Algorithms for Massive Datasets

Finding Similar Items

Gregorio Luigi Saporito
ID: 941503

April 19, 2021

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.

Contents

1 Introduction

The aim of this project is to identify a scalable solution to detect pairs of similar items. Finding all the possible exact matches can already be a challenging task when the dataset considered is large. This challenge is further exacerbated when, instead of exact equality, similarity is considered. The dataset chosen consists of Stack Overflow questions retrieved from Kaggle. To find similar items an application of locality sensitive hashing (LSH) designed for the Jaccard distance is considered. This approach relies on hashing techniques which allow finding pairs that are likely to be similar. This algorithm is more scalable than a typical "brute force" strategy involving the comparison of all possible pairs. The latter process would indeed be considered quadratic with its complexity rapidly increasing as the dataset size grows. The similarity metric chosen in this context is the Jaccard similarity since documents can be thought of as sets of tokens. After the application of some preprocessing techniques like the removal of stop words the idea is that documents with many tokens in common have high chances of being about the same topic. LSH can have many applications also in other contexts like identifying mirror sites and recommendation systems. In the particular scenario considered in this project (i.e. finding similar Stack Overflow questions), LSH can be exploited to avoid redundant questions and direct users with similar questions in the same area. A series of experiments are carried out and discussed to see how Spark (set up on Google Colab) scales as the dimensionality of the dataset grows. The final results of the algorithm are then shared along with some examples of questions inferred as similar.

2 The Dataset

The data processed in this project is retrieved from Kaggle[1] and imported in Google Colab through Kaggle's API. The data is called **StackSample** and it contains information about Stack Overflow questions and answers centred around coding topics. In particular, the data consists of three relational tables:

- **Answers.csv** (1.5 GB) which contains answers to specific questions raised on the platform and it includes variables like the answer's id, date, and its content
- **Questions.csv** (1.79 GB) which revolves around questions posed on Stack Overflow and it includes features such as the question's title, data, and its content in string format
- **Tags.csv** (62.44 MB) which stores all the tags referred to a specific question containing some relevant keywords

[1]The link to the dataset is `https://www.kaggle.com/stackoverflow/stacksample`. The data is released under the CC-BY-SA 3.0 license, with attribution required.

In this project only the **Questions.csv** file is loaded into Spark and it contains 6 variables:

- **Id** which is the key of the relational table and it univocally identifies each question
- **OwnerUserId** which univocally identifies the user
- **CreationDate** which stores the day the question was posed
- **ClosedDate** (if applicable) which stores the closing date
- **Score** which is calculated as the number of upvotes - number of downvotes
- **Title** which tracks each question's title
- **Body** which stores the actual content of the question

The columns **Body** and **Id** are those that are considered in the analysis and during the data processing stages.

3 Data Organisation

The data is imported directly in Colab through Kaggle's API and unzipped. The files Answers.csv and Tags.csv are removed to free up memory space because they are not needed for the analysis. A Spark environment is set up to load the Questions.csv file as a Spark DataFrame by using the `spark.read` command. For performance reasons, Spark reads .csv files line by line and newline characters can cause problems for the parser. In this case, the Body column in Questions.csv contains HTML syntax with characters like `"\n"` and `"\r"` which can compromise the correct loading of the dataset. Without specifying a suitable escape character, the parser would load part of the string until it finds a newline character. The remaining part of the string would be then loaded in a new row of the DataFrame incorrectly splitting the original string and often assigning part of it to the wrong column. Most of the remaining columns in the row would be then filled with null values. To solve this problem the escape character `"\""` is manually specified as an option when loading the .csv file with Spark, resulting in the correct loading of the dataset. A total number of **1,264,216** observations are counted with the Spark `.count` command. Table 1 shows a snapshot with a few rows of the loaded dataset and the selected variables of interest (i.e. the columns Id and Body).

Id	Body
80	<p>I've written a database generation script in...
90	<p>Are there any really good tutorials explaini...
120	<p>Has anyone got experience creating S...
180	<p>This is something I've pseudo-solved many ti...
260	<p>I have a little game written in C#. It uses ...

Table 1: Snapshot of the first five rows of the dataset loaded into Spark

4 Data Cleaning and Preprocessing

Firstly, an inspection to identify the potential presence of missing values is carried out. For this purpose, a counter to detect nan and null values is constructed. As Table 2 shows, no missing values are present in the columns Id and Body.

Id	Body
0	0

Table 2: Inspecting missing values in the columns Id and Body

The next step in the process is removing the HTML syntax, including the newline characters mentioned in the data organisation section. This is achieved using regular expressions and constructing a function that is then applied to the Body column with the `.select` command. Table 3 shows how the first rows of the DataFrame look like after the data cleaning function is applied to the Body column.

Id	Body
80	I've written a database generation script in SQL...
90	Are there any really good tutorials explaining b...
120	Has anyone got experience creating SQL-based ASP...
180	This is something I've pseudo-solved many times ...
260	I have a little game written in C#. It uses a da...

Table 3: How 5 rows of the DataFrame look like after removing the HTML syntax

The text can now be tokenised using the `pyspark.ml.feature` `RegexTokenizer` and applying the method `.transform` on the DataFrame concerned. After the strings are tokenised stop words are removed to reduce the number of tokens to process and remove features that provide little information about how similar two documents are since they occur very frequently. This is achieved with the `pyspark.ml.feature` `StopWordsRemover`. Since the mathematical abstraction adopted in this context is

grounded on set theory no duplicate tokens are allowed for each document (a set is simply an unordered collection of objects and the Jaccard similarity is designed to work with sets). To remove duplicates the method `array_distinct` from the `pyspark.sql.functions` module is used. Table 4 summarises the preprocessing steps which are applied to the documents.

Id	Body	Tokens	No Stop Words	No Duplicates
80	I've writt...	[i, ve,...	[ve, written,...	[ve, written,...
90	Are there ...	[are, there,...	[really, good,...	[really, good,...
120	Has anyone...	[has, anyone,...	[anyone, got,...	[anyone, got,...
180	This is so...	[this, is,...	[something, ve,...	[something, ve,...
260	I have a l...	[i, have,...	[little, game,...	[little, game,...

Table 4: 5 rows showing the preprocessing pipeline applied to the documents

5 Algorithms Considered and Implementation

Finding exact matches can already be challenging with large datasets. This issue is further exacerbated when, instead of perfect equality, the goal is to find similar items. Since the aim of this project is to find a solution that scales with large datasets, the MinHash algorithm for the Jaccard Distance is chosen and it is implemented using Spark 3.1.1[2]. Minhash is designed to work with the Jaccard distance and it belongs to the LSH family. The advantage of using hashing relies on its speed and higher scalability compared with a naive brute force approach which will be discussed more in detail in the scalability section of this report. The main disadvantage of MinHashing, however, is that it is a non-deterministic algorithm. It is therefore prone to have false positives and negatives but, if properly designed, these can be minimised resulting in an efficient and effective performance. This is particularly true when the goal is to identify the most similar items above a user-defined similarity threshold.

The Jaccard similarity is defined as the cardinality of two sets' intersection over their union and it is described by the following mathematical expression:

$$s(I, J) = \frac{|I \cap J|}{|I \cup J|}$$

The Jaccard distance corresponds to 1 minus the Jaccard similarity of the corresponding sets. As specified in the Spark documentation, MinHash computes the

[2] `https://spark.apache.org/releases/spark-release-3-1-1.html`

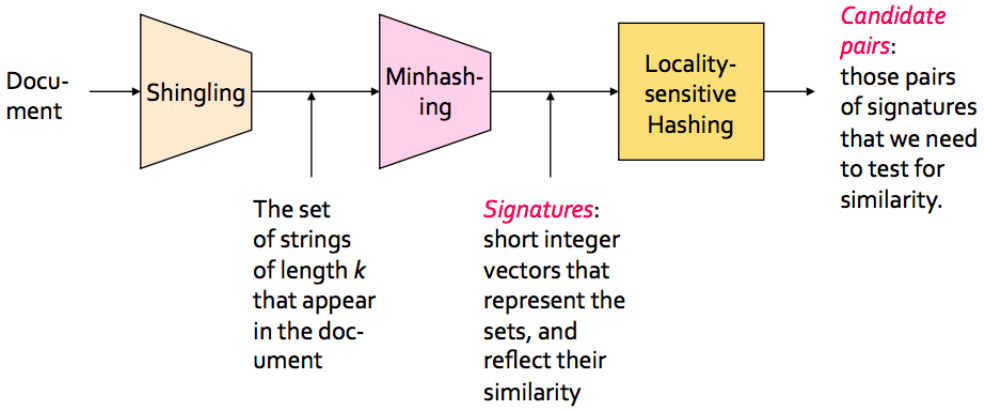

Figure 1: Process to find candidate pairs (Leskovec et al., 2019).

minimum of all hashed values by applying a random hash function g to each set element[3].

$$h(I) = \min_{i \in I} g(i)$$

Mathematically it can be proven that the probability that a hash function returns the same value for two documents corresponds to the Jaccard similarity of the documents concerned. The inputs of the MinHash algorithm are dichotomous vectors with zeros indicating the absence of a specific element in the set and 1 indicating its presence. Operationally, Spark allows for both dense and sparse vectors. Then `.approxSimilarityJoin` allows to approximately join two datasets on rows whose corresponding distance metric (in this case the Jaccard distance) is smaller than a user-defined threshold. A threhsold of 0.5 is set in this project but it can be tuned accordingly based on a user's specific requirements. While this approximate similarity join supports the use of two distinct datasets, for the purposes of this project a self-join is considered allowing us to identify the possible pairs of documents in the StackSample corpus. Figure 1 illustrates the process to find candidate pairs with Minhash and LSH.

6 Scalability

The solution proposed in this project is scalable for two main reasons. Firstly, it is implemented in Spark. By relying on in-memory computations and a distributed logic Spark can scale more than a traditional computational environment while being generally faster than Hadoop. Should the dataset be larger than the one considered

[3] `https://spark.apache.org/docs/3.1.1/ml-features.html#minhash-for-jaccard-distance`

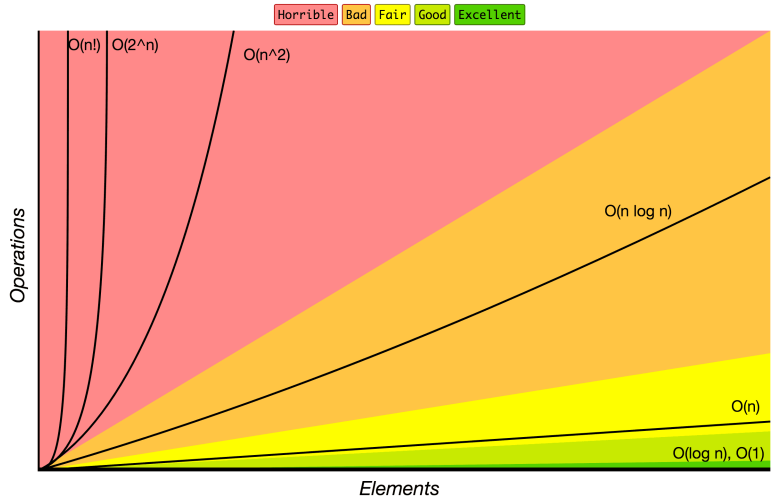

Figure 2: Big O Complexity Chart.

in this project more machines could be simply added to the cluster allowing the environment to handle more data-intensive operations. Secondly, to address the task of finding similar items Minhash in the LSH family is implemented. The advantage of this particular hashing technique relies precisely on its scalability. As a benchmark, we can consider the typical brute force approach required to compare all possible pairs of items which would be deemed as quadratic (i.e. `O(n^2)`) and therefore computationally challenging. As a reference, Figure 2 describes the order of complexity of some computational problems. To summarise, the theoretical solution adopted in this project and its operational implementation with Spark are in a good position to scale with larger datasets. From an experimental standpoint, the scalability of this specific project is put to the test by running a series of experiments. The size of the StackSample dataset is progressively increased while tracking the time required for the computations. More details about the experiments are discussed in the experiments section.

7 Experiments

A total of 16 experiments are carried out to track the computational time required to run the LSH algorithm mentioned in section 5 as the size of the dataset grows. The results are obtained by progressively increasing the parameter n in the `.limit(n)` clause applied to the Spark input DataFrame (the parameter allows us to work on a smaller portion of the data). The time required for the LSH algorithm to return the results is then tracked with the `.time()` method from the `time` module. Figure 3 shows a visual representation of the performance of the experiments (in blue) coupled with a linear trend (in green). Now assuming an analogous behaviour for

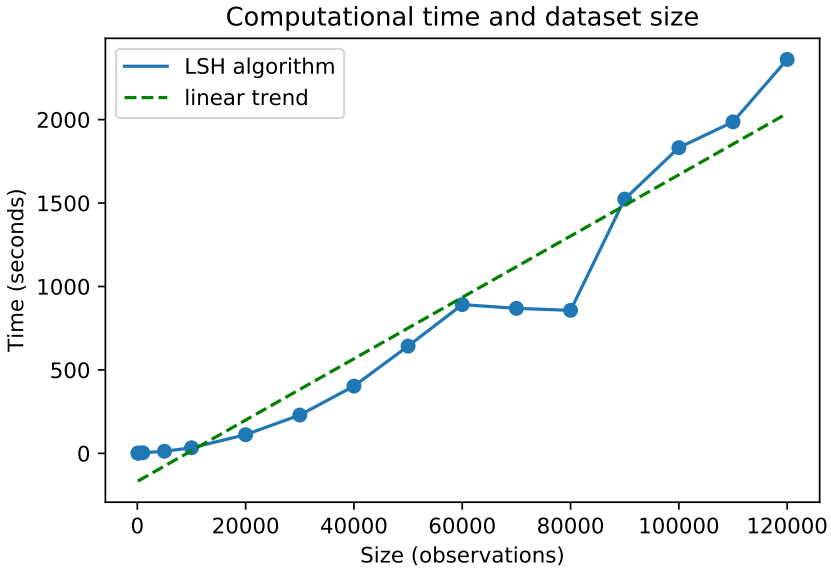

Figure 3: Experiments showing the time required to run the LSH algorithm

the out of sample observations and assuming a linear trend, the time required to process the entire StackSample dataset (around 1.2 million observations) with the technology considered would be of the order of hours.

8 Results

For simplicity and faster reproducibility of the results, the final analysis is performed on a smaller portion of the dataset (20,000 observations). Arrays with empty tokens in the input dataset are filtered out since Minhash cannot work with empty sets[4]. The initial pipeline consists of two stages:

- the use of the HashingTF transformer to count the number of tokens present in each document (the transformer exploits a hashing trick to further enhance the scalability of the solution)
- the application of MinHashLSH.

The standard `.fit` and `.transform` methods are applied to the input DataFrame. Finally, `approxSimilarityJoin` is applied over the hashed outputs to perform an approximate self-join on a Jaccard distance smaller than 0.5. Table 5 shows the first rows of the outputs showing the matches identified. As expected, matches

[4] `https://spark.apache.org/docs/latest/ml-features#minhash-for-jaccard-distance`

corresponding to the same Id have a Jaccard distance of zero which satisfies one of the mathematical properties that a distance should have.

Id	Id	Distance
80	80	0.0
90	90	0.0
120	120	0.0
180	180	0.0
260	260	0.0

Table 5: Matches found (including duplicates)

Table 6 shows the matches identified after the removal of duplicate pairs in the smaller portion of the dataset considered.

Id	Id	Distance
446500	446600	0.489
503310	835280	0.375
612820	634630	0.491
1041520	1042370	0.304

Table 6: Matches found (excluding duplicates)

To contextualise these results, some examples of similar questions are shown in table 7.

9 Conclusions

To summarise, this project aimed to identify a scalable solution to detect pairs of similar items. The intended goal was to find a scalable solution both from a theoretical and operational standpoint. This was achieved by using LSH for the Jaccard distance and by implementing the solution in Spark 3.1.1 respectively. The data was imported, cleaned, and preprocessed before giving it as input to the algorithm. A series of experiments were carried out to see how Spark combined with LSH scales as the size of the dataset grows. The final results were shown along with some examples of similar question to assess the performance of the algorithm.

Id	Body	Distance
612820	I have a property grid that helps me manage all of the controls on a form. These controls are for designer-type folks, so I'm not really worried that much about the user interface... until someone selects multiple objects.I have a UITypeEditor for the "BottomDiameter" property on these common objects. [...]	0.491
634630	I have a property grid that helps me manage all of the controls on a form. These controls are for designer-type folks, so I'm not really worried that much about the user interface... until someone selects multiple objects.I have a UITypeEditor for the "EffectiveDiameter" property on these common objects. [...]	
1041520	I have a rewrite rule set up in my [...] Apache throws a 404. The PHP logic defaults to page 1 without a page specified, so I know the script is fine.What can I do?ThanksNick	0.304
1042370	Hey everyone, I'm having rewite issues. [...] Apache throws a 404. The PHP logic defaults to page 1 without a page specified, so I know the script is fine. [...] What can I do?Thanks Nick	
503310	Is it possible to map an enum as a string using Fluent Nhibernate?	0.375
835280	It's possible to map a VIEW using Fluent NHibernate? If so, how?	

Table 7: Examples of similar questions detected