Imperial College
London

Refinitiv Project

Technical Report

Intelligent Web Crawling

ICDSS Advanced Data Science Team

Jakob Torben

Ricardo Mokhtari

Yousef Nami

Imperial College London

February 2021

Abstract

Web scraping for the same information from different company websites (such as board member names of the FTSE100) is a highly labour intensive process, as they have completely different structures and subdirectories, each requiring custom parsers. The scale of this problem is amplified when the scrapers are to automatically adapt to websites whose structures are bound to change over time. Despite developments in web scraping practices and tools, the problem of automatic web scraping is largely unsolved, with limited literature. Much of this is due to the lack of robust methods for calculating the difference between two parsers as a Cost function, rendering data-driven solutions useless.

This paper presents a summary of research conducted to determine a Cost function by representing parsers in two ways: a) Abstract Syntax Tree and b) post-fix. A total of 18 parsers are analyzed. These cover 6 FTSE100 company websites, each with 1 current and 2 historical webpages. Future work may conduct more thorough sensitivity testing of the two methods of computing the difference between parsers that we present. Additionally, we encourage studies exploring how these metrics can be incorporated into a predictive model that could suggest edits to parsers, thus bringing automated web crawling closer to realisation.

Contents

1 Introduction

Most companies - especially those listed on the Financial Times Stock Exchange (FTSE) - present information about their Executive Board on their websites, sometimes even on separate Corporate websites. Though to the average person this is trivial, companies such as Refinitiv make use of this publicly available data by collecting it through web scraping.

Web scraping is, in simple terms, the extraction of data from websites. In principle this involves making HTTP(s) requests to URLs, getting a HTML response and parsing the Document-Object-Model (DOM) to find the relevant text.

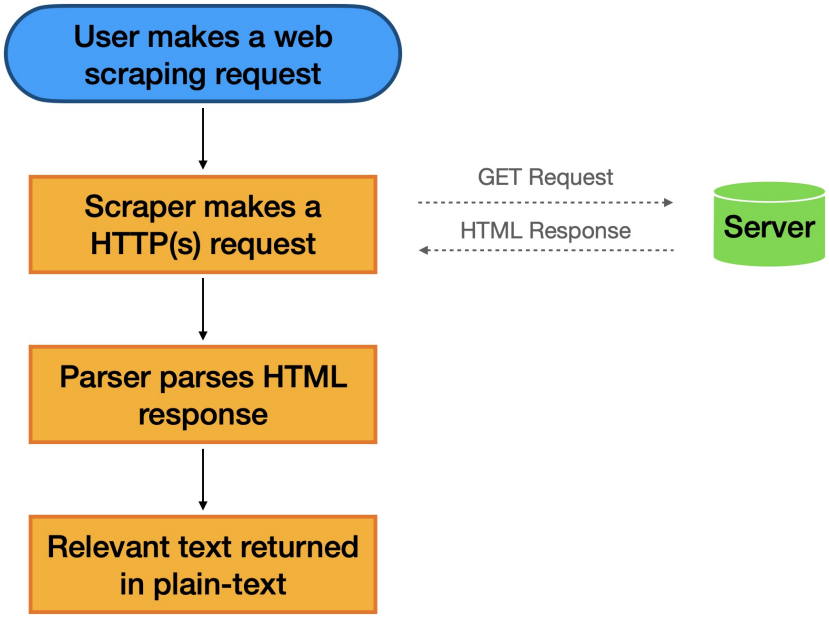


Figure 1: A high-level flowchart showing how a web scraper works.

Multiple web scraping tools exist that allow non-programmers to easily scrape the web, requiring only a minimal knowledge of the DOM to correctly parse the HTML responses. However, though this works well for single instances, the process becomes a lot more laborious when users target many websites and attempts to automate the web scraping process.

This is precisely the problem when creating an automatic scraper to extract Executive Board member names and occupations from the FTSE 100 companies. There are two significant challenges:

1. **Temporality:** websites change over time. This includes changes in the DOM structure and changes in domain names.

2. **Scale:** there are 100 companies, all of which have completely different DOM structures, as well as different subdomains where the relevant information can be found (some of which include duplicate Board members)

Refinitiv's solution to this problem currently relies on having hundreds of manually coded parsers. A hashing function checks websites to see if there have been any changes, and if so, an alert is made.

The next step requires manual labour to examine the changes on the websites to see if a new updated parser has to be written.

An ideal solution would be having a Reinforcement Learning driven parser that detects changes in websites and makes recommendations for changes in the code (or writes them on its own).

One requirement to get to that stage is determining a cost function that compares different parsers for similarities and differences. This is a largely unsolved problem in this context. Therefore, this project explores different cost functions for capturing the differences between parsers. Our code is available at `https://github.com/jakobtorben/Intelligent-public-web-data-extraction`.

2 Methodology and Results

2.1 Parsers

A crucial first step of this project was to write parsers for capturing both current and historical information relating to a company's leadership. To write these parsers we used Scrapy[1], which is a Python library for web crawling. Scrapy parsers can be written in a number of ways. However, since we are interested in comparing parsers, it is important to minimise the effect of different styles of coding between team members. For this reason, all parsers were written following the same general style:

1. We are interested in extracting leadership information. This is usually the name and title of all board members, which are stored in many repeating HTML elements on the company's page. The first step invovles extracting these repeating elements using Scrapy's `response.css()` functionality.

2. We then iterate through each returned HTML element and parse out the relevant information by retrieving the text corresponding to the board member's name and title.

3. The parsed information is exported as a .csv file.

4. It was decided that Natural Language Processing techniques would not be used to clean the response, as that would add unwanted noise to the parsers

In order to capture historical information, we made use of the "Wayback Machine", which regularly takes snapshots of websites and allows emulation of previous versions of a website throughout its history. For parsing websites viewed through the Wayback Machine, we applied the same general style as above. Some extra parsing was required in order to obtain the relevant year. In this way we are able to parse both the current information by parsing the current website and historical information by parsing archived versions of the same site. An example of a complete parser is shown in Appendix C.

[1]Other web scraping libraries were consdiered too, such as BeautifulSoup and Selenium. It was determined that Scrapy would be most appropriate given how well it scales.

Although this method achieves our goal of capturing both current and historical information, there are several limitations we identified. Firstly, we refer to the above methods as manual parsing because writing the parser involves manually inspecting the web page's HTML in order to determine the class names and structure of the elements. Due to this, we estimate that writing a new parser can take between 15 minutes and an hour, depending on the complexity of the web page. Shell's leadership page, for example, has a very complex structure with many repeating class names, requiring very careful parsing. The team also found webpages that were not able to be parsed with the proposed method, such as Coca-Cola's. Table 1 summarises the parsers that were created, and provides links for sample websites where they work (note that this is not an exhaustive list).

Version	**AstraZeneca**	**HSBC**	**GSK**	**Shell**	**Tesco**	**Unilever**
Current	Present	Present	Present	Present	Present	Present
Historical 1	Before 2016	Before 2018	Before 2018	Shell 2018	Before 2012	Before 2020
Historical 2	Before 2002	Before 2014	Before 2016	Shell 2013	Before 2011	Before 2015

Table 1: A summary of the different parsers made for each company with links to relevant webpage

Due to the manual effort required, this method scales very poorly and demonstrates the need for automated and intelligent web crawling. More generally, any change in the domain name (or HTML structure) of the current web page will require manual intervention. Additionally, since our method of parsing historical data depends entirely on the Wayback Machine, a single change in the domain name would threaten to stop all the parsers from working, requiring manual intervention. However, since these parsers are not being used in production, and are instead being used to develop a similarity metric between parsers, these limitations have little impact on the focus of this project, and suit our needs.

Bearing in mind that we now have many distinct parsers, the next logical step is developing a method to compute the similarity between parsers. This is what the next section will address.

2.2 Computing the Similarity Between Parsers

In order to compute the difference / similarity between parsers, they must first be represented in a way that allows them to be compared. The first is representing the code as Abstract Syntax Trees (AST) (used to represent code before it is compiled) and the second, inspired by Natural Language Processing, is representing code elements as tokens stored in the post-fix order as a vector. The next subsections will explain these concepts in more detail.

2.2.1 Similarity using Abstract Syntax Tree representation of code

What is an Abstract Syntax Tree?

In simple terms, an Abstract Syntax Tree represents code and logic elements in a tree-like format. In a very high level form, this could be represented as a Lexical Analysis stage followed by a Syntactic Analysis stage [1].

The first stage converts the code's elements into tokens (i.e. components), where the code is read as a string and converted into 'Lexical' elements.

For example, the command `print('Hello World!')` could be read 'Lexically' as follows:

- `print`: an identifier that identifies the name of the method
- `(`: a punctuation element signalling the start of the method
- `'Hello World!'`: a string (or argument) of the method
- `)`: a punctuation element signalling the end of the method

It's worth noting that at this level, the logic of the code is not captured. So though the elements are clear, it is unclear what they represent to the computer (though it is clear to humans who understand code). This stage is similar to how a language's lexical elements might be identified, for example, `I walked my dog.` would be read as:

- `I`: a pronoun referring to first person singular
- `walked`: a verb in past tense
- `my`: a possessive pronoun for first person singular
- `dog`: a noun
- `.`: punctuation

The next step is to perform Syntactic analysis, where the code is converted to logic and is given structure.

With the code example above, the structure would represent something like this:

1. `ExpressionStatement`, identifier `print`
    - arguments: [type: string, value: 'Hello World!']

The lexical features (i.e. the punctuation for example) are used to build the logic of the tree.

This is similar to how the syntax in a language is determined. For example, with the dog walking example above `I` would be the subject, `walked` the verb, `dog` the direct object represented by identifier `my`, and finally `.` declares that the sentence ends.

In reality, these are far more complex that represented here, but the Python libraries `ctree` and `astvisualizer` greatly help visualise them. Sample images of this applied to parsers are shown under Appendix A.

Finding similarities between ASTs using Pycode Similar

With the ASTs defined for the parsers, a literature review was performed to identify potential methods to calculate the similarities between the parsers. In literature, there are some works that considers comparison of code, in particular for plagiarism detection. Abstract syntax tree kernel is

a common method used for measuring the similarity between a pair of programs [2]. By using a AST rather than simply comparing different lines of code, tricks like swapping the order of lines, whitespace padding and renaming variables are more easily detected. This feature fits well for our purpose, as we want to capture the intent of the parsers, not the programming style.

In a recent paper, this method was extended by adding weights to the nodes according to the tree structural similarity [3]. The implementation firs converts the source code of a program to an abstract syntax tree and then gets the similarity by calculating the tree kernel of two abstract syntax trees. This method was found to perform significantly better than other popular plagiarism checkers, such as Sim and JPlag.

In our implementation, we adopted the Python library Pycode Similar, which uses ASTs to represent the source code [4]. It includes two methods to find the difference, a simple line difference algorithm and the more sophisticated TreeDiff method. The latter uses tree edit distance, where the distance between two ordered trees is considered to be the weighted number of edit operations (insert, delete, and modify) to transform one tree to another [5]. The original implementation was optimised for comparing several programs to a reference program and for that reason had a zero insertion cost. For our use, we need to have a symmetric cost function, which lead us to modify the implementation to use a similar cost for the insert and delete. A plot of the similarity before making this modification can be found in the Appendix in Figure 7.

After this modification, the similarity between the present parsers were found and plotted in a heatmap in Figure 2a. The cost function shows a roughly symmetric result, with some differences. It identifies high similarities between parsers such as HSBC, Unilever, GSK and Shell. While AstraZeneca and Tesco differs. The parsers for HSBC and Unilever, has almost the exact same code, except a difference in the html parser argument, leading to a 96 % similarity.

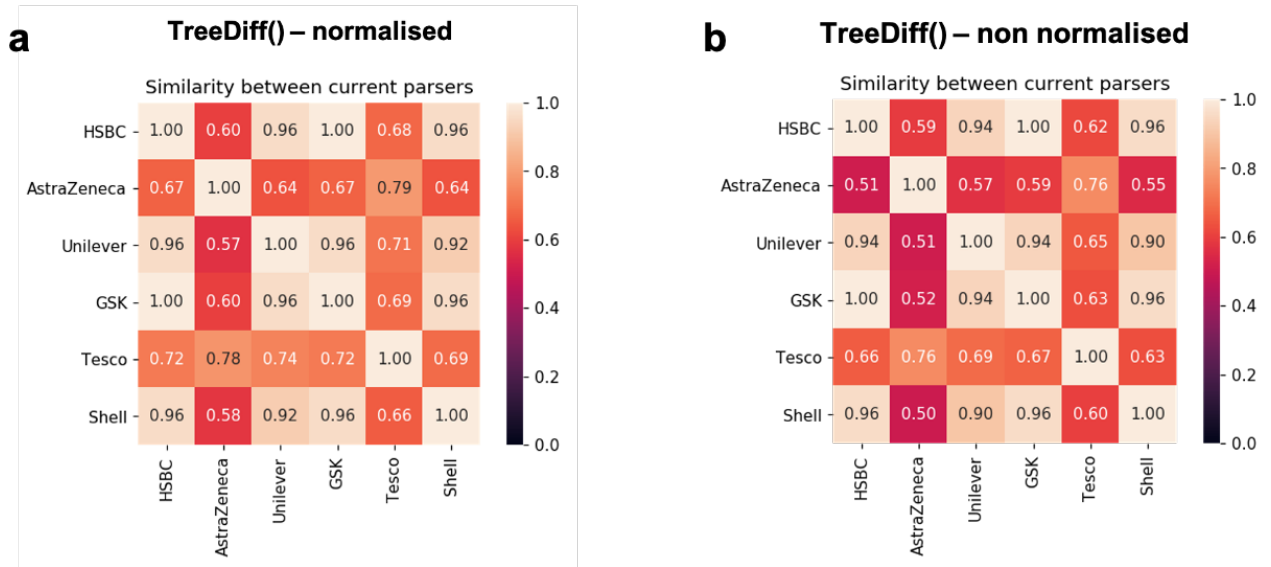


Figure 2: Heatmaps of the similarity of the different parsers for the present website. **a)** Similarity calculated by using the tree edit distance, in an implementation adapted from Pycode Similar. **b)** Similarity calculated using tree edit distance, without normalisation.

In the original implementation, the nodes in the AST is normalised by removing all argument entries and attributes. For this project, we are interested in finding the right parser argument and attribute method to extract the desired information from the html document. For that reason, this normalisation was removed in our implementation. Figure 2b shows an updated plot of the similarity, without the normalisation. The two plots show a similar result, however, the non-normalised version captures different arguments as well. For instance, now HSBC is reduced from 96 % to 94 %, which is a small but important difference. If weighted ASTs were to be implemented, this would have a larger impact on the similarity.

Finally, the similarity between the present and historic parsers were considered. Figure 3 shows heatmaps of each company. It illustrates how the structure of the website affects the different parsers. For instance, HSBC and Tesco had limited changes to their website in resent years, such that similar parsers could be used. While websites such as Unilever and Shell had significant changes to their structure, making the parsing process more complicated.

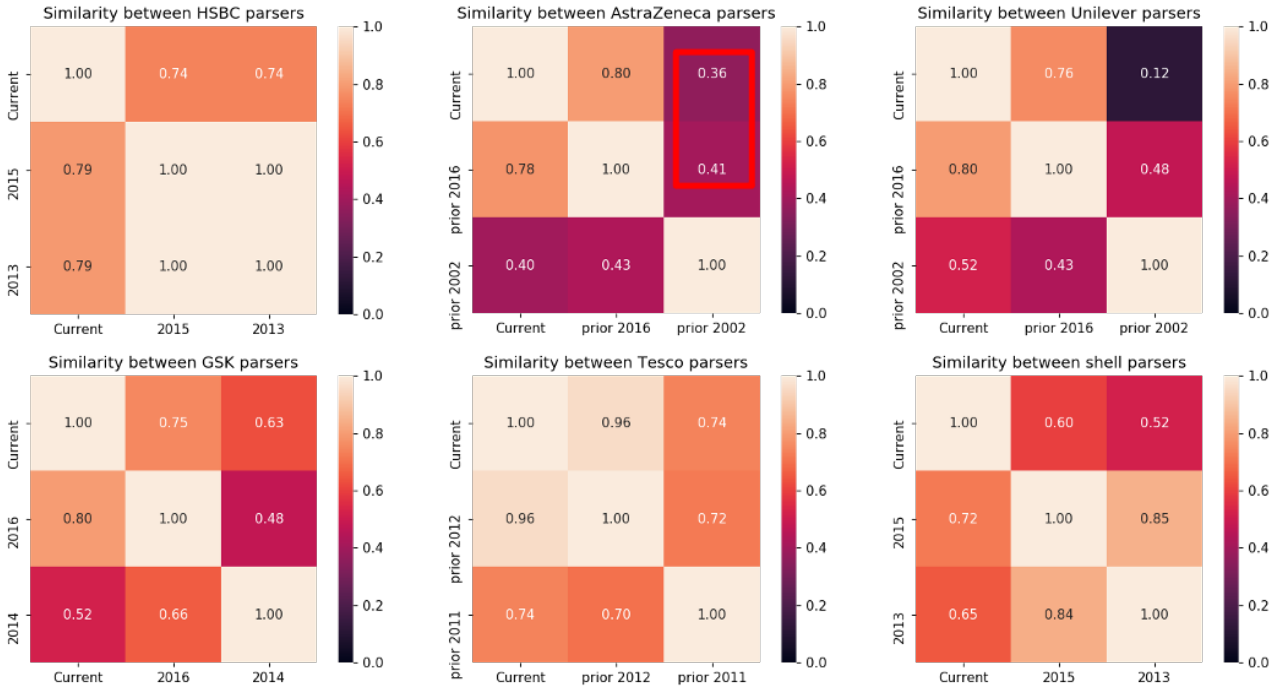


Figure 3: Heatmap of the similarity between the parsers of the archived versions of the website. The parsers in the red rectangle for AstraZeneca had a negative tree edit distance, due to the large difference in parsers.

Overall, it was found that the similarities were too close to one another for this method to be robust in trying to accurately describe different parsers. This might have been improved by adding weights to the nodes, based on the tree structure symmetry between the two parsers being considered and all the parsers in the database. Another option is to consider other methods for representing the ASTs. To find the optimal method, the team considered a different approach to representing and comparing the ASTs.

2.2.2 Vector representation of AST

Directly computing the difference between two ASTs is a highly non-trivial problem. Therefore, it may be beneficial to work with a representation of the AST that allows for a more straight-forward difference calculation. Computing the difference between vectors, for example by calculating the cosine similarity, is very straightforward and used ubiquitously across many domains. If it is possible to faithfully represent ASTs as vectors, then computing the difference between the ASTs would be much more straightforward.

There are 3 standard notations for writing expressions: prefix, infix and postfix. These 3 notations refer to the order of operators and the operands on which they operate. For example, the infix notation is the standard way that mathematical expressions are written: "(2+5) / 3". In postfix notation the operators come after the operands, while in prefix notation the operators come before the operands. Therfore, the postfix representation of "(2+5) / 3" is "2 5 + 3 /" and the prefix representation is "/ + 2 5 3". More generally, these 3 notations represent 3 different ways of representing the same expression tree (Figure 4).

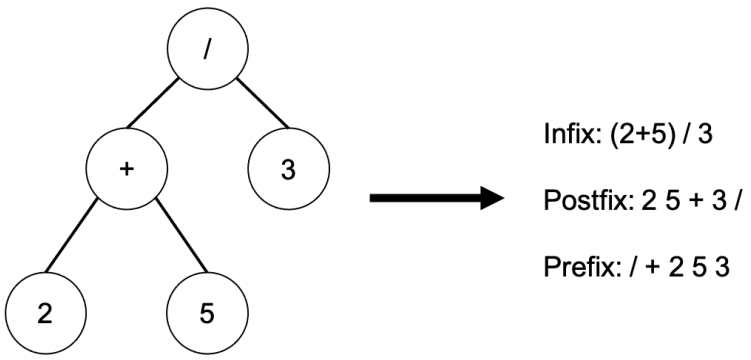


Figure 4: Example expression tree and conversion to infix, postfix and prefix notations.

These different vectorised notations can be generated through different ways of traversing the expression tree. For example, the postfix representation is generated via a postorder traversal of the tree, while the infix representation is generated through an inorder traversal of the same tree. Therefore, if we replace the expression tree with an AST, and apply a an appropriate traversal, we should be able to generate a vector representation of the tree.

This was implemented in Python by using the `ast` module. First the AST is generated using `ast.parse()` followed by a postorder traversal to generate the postfix representation of the tree. An example applied to a short code snippet is shown in Figure 5.

This method may appear convoluted when applied to such a small code fragment as in Figure 5. However, when applied to larger pieces of code, such as an entire parser, the conversion of the code first to an AST and then traversing this tree is absolutely critical for faithfully capturing the correct operator-operand relationship in the code.

Given the time frame of the project, it was difficult to explore methods of comparing the vectorised ASTs. Some suggested methods are described under future work.

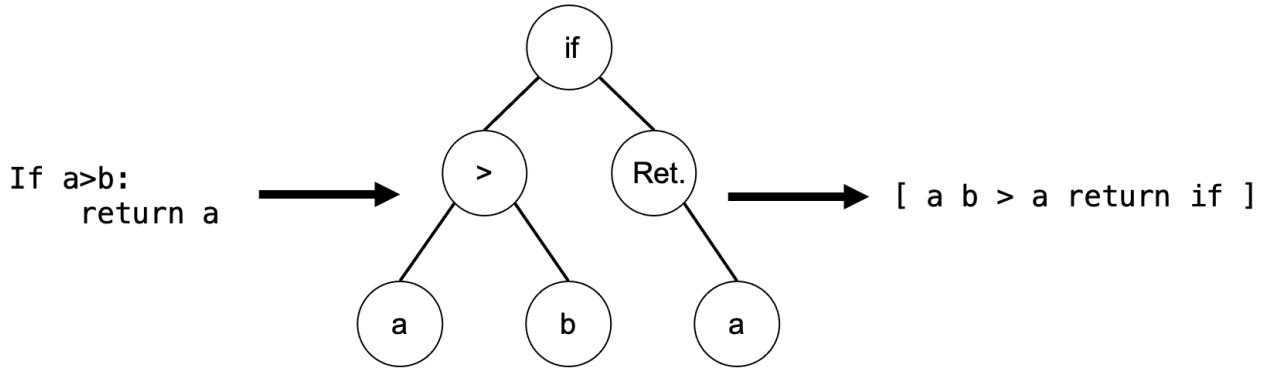


Figure 5: Conversion of code to postfix vector. Code is first parsed into an AST. Postorder traversal of the AST generates the postfix vector form.

3 Future Work

So far in this project, we have been experimenting with different methods to represent source code of the parsers, in addition to methods for finding the similarity between them. This is a non-trivial problem, with no one right solution. Using the ASTs to store the parsers and tree edit distance to find the similarity between them, we successfully managed to identify parsers that required little change to work on historical websites. However, this method was found to lack sensitivity, and yielded too small differences between some of the parsers. In future work, this could have been resolved by implementing a more sophisticated method to store the ASTs, such as the weighted AST kernel, as discussed in [3]. This would involve adding weights to the nodes according to the tree structural similarity between the two parsers being compared and common structures of all parsers in the dataset.

Another possible approach, would be to store the ASTs in a vectors instead, using a postfix representation. This would make it easier to find the similarity between two parsers. There are several possible methods to find the similarity between two tokenised vectors. Jaccard index is a conventional statistic, given by the intersection of the two sets, divided by the size of the union. However, this method is not well suited for cases where we care about the order of the vector. A more suited approach for this case is the Levenshtein distance [6]. This method considers the number of edits required to change one vector into another. With the ASTs represented as vectors, this method could be tested for finding the similarity between parsers in future work.

Extensive further testing would need to be carried out on the methods we have presented. However, given an appropriate method for finding the difference between parsers, the next step would be to suggest changes to the parsers whenever a website changes. This is an open-ended problem with many possible solutions. Related work in intelligent web crawling utilises reinforcement learning to automatically retrieve relevant pages [7]. In this context, it may be possible to use the difference metrics we have investigated as a reward function for an agent continuously crawling the web - we encourage future studies exploring this. We also suggest investigating more advanced methods of utilising similarity metrics in machine learning, such as triplet loss.

References

[1] D. Kundel, *Asts - what are they and how to use them*. [Online]. Available: https://www.twilio.com/blog/abstract-syntax-trees.

[2] M. Collins and N. Duffy, *Convolution kernels for natural language*, 2001.

[3] F. Deqiang, X. Yanyan, Y. Haoran, and Y. Boyang, *Wastk: A weighted abstract syntax tree kernel method for source code plagiarism detection*, 2017. DOI: https://doi.org/10.1155/2017/7809047.

[4] Fyrestone, *Pycode similar*. [Online]. Available: https://github.com/fyrestone/pycode_similar.

[5] Z. Kaizhong and S. Sasha, *Simple fast algorithms for the editing distance between trees and related problems*, 1989.

[6] V. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1965.

[7] M. Han, P. Wuillemin, and P. Senellart, *Focused crawling through reinforcement learning*, 2018.

A Figures

a

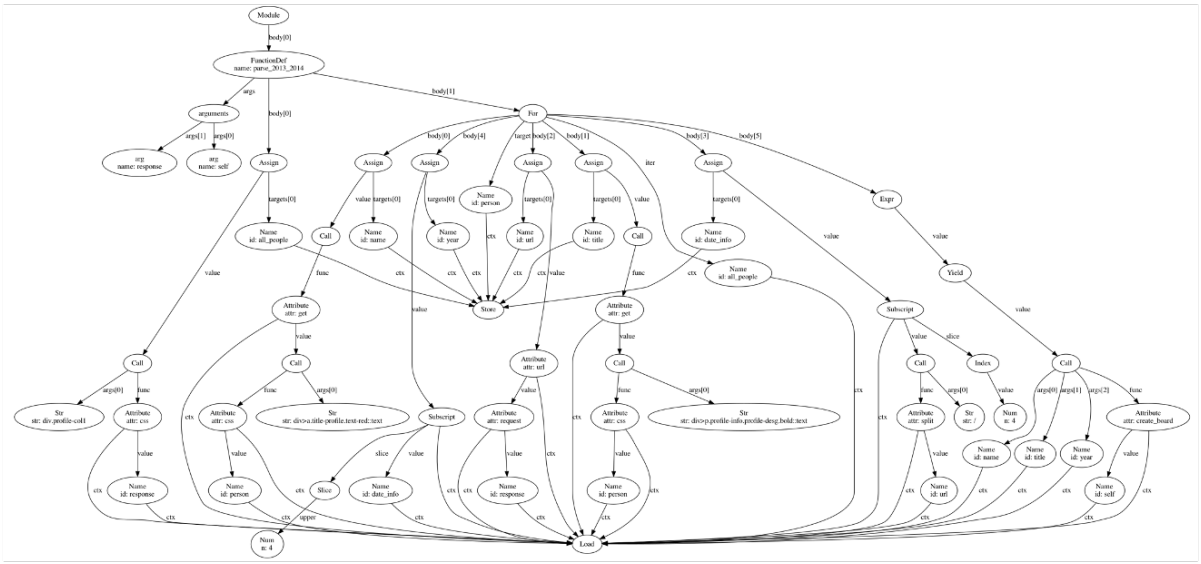
```python
# parse the board from 2013-2014
def parse_2013_2014(self, response):
    # define selector that contains all items
    all_people = response.css("div.profile-col1")

    # iterate through items
    for person in all_people:
        # manually parse name
        name = person.css("div>a.title-profile.text-red::text").get()
        # manually parse title
        title = person.css("div>p.profile-info.profile-desg.bold::text").get()

        # find the year from the crawled URL
        url = response.request.url
        date_info = url.split("/")[4]
        year = date_info[:4]

        # Return item
        yield self.create_board(name,title,year)
```

b



c

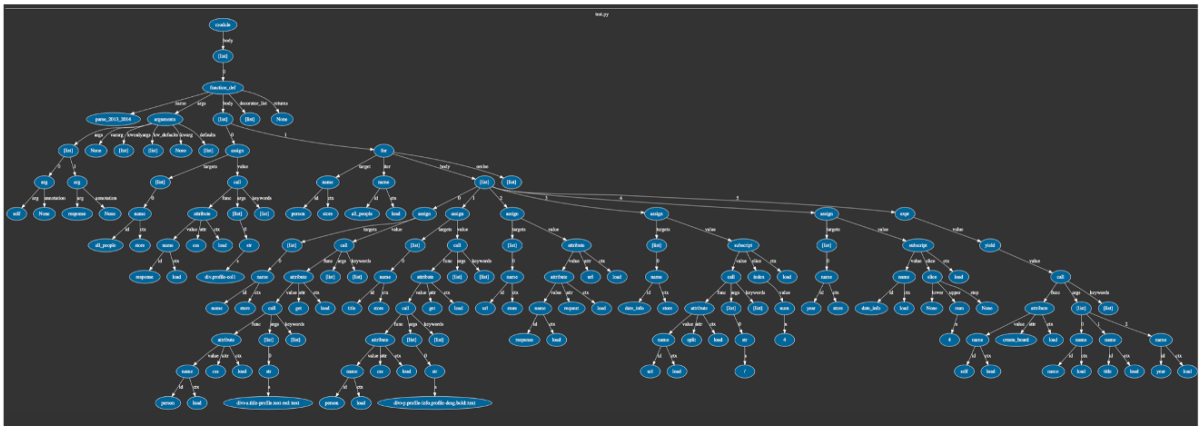


Figure 6: **a)** Example parser. **b)** A representation of the parser's AST using `ctree`. **c)** A representation of the parser's AST using `astvisualizer`.

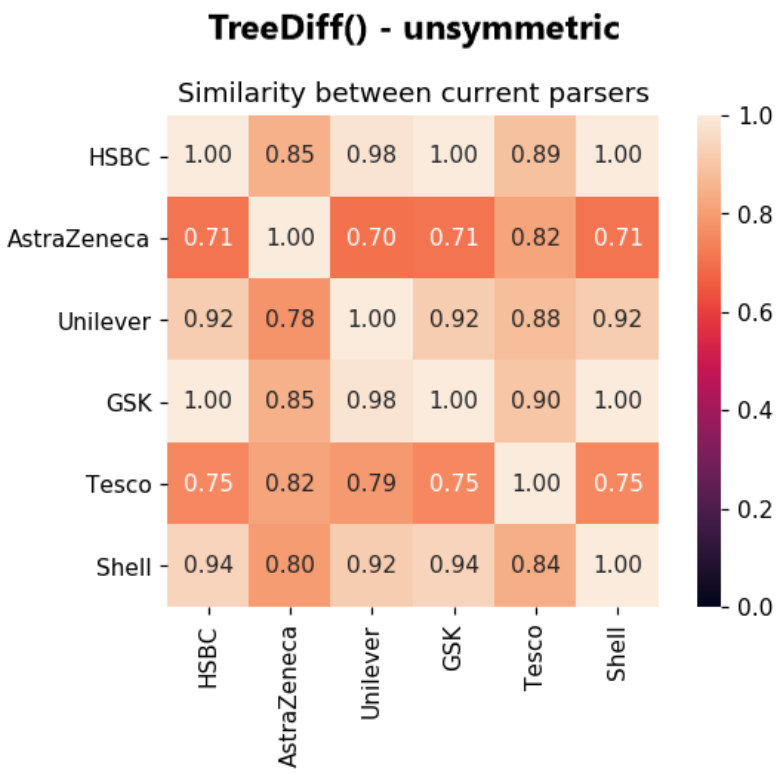


Figure 7: Heatmap of the similarity of the different parsers for the present website. The similarity was calculated by using the tree edit distance, but with a zero cost of insertions between the two trees being compared, leading to an unsymmetrical cost function.

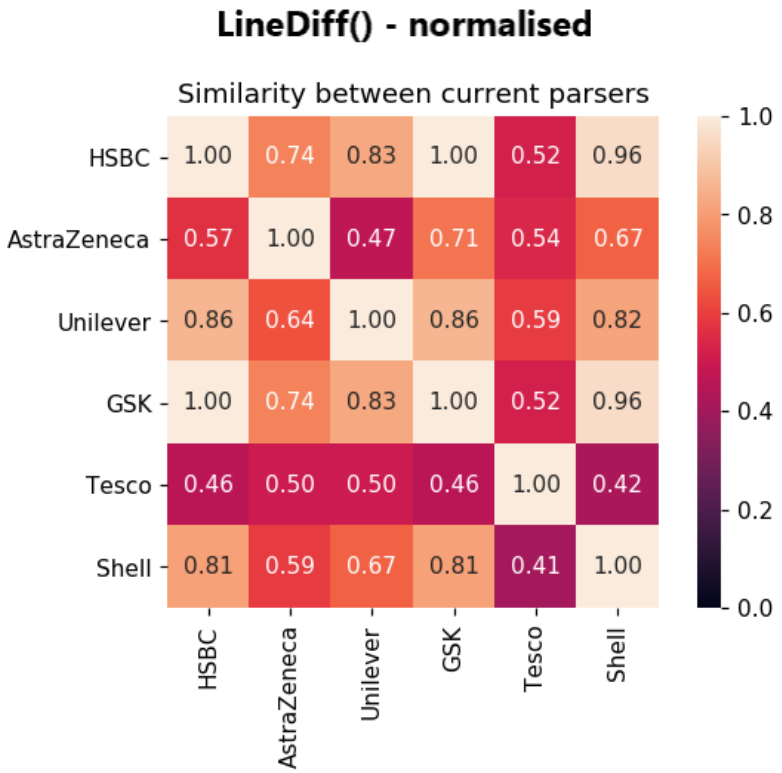


Figure 8: Heatmap of the similarity of the different parsers for the present website. The similarity was calculated by using a linediff algorithm, in an implementation adapted from pycode similar.

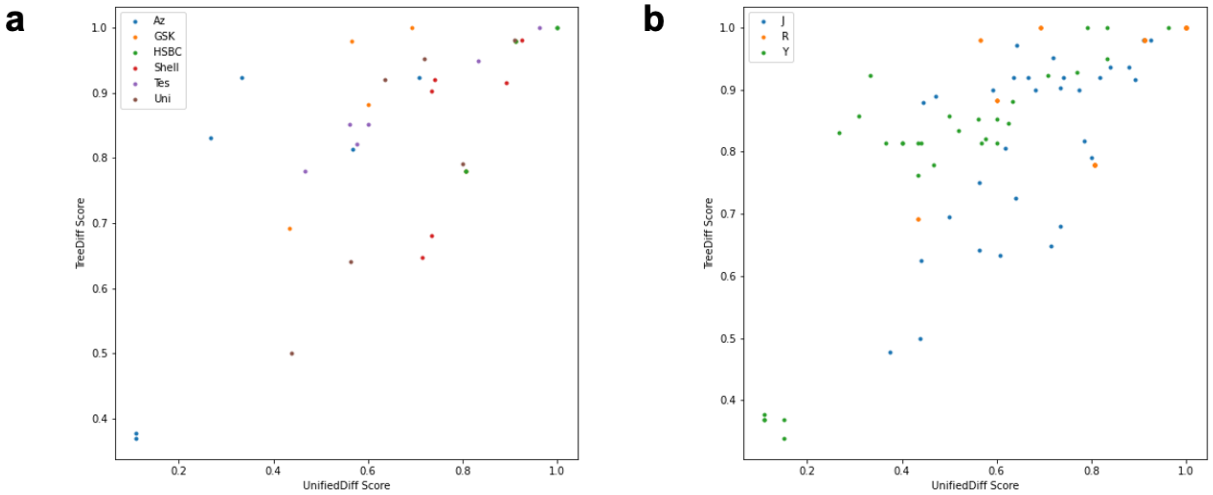


Figure 9: **a)** A scatterplot of the different parsers grouped by company plotted based on their similarity scores. **b)** A scatterplot of the different parsers created by the same authors plotted based on their similarity scores

B Code

```
# imports
import scrapy
from ..items import Board
import datetime

class HSBC_board(scrapy.Spider):
    name = 'HSBC_board'

    # define URLs
    allowed_domains = ['www.hsbc.com/']

    # define URLs and parsing method
    def start_requests(self):
        # current site
        yield scrapy.Request('http://hsbc.com/who-we-are/leadership/',self.parse_current)

        # archived sites
        yield scrapy.Request('http://web.archive.org/web/20181109052314/https://www.hsbc.com/about-hsbc/leadership', self.parse_2015_2018)
        yield scrapy.Request('http://web.archive.org/web/20171109052314/https://www.hsbc.com/about-hsbc/leadership', self.parse_2015_2018)
        yield scrapy.Request('http://web.archive.org/web/20161109052314/https://www.hsbc.com/about-hsbc/leadership', self.parse_2015_2018)
        yield scrapy.Request('http://web.archive.org/web/20151109052314/https://www.hsbc.com/about-hsbc/leadership', self.parse_2015_2018)
        yield scrapy.Request('https://web.archive.org/web/20141202033024/http://www.hsbc.com/about-hsbc/leadership', self.parse_2013_2014)
        yield scrapy.Request('https://web.archive.org/web/20131121204902/http://www.hsbc.com/about-hsbc/leadership', self.parse_2013_2014)

    def create_board(self, name, title, year):

        # create item for export
        item = Board()

        # asign fields
        item['company'] = 'HSBC'
        item['title'] = title
        item['year'] = year
        item['name'] = name

        return item

    # parse the current HSBC board
    def parse_current(self, response):
        # define selector that contains all items
        all_people = response.css("li.directors-index__item")

        # iterate through items
        for person in all_people:
            # manually parse name
            name = person.css("a>div>div>h3.contact-large-image-teaser__header::text").get()
            print(name)
            # manually parse title
            title = person.css("a>div>div>p>span::text").get()

```

```
            now = datetime.datetime.now()
            year = now.year

            # Return item
            yield self.create_board(name,title,year)

    # parse the board from 2015-2018
    def parse_2015_2018(self, response):
        # define selector that contains all items
        all_people = response.css("li.profile-col1")

        # iterate through items
        for person in all_people:
            # manually parse name
            name = person.css("div>h3>a.title-profile.text-red::text").get()
            # manually parse title
            title = person.css("div>p.profile-info::text").get()

            # find the year from the crawled URL
            url = response.request.url
            date_info = url.split("/")[4]
            year = date_info[:4]

            # Return item
            yield self.create_board(name,title,year)

    # parse the board from 2013-2014
    def parse_2013_2014(self, response):
        # define selector that contains all items
        all_people = response.css("div.profile-col1")

        # iterate through items
        for person in all_people:
            # manually parse name
            name = person.css("div>a.title-profile.text-red::text").get()
            # manually parse title
            title = person.css("div>p.profile-info.profile-desg.bold::text").get()

            # find the year from the crawled URL
            url = response.request.url
            date_info = url.split("/")[4]
            year = date_info[:4]

            # Return item
            yield self.create_board(name,title,year)
```

Listing 1: An example of a complete parser for the HSBC website. The `parse_current()` function parses the current webpage. Other parsers use the Wayback Machine to parse historical data.