Federico Matteucci - 955753

Project of Algorithms for Massive Datasets

Abstract

Goal of this report is to show methods to find similar documents in a corpus. Specifically, I will analyze the procedure of comparing documents from preprocessing and text cleaning to the definition of a similarity score. All of the data structures used in the attached notebook are defined in the `pyspark` module, relying on the Spark framework for scalability.

Dataset

The dataset I will refer to throughout this analysis is `Questions.csv`, part of the Stacksample dataset. It contains 1264204 non-null entries of questions posted on StackOverflow, with some metadata that falls out of the scope of this analysis.

The dataset was loaded in a `pyspark.sql.DataFrame` object (later referred to as `dataframe`) named `df`, that will be used later to retrieve information about documents of interest.

Schema of `df`

```
df

|-- Id:  string (nullable = true)

|-- OwnerUserId:  string (nullable = true)

|-- CreationDate:  string (nullable = true)

|-- ClosedDate:  string (nullable = true)

|-- Score:  string (nullable = true)

|-- Title:  string (nullable = true)

|-- Body:  string (nullable = true)
```

In further analysis I will only consider a sample of about one tenth of `df.select([’Id’, ’Body’])`, named `df1`.

Processing

Code column

As this report is dealing with StackOverflow questions that often contain code sections (delimited by the `<code>` HTML tag), comparing separately the code and the text of two questions can yield better results. Therefore, the very first action is to go through each document, isolate the code section and remove it from the text of the document.

Note that it is possible for a document to have an empty `Code` or `Body` field, so this is taken into account when computing the similarity. About 74% of documents considered in the sample contain code.

Extracting text from the sample dataframe `df1` yields `df2`.

Schema of `df2`

```
df2

|-- Id:  string (nullable = true)

|-- Code:  string (nullable = true)

|-- Body:  string (nullable = true)
```

Tokenization

In order to compare two documents, they need to be transformed into numeric arrays. Prior to that, they need to be tokenized, that means they have to be reshaped into a list of tokens.

Tokens are usually made of *n*-grams, contiguous sequences of *n* words (a word is delimited by punctuation and whitespace characters), although it is possible to generalize this notion to skipgrams (to name one) that allow a better grasp over the context in which a word appears.

For the sake of this analysis, I will use unigrams for both code and body tokenization. I built two tokenizers , one for the code and one for the body section, that differ for some characteristics (such as removing stopwords and conversion of lower case).

The tokenization of `df2` is `df3`.

Schema of `df3`

```
df3
```

```
|-- Id:  string (nullable = true)
|-- tkBody:  array (nullable = true)
| |-- element:  string (containsNull = true)
|-- tkCode:  array (nullable = true)
| |-- element:  string (containsNull = true)
```

Vectorization

Vectorizing a tokenized document is the action of mapping the tokenization into an n dimensional euclidean space, where n is the size of the vocabulary (the number of unique tokens in the corpus). To prevent sparsity, one can represent a vectorized document as a python `dictionary`whose keys are the tokens appearing in the tokenized document and whose values depict how relevant the token is to the document.

One possible measure of this relevance`tf-idf` , that requires the **term-frequency** (`tf`, the relative frequency of the token in the document) and the **inverse-document-frequency** (`idf`, the inverse of the natural logarithm of the number of documents a token appears in).

As `idf` is expensive to compute and the focus of this report is on computational complexity rather than quality of the results, I will run a naive `tf` vectorization (also, as stopwords were removed, the problem of very frequent words in the corpus is at least partly relieved).

The only operation performed at this stage is the vectorization of `df3` in `df4`.

Schema of `df4`

```
df4
|-- Id:  string (nullable = true)
|-- vectkBody:  map (nullable = true)
| |-- key:  string
| |-- value:  double (valueContainsNull = true)
|-- vectkCode:  map (nullable = true)
| |-- key:  string
| |-- value:  double (valueContainsNull = true)
```

Similarity

To measure the affinity of two documents I will use `cosine similarity` for mainly these reasons:

1. it does not depend on the length of the documents involved nor on their norm
2. it can be natively calculated on a dictionary representation of the two documents
3. it takes values between 0 and 1 so it is interpretable (at least to some extent).

As already mentioned above, some documents can lack either the code section or the body section. In case this happens, I will set their cosine similarity to a default of -1 and ignore it in the next computations.

As for the formula itself: given two arrays $\underline{x}$ and $ y $ their cosine similarity is the cosine of the angle between the two, that is: $cos\_sim(\underline{x}, \underline{y}) = \frac{\underline{x} \cdot \underline{y}}{||\underline{x}|| \cdot ||\underline{y}||}$.

When translated in the dictionary language, it becomes: $cos\_sim(D, D') = \frac{\sum\limits_{j \in K \cap K'} v_k \cdot v'_k}{||D|| \cdot ||D'||}$, where:

- D, D' are dictionaries with keys K, K' respectively
- v_k is the value associated to key k in D (v'_k is defined analogously)
- $||D||^2 = \sum\limits_{k \in K} v_k^2$

Eventually, to aggregate the two scores (of body similarity and code similarity) into a single index, I aggregated them using an harmonic mean. Why the harmonic mean? Because it penalizes pair of documents which are very similar with respect to one measure and very little with respect to the other. The functions to compute the harmonic means also deal with the zeros and negative values output from cosine similarity.

Analysis

Now that all of the instruments are defined and ready, I can use them to solve the problem of finding similar items.

One vs all

A first application of what seen above is to have a document D and a corpus C as inputs and compare D with all of the documents in C to find the most similar one.

Operatively, this requires to compute the similarity of each row of the vectorized dataframe `df4` with D , storing the results in new columns and eventually sorting on `Similarity_with_[D]`, obtaining `df6`.

Case study

I chose to compare document `13160` against `df4`. The result is that the most similar document is `10560310`, with a body similarity of 0.45, a code similarity of -1 and an overall similarity of 0.45 (quite a low score).

Let's see them:

```
13160 :  Best practice for webservices

10560310 How do I create unicode characters with variable numbers?
```

This result is not encouraging, as the two documents are clearly very different in their bodies (I provide the body content in the appendix and in the notebook). My guess is that the algorithm was tricked by the fact that document `10560310` is very short and most of the words appearing there are in `13160` as well.

Schema of `df6`

```
df6

|-- Id:  string (nullable = true)

|-- CodeSim_with_13160:  double (nullable = true)

|-- BodySim_with_13160:  double (nullable = true)

|-- Similarity_with_13160:  double (nullable = true) | sorted
desc
```

All vs all

Finally, let's compare a corpus with itself and return the pair of documents inferred to be the most similar.

The operations performed are:

1. Sample the dataframe containing the vectorizations (`df4`) obtaining `df8`

    - Required to keep space and time complexity under control, as this section runs in $\mathcal{O}(N^2 \log(N))$

2. Cross-join the vectorized dataframe (`df8`) with itself, obtaining `df9`
    - In order to further optimize the computation I kept only the pairs where the `Id` of the first document is strictly less than that of the second one, removing duplicates and pairs of twins
3. Compute the similarities (body, code and overall) of each row of `df9`
4. Sort `df9` over its `Similarity` column, obtaining `dfA`
5. Show the first rows of the result

Results

In the sample considered, the most similar documents are `27293230` and `6547590`, with a `BodySim` of 0.78, a `CodeSim` of -1 and therefore a `Similarity` of 0.78.

Let's confront them directly (their entire texts are on display in the Appendix and on the notebook):

```
6547590 :  EditText in Google Android
```

```
27293230 :  How to get index of UI?
```

Their titles are quite different and their bodies are both small. More, one of them does not contain any code. This is a clear example of a false positive, as the documents do share some words but the overall meaning is completely different.

More interesting results

Let's instead consider the most similar pair with positive `CodeSim` and `BodySim`: that is pair `32332430` (`Fetch array after mysqli_multiple_query() function`) and `39351520` (`some css not working`).

These are much longer documents with extensive HTML-like code. I am quite satisfied with this result even though I think the algorithm was tricked by the high number of `&lt;` and `&gt;` symbols, that are frequent in HTML as they delimit tags.

DB Schema of `dfA`

```
dfA

|-- Id1:  string (nullable = true)

|-- Id2:  string (nullable = true)
|-- BodySim:  double (nullable = true)
|-- CodeSim:  double (nullable = true)
|-- Similarity:  double (nullable = true) | sorted desc
```

Time complexity

Assume that the longest document has length L and that the corpus is composed of N documents.

The steps are:

1. **Extracting code from a text :** $\mathcal{O}(L)$ [*it's a regex call , source*]
2. **Tokenizing a document :** $\mathcal{O}(L)$ [*splitting a string of length* $\leq L$]
3. **Vectorizing a document :** $\mathcal{O}(L)$ [*counting over a list of length* $\leq L$]
4. **Computing the similarity of two documents :** $\mathcal{O}(L)$ [*constant time operations repeated once for each common key in the dictionary representation of the two documents*]
5. **Cross join :** $\mathcal{O}(N^2)$ [*taking pairs of records*]
6. **Sorting the cross-joined dataset :** $\mathcal{O}(N^2 \log N)$ [*source*]

We perform operations $1-3$ once for each document, operation 4 once for each pair of documents, operations 5 and 6 once overall.

Therefore, the time complexity of the algorithm falls into $\mathcal{O}(L \cdot N^2 \cdot \log N)$.

Conclusions

The techniques shown here give hints of what a more refined and computationally requiring algorithm can achieve in terms of document comparison. Some possible improvements:

- Tokenizing in skipgrams instead of unigrams
- Identifying the programming language involved in the question, if any
  - For instance, one could recognize a strongly typed language from its syntax or even from stylistic properties (such as SQL keywords being most often written upper case)
- Lemmatization / stemming
- Using `tf-idf` vectorization rather than `tf` vectorization
- Dropping questions that are too small, as they are likely to give false positives (that is, theye are likely to be considered similar to other small documents)

Sources

- https://databricks.com/glossary/catalyst-optimizer
- https://stackoverflow.com/questions/32123887/apache-spark-rdd-sortbykey-algorithm-and-time-complexity
- https://stackoverflow.com/questions/4378455/what-is-the-complexity-of-regular-expression
- *Lecture notes*
- *Leskovec, Rajarama, Ullman: Mining of Massive Datasets*

Appendix

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 thatn 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.

Documents

For a better formatting of the code sections, check the attached notebook.

One vs all

Document 13160

- Id: `13160`
- Title: `Best practice for webservices`

I've created a webservice and when I want to use its methods I instantiate it in the a procedure, call the method, and I finally I dispose it, however I think also it could be okay to instantiate the webservice in the "private void Main_Load(object sender, EventArgs e)" event.

The thing is that if I do it the first way I have to instantiate the webservice every time I need one of its methods but in the other way I have to keep a webservice connected all the time when I use it in a form for example.

I would like to know which of these practices are better or if there's a much better way to do it

Strategy 1

Strategy 2

Document 10560310

- Id: `10560310`
- Title: `How do I create unicode characters with variable numbers?`

Basically what I want to do is print u'1̆001', but I do not want the 1001 hardcoded. Is there a way I can use a variable or string for this? Also, it would be nice if I could retrieve this code again, when I use the output as input.

All vs all - best result

Document 6547590

- Id: `6547590`
- Title: `EditText in Google Android`

I want to create the EditText which allow Numeric Values, Comma and Delete and other values are ignore.

So How I can achieve this by programming code ?

Thanks.

Document 27293230

- Id: `27293230`
- Title: `How to get index of UI?`

I want to get index of UI. I did like following, but I couldn't. Is there any way?

source:

result:

expected result:

All vs all - Best complete result

Document 32332430

- Id: `32332430`
- Title: `Fetch array after mysqli_multiple_query() function`

In the following statement I am trying to fetch the result of the subjects table in the widget_crop database. But I am receiving the down error message after using the mysqli_multi_query() function in order to be able to use multiple mysql queries inside my php code . How can i correct this and print the table correctly ?

Error page

Document 39351520

- Id: `39351520`
- Title: `some css not working`

So I have been following this tutorial for making a calendar widget with html, css, and js but I am stuck on the CSS not working. Did I miss something or miss-type? Where is my html/css going wrong to not end up the same final result? When I try to edit certain CSS it doesn't seem to affect the output.

The tutorial is here for reference to how it is suppose to look like.

Here is my HTML: