Tree similarity: Finding similar html pages based on structure

Pim Wijn (s1758888)

May 30, 2017

Contents

1 Introduction

In data-mining research and education there seems to be a lot of focus on the extraction and analysis of unstructured (textual) data. This is no surprise since lots of information can be extracted from such data. However there is also interesting information to be found in (semi) structured data such as xml and html documents. Not only the text contains information but the page structure gives us lots of information as well. Take for example a website such as kieskeurig.nl, when looking for products on this website on will be presented with a page containing results. Each of these result blocks doesn't only contain a description but it will also contain specific ratings, pricing information, dates of release, availability, etc. This data is especially suitable for structured extraction. For example with a query language for xml/html documents such as XPath. It is not too difficult to build a model for this extraction manually, but it is still quite time consuming. But now imagine that we want to do this on a large scale. You would not want to do this manually for every website.
A solution would be to find web pages with similar structure to apply the same model with minimal modifications. The goal of this paper is to describe a way to aid in (semi) automatically applying existing models to new structurally similar content.

Such a model would need to be fast to compute, be able to handle large amounts of data and easy and quick to query. In the next section I will describe an example of data that can be used as input. In section 3 I will describe the general steps and algorithms necessary to achieve such a results. In section 4 we will take a look at specific implementation challenges and finally we will evaluate the result and look at improvements that could still be made.

2 Dataset

To build a such model a large amount of html pages are needed. It is possible to gather this yourself using web-crawling frameworks. Luckily this is not necessary since large datasets are available containing more than enough web pages to build any model.

2.1 Common Crawl

An example of such a dataset is the one created by the *Common Crawl foundation*, the foundation's aim is to archive as much of the web as possible. Data is being gathered since 2008 and from the start of 2014 there have been monthly datasets. The dataset that was used to build the LSH model is part of the archive from march 2017. This set contains 66500 files which combined contain 250+ TiB (60+ TiB compressed) of web pages. Each of those files contains around 1GB of compressed data which corresponds to between 40.000 and 50.000 web pages. This dataset can be found on http://commoncrawl.org/2017/04/march-2017-crawl-archive-now-available/

Since the size of this dataset is too large to use for computation on this 'small' scale and such an amount is not necessary to show the properties of such a model in this paper on 20 of those compressed files where used for the model described , combined containing a bit over a million usable web pages.

2.2 WARC File Format

The data files contain web pages in the Web ARChive format. This format contains meta-data, response headers and the retrieved html. An example of such a file can be found in appendix A. The file format is created to provide a standard way to store data received by crawling web pages and is specifically made to support large amounts of objects in one file.

3 Pipeline

In this section we will take a look at the steps that need to be taken to build and query a model that efficiently finds similar items. This description will mostly look at the algorithms necessary and try to ignore as much implementation detail as possible.
First we will follow the transformations each web page has to go trough before it can be included in the the LSH model. Generally LSH requires either a set that can be used for MinHashing or an array/vector containing decimal values. For each page first the the html needs to be extracted from the WARC files, next the html needs to be parsed into an easy to use format. Then the data should be transformed to a format that can be used to calculate hashes to use in the LSH model.
In the final part of this section we will take a look at the way to query an LSH model.

3.1 WARC extraction

The first step in the data processing pipeline is the extraction of html strings from the WARC files. From each of the WARC files separate WARC objects can be extracted using a python library [1]. Since the python library can directly handle the compressed files it is not necessary to extract those files, reducing the needed storage. This library extracts the response and metadata in a way that is easy to access. From the resulting object relevant meta-data such as the url is extracted and the response headers are stripped from the response, resulting in just the html.

In this step html pages can be filtered, for example a page with html that is too short and cannot be used or HTTP redirect responses.

Here an example html structure is shown, this specific html will be used to illustrate the result of each step in the pipeline. The data resulting from the extraction would be similar to the example html string.

```
<html>
    <head>
        <meta></meta>
        <meta></meta>
        <meta></meta>
    </head>
    <body>
        <ul>
            <li></li>
            <li></li>
            <li></li>
        </ul>
    </body>
</html>
```

3.2 Html Parsing

Once the html is extracted the next step is to transform it into a format that is easy to manipulate for tree algorithms. It is also advantageous if the size of the representation becomes smaller, this lessens the transfer of data between worker nodes.
While parsing, irrelevant or non-structural html tags can be ignored and the tags can be replaced with integers. Further reducing the size of the tree. Integer values for tags can directly be used in further transformation steps. To keep the replacements and filtering synchronized between worker nodes it is necessary to keep a global replacement dictionary that is sent to each worker node. If necessary this dictionary can also be used to translate the integer values back to html tags.
Generally html parsing libraries return an elaborate tree object containing all information for each node including all kinds of helper functions. For the purpose of our algorithms we don't need all this information. Therefore it is necessary to traverse the parsed tree once more, extract relevant tags and store them in a recursive array structure. For each tag the structure contains a 2-tuple of a tag itself with an array containing each of its children. During the traversal of the tree tags are also replaced and filtered as described previously. For each of the filtered tags the children of are stored as direct children of the parent node.
Finally the result will be an array like this:

```
(html, [
    (head, [
        (meta, []),
        (meta, []),
        (meta, [])
    ]),
    (body, [
        (ul, [
            (li, []),
            (li, []),
            (li, [])
        ])
    ]),
])
```

3.3 Matrix representation

Now we can start with the most exciting part of the preprocessing of the data, all the steps taken up till now were to provide a suitable structure for the coming algorithm(s). We currently still have data with a tree structure, while LSH needs a set of n-grams or an array containing decimal values.
This transformation will be done using an algorithm described in a paper by Karau et al. [6]. This algorithm has two parts. The first part transforms an ordered tree into a binary branching tree. The tree is then used to generate a matrix containing the binary branches of the tree. These binary branches are similar to n-grams in text, however in this case the n-gram is described by the parent, the left child and the right child. For example: "$root.left.right$". These n-grams are also added to a set to use for minhashes.
This algorithm requires the traversal of the tree only once, since the resulting transformation for each node depends only on its children and direct siblings. Therefore the time complexity is $O(n)$. Since the html trees can contain 10.000's of nodes this is necessary to keep the execution time reasonable.

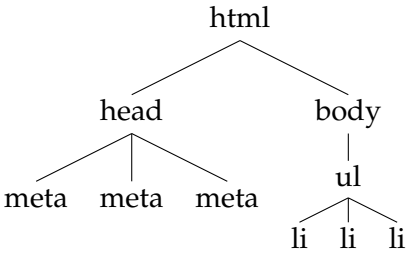
Binary Branching Tree

The algorithm used to generate a binary tree is based on the algorithm described in *The Art of Computer Programming* [4, p.340], in practice this algorithm is often used to transform algebraic expressions. Although the transformation seems somewhat complex, the description of the algorithm is quite simple:

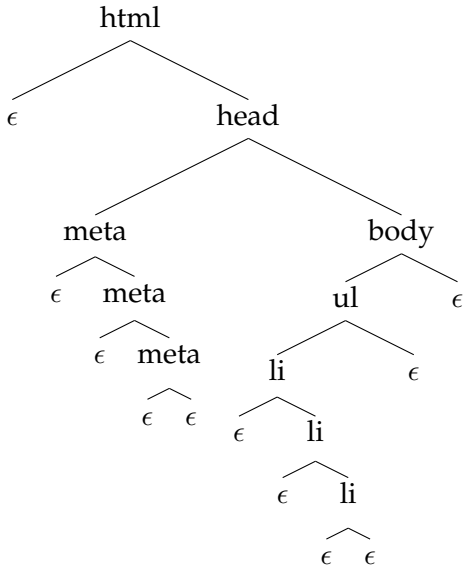
1. Link all siblings in the tree with edges. (Siblings become children as well)
2. For each node remove all edges to its children, except the edge connecting it to the first child.
3. When the left child (first child) is empty it becomes ϵ.
4. When the right child (first right sibling) is empty it becomes ϵ

The result of this algorithm executed on the previously extracted tree looks like this:

Original:



Binary:



Building the Matrix

The matrix that is going to represent the tree is a 3 dimensional matrix, this is $O(n^3)$ space, which is a large amount of memory required for each html page. However each html page contains only a small subset of all possible n-grams, making it possible to use a sparse matrix requiring much less space.
To represent the tree in this kind of matrix gives several advantages. First it makes it easy to compute a good approximation of the distance between two trees. This distance is a lower bound on the edit distance, with the edit distance being at most five times the difference between two matrices [6]. Second by having a matrix or array with decimal values it is possible to apply different distance measures, such as cosine similarity.

To generate this sparse matrix all binary branches or 3-grams have to be added to the matrix. In this case the first binary branch would be $html.\epsilon.head$. Normally we would need to traverse the tree again to get the binary branch of each node. However it is possible to retrieve this information during the previous algorithm building this binary tree and add the binary branches to the matrix during the transformation.

The resulting matrix will look like this, with zeros for all non-occurring binary branches:

⋮	⋮	⋮	⋮
html	ϵ	*head*	1
⋮	⋮	⋮	⋮
head	*meta*	*body*	1
⋮	⋮	⋮	⋮
meta	ϵ	*meta*	2
meta	ϵ	ϵ	1
⋮	⋮	⋮	⋮

⋮	⋮	⋮	⋮
body	*ul*	ϵ	1
⋮	⋮	⋮	⋮
ul	*li*	ϵ	1
⋮	⋮	⋮	⋮
li	ϵ	*li*	2
li	ϵ	ϵ	1
⋮	⋮	⋮	⋮

3.4 LSH

At this point all the preprocessing is done and it is time to build a model we can use. When working with big data such a model needs to have several characteristics. It needs to update quickly, execute queries quickly and preferably it should be distributable. Most models for finding similar items either require a long time to update with new items or queries depend on the size of the model. Which is why in this pipeline LSH or Locality-Sensitive Hashing is used. It is an algorithm that provides a probabilistic way to find similar items based of fixed size signatures that represent the item. The time required to update the model only depends on the sizes of the objects and the amount of hashes that will be generated for the signature, making the time complexity constant for a specific implementation. This means that completely building the model takes $O(n)$ time. Querying time complexity grows slowly with the amount of resulting candidates and the space required to store an LSH model depends only on the size of the hashes and the amount of items in the model. With the size of the hashes fixed this means that the space complexity is also $O(n)$. An added advantage of LSH is the option to distribute almost every part of the updating and querying over multiple workers.

For the implementation described here we focus only on MinHashing, though other distance measures are possible [5, ch. 3].

MinHash

Before finding candidates it is necessary to generate a signature for the object. This is done by means of MinHashing. To calculate a MinHash for a set we need to randomly permute the n-grams and then find the index of the first non-zero element in that permutation. For MinHashing a hash function is used to prevent the actual permutation of the array by approximating it.
To generate a signature of length k it is necessary to calculate k different MinHashes on the set. A very useful property of this way of hashing is that the probability of two MinHashes on two sets corresponds to the Jaccard-similarity of those sets. Defined by: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.

Finding Candidates

Once we have calculated the MinHashes it is possible to select candidates. Previously the signatures of size k were calculated. These are now divided into b bands of r rows. A candidate will be selected if two signatures agree in all rows in a single band. The chances of becoming a candidate is $1 - (1 - s^r)^b$.
Of course we want to maximize the amount of correct candidates and minimize the amount of false positives and false negatives. To do this we have to chose a cut-off point where we decide whether an item is a candidate or not. To calculate which values of b and r are needed to achieve

that cut-off point, the formula $(\frac{1}{b})^{\frac{1}{r}}$ may be used. The precision can be improved by increasing the size of the signatures.

A more detailed explanation of the LSH algorithm can be found in the *Mining of Massive Datasets* book [5].

3.5 Querying

Once the model has been created it becomes fairly easy to query the model with a new item. To do this we first need to compute the MinHash signature as described in the previous section. Once this signature has been computed we can compare the rows according to the rows and bands that were decided on previously. From this we get candidates.

Once a list of candidates has been retrieved we calculate the actual distance between the input and the candidates to see if each candidate is actually similar. For this Jaccard-similarity can be used but other distance measures such as cosine-distance are also suitable.
In this specific implementation we use cosine-distance since the matrix representation of trees gives us a good basis. Furthermore more tags in web pages doesn't mean that a page is not similar, if a page of a webshop shows a different amount of items it is still the same (or a similar) page. Even though the amount of tags on the page will be different. This fits well with cosine-distance.

4 Implementation

Now we have seen how each of the required steps for the model works in general, it is time to look at the specifics of this implementation of the pipeline. The focus will mostly lie on aspects that are relevant for the distributed computation of such a system. We will also take a look at a couple of libraries that shape this implementation. Finally we will take a look at the performance of the implementation and look at the ways it can still be improved.

4.1 Libraries

First we will briefly take a look at the main libraries used in the implementation. These have major impacts on the design and code, some knowledge of them will be helpful in evaluating the design and performance of the system. The three main libraries that are most important are, Spark for the execution and distribution of tasks, Lxml for the parsing and traversing of html-trees and DataSketch which provides an interface for the implementation of LSH.

Spark

Spark [3] is a cluster computing framework and the spiritual successor of Hadoop. It is mostly used for huge distributed jobs, for which it provides several helpful tools. Probably the most important of these are RDDs or resilient distributed datasets. These are non-persistent datasets that can easily be distributed over multiple worker nodes. RDDs also provide a basis to apply transformations on each separate data item and build a pipeline chaining multiple transformations. Once all computation is done the result can be collected from an RDD. For this Spark uses lazy evaluation to minimize computation.
Under the hood Spark handles all scheduling, serialization/de-serialization (sending objects between workers) and collection of data. It also provides a nice web interface to monitor your cluster and manage jobs.

Spark also offers Streaming, SQL solutions and machine learning algorithms which were not used in this implementation.

Lxml

Lxml [2] is a python library designed for the parsing and manipulation of xml and html documents. This library was used because it is fast (a Python wrapper around a C library), robust and it provides an object that is easy to work with. The library allows for easy traversal and attribute extraction which are extensively used to transform the html document from a text string into a simplified recursive tree structure as described in section 3.2.

DataSketch

DataSketch [7] is the python library used as basis for the LSH implementation. It is directly based on the algorithms in the *Mining of Massive Datasets* book [5] and provides a simple implementation for MinHashes and LSH. Because it is simple it doesn't support distribution or data too large for memory out of the box. We therefore need to distribute handle this in our own implementation. It does however provide options for tuning MinHashes, the cut-off point for candidates and the balance between false-positives and false-negatives.

4.2 Design Decisions, Challenges (and somewhat less elegant fixes)

Now we have a clear view of the global workings of the pipeline and the LSH model, and have seen some of the main tools. Which means that we can take a look at the architecture and design decisions for this application.

Data Input

To start the Spark pipeline an initial RDD is needed, for some standard formats Spark can directly initialize an RDD. This is not possible for a WARC file since it is a file buffer which is not completely loaded in memory. Therefore it is necessary to first load a batch op html documents from the WARC file and manually create an RDD from them using Spark.
In the internals of the WARC file loader the batches automatically continue in new files when needed. The advantage of this that you have much more control over the input, which can also influence later stages of the process.

Memory Optimizations

To optimize memory usage there are several steps that can be taken. The most obvious one is filtering all unnecessary data, this is done in the first step where all non-relevant html-documents are filtered. During the preprocessing irrelevant tags are filtered as soon as possible and the tree is directly transformed to a more memory-efficient format.
This leaves us with only a few places where large gains can be made. But in the model there are still two types of objects that grow linearly with the size of the input, namely minhashes and matrices. Matrices can be compressed by using sparse matrices, which can reduce memory usage by several times. The minhashes used in its simplest form are mutable objects. For python this means quite some memory overhead. To solve this they can be transformed into immutable objects allowing python to store the objects more efficiently. Another way to store python objects more efficiently is by specifying the serialization protocol. Which reduces the memory usage to

1/3.
The advantages of minimizing memory usage are threefold. First, the amount that can be put in memory on the worker nodes grows when less memory is required per item. Second, serialization/de-serialization and the transportation of data between worker nodes takes less time. Finally, the storage required to store the items is less, so bigger models can be stored.

Building the Model

The main LSH model is built on the master node, the actual computation doesn't take much time since the only action it has to take is storing the data in the model. The main concern is the usage of memory when building the model. This is where the batch input shows its advantages. Because the data is processed in batches, each batch can be stored in the model and the rest of the memory can be cleared. To further improve this further the matrices that are still used for querying are stored in a database as soon as possible. The relatively smaller minhashes are added to the LSH model which stays in memory until a suitable size is reached.
When the LSH model is large enough the object is serialized and stored in a file. This file can easily be de-serialized in python when needed again.

Querying

To query the stored model a web page is retrieved from the Internet given an URL. The resulting html goes through a non-distributed version of the pipeline on the master node. The master nodes then loads the previously built model parts. Each of these parts is de-serialized and then queried with the signature of the query document. The resulting candidates are combined and for each of these candidates the corresponding matrix representation is retrieved from the database. For each of these candidates the cosine-distance with the query page is calculated. Finally the candidates are sorted based on the distance and the top 10 is returned.
Due to an old version of numpy (used by DataSketch) the de-serialized minhash internal representation was different than the representation directly after it was calculated. To remedy this it was necessary to serialize and de-serialize the query minhash before actually querying the model.

Interface

The implementation provides a simple command-line interface the user to specify a model name and either an URL or model size depending whether the model is going to be created or queried. The interface looks like this:

```
Create:
~/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --master spark://<master_url>
main.py --train <size> --modelname <modelname>

Query:
~/spark-2.1.1-bin-hadoop2.7/bin/spark-submit --master spark://<master_url>
main.py --query '<url>' --modelname <modelname>
```

Spark

One of the more challenging parts of the implementation was getting the code, which was written locally, to work on the DAS4 Spark cluster. The worker nodes were outfitted with a limited

amount of libraries with old versions. (At the moment I wasn't aware of how to install new software on each worker node). This meant that I changed the code to work with old versions of libraries and I had to make some changes inside libraries to make them compatible. This also lead to the previously described minhash incompatibility.
Spark also requires the inclusion of extra libraries and classes in a special way to distribute them to the worker nodes. This also changes the way these files are imported and makes it difficult to run code separately without starting spark.

4.3 Performance

The final model was run on DAS4 with 16 worker nodes. The performance of the model created from 1000000 wep pages takes 2.3 hours (8158 seconds) to compute. The storage sizes for the model are 1.3 GB for the LSH model itself and 3.9GB for the database of matrices. This information was based on a single run since Spark crashed afterwards. However based on the small scale test the performance is quite constant.
On smaller scale tests of 50000 urls the memory of the model was 166MB/550MB and for 1000 urls it was 3.3MB/11MB for the LSH model/matrices. This shows that the memory grows linearly with the size of the input. The running time also grows linearly with the size of the data, while computing the model. However it is easy to see the startup overhead of Spark before the actual execution of the pipeline.
This overhead is especially notable when querying the model. Quit a large part of the measured time is Spark startup. The query time on the full sized model is 2 minutes, whereas the query time on 50000 items takes 79 seconds.
The expected running time on my laptop based on the local runs and the behavior of the pipeline is 6-8 hours. This means that increase in computation speed is around three times. Way lower than you would expect when having access to DAS4 with the amount of worker nodes that were available.
The most likely explanation for this is the design of Spark, it assumes that the tasks are quite small to be able to minimize the sending and receiving of data between worker nodes. Possibly this also limits the ability of Spark to schedule tasks.

4.4 Improvements and Future options

Since this implementation is just a proof of concept and explores the possibilities of this approach. This means that there enough aspects that could be improved and in this section we discuss some of these improvements.

Distribution

These less than satisfactory performance of Spark suggest that it might be necessary to take an approach that manually minimizes data transfer between nodes. This could be done by execute the whole pipeline for one item on only one worker node, immediately distribution storage and the models as well. This also makes in easier to use Spark or any other distribution framework to query the model in a distributed manner. Spark can then directly sort the candidates and only request the top 10 on the master node.
It is also possible to go a step further and completely distribute the LSH algorithm. First distribute each item, second distribute each minhash calculation, then combine them into the signature. Finally to query, distribute the similarity checks for each bucket and then combine the hits to generate the candidate pairs.
This approach could be implemented manually, but in the next version of Spark (2.2) this will be integrated and can be easily and directly used. This will use a more efficient way to store the

minhashes (not like serialized objects), which should again greatly decrease the memory needed for the LSH model.

Algorithms

Currently there is one large self implemented algorithm in this implementation. The algorithm to transform a tree into a matrix representation, this is implemented as a recursive algorithm, which is conceptually better. However this recursive approach can lead to problems, since most languages limit stack size. This becomes problematic even faster because the binary trees easily become very deep.
The solution would be to implement and iterative version of the algorithm in which you keep track of your own stack structure, which will be allocated on the head, which does not have those limitations.

Data

In this sort of experimental setup the data is stored in several files which are directly downloaded from the Common Crawl server. In a real world situations you would probably retrieve your data from an efficient large data archive, which can also quickly retrieve single web pages. This would make it easy to just distribute URL lists and have the worker nodes take care of retrieval themselves.
In combination with the improved distribution, the need for batch processing is removed and it will also improve task scheduling and decrease the chances of one node allocating too much memory.
If storage is an issue only the minhashes could be saved and the matrices which require more space could be calculated on the fly requiring only a small amount of extra work.
In a real world application the data will also be filtered to include only relevant html pages, which for example already have corresponding models.

5 Conclusion

As we can see even a not very complex model requires several steps of preprocessing and computation and several interesting algorithms such as the binary tree transformation, matrix representation and LSH as we see in section 3. This is the case without even looking at the distribution of parts of these algorithms. Once we look at distribution things get even more interesting as we have seen in section 4. Not only requires each part of the computation careful examination to see where memory use can be minimized and what parts can be distributed over multiple worker nodes. As we have seen in section 4.4 this brings several challenges also depending on the context in which the application is developed. We have looked at the result and performance of the implementation. After that we looked at solutions to still existing flaws and future possibilities.
With this report I hope to have given the reader good understanding of all parts of the project and a good basis to further experiment or try similar or related research.

A WARC File Extract

This WARC extract shows the structure of a WARC file. The first section contains the metadata from the webcrawler itself, such as spiderdate, target and hashes. The second part of the extract shows response headers that resulted from the http request. Finally the file contains the resulting html that can be processed in further steps of your pipeline. This extract is taken from `http:`

//commoncrawl.org/the-data/get-started/ a full example of a WARC file can also be found on this url.

```
WARC/1.0
WARC-Type: response
WARC-Date: 2014-08-02T09:52:13Z
WARC-Record-ID:
Content-Length: 43428
Content-Type: application/http; msgtype=response
WARC-Warcinfo-ID:
WARC-Concurrent-To:
WARC-IP-Address: 212.58.244.61
WARC-Target-URI: http://news.bbc.co.uk/2/hi/africa/3414345.stm
WARC-Payload-Digest: sha1:M63W6MNGFDWXDSLTHF7GWUPCJUH4JK3J
WARC-Block-Digest: sha1:YHKQUSBOS4CLYFEKQDVGJ457OAPD6IJO
WARC-Truncated: length

HTTP/1.1 200 OK
Server: Apache
Vary: X-CDN
Cache-Control: max-age=0
Content-Type: text/html
Date: Sat, 02 Aug 2014 09:52:13 GMT
Expires: Sat, 02 Aug 2014 09:52:13 GMT
Connection: close
Set-Cookie: BBC-UID=...; expires=Sun, 02-Aug-15 09:52:13 GMT;
 path=/; domain=bbc.co.uk;

<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN"
 "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>
BBC NEWS | Africa | Namibia braces for Nujoma exit
</title>
...
```

References

[1] Internet Archive. Warc documentation, 2012.

[2] Stefan Behnel. Lxml documentation, 2017.

[3] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'reilly, 2015.

[4] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Pub Co, 1997.

[5] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2010.

[6] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similariy evaluation on tree-structured data. 2005.

[7] Eric Zhu. Datasketch documentation, 2017.