Finding Similar items in StackSample

Algorithms for Massive Datasets

Università degli Studi di Milano

Pramodh Chinthareddy (matr. 936461)
pramodh.chinthareddy@studenti.unimi.it

Ricardo Murillo Rapso (matr. 923159)
ricardo.murillorapso@studenti.unimi.it

January 8, 2021

Abstract The present assignment finds similar items from the Questions found in the *StackSample: 10% of StackOverflow Q&A*, by using Term Frequency Inverse Document Frequency, Hashing Term Frequency, Locality Sensitive Hashing with the Jaccard Distance, to finally complete a predictor with Approximate Neighbor Search. The solution to this task is given in scalable and reproductible form. All the materials for this project can be found on the following GitHub entry.

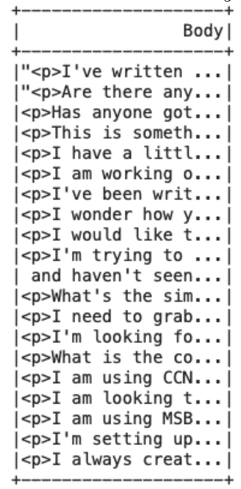
Keywords: *Finding similar items, Large Datasets, Nearest Neighbor Search, Hashing*

Contents

1 Data description

One of the problems of data science is finding similar items [1], specially in the context of large data sets. This is because it is not possible to make analysis by simple human examination, and specialized computer method, and a great effort on data preprocessing and cleansing is needed [2].
On the context of this assignment, the *StackSample* dataset that is found in *Kaggle* [3] will be used to find similar items from the different questions that conform it . The dataset is made with the text of 10% of questions and answers from the Stack Overflow programming Q&A website and it is organized in three tables, *Questions, Answers and Tags* [4], from which this project is only interested in the first one.
The table *Questions*, contains the title, body, creation date, closed date score, and owner ID for each of the questions of the database. Each user is assigned with an ID which is a multiple of 10.

Figure 1: Sample of entries from *Body* in *Questions* table.



```
+--------------------+
|                Body|
+--------------------+
|"<p>I've written ...|
|"<p>Are there any...|
|<p>Has anyone got...|
|<p>This is someth...|
|<p>I have a littl...|
|<p>I am working o...|
|<p>I've been writ...|
|<p>I wonder how y...|
|<p>I would like t...|
|<p>I'm trying to ...|
| and haven't seen...|
|<p>What's the sim...|
|<p>I need to grab...|
|<p>I'm looking fo...|
|<p>What is the co...|
|<p>I am using CCN...|
|<p>I am looking t...|
|<p>I am using MSB...|
|<p>I'm setting up...|
|<p>I always creat...|
+--------------------+
```

Questions has a total of 33,149,724 observations, of which 1,404,281 correspond to the non-null entries of the variable *Body.* Figure 1 shows a small snippet of the questions from the Body column in the database. It can be noted almost immediately that the database is in need of cleaning just by the presence of elements such as *<p>*. The second table, *Answers*, contains also a *Body* variable corresponding to each of the answers, an *Id* for the owner of the answer, and a *Parent ID* variable, that provides an association with the *Questions* table. Lastly, the *Tags* table, contains tags for each of the questions [3].

2 Data Organization

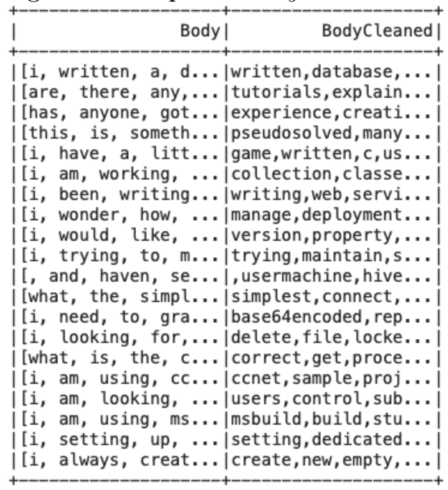
The *Spark DataFrame API* was used for loading the data, preprocessing and building the detector of similar items. *Spark DataFrame* is a dataset organized into named columns, conceptually equivalent to a table in a relational database or a data frame used in by programs such as *R* or *Python*, but capable of better optimization[5]. *Spark SQL* is a module for structured data processing, that provides information about the

structure of both the data and the computation that is being performed, and provides rich integration between *SQL* and regular *Python, Java or Scala* code, including the ability to join *RDDs*, represent a collection of items distributed across many compute nodes that can be manipulated in parallel, and SQL tables [5]. Internally, *Spark SQL* uses this extra information to perform extra optimizations.
There are several ways to interact with *Spark SQL*, including *SQL*, and *Dataset API*, and when computing a result, the same execution engine is used, independent of which API/language is used to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation [6], meaning that when running *SQL* from within another programming language, like it will be done on this project, the results will be returned in the form of a Dataset or DataFrame.

3 Data Pre-Processing

Data cleaning is the first step and also one of the most important ones in every data science project [2]. As it was noted previously, most of the entries in the *Body* column of the *Questions* table are *null*. These were removed reducing the amount of entries in the database from 35 million to about 1.4 million rows, each representing a different question entry.
All these rows contained strings such as *<p>, </p>, <code>, </code>, <strong>, </strong>*, etc, and since these string did not add up any useful information to the analysis, they were removed from the data using Regular expression replace, *regexp_replace* function from the *pyspark.sql.functions* library. It is worth to note that the data cleaning process for 1.4M rows was done in seconds, compared to an equivalent operation in *Pandas* that could take several minutes. Similarly, word contractions such as *I've, That's, I'm, can't*, were stripped of their string patterns in the end. HTML tags were also removed.

Figure 2: Sample of *Body* after cleaning



```
+--------------------+--------------------+
|                Body|         BodyCleaned|
+--------------------+--------------------+
|[i, written, a, d...|written,database,...|
|[are, there, any,...|tutorials,explain...|
|[has, anyone, got...|experience,creati...|
|[this, is, someth...|pseudosolved,many...|
|[i, have, a, litt...|game,written,c,us...|
|[i, am, working, ...|collection,classe...|
|[i, been, writing...|writing,web,servi...|
|[i, wonder, how, ...|manage,deployment...|
|[i, would, like, ...|version,property,...|
|[i, trying, to, m...|trying,maintain,s...|
|[, and, haven, se...|,usermachine,hive...|
|[what, the, simpl...|simplest,connect,...|
|[i, need, to, gra...|base64encoded,rep...|
|[i, looking, for,...|delete,file,locke...|
|[what, is, the, c...|correct,get,proce...|
|[i, am, using, cc...|ccnet,sample,proj...|
|[i, am, looking, ...|users,control,sub...|
|[i, am, using, ms...|msbuild,build,stu...|
|[i, setting, up, ...|setting,dedicated...|
|[i, always, creat...|create,new,empty,...|
+--------------------+--------------------+
```

The next step was removing stop-words, using the *StopWordsRemover* function from *pyspark.ml.feature* library. Unfortunately, this is not sufficient to eliminate all the unnecessary words such as *really, are , good , anyone , someone*,and others, so, a custom stop word list was added to the existing library. Figure 2 shows a sample of how the preprocessed data looks after the cleaning process. It can be noted that the column *BodyCleaned* does not contain any of the above-mentioned unnecessary strings or stop-words.

4 Implementation of the classification algorithm

Various algorithms used to build the similar items detector, starting with the *tokenizer*, which takes text and divides it into individual words mostly [6].

4.1 TF-IDF

Term frequency-inverse document frequency (*TF-IDF*) is a feature vectorization method that weights the importance of a term to a document in the corpus[2]. If we have a term *t*, a document by *d*, and corpus *C*, the *Term frequency TF(t,d)* is the number of times that term t appears in document *d*, while document frequency *DF(t,C)* is the number of documents that contains term *t* [6].
IDF has as limitation that it sees words as *bags of words*[2], meaning that, it does not take into account their context. A measure of importance of each term can be that one used in Equation 1, where is a numerical measure of how much information a term provides:

$$IDF(t, C) = \log \frac{|C| + 1}{DF(t, C + 1} \tag{1}$$

|C| is the total number of documents in the corpus. Since logarithm is used, if a term appears in all documents, its IDF value becomes 0. Note that a smoothing term is applied to avoid dividing by zero for terms outside the corpus. The TF-IDF, see Equation 2measure is simply the product of TF and IDF:

$$TFIDF(t, dC) = TF(t, d).IDF(t, C) \tag{2}$$

HashingTF was used to generate the term frequency vectors. A hash function h takes a hash-key value as an argument and produces a *bucket* number as a result[1]. The bucket number is an integer, that belong to the interval 0 to $B - 1$, where B is the number of buckets[1, 6].
The algorithm sets of terms and converts those sets into fixed-length feature vectors, if hash-keys are an array, set, or bag of elements of some one type, in this case, elements are strings, the algorithm converts each character to its ASCII or Unicode equivalent, which can be interpreted as a small integer. After this, it sums the integers, and then divides them by B [1]. *HashingTF* utilizes the hashing trick[6] which consists on mapping into an indexes by applying a hash function, in this case *MurmurHash 3.* A *collision* is when two rows get the same hash value [1], and in order to prevent them, the target feature dimension was increased in the power of 2^3.

4.2 Inverse Document Frequency

Inverse Document Frequency *(IDF)* scales the features of vectors product of HashingTF with weights, and down-weights features which appear frequently in a corpus[6]. So far the process has started with a set of sentences, in the form of a dataset, after pre-processing the *Questions* table, then, each sentence was divided into words using *Tokenizer*. For each sentence, or bag of words, *HashingTF* was used to hash the sentence into a feature vector. Later, IDF was used to rescale the feature vectors, improving performance when using text as features. The feature vectors are now ready to be passed to a learning algorithm *LSH (Locality Sensitive Hashing)*.

4.3 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a class of hashing techniques, which consists in hashing items several times in a form that the most similar ones will have a larger probability to be hashed to the same bucket, than disimilar ones [1]. *LSH* is used in clustering, approximate nearest neighbor search and outlier detection with large datasets[6]. There are several *LSH* families that can be used depending on the data, and for this purpose *Minhash for Jaccard Distance* was used with a total amount of five buckets.

4.4 MinHash for Jaccard Distance

As is was established before, *MinHash* is an *LSH* family for *Jaccard distance* where the input features are sets of natural numbers. The *Jaccard distance* of two sets is defined by 1 - the ratio of the sizes of the intersection and union of each of the sets[1], as it can be seen in Equation 3:

$$d(\mathbf{A},\mathbf{B}) = 1 - \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|} \tag{3}$$

MinHash applies a random hash function g to each element in the set and take the minimum of all hashed values[6]:

$$h(\mathbf{A}) = \min_{a \in \mathbf{A}}(g(a)) \tag{4}$$

A *MinHash* function on sets is based on a permutation of the universal set, where the minhash value for a set is that element of the set that appears first in the permuted order[1]. For example, Vectors.sparse(10, Array[(2, 1.0), (3, 1.0), (5, 1.0)]) means there are 10 elements in the space. This set contains elem 2, elem 3 and elem 5. All non-zero values are treated as binary "1" values. Note: Empty sets cannot be transformed by MinHash, which means any input vector must have at least 1 non-zero entry.

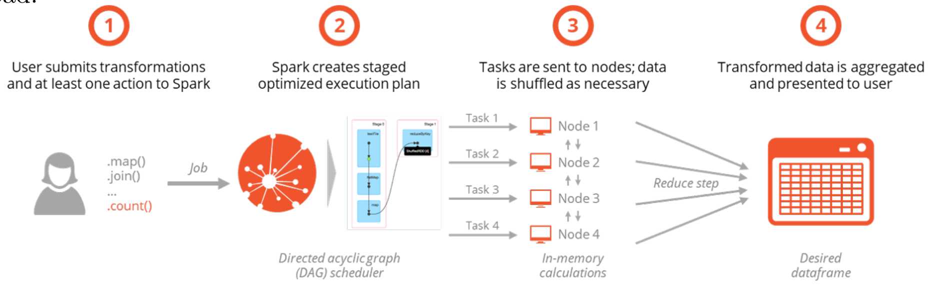
4.5 Approximate Nearest Neighbor Search

Approximate nearest neighbor search takes a dataset (of feature vectors) and a key (a single feature vector), and it approximately returns a specified number of rows in the dataset that are closest to the vector. Approximate nearest neighbor search accepts both transformed and untransformed datasets as input. If an non transformed dataset is used, it will be transformed automatically. In this case, the hash signature will be created as outputCol. A distance column will be added to the output dataset to show the true distance between each output row and the searched key. Note: Approximate nearest neighbor search will return fewer than k rows when there are not enough candidates in the hash bucket.

5 Scalability

Spark can load data into memory on the worker nodes, many distributed computations, even ones that process terabytes of data across dozens of machines, can run in a few seconds. This makes the sort of iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark. Spark provides both Python and Scala shells that have been augmented to support connecting to a cluster [5]

Figure 3: Sketch by *Towards Data Science*[7] illustrating how *Spark* optimizes workload.


1
User submits transformations and at least one action to Spark
2
Spark creates staged optimized execution plan
3
Tasks are sent to nodes; data is shuffled as necessary
4
Transformed data is aggregated and presented to user
.map()
.join()
...
.count()
Job
Task 1
Task 2
Task 3
Task 4
Node 1
Node 2
Node 3
Node 4
Reduce step
Directed acyclic graph (DAG) scheduler
In-memory calculations
Desired dataframe

Our detector is fast and efficient and scales with the data size. This is because of the usage of Spark, which uses Regular expressions and Spark DataFrame based API, and the default datatype is “String”, which occupies relatively very less memory. Apache Spark is a unified analytics engine for large-scale data processing. It achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine. Spark’s Catalyst optimizer converts user inputs into a *Directed Acyclic Graph (DAG)* of transformations before distributing the optimized tasks across a network of computers for in-memory processing. Hence, the reason, the data cleaning was done in seconds.

6 Description of experiments

Every question (each row/also referred as document) is Tokenized first. The extracted tokens for each row are vectorized (of 8 vectors each) using HashingTF (i.e., Term Frequency). These features are used to compute the Inverse document frequencies (using IDF). Finally the TfIdf features for each row (question) are computed (of 8 vectors each). Our feature vectors are passed to a learning algorithm LSH (Locality Sensitive Hashing). Here we classified the questions into 5 buckets using MinHashLSH (a family of LSH algorithms). Finally, we use approxNearestNeighbors method to group the similar questions by passing the TfIdf feature vectors and a single feature as a key.

7 Results and Discussion

We have built a detector to group similar questions from the Questions posted in the Stack Overflow website (10% of it's questions). Spark ML library is a reliable source for this task, while we used the Spark DataFrame based API, the same could be implemented using the RDD based API. Due to it's unified analytics engine, the transformations are done in a distributed manner, and hence the pipeline scales with data size.

References

[1] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets.* Cambridge University Press, 3 ed., 2014.

[2] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark.* O'Reilly Media Inc., 1 ed., 2015.

[3] Kaggle, "StackSample: 10% of StackOverflow Q&A," 2020. Retrieved from Kaggle, `https://www.kaggle.com/stackoverflow/stacksample`.

[4] Stack Overflow, "StackSample: 10% of StackOverflow Q&A," 2019. data retrieved from Kaggle, `https://www.kaggle.com/stackoverflow/stacksample`.

[5] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark.* O'Reilly Media Inc., 1 ed., 2015.

[6] Apache.org, "PySpark 2.2.0 documentation: pyspark.ml package," 2020. Retrieved from Apache.org, `https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.linalg.SparseVector`.

[7] Towards Data Science, "Will it scale?," 2019. Retrieved from Towards Data Science, `https://towardsdatascience.com/will-it-scale-1f4996e940ff`.