Project Report: Mining Data Records in Python

Data Wrangling with Leonid Libkin & Pierre Senellart

Master IASD - PSL University

João Paulo Casagrande Bertoldo
`joaopcbertoldo@gmail.com`

April 2020

Abstract

This project is part of the evaluation of the course Data Wrangling with Professors Leonid Libkin and Pierre Senellart as part of the Master's degree IASD at the PSL University (session 2019/2020).

This report presents a review on my Python implementation of the algorithm *Mining Data Records in Web Pages*, designed by Liu, Grossman, and Zhai (2003a).

Contents

1 Introduction **2**

2 Original Paper **2**
2.1 Authors 2
2.2 Overview 3

3 Implementation **4**
3.1 Overview 4
3.2 Core 5
3.2.1 Dependencies 5
3.2.2 Classes 6
3.2.3 Intermediate structures 6
3.2.4 Functions 7
3.2.5 Parameters 8
3.3 Web API and Chrome Extension 8
3.4 Other modules 8
3.5 Unit tests 9

4 Experiments **9**
4.1 Analyzed factors 10
4.2 Procedure 10
4.3 Evaluation 11
4.4 Results 12

5 Conclusion **13**

1 Introduction

The initial goal of this project was to implement an open version of the algorithm created by Liu et al. (2003a) in Python. In addition to it, the project aimed to include a visualization feature that would allow the user to inspect the code execution. Unfortunately, as we will see in Section "Implementation", the algorithm did not work as expected at first.

As a consequence, I changed the project's goals to investigate that and try to correct the problem by changing it's parameters. For this reason, the visualization was not implemented. Although, the final results are presented in a rudimentary version of that by coloring data records directly in the HTML pages.

I argue, based on my experiments, that this algorithm does not seem to work with today's websites. I experimented MDR's main parameter, the edit distance threshold, to show that. Besides, I show that, using Liu et al.'s assumptions, today's pages do not even return results at all.

In the following sections, first, I summarize the original paper that introduced the algorithm *Mining Data Records* (MDR), then I present the main components of my implementation, and finally explain how I tested it and what the results show.

The project is stored in an open GitHub repository under the MIT License: link to the repository.

2 Original Paper

To implement this algorithm, I based myself on the paper *Mining Data Records in Web Pages*, by Liu, Grossman, and Zhai (2003b). According to Semantic Scholar[1], it has been cited over 400 times, of which more than 90 articles were highly influenced by it.

2.1 Authors

Bing Liu is a Distinguished Professor of Computer Science at the University of Illinois at Chicago (UIC). He published many articles, like the one used for this project, about data mining applied to data from the web and wrote of the book *Web Data Mining* (Liu, 2007).[2]

Robert L. Grossman was a professor at the University of Illinois at Chicago for more than 20 years. Nowadays he is a professor at the University of Chicago. Grossman has received several awards and makes part of a number of advisory boards. He is Chair of the Open Commons Consortium (OCC) and ACM Fellow since 2016.[3]

Yanhong Zhai was, by the time of the publication, a Ph.D. student at the University of Illinois at Chicago. Since her Ph.D. degree in 2006, Yanhong has been working in Microsoft as Software Development Engineer (SDE) for more than 10 years. Notably, she has worked in web data mining and is now a Principal SDE in the company. [4]

[1] *"Mining Data Records in Web Pages" on Semantic Scholar* (2020).
[2] *Bing Liu's page on the Computer Science Department of the University of Illinois at Chicago's website* (2020).
[3] *Robert L. Grossman's LinkedIn profile* (2020); *Robert L. Grossman's page* (2020).
[4] *Yanhong Zha's LinkedIn profile* (2020); *Yanhong Zha's page on the Computer Science Department of the University of Illinois at Chicago's website* (2020).

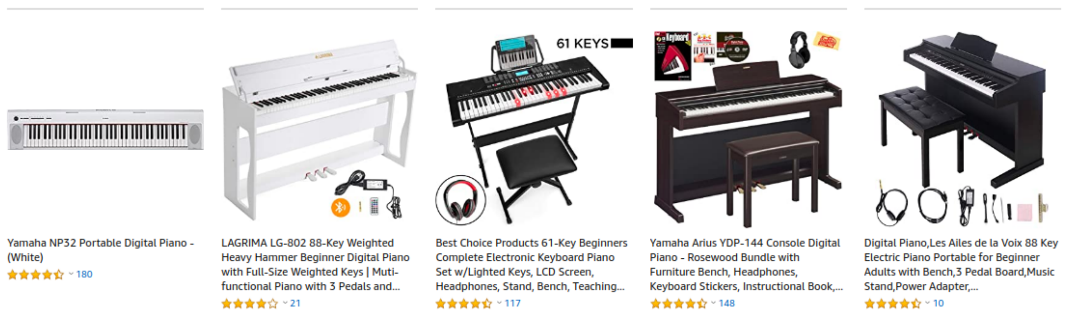

61 KEYS
Yamaha NP32 Portable Digital Piano - (White)
180
LAGRIMA LG-802 88-Key Weighted Heavy Hammer Beginner Digital Piano with Full-Size Weighted Keys | Muti-functional Piano with 3 Pedals and...
21
Best Choice Products 61-Key Beginners Complete Electronic Keyboard Piano Set w/Lighted Keys, LCD Screen, Headphones, Stand, Bench, Teaching...
117
Yamaha Arius YDP-144 Console Digital Piano - Rosewood Bundle with Furniture Bench, Headphones, Keyboard Stickers, Instructional Book,...
148
Digital Piano,Les Ailes de la Voix 88 Key Electric Piano Portable for Beginner Adults with Bench,3 Pedal Board,Music Stand,Power Adapter,...
10

Figure 1: Sample page from www.amazon.com with data records about pianos. Each product offered on the page is a data record - that is, an object containing an image, a title, a rating, and the number of purchases of the product.

2.2 Overview

The article introduces and explains the algorithm *Mining Data Records* (MDR). A *data record* is a structured object on a page that holds information about some entity. The latter can be, supposedly, from a database of records exposed on the website. For instance, it can be a product on a market place website like Amazon (Figure 1).

MDR finds data records inside table tags in HTML pages without using any heuristics about the page's contents or topic. Here below, I briefly describe the article's contents, but the reader is encouraged to read the short (Liu et al., 2003a) or technical (Liu et al., 2003b) version of the original publication for more details.

Rationale Consider an HTML document as a tree structure where tags inside another are children of the latter (see Figure 2a). The core of the algorithm is based on three ideas: the notion of Generalized Node (noted `GNode` for short), Data Region (noted `DataRegion`), and a measure of the distance between `GNode`s.

A `GNode` is a set of $n \in \{1, ..., N_{max}\}$ adjacent nodes (thus, under the same parent node). A `DataRegion` is a sequence of adjacent `GNode`s that are *similar*. See Figure 2b for an illustration of these two concepts. In this context, *similar* means that the distance between each two directly adjacent `GNode`s is less than the threshold $D_{threshold}$. The measure of distance is the Normalized Levenshtein Edit Distance [5]:

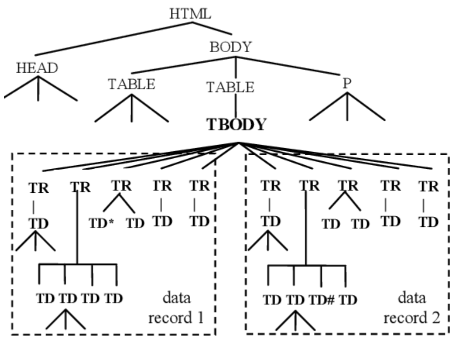
$$d\left(s_1, s_2\right) = \frac{LevenshteinEditDist\left(s_1, s_2\right)}{\frac{\text{length}(s_1)+\text{length}(s_2)}{2}} \in [0, 1] \tag{1}$$

The values of N_{max} and $D_{threshold}$ are parameters of the algorithm and the authors reported using, respectively, 10 and 0.30. Once the `DataRegion`s are found in the HTML, there is a scan over them that figure out where the data records are.

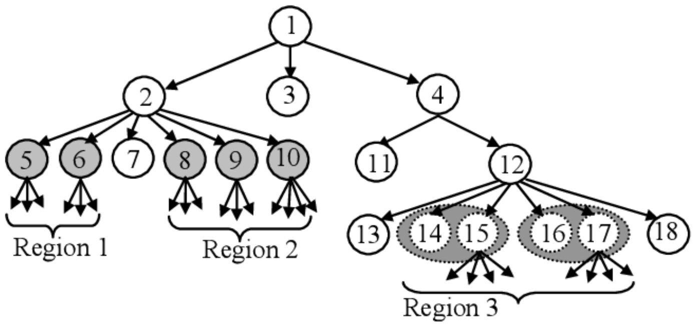
Experimental results The authors tested MDR on 46 different pages, with a total of 621 objects (data records). They reported precision and recall of, respectively, 100% and 99.8%, outperforming (by far) other previously proposed algorithms. The list of websites used by them can be seen in Table 1 in Liu et al. (2003a).

Issues Unfortunately, the authors did not make available any reference implementation. They also did not make public the specific cached HTML pages that they used to test the algorithm on, or even the

[5]See the *"Levenshtein Distance" article on Wikipedia* (2020) for more information.



(a) HTML as a tree structure.



(b) GNode and DataRegion concepts illustrated. Region 1 is a DataRegion with two GNodes of size $n = 1$. Region 2 is a DataRegion with three GNodes of size $n = 1$. Region 3 is a DataRegion with two GNodes of size $n = 2$.

Figure 2: Key concepts. Source: Liu et al. (2003a).

specific URLs of the pages used (only the websites). Some of the websites mentioned in their test set do not exist anymore (like *bookbuyer.com* and *codysbooks.com*) or became a completely different website (such as *eveonline.com*). Moreover, they do not describe how the data was labeled or inspected, making it hard to reproduce their metrics.

3 Implementation

3.1 Overview

The first implementation of the algorithm, with the suggested parameters, did not work as the authors claim. For this reason, I decided to investigate the effect of some parameters and variations of the algorithm. To test my implementation, I collected HTML pages and annotated the number of data records in each of them. To facilitate the annotation task, I created a Google Chrome extension to interface with a Flask[6]web API. Next, I ran the algorithm's steps individually, saving intermediate results for inspection.

Code structure The next subsections will explain the main components (modules, classes, functions, etc) of my code. (Most of) The code referenced in this section is organized in the structure shown in Listing 1.

[6]Link to Flask's official page.

```
# project root
/
|--/src
|   |--/api
|   |   |-- main.py
|   |
|   |--/extension
|   |   |-- readme.txt
|   |   |-- manifest.json
|   |       ...
|   |
|   |-- core.py
|   |-- files_management.py
|   |-- prepostprocessing.py
|   |-- utils.py
|
|--/test
|    ...
|
|--/dev
|   |--/training
|   |   |-- evaluating_stuff-with-list.ipynb
|   |   |-- evaluating_stuff-with-list-with-cleanup.ipynb
|   |   |-- evaluating_stuff-withOUT-list.ipynb
|   |   |-- evaluating_stuff-withOUT-list-with-cleanup.ipynb
|   |   |-- preprocess_all.py
|   ...
```

Listing 1: Main code structure.

3.2 Core

This is where the algorithm itself and the entities (classes) that interact with it live. The code in this module is not supposed to manage any of the files, inputs or outputs formats, nor treat the way they are loaded and saved. Instead, it expects all the inputs to be in known specific formats.

The main input types are `HtmlElement` (from `lxml.html`), the classes defined in the module (discussed next), and a few dictionary structures, which are specified with `typing`[7]for documentation[8].

3.2.1 Dependencies

`lxml` After reading about[9]and experimenting with a few Python libraries for HTML handling, I chose `lxml` for a few reasons. First, it is designed to work with the `ElementTree` API, making it simple to treat HTML documents as a tree structure like the original paper does. Second, it is fairly easy to parse the HTML into `HtmlElement` and serialize it back, which are the main functionalities I used. And finally, unlike other libraries, it is not full of fancy features that I did not need.

`graphviz` Since the original goal was to generate visualizations, I also reviewed several options for graph rendering and chose `graphviz` because it is open source. Also, it provides a fairly simple interface for the features I was looking for. In the end, it was not used in the core code, but you may find some utility functions that can be used to render a graph from an `HtmlElement`.

[8]Link to the `typing` library's page in Python 3.6.
[8]Link to their definitions in `core.py` in the repository.
[9]For instance, this review on *tomassetti.me*.

python-Levenshtein One of the core tasks in the algorithm is to measure the (normalized) Levenshtein edit distance between strings - remember that `GNode`s are *similar* if this distance is below a threshold. I initially used the library `strsim`[10], but it started creating problems (which I could not debug) at some point. So I switched to the library `python-Levenshtein`[11].

3.2.2 Classes

GNode corresponds exactly to the concept of *Generalized Node* introduced in the original paper. It represents a set of subsequent HTML tags in the document graph and it includes all the subtrees starting at its nodes. In practice, it is referenced by its parent's name and the (0-starting) indices of the first and last HTML tags of the `GNode`.

DataRegion is a tuple with the `GNode` size n (i.e. the number of HTML tags in it), the index of the first tag of the first `GNode`, and the number of tags covered by the `GNode`. This notation is not very intuitive but it is handy inside the algorithms that use it. For practical reasons, I also added to it the parent's name - which is, by definition, the same as its covered `GNode`s' parent.

DataRecord Most of the time, `DataRecord`s correspond exactly to one `DataRegion`. However, some data records might be split into more than one `DataRegion`. For instance, when the titles of the objects are in cells of a table row and the description is in the respective cells of the next row[12]. Therefore, a `DataRecord` object is defined as a list of `GNode`s, so both cases can be treated uniformly.

NodeNamer Unfortunately, `lxml`'s implementation does not have a consistent `id()` (same for `__hash__()`). I do not know the reason, but I empirically checked that an `HtmlElement` object can return different values from those functions between calls (during the same run time). For this reason, it is not easy to uniquely reference any node of a generic HTML document. This limitation is crucial because the intermediate data structures (see the next section) map partial results per node. My solution was to add a node name as a tag attribute to the `HtmlElement`s of the document with a sequential value. The `NodeNamer` object serves as a stateful function that is used as a shortcut to get that value from the `HtmlElement`s - avoiding a snippet repetition and making it consistent.

MDR puts together all the pieces of the algorithm (written in functions). It makes it easier to access some variables and mappings (the data structures described below) that are passed between the functions.

Others During my tests, I considered using different distance thresholds in the different places where this notion is used (there are three of them). The class *MDREditDistanceThresholds* would be used to allow this change to be made easily, but in the end, it was not used. Finally, *WithBasicFormat* is a utility for printing the other data structure in a more readable way.

3.2.3 Intermediate structures

These two are important data structures for understanding the code because they act like mappings of information. They are used to pass sub-results between the sub-algorithms.

[11]Link to `strsim`'s PiPy page.
[11]Link to `python-Levenshtein`'s PiPy page.
[12]This particular type of case is better explained in Section *Non-contiguous object descriptions* in Liu et al. (2003a)

DISTANCES_DICT_FORMAT It keeps all the distances between pairs of `GNode`s, which are computed before being used. This dictionary's keys are the node names and it maps a node to another dictionary. The latter maps the possible `GNode` sizes n (integers from 1 up to N_{max}) to yet another dictionary. This last one maps pairs of `GNode` to distances (a float between 0 and 1). See the example below.

```
{
    # node name
    "table-00000": {
        # GNode size
        1: {
            # GNode pair
            (
                GNode("table-00000", 0, 1),
                GNode("table-00000", 1, 2)
            ): 0.1,

            (GNode(..., 1, 2), GNode(..., 2, 3)): 0.2,
# the distance
            (GNode(..., 2, 3), GNode(..., 3, 4)): 0.5,
            # ...
        },
        2: {...},
        # ...
        10: {...},
    },
    "table-00001": {...},
    ...
}
```

Listing 2: Illustration of a `DISTANCES_DICT_FORMAT`.

DATA_REGION_DICT_FORMAT This one maps the nodes' names the set of `DataRegion`s found under that node.

```
{
    # node name
    "table-00000": set({
        DataRegion("table-00000", 2, 0, 4),
        DataRegion("table-00000", 3, 6, 9),
    }),
    "table-00001": ...,
}
```

Listing 3: Illustration of a `DISTANCES_DICT_FORMAT`.

3.2.4 Functions

The main functions correspond almost exactly to the pseudo-codes described in the original paper. Their behaviors can be summarized as follows:

- `compute_distances()`: recursively iterates over all the nodes to compute the distances dictionary;
- `_compare_combinations()`: on a given node, effectively goes through all the possible combinations of pairs of `GNode`s of all sizes and computes the distances;
- `find_data_regions()`: recursively find all the data regions under a subtree of HTML document;

- `_identify_data_regions()`: on a given node, find all the data regions such that each one is the largest (in terms of covered nodes) possible;
- `find_data_records()`[13]: scans all the`DataRegion`s to extract `DataRecord`s accordingly to two different cases: when the `GNode` sizes are of size 1 or larger;
- `_find_records_1()`: creates `DataRecord`s either from all the `GNode` s of the `DataRegion` or from its children;
- `_find_records_n()`: creates `DataRecord`s directly from the `GNode`s covered by the `DataRegion` or from pieces of them to handle the case where the records are not contiguous (see the paragraph about the `DataRecord` class).

3.2.5 Parameters

The algorithm has three parameters that are passed to the initialization of the MDR object:

- **`minimum_depth`**: all nodes with a tree depth[14] are not taken into consideration in the recursive calls of the functions described above. As a consequence, any direct child of these nodes can never be part of a `DataRecord`;
- **max_tag_per_gnode** (N_{max}): the largest `GNode` size taken into consideration;
- **edit_distance_threshold**: the maximum distance between two `GNode` s for them to be considered *close* - therefore, possibly in the same `DataRegion`.

3.3 Web API and Chrome Extension

To make it easier to use and test *pymdr*, I developed a minimalist Google Chrome extension that allows the user to call the algorithm on a (locally running) Flask API. The extension has two functionalities:

1. It copies and sends to the API the URL of the currently open page (button *analyze*). Then the API executes the algorithm and returns a local file path (button *copy output path*). The file is a local copy of the HTML page with all data records painted with different colors.
2. The user inputs the correct number of data records on the page (input box *nDataRecords*) and the API saves it (together with the page's URL) locally.

The extension must be installed using Chrome's developer mode on and the API must be running locally. To install the extension, follow the instruction in the `readme.txt` file inside the extension's folder[15]. To start the API, launch the script `launch-me.sh` in the root of the project[16].

3.4 Other modules

The other two modules are rather for support tasks like managing files and treating the HTML files. They are mostly used in the API and in the script `preprocess_all.py` (explained in Section "Procedure").

`files_management` defines utilities to write and read from files easily and keeping their locations consistent. The organization of the files mostly depends on the class `PageMeta`, which is used to access metadata from an annotated page. The files corresponding to a given page are referenced by a (shortened)

[13]This algorithm is not directly given in the paper like the other, but only described in the text.
[14]Link to the definition of "Tree depth" on Wikipedia.
[16]Link to the extension's `readme.txt` file in the repository.
[16]Link to the `launch-me.sh` file in the repository.

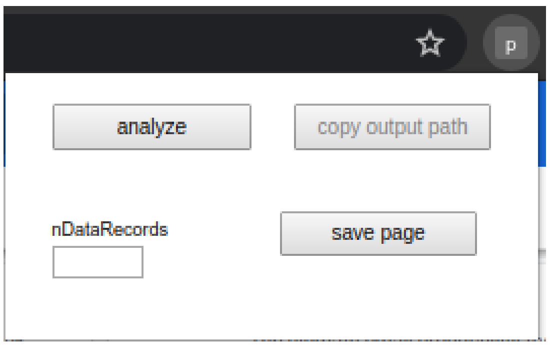


Figure 3: Screenshot of the Pymdr Google Chrome extension.

hash of the URL where the page comes from. They are all located in the directory `outputs` at the root of the project. This directory's location can be changed in the file `config.yml` in the root of the project.

`prepostprocessing` wraps each sub-algorithm that compose MDR (see Section Functions). They save intermediate results in files, load them from previous steps, and ensure that the files already processed are skipped. This is useful to keep executions consistent between calls, especially in the script (see "Procedure") used to process all the pages.

3.5 Unit tests

The directory `test` has the same structure as `src` and the respective files contain unit tests for those in the latter (see the Listing 1). They use Python's builtin package `unittest`. For maintenance purposes, I did not create unit tests for everything, but only for the most important parts of the algorithm. They were indeed very useful during my iterations of the code making sure to keep the code correct. These tests have rather simple input/output tests. So, it might be a useful entry point to understand those functions' interfaces.

4 Experiments

From the very beginning, I realized it would be painful to test and inspect the results by managing the inputs and outputs of the core module manually. For this reason, I wrote the first version of the extension and the API only for caching the HTML pages, executing MDR, and returning another file with colored HTML tags to identify the data records.

(Very) Slow first results My very first tests showed that my code was extremely slow. Although I did not record execution times, the typical execution time was in the tens of seconds (or even minutes). This is way above the authors' claims of sub-second execution times.

After re-reading the technical paper carefully, I realized that the algorithm is only supposed to consider table-related tags (table, tr, th, td, ...). In addition to that, the results did not seem correct at all. The algorithm was not capable of identifying any data record. This made sense because the algorithm was simply not intended to work with all the tags.

I then started ignoring all other tags. It vastly accelerated the execution to a few seconds in the worst case. However, many of the pages did not have any data records at all. After manually inspecting a few, I realized that many of the pages simply did not use table tags, but lists (`<ol>` and `<ul>`) for organizing the data records.

I hypothesize that web pages, today, use table tags way less often than when the algorithm was developed. According to *"HTML table basics" on MDN web docs* (2020): "...a lot of people used to use HTML tables to lay out web pages...". This essentially breaks the algorithm because it is made to work well with these tags. Therefore, I decided to make sure it was not a matter of bad parameters. So I tested the algorithm with different threshold values since this is the most important parameter.

4.1 Analyzed factors

Threshold The original recommended value was 0.30, so it makes sense to keep the threshold range *around* 0.30. Also, the slowest part of the algorithm is (by far) the computation of the distances (see Functions). Fortunately, it does **not** dependent on the threshold choice. So, having a large value set for this parameter should not increase much the execution of the tests. Therefore, I tested with all the values in $0.05, 0.06, ..., 0.49, 0.50$.

HTML Lists MDR was designed to work with table tags, so considering all the HTML tags did not make any sense. On the other hand, considering only tables seemed too restrictive (it will be explained in Section Results). Thus, I considered a version only with tables and another with lists additionally. This behavior can be controlled by changing the function `should_process_node`[17]in `core.py`.

Edit distance distortion While reviewing and cleaning my code, I realized that I introduced an issue to the algorithm by distorting the edit distances. This happened due to the naming solution I used to reference the nodes (see the class `NodeNamer` in Section Classes). It adds a new attribute to the HTML tags, so, while serializing subtrees, this attribute ends up as part of the string. This affects the distance measure especially for nodes with little content because the sizes of the attribute's key and value make the distances decrease artificially.

For this reason, I made a hotfix that cleans all the attributes from the subtrees before measuring the distances. The parameter used to control this behavior is the constant `STR_DIST_USE_NODE_NAME_CLEANUP`[18]in `core.py`.

Therefore, there are four tests, corresponding to the combinations of the two previous parameters' values: $\{\text{with lists}, \text{without lists}\} \times \{\text{with correction}, \text{without correction}\}$. Here, *correction* refers to the distance distortion hotfix. For each of these combinations, all the threshold values are tested. Then, I looked up for the best values and inspected *promising* results manually in detail.

4.2 Procedure

Test pages First, I collected sample pages from the same websites mentioned in Liu et al. (2003a)[19]. Several of them did not exist or completely changed domain/content. In total, I collected 94 pages, of which 86 were processed - the others went through some problems in the process.

I collected these pages using the Chrome extension to annotate the ground truth number of data records of each page. The pages' URLs and their annotations can be seen in the file `outputs/pages-meta.yml`[20].

[17]Link to the function `should_process_node` in the repository.
[18]Link to the constant `STR_DIST_USE_NODE_NAME_CLEANUP` in the repository.
[19]I looked for the original cached pages used by the authors and even tried to get in touch with them to find them. Unfortunately, I could not find them out. I also tried to look up for the pages used in the two main references of the paper (OMNI and IEPAD), but they are also not available.
[20]Link to the file `pages-meta.yml` in the repository.

Script I used the script `dev/training/preprocess_all.py`[21] to go through all the thresholds, all the pages, step by step. To control the other parameters, one should directly change the code in `core.py`. It is possible to choose, directly in the script, which steps should be executed to avoid useless executions when running it multiple times.

Outputs The results and intermediate files generated by the script are organized in the folder `outputs` under the root of the project. It contains the following subdirectories:

- `raw_htmls`: contains the cached HTML pages without any modification;
- `preprocessed_htmls`: contains HTML files after two phases of preprocessing: (1) without comments, script, and style tags; (2) with the node names added to all the tags (see `NodeNamer` in Section "Classes").
- `intermediate_results`: contains two types of file: (1) the precomputed distances, returned by the function `compute_distances()`; (2) the precomputed `DataRegion`s, whose files are separated by different threshold values[22] because this parameter directly impacts the results.
- `results`: contains the `DataRecord`s in separate files following the same logic of the `DataRegion` files (i.e. by threshold value). It may also contain HTML files where the background of the tags in the same `DataRecord` are of the same color for inspection.

Reproduction All the result files can be found in the file `outputs/bkps-all.zip`[23]. To inspect them, extract everything to `outputs`. You will see four directories starting with "bkp". They correspond to the four different cases tested.

There are four Jupyter Notebooks inside `dev/training`. Again, they correspond to the combinations of parameters. They have all been executed with the parameters indicated in the files' names (also indicated in their first cells). To execute them, move the two directories inside one of the backup directories (`intermediate_results` and `results`) to `outputs`. I recommend looking, first, at `evaluating_stuff-with-list-with-cleanup.ipynb`[24] because it is better commented with explanations. All the others have the same code but were not as much documented.

The reader can also easily rerun all the tests by executing the script in `dev/training/preprocess_all.py`. Please, note that the instructions in the `README.md` file should be followed before. Make sure that you empty the `outputs` folder before executing (except for the `pages-meta.yml`). You should also verify which steps of the process should be executed by setting the boolean parameters of the function `main()` inside the script.

4.3 Evaluation

Partial annotations Since the task of manually annotating the data records inside each HTML file would have been too laborious, I opted for another solution. As mentioned before, I only annotated the correct number of data records on the pages, which is relatively easy to execute. For instance, see *this page*: it has 12 records. Note that, if the algorithm finds 12 data records, it does not necessarily mean that it worked. The results might be in a completely different part of the HTML page that does not have any data record *de facto*.

[21] Link to the script `preprocess_all.py` in the repository.
[22] Values are rounded up to two decimal places.
[23] Link to the file `bkps-all.zip` in the repository.
[24] Link to the file `evaluating_stuff-with-list-with-cleanup.ipynb` in the repository.

Simplification assumption However, if it does not have 12 records in the results (or, say, 'almost'), it probably did not find the correct records. This is a reasonable assumption because the data records are usually adjacent. Therefore, finding the correct region should entail finding the other ones as well. An exception might occur when there is a disconnected data record[25].

Therefore, we will assume that this *soft* annotation can be used as a valid way to check if the algorithm is working properly. Besides, it is laborious to inspect raw HTML. This framework was also useful to filter out bad candidates (e.g. eliminating cases with 60 data records where there should be 10) to inspect the good ones in detail.

Visualizations Since it is not reliable to compute accuracy metrics on this signal, I manually inspected many results and generated several visualizations to find good candidates. This is not in any measure negative for the sake of comparing results with Liu et al. (2003a). They did not explain precisely how their data was annotated, how the precision and were computed, and did not make the data available. Therefore, having those metrics would not be of more utility than checking the results individually for the sake of this project.

Figure 4 shows, on the y-axis, the pages (there is no correlation between the number and the page itself) and, on the x-axis, the threshold values tested for each of them. The dots displayed represent $(\text{page}, \text{threshold})$ pairs where the number of data records returned by MDR is within a margin of ± 5 relative to the ground truth. In the previously mentioned Jupyter Notebooks, you will find more detailed visualizations that I used.

4.4 Results

Despite trying to adapt the algorithm to today's pages, it does not seem to adapt anymore. Figure 4 shows that no threshold seems to adapt well to the pages and, more generally, the algorithm does not succeed in finding data records properly.

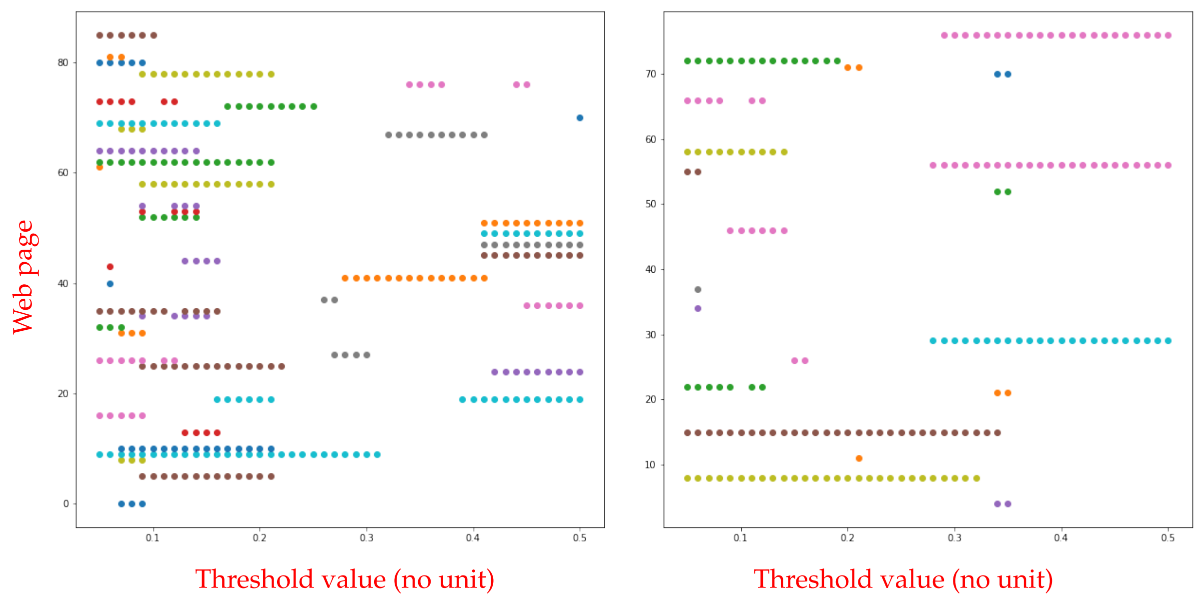
As mentioned in Section "Analyzed factors", without considering list elements almost nothing is detected at all. This can be seen by comparing Figures 4b and 4a. The latter has way fewer results than the former. I also manually checked that in many of these cases it is due simply to the fact the tables are no longer used in the pages.

Out of the four combinations of parameters that I experimented with, the most promising was "with list elements" and "with correction". There is no reason to believe that the edit distance distortion would make it better and, as argued above, it does not make sense to use the algorithm only with HTML tables.

Notice that the margin of ± 5 used for this plot is relatively loose. A large part of the pages have only a few tens of data records, so having 5 data records identified is already bad enough. With this in mind, there are a few things to conclude from this graph:

1. **There are many blank lines**: regardless of the threshold, the number of data records is *far* from the ground truth.
2. **There is no gold threshold**: the authors of MDR claim to achieve very high accuracy with a threshold of 0.30. Here we see that no value is capable of having relevant results in most of the pages. If it was the case, there would be a visible vertical line or column full of dots.
3. **The sensitivity varies enormously**: we can see pages where the value of the threshold seems to not even matter, while, for others, there is are very few (or even one) values with relevant results.

[25]See Section "3.4 Data Records not in Data Regions" in Liu et al. (2003a) for further detail



(a) With list elements.

(b) Only table elements.

Figure 4: Closest results per page in a margin of ± 5. Each point is a run where the number of data records is within a margin of ± 5 relative to the ground truth. The x-axis is the threshold distance used for that run and the y-axis is merely for different pages.

Therefore, we can see from Figure 4a that MDR does not perform well overall. More rigorously, the graph tells us that it *probably* is not working well. I further inspected those pages using a smaller margin and a few random ones and the algorithm does not correctly find the data regions. For many of the cases where the number of data records coincides with the ground truth, the algorithm is capturing menus, sidebars, and button panels on the web pages.

5 Conclusion

My implementation of the Mining Data Records (MDR) ended up being rather correct in terms of software engineering, but the algorithm simply does not properly work on today's pages anymore. I can see two clear improvements that could be useful for the code structure itself: (1) replacing `lxml` and (2) implementing more unit tests.

Although `lxml` has useful features and does fit well most of the needs of the project, it introduced the problem with nodes identification (explained in Section "Classes"). This issue consequently introduced the need for another solution that is not clean and makes the implementation stateful.

The unit tests showed to be very handy to debug my code as I iteratively made modifications to my code. It would probably be useful to have some examples a bit more complex in the core functions. The module `files_management` has some important functionalities that make it easier to manipulate the core code, so it would be useful to have unit tests for it as well.

As shown in the results, unfortunately, this algorithm does not perform well on modern pages regardless of the parameters used. This work could further be improved by investigating the local behavior of the algorithm. If we verify that it is indeed possible to still use MDR with data records embedded in lists, it could be possible to make it work. For instance, by tweaking the `minimum_depth` it could be possible to get rid of the menus and panels.

References

Bing Liu's page on the Computer Science Department of the University of Illinois at Chicago's website. (2020, 04). www.cs.uic.edu/ liub. (Accessed on 2020-04-18)

"HTML table basics" on MDN web docs. (2020, 04). developer.mozilla.org. (Accessed on 2020-04-24)

"Levenshtein Distance" article on Wikipedia. (2020, 04). en.wikipedia.org. (Accessed on 2020-04-18)

Liu, B. (2007). *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data.* doi: 10.1007/978-3-642-19460-3

Liu, B., Grossman, R., & Zhai, Y. (2003a). Mining data records in web pages. In *Proceedings of the ninth acm sigkdd international conference on knowledge discovery and data mining* (p. 601–606). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/956750.956826` doi: 10.1145/956750.956826

Liu, B., Grossman, R., & Zhai, Y. (2003b). "Mining Data Records in Web Pages" UIC Technical Report.. Retrieved from `https://www.cs.uic.edu/~liub/publications/KDD-03-techReport.pdf`

"Mining Data Records in Web Pages" on Semantic Scholar. (2020, 04). www.semanticscholar.org. (Accessed on 2020-04-18)

Robert L. Grossman's LinkedIn profile. (2020, 04). www.linkedin.com. (Accessed on 2020-04-18)

Robert L. Grossman's page. (2020, 04). rgrossman.com. (Accessed on 2020-04-18)

Yanhong Zha's LinkedIn profile. (2020, 04). www.linkedin.com. (Accessed on 2020-04-18)

Yanhong Zha's page on the Computer Science Department of the University of Illinois at Chicago's website. (2020, 04). www.cs.uic.edu/ yzhai. (Accessed on 2020-04-18)