Paper Title*

1st Zoé Giacomi
MSc in Finance
HEC Lausanne,
Lausanne, Switzerland
zoe.giacomi@unil.ch

2nd Gino Mazzoni
MSc in Finance
HEC Lausanne,
Lausanne, Switzerland
gino.mazzoni@unil.ch

3rd Alexander Semionov
MSc in Finance
HEC Lausanne,
Lausanne, Switzerland
alexander.semionov@unil.ch

***Abstract*—Real estate has been for a long time an opaque, decentralized market. As such, the Swiss housing market is subject to information asymmetry between buyers and sellers, creating potential adverse selection. The housing market is dominated by real estate agents who possess more information than potential buyers which contributes to skewing the price of houses. Though transparency has positively evolved with the apparition of the internet and online properties platforms, there is still a lack of publicly available historical data and comparative databases. As such, our research is focused on testing whether creating a continuously updated open-source database of all present and historical listings in Switzerland is feasible.**

***Index Terms*—swiss real estate, web scraping, comparis, scrapy, selenium, user interface, transparency, economic research**

I. Introduction

Real Estate Transparency has long been associated with favorable business environments [2]. While Switzerland is ranked 11th in the JLL "Global Real Estate Transparency Index" [1], it is lagging many other European countries, such as France, the UK, Sweden, and Germany. By empirical observation, the Swiss market is dominated by private valuers and brokers ("Agents"), who refrain from citing exact addresses and informing buyers the right price of properties on the market. Hence, leading to vast information asymmetry between Agents and potential buyers. However, the rise of web platforms is slowly erasing information asymmetry, where potential buyers may compare different properties without the use of Agents. Nevertheless, even with the use of these revolutionary real estate web platforms, it is still difficult to assess for a potential buyer what drives the price of their target property. Is it undervalued? Overvalued? At the right price? Therefore, despite advancements in enhancing transparency of the housing market, there are still improvements to be made to minimize information asymmetry, so that potential buyers may make rational and informed decisions before investing in real property.

II. Research Question

A. Problem

Real Estate has long been a very opaque asset class, with subjective pricing methods; as opposed to asset classes that trade in the open market (stocks, bonds, derivatives, etc.). Historical transaction data is difficult to obtain, and the fact that each property is radically different in its characteristics makes it difficult for unexperienced buyers to assess property value. Therefore, buyers must rely on Agents, who have better knowledge of the housing market. Rutherford, Springer and Yavas (2005) [11] show that Agents sell their own houses for a premium of approximately 4.5%, whereas Levitt and Syverson (2008) [9] find that houses owned by Agents sell for approximately 3.7% more than other houses. The evidence suggests the presence of Agents' information advantage in the housing market. Why do Agents pay lower prices when buying their own houses? One explanation is related to information asymmetries in the housing market. Real estate agents have information advantages over less informed "nonagent" buyers. These studies also show that Agents will use this information to their own personal advantage, which may enter in conflict with the interest of their clients. With the apparition of e-commerce real estate platforms, potential buyers have access to current listings with prices, characteristics, and pictures. This may help in reducing information asymmetry to a certain extent. However, these listing are actual listings, where historical listings are not available. Moreover, the data is not formatted and structured into databases to make informed comparisons. Finally, the price that is reflected in listings may not be the final prices, as there is generally a negotiation process involved before the transaction closes. Therefore, we find the lack of a real open-source platform, showing past listings in a structured fashion to be necessary in reducing information asymmetry and enhancing transparency of the housing market in Switzerland.

B. Objective

Our goal is to provide Swiss potential home buyers with an open-source platform, that lists all previous past listings, which are ordered by NPA (Swiss postal codes), price, square meters, and number of rooms. This platform would pull data automatically in certain periodic intervals, which would provide with a wide range of historical data, useful to understand trends. This platform would give an edge both to professional Agents and potential buyers, effectively reducing information asymmetry between the two parties. Such platforms would also be relevant for statistics by authorities, insurances, mortgage lenders, as well as advertisers.

C. Scope

Considering the wide array of properties in Switzerland, we decided to focus our attention to Lausanne and its suburbs.

This will permit us to test out the first version of our platform, with limited property data and historical data. However, the platform is scalable to become a nationwide platform.

III. Methodology

A big difference between real estate and other publicly traded assets is that there is no centralized platform where offers can be tracked. In the case of asset classes such as bonds and equity this service is provided by multiple sources such as Bloomberg, Refinitiv and Morningstar. Historical prices, key financial data, plots and proxies are all available on one single platform. Something similar is missing for the Swiss real estate market.

Usually real estate on the various websites is presented as a list of single "tiles", along with a brief overview of the key data and pictures of the corresponding property (for reference see figure 1). Whilst this is already a solid foundation, there is no possibility to, for example, see an average cost of a property per ZIP code and no real way to compare similar estates.

Given that the data is neither available via an API nor through an already established database, the decision was made to write an algorithm that scrapes the required data from a website.

A. Choice of Website

Whilst there exist numerous websites that act as Agents between buyer and seller such as `www.immoscout24.ch` and `www.homegate.ch`, all of them suffer from the same underlying problem mentioned in the introduction. Selecting and pricing real estate by clicking through every listing is cumbersome and inefficient. Furthermore, with the data being split up over many websites, a buyer might not find his ideal new property, simply by searching on the wrong platform.

An attempt at creating a more transparent and efficient real estate market was made by Comparis. On their website, listings from many sources are aggregated and displayed according to their key characteristics. However, also `www.comparis.ch` does not offer tools for further analysis and does not readily display historical data, except for increases or decreases in the price in percentage.

For the purpose of creating a database which provides a user with information on as many properties in a given area as possible, `www.comparis.ch` is the ideal target for scraping data.

B. Web Scraping

When creating a web scraping application in python, there are a number of packages and tools to choose from. The modules include, but are not limited to `requests`, `BeautifulSoup`, `Scrapy` and last but not least, `Selenium`. Each come with their advantages and disadvantages. The former two modules, `requests` and `BeautifulSoup` both come with the bonus of ease of use. The latter two modules offer more functionality, but have a steeper learning curve.

After several attempts, the choice boiled down to a combination of `Selenium` and `Scrapy`. This decision was based on the need to not only collect data on the "top level" of the listing (the kind of data seen in figure 1), but also the information on the detail page of every single URL.

Fig. 1: Typical Listing on Comparis

The need for more data was not the only factor that led to this choice. Today's websites are populated with JavaScript's that procedurally generate the content a user can see. The JavaScript's may load further information based on simple actions such as scrolling on the page, clicking a button, or by more complex actions such as hovering over a certain area or logging in somewhere.

For a web scraping service that relies on parsing HTML (such as `requests` and `BeautifulSoup`), this poses a major issue. The way `requests` works is that it sends a GET request to the website and subsequently returns the source code (HTML) for further analysis. However, simply sending a request is not enough to trigger a JavaScript since it needs interaction from the user to generate the content and thus, "add" source code that can then be analyzed. The source code returned by `requests` for JavaScript heavy websites generally offers very limited insights. A common workaround is the scraping the "hidden API" [7] [8]. Said "hidden API" consists of a JSON file which, in certain cases contains all the information the JavaScript would generate based on the user action. Since this JSON has its own URL, `requests` can get the data from there, and therefore effectively get around the "JavaScript barrier". Due to the structure of Comparis' website, this was not an option.

C. Scrapy

Of the largest web scrapers available for python, `Scrapy` usually serves the needs of larger scale web scraping projects. `Scrapy`'s main feature is the so-called `spider` which can be customized to serve many needs. A `spider` is set up by using the `scrapy genspider <example> <example.com>` command (after running the `scrapy startproject` command in the terminal). Along with the project- and spider folders, `Spider` generates several `.py` files that allow granular customization of the spider. To use a spider, the user has to define a spider class (which inherits all attributes from the class `scrapy.Spider`), that tells `Scrapy` how and in what sequence to scrape which content of a given website.

Websites are created with the user experience in mind first. This entails that the output of a scraping activity can be disorganized and filled with unwanted HTML tags. Here `Scrapy` offers a very attractive solution, namely the `ItemLoader`, which is defined as a class in the `items.py` file. Whilst `ItemLoader` certainly adds to the steepness of the learning

curve, it is a worthwhile endeavor, since the scraped data will come out ”clean” already (”clean” means free of unwanted HTML tags or strings, for more information, refer to section IV-B3). `ItemLoader` ”cleans” the data when it comes in, stores it while the spider is running, and generates an output based on the user's preference.

`Scrapy` can send many requests to a website at once, thus speeding up the data gathering process tremendously (by default `Scrapy` sends around 16 requests a time). However, even when using all the request headers that a normal browser sends to a website, the `Scrapy` spider's were constantly IP-blocked.

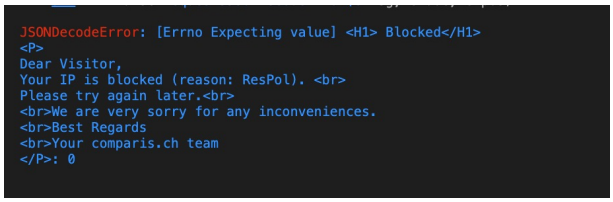


Fig. 2: IP block when running the spiders

D. Selenium

Whilst `Scrapy` works well with extracting data from the website's source code (especially using XPATHs's), it is still not possible to navigate JavaScript heavy websites and trigger the scripts that give access to the important information in the source code. Furthermore, even with the implementation of rotating proxies in `Scrapy` , the IP-block could not be circumvented.

This is where `Selenium` comes in: `Selenium` allows a user to control any browser environment, provided the corresponding driver has been installed. Controlling a browser offers several advantages, the primary advantage being that the request comes from an actual browser, which means it will very rarely get blocked. The downside is that by sending the requests through a browser window, the amount of requests per time is practically reduced to one. This increases the runtime of the program drastically, however it adds the upside that the data collection is successful.

For this project it was decided that by combining `Selenium` with `Scrapy` , the best of both worlds can be obtained: reliable data gathering on the one hand, and clean data extraction on the other hand.

E. User platform

To make the dataset scraped from Comparis more user-friendly we decided to build a Graphical User Interface (GUI). This way the user will be able to search for a property within a certain price range, zip code or with specific features, instead of having to go through the full scraped dataset. We used the open source library `tkinter` **??** to model the GUI. We chose this library as it is pre-installed in python. Moreover it is stable and flexible and provides simple syntax. `tkinter` is the perfect tool for this first prototype. However, if the project grows `tkinter` will not be powerful enough, therefore we recommend switching to PyQT another GUI or even REACT for a more professional look.

IV. Implementation

The whole project can be found in a repository on Github. There the README will provide the necessary information regarding the implementation of the scraper as well as how to use the interface. All of the files were written and run in python version 3.9.12. Below a list of all the packages plus their respective versions that were used can be found.

- `Scrapy` - version 2.6.1
- `Selenium` - version 4.1.5
- `Webdriver_manager` - version 3.5.4
- `Numpy` - version 1.22.3
- `Pandas` - version 1.4.2
- `Time`
- `Datetime`
- `Openpyxl` - version 3.0.9
- `Tk (tkinter)` - version 0.1.0
- `Pillow` - version 9.1.1

A. Structure of the project

```
/.
├── comparis_webscraper
│   ├── comparis_webscraper
│   │   ├── spiders
│   │   │   ├── comparis_scraper.py
│   │   │   └── property_code_scraper.py
│   │   ├── items.py
│   │   ├── middlewares.py
│   │   ├── pipelines.py
│   │   └── settings.py
│   └── scrapy.cfg
├── data
│   ├── database.xlsx
│   ├── property_codes.csv
│   └── property_details.csv
├── GUI
│   ├── gui_data
│   │   ├── data.xlsx
│   │   ├── MeanPriceRooms.xlsx
│   │   └── MeanPriceZip.xlsx
│   ├── cleaning_database.py
│   ├── graphs_GUI.py
│   └── main.py
├── Heatmap
│   ├── PLZO_SHP_LV95
│   ├── MAP1.py
│   └── data.xlsx
├── README.md
├── Makefile
├── .gitignore
└── .git
```

B. Implementation of the Web Scraper

As mentioned previously, the web scraping was done by utilizing two different tools, `Scrapy` and `Selenium`. When programming the spider, the main challenge was to implement the vision of what it should do. When called, the spider should be able to do the following:

1) Activate the browser via `Selenium`
2) Access a list of predefined URLs
3) Loop through every URL and activate the JavaScript
4) Download 22 datapoints from several locations on the website
   - If not available, no value was returned
5) Process the data via the input processor of the `ItemLoader`
6) Store data in the `ItemLoader` until the loop has finished
7) Output (yield) the data from the `ItemLoader` into the "data" directory as `.csv` file with the download date in the name

Before explaining how the main spider (`property_scraper`) was built, we provide a brief explanation of how the URL of every single property was obtained without getting IP-blocked.

1) Obtaining the List of URLs: Web scraping turned out to be a mix of close investigation of HTML source codes, creative problem solving and a lot of trial and error. Obtaining all the property URLs was a clear example of this.

Although on a smaller scale, it also required a combination of `Scrapy` and `Selenium` in order to trigger the JavaScriptss.

When a user runs a search on Comparis (e.g. for a house for purchase in a specific location), the website will filter through **all** the listings and return the corresponding properties. If the user triggers the JavaScript by scrolling, Comparis will load a JSON containing the unique ID's of **all** search results at the bottom of their HTML. This can be seen when right clicking on the page and selecting "Inspect" and entering `__NEXT_DATA__` in the search field.

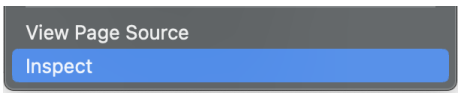


Fig. 3: Inspect Elements on Browser

From there on, `Selenium` will pass the "current state" (i.e. state of HTML after JavaScript manipulation) to the spider for the analysis. The spider then yields the ID's and creates URLs by adding them to the end of the path component of the URL (which is how Comparis refers to their detail pages). The data is subsequently saved as `.csv` to the "data" folder under the name `property_codes_YYYYMMDD.csv`. Adding the current date to the name is necessary with regards to the possibility of creating a database that scrapes the web regularly for available real estate.

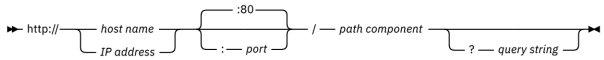


Fig. 4: Structure of a URL [4]

2) Creating the "property-scraper" Spider: Whilst the idea of combining two different web scraping services into one project came up early on, a first instance of a working example was presented in an article on towardsdatascience.com [10]. However, the given example had to be heavily adapted, especially since in our project, using the `ItemLoader` was of high importance. The `ItemLoader` allows for efficient

input- and output processing of data obtained from scraping HTML [3]. The goal was that the data required a minimum of cleaning (i.e. no extraction of text, no deletion of HTML tags outside of the spider). The following lines will explain how this goal was reached.

Given that the spider is a class, the user needs to define its methods. The `def __init__()` method is not required, since the `scrapy.Spider` inherits it already from its parent class. Following the outline given in section IV-B, a browser window using `Selenium` is opened. Using a `for` loop, the URLs in `property_codes.csv` are passed to `Selenium`, which navigates to every single website and scrolls to the bottom of the page to, again, activate the JavaScripts. From there on, the source code is passed to the `ItemLoader` which processes the 22 required fields. The data is yielded all at once and put into a `.csv` file (again along with the current date).

3) Implementation of the ItemLoader: An intermediary step which has been overlooked thus far is the creation of the `ItemLoader` within the `items.py` file. While writing the `property-scraper` spider, it was necessary to identify the so-called fields within the HTML that needed to be scraped. As is customary in programming, there are many ways to get the same result, and the same goes for selecting fields within an HTML.

The two main ways to identify an HTML element are known as the CSS and XPATH selectors (for more information regarding HTML selectors, please refer to this blog post).

For this project XPATH's were selected mainly because of ease of use. The compact syntax offers a quick way to find data within the HTML and was thus a decisive factor. After gathering all the required XPATH's (which can be verified individually using "Inspect" on the website), they were added to the `items.py` file.

In `items.py` every HTML field that needs to be scraped is of class `scrapy.Item` and is assigned to a variable that processes the input and output.

```
class ComparisWebscraperItem(scrapy.Item):
    """
    Defines how the input from the scraped fields is processed.

    Also defines how the output is processed.
    """
    address = scrapy.Field(
        input_processor = MapCompose(remove_tags,
                                     remove_string,
                                     remove_comma),
        output_processor = Join())
```

Fig. 5: Defining an Item

As a consequence the `ItemLoader` is created. `Scrapy` provides the user with six different input processors, including `MapCompose()` which was used extensively. These work like normal functions [12]. If need be, custom functions can be added, which was also done extensively in this project.

For illustration purposes we provide figure 5. There the definition of a field in the `ItemLoader` can be seen in action.

When scraping an HTML field, there are two broad scenarios:

1) The field is available on every website's HTML
   - This is the case with fields that contain price, square meters or address for example
2) The field is only available sometimes
   - ...and is of categorical nature (e.g. *"Does the property have a washing machine (Yes/No)?"*)
   - ...has content that needs to be scraped (e.g. the construction year, if present, will be scraped)

When scraping the field "address" (scenario 1 applies) several input- and output processors are defined (see figure 5). The input processor is primarily `MapCompose()` and `remove_tags`. The latter is technically a function and is imported from the `w3lib.html` library. In figure 5 `remove_string` and `remove_comma` can also be seen. They are instances of custom functions that were defined manually in the `items.py` file during the process of scraping. As with the `remove_tags` they can then be passed to the `MapCompose()` processor. This worked very well for fields which were always present (i.e. price, address, square meters etc.).

However, how does one deal with fields that are not always present (e.g. fields that state whether the apartment is equipped with a washing machine or not)? This seemingly complicated problem can be resolved quickly with three lines of code:

```
def parse_equipment(equipment):
    """
    If the string is found in the html, then 'True' will be inserted by the itemloader,
    otherwise nothing will be inserted.

    The variables are either true or false, which makes this an effective method.
    """
    return 'None' if equipment == None else True
```

Fig. 6: Function for fields that are not always present

Here the properties of both the XPATH and the functions are leveraged. The XPATH of a given field, for example the one for the "elevator" field, will only return a result when it finds the keyword "Ascenseur" within a specific range of the HTML. Given this property of the XPATH and given that the keywords are always the same, it is natural to write a function that can evaluate this binary problem. The function in figure 6 returns `True` when the keyword is found, and an empty value (`N/A`) otherwise.

By importing the class `ComparisWebscraperItem` into the `comparis_scraper.py` file, the `ItemLoader` could be put to use.

C. Implementation of the GUI

1) Organizing the scraped data: The first step to building the GUI is organizing the data that we obtain from the web scraping. For this step we use the same progam than in our parallele data analysis project, named `cleaning_database_GUI.py` . We first analyze the two separate datasets that we obtained from the

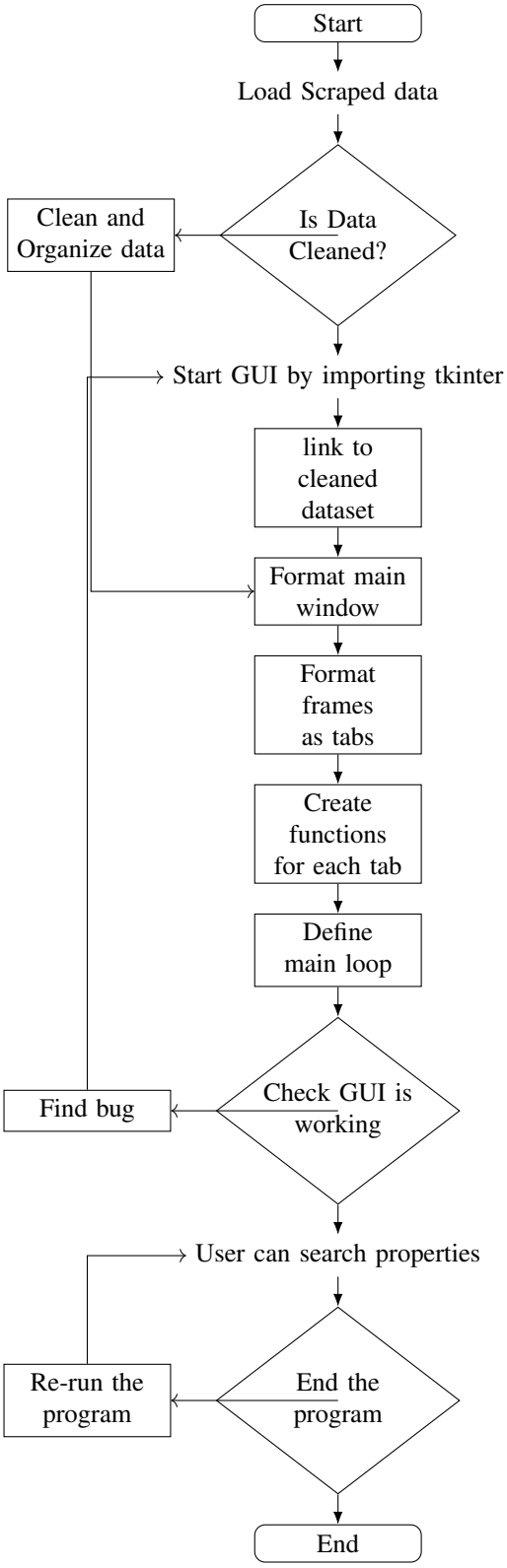
scraping. These are named `property_codes.csv` and `property_details.csv` . We analyze them by printing the headings and the description to check the data and the types. We then merge them. After this, we sort through the values, removing all the missing variables and errors. We create dummy variables for the variables that might be interesting for graphs or regression analysis. We then proceed to split the address from the zip code, to get the zip code in a separate column. We do this as we do not have precise addresses and we want to be able to search properties within a certain zip code. Finally, we drop the variables that are not needed and save the sorted data into the excel file `/GUI/data.xlsx` . This will be the dataset we use for our GUI program.

We also create a short program named `graphs_GUI/.py` to prepare the data for the graphs we will want to display on the interface. We use the excel file we have just created named `data.xlsx` . We first drop all the variables we do not want to keep. We then group the remaining variables, for a given category, such zip code and number of rooms, by average price. We save these grouped datasets into two separate excel files, namely `MeanPriceRooms.xlsx` and `PricebyZip.xlsx` in the GUI folder. These last datasets will only be used for the graphs.

2) Building the GUI: To build the GUI we created a new program named `main5.py` . We first defined what we wanted our GUI to do. We decided, we wanted the user to be able to search for a property within a certain price range, or with a specific number of rooms or within a specific zip code. We also wanted to display graphs for general information. Thefore to build this GUI we needed to have different tabs to search for these separately. We also needed to be able to clear the search to be able to run the program multiple times.

To start the program that will create the interface, we start by defining the characteristics of the root widget otherwise known the main window, such as height, width and title. We then add the notebook instance to add tabs to the interface. In `tkinter` , these are knows as frame widgets. From there, we format the frames and add the labels, entries and buttons we wish to have in each tab. For the formatting we chose to use the grid method. This defines the place of each item, also called widget, within the frame based on colums and rows. The grid manager automatically adapts the size of grid based on the number of widgets inside the frame. We then add the scrollbar to the root window to be able to scroll through the returned properties if the list is long. We also add a treeview instance so that the pulled data from the dataset is displayed in a hierarchical and tabular structure. Finally we define the functions. We use three main functions, one to search through the database and retrieve the data to the interface. The second function to clear the search and delete the displayed data. The last function creates and displays the graphs. Each function is adapted to the chosen variable, such as price, number of rooms or zipcode, to display the correct information

This is a very basic GUI to make it easier for the user to interact with the scraped data. The aim would be to develop the interface to handle searches that are more complex, such as conditional searches with more than one characteristic. More graphs could also be displayed to give diverse information to the user.



V. Maintenance and Upkeep

The project is based on data that evolves daily. Therefore running all the project's programs daily is recommended to have the latest data and the corresponding GUI. Moreover, the time stamps automatically ensure a chronological order and could make multivariate time series analysis possible. This will give a clearer overview of the housing market.

When re-running the web scraper, the framework should be maintained identically to ensure all file names and variable names stay the same. This will ensure the accuracy of the algorithms which are based on the scraped datasets and hence, an accurate outcome for the GUI.

The codebase for this project can be found on its Github public repository. The repository contains a detailed `readme.md` file explaining the objective, the structure of the project and the packages that are needed to set up the project. All the datasets and programs can be found with detailed comments and explanations.

Special attention should be brought to the numerous packages that are needed to carry out the project. These packages should be updated regularly. The instances and methods used should also be modified in accordance to the packages to avoid paths being deprecated. The website which we base our scraping on should also be checked for any updates or changes in its framework, as this could potentially block the scraping program. Hence the whole project would be affected.

Finally, as the database grows the GUI might have to be modified. The `tkinter` package has difficulty handling large datasets. Therefore, a migration to a more powerful GUI might be necessary.

A. Web Scraper

The efficacy of the web scraper stands and falls with the structure of the website. This iteration of the comparis webscraper is adapted only to the real estate section of the website. This means the data in any location within any radius can be scraped with this web scraper. Should there be an additional datapoint of interest, it would need to be added in three steps:

1) Identifying the field
   - Is it available on every page?
   - If yes, the implementation will
2) By using "Inspect" (see figure 3) and the search bar in the HTML, write an XPATH pointing to the required field.
3) Add the field to the `items.py` file and thus
   a) Define the field name
   b) Define the input processor
   c) Define the output processor

B. GUI

C. Scalability

In its current state, the scraper can download real estate data from any village, town, region or canton in Switzerland, as long as the provided URL is from Comparis. Currently the code has to be activated manually. During the runtime of the program there is no user input required whatsoever. The data is stored automatically and the browser window is closed autonomously.

D. Where to Go From Here?

VI. Results

The implementation of all the programs contained in the project result in obtaining the data from the web scrapper, then transforming it to build the GUI. This will help the user to get information on the lausanne housing market, see price listings, historical listings and navigate easily between available properties.

A. Dataset

The dataset we used for the building the GUI is reduced compared to all the data we initially have access to. We chose to keep this reduced version of the dataset, to keep the GUI easy to use and read for the user. Figure 7 and 8 show the the first few rows of the two cvs files: `property_codes.csv` and `property_details.csv` returned from the webscrapping. Figure 9 shows the data we used to feed the GUI. Figure 10 shows the data we used for the average price by zip code graph. Figure 11 shows the data we used for the average price by number of rooms graph.

	Property_Codes	url
0	27884987	https://fr.comparis.ch/immobilien/marktplatz/d...
1	27884155	https://fr.comparis.ch/immobilien/marktplatz/d...
2	27883605	https://fr.comparis.ch/immobilien/marktplatz/d...
3	27883164	https://fr.comparis.ch/immobilien/marktplatz/d...
4	27884625	https://fr.comparis.ch/immobilien/marktplatz/d...
5	27882768	https://fr.comparis.ch/immobilien/marktplatz/d...
6	27881358	https://fr.comparis.ch/immobilien/marktplatz/d...
7	27880295	https://fr.comparis.ch/immobilien/marktplatz/d...
8	27879577	https://fr.comparis.ch/immobilien/marktplatz/d...

Fig. 7: First rows of `property_codes.csv`

	address	available_from	balcony	closest_shop	contruction_year	dishwasher	...	rooms	secon_school	sq_meters	tv	type	wash_m
0	1012 Lausanne	À convenir	NaN	NaN	1933	NaN	...	8.5	NaN	337	NaN	Villa	NaN
1	1004 Lausanne	Immédiate	NaN	NaN	Non disponible	NaN	...	Non disponible	NaN	Non disponible	NaN	Immeuble	NaN
2	La Conversion 1093 La Conversion	01.07.2022	NaN	403.0	2001	NaN	...	6	512.0	180	NaN	Maison mitoyenne	NaN
3	1026 Echandens	À convenir	NaN	NaN	Non disponible	True	...	6	NaN	210	True	Maison jumelée	NaN
4	1066 Epalinges	À convenir	NaN	NaN	1972	NaN	...	7.5	NaN	604	NaN	Maison	NaN
5	Rue du Centre 19 1023 Crissier	À convenir	True	NaN	1850	NaN	...	7.5	NaN	250	NaN	Ferme	NaN
6	1095 Lutry	À convenir	NaN	NaN	2009	NaN	...	3	NaN	163	NaN	Appartement	NaN
7	1026 Echandens-Denges	À convenir	NaN	NaN	Non disponible	NaN	...	Non disponible	NaN	188	NaN	Maison jumelée	NaN
8	1023 Crissier	À convenir	True	NaN	1850	True	...	7.5	NaN	Non disponible	NaN	Autre	NaN
9	1053 Cugy VD	À convenir	NaN	116.0	1987	NaN	...	2	NaN	48	NaN	Appartement	NaN

Fig. 8: First rows of `property_details.csv`

	price	rooms	sq_meters	construction_year	property_type	url	zip_code
0	8.5	337	1933	Villa	https://fr.comparis.ch/immobilien/marktplatz/d...	1012	
0	0.0	0	0	Immeuble	https://fr.comparis.ch/immobilien/marktplatz/d...	1004	
1790000	6.0	180	2001	Maison mitoyenne	https://fr.comparis.ch/immobilien/marktplatz/d...	1093	
1950000	6.0	210	0	Maison jumelée	https://fr.comparis.ch/immobilien/marktplatz/d...	1026	
0	7.5	604	1972	Maison	https://fr.comparis.ch/immobilien/marktplatz/d...	1066	
0	7.5	250	1850	Ferme	https://fr.comparis.ch/immobilien/marktplatz/d...	1023	
2250000	3.0	163	2009	Appartement	https://fr.comparis.ch/immobilien/marktplatz/d...	1095	
1990000	0.0	188	0	Maison jumelée	https://fr.comparis.ch/immobilien/marktplatz/d...	1026	
0	7.5	0	1850	Autre	https://fr.comparis.ch/immobilien/marktplatz/d...	1023	
450000	2.0	48	1987	Appartement	https://fr.comparis.ch/immobilien/marktplatz/d...	1053	

Fig. 9: First rows of `data.xlsx`

B. GUI

The GUI presents the scrapped data in an orderly manner.

1) Main Window: The main window contains the four available tabs namely, price range, Rooms, Zip Code and Graphs and the empty treeview. (see 12)

zip_code	price
1000	2.068504e+06
1003	9.634583e+05
1004	9.795278e+05
1005	1.678750e+06
1006	2.281701e+06
1007	1.222663e+06
1008	1.773321e+06
1009	2.083571e+06
1010	1.163966e+06
1012	2.206444e+06

Fig. 10: First rows of data used for the average price by zip code graph

rooms	price
0.0	1.500108e+06
1.0	2.940528e+05
1.5	4.541636e+05
2.0	5.551256e+05
2.5	7.196087e+05
3.0	1.043000e+06
3.5	8.988854e+05
4.0	1.056538e+06
4.5	1.492171e+06
5.0	1.946793e+06

Fig. 11: First rows of data used for the average price by number of rooms graph

2) Frame 1: The first frame allows the user to search within a specific price range. The user enters a minimum value and maximum value within each of the boxes and presses the search button. The treeview returns all the properties within the chosen price range. (see 13) To restart the process, the user can either press the clear button or enter new values.

3) Frame 2: The second frame allows the user to search within a specific number of rooms. The user enters a minimum value and maximum value within each of the boxes and presses the search button. The treeview returns all the properties within the chosen number of rooms. (see 14) To restart the process, the user can either press the clear button or enter new values.

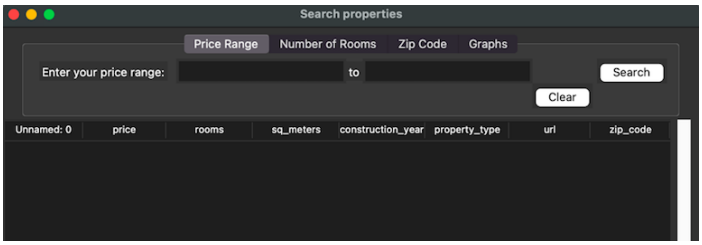


Fig. 12: View of the main window

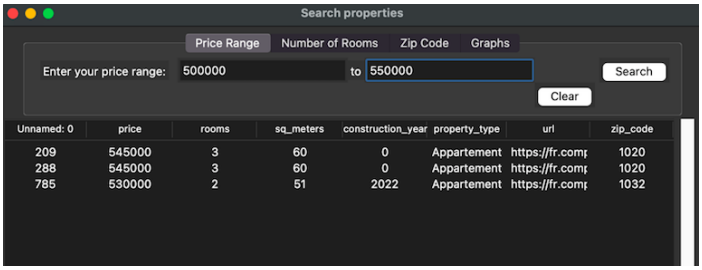


Fig. 13: View of the first frame

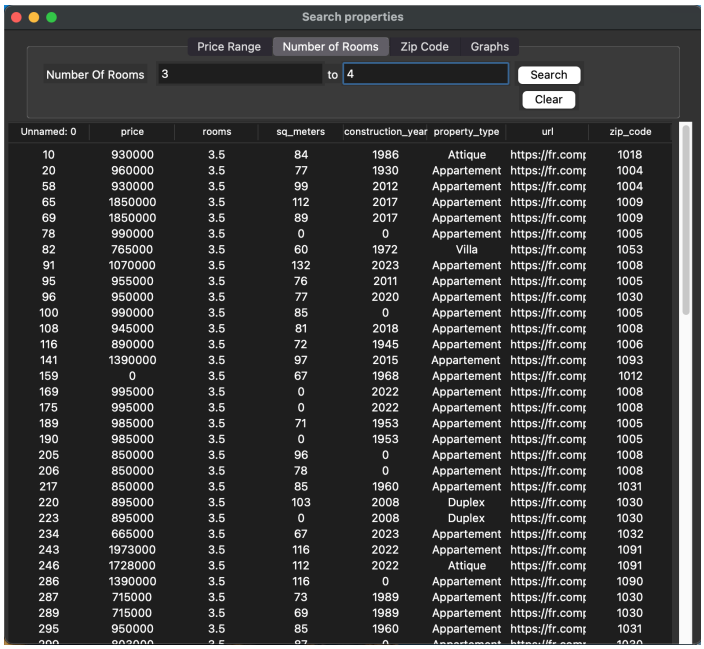


Fig. 14: View of the second frame

4) Frame 3: The third frame allows the user to search for properties in a specific zip code. The user enters the wished zip code in the box and presses the search button. The treeview returns all the properties within the chosen zip code.(see 15) To restart the process, the user can either press the clear button or enter a new value.

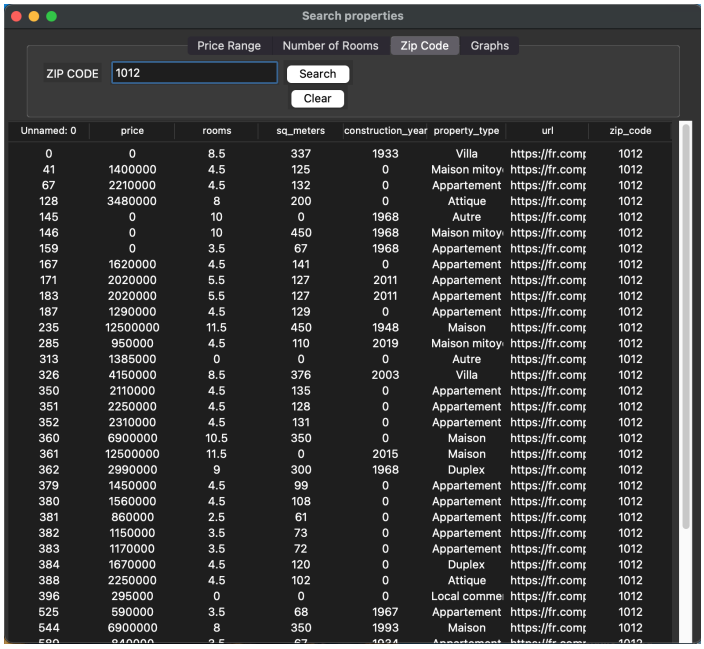


Fig. 15: View of the third frame

5) Frame 4: The fourth frame displays two buttons corresponding to two different bar graphs, namely the average price by zip code and the average price by number of rooms. The user must simply press on the corresponding button and the graph is displayed. The figure 16 displays the average price per zip code in millions of CHF. At the time when we run the program we can see that the zip code 1094 correspnding to Belmont sur Lausanne has the highest average price, namely of 3.7 million CHF. In general, the zip codes corresponding to the outskirts of Lausanne have a higher average price. This could be due to those areas being less built up, hence the properties there tend to be houses and villas rather than apartments. has a much higher price.

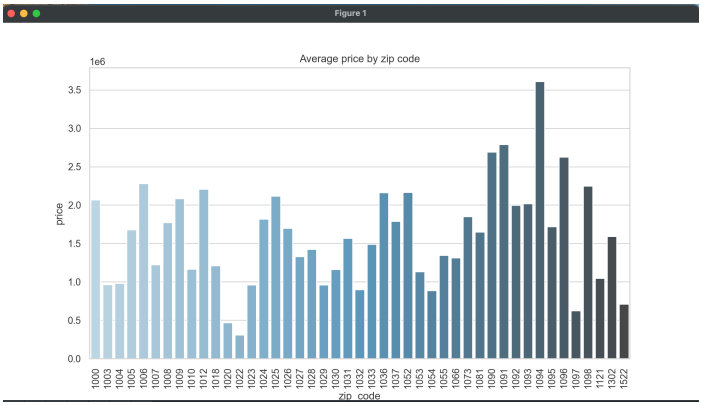


Fig. 16: Average price by zip code

The figure 17 displays the average price per number of rooms by a base of 10 million CHF . At the time when we run the program we can see that the price per number of rooms does not grow exponentially as the number of rooms increases. This could be due to the dataset containing many types of properties such as: commercial properties, apartments, houses and hotels, which messes with overall average. Moreover a lack of data can also explain the lack of exponential growth as the number of rooms increases.

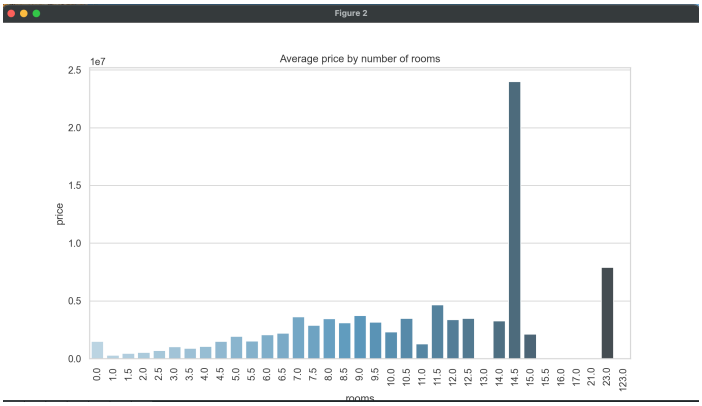


Fig. 17: Average price by zip code

6) Heatmap: Iniatially we wanted to display a heatmap of Lausanne based on the average price by zip code. This map is built with the help of the geopandas and folium packages. To build it we merge the MeanPriceZip file with the geolocations of the Lausanne's zip codes. Nonetheless, due to a merging and geolocation issue we were unable to display the map.

C. Limitations

1) Legality: "Data Scraping", "Data Crawling" or "Data Mining" are classified as "Data Harvesting" in regards to the law. These practices are subject to the European Union General Data Protection Regulation ("EU GDPR") and to the Switzerland Federal Act on Data Protection ("FADP") [6]. As we're currently not established in the EU and do not have affairs with the EU, EU GDPR is not applicable, but will need to be considered should our platform grows internationally. The FADP allows Data Harvesting for research and statistics purposes but does not mention about the legality of using such data, should we release it to the public free of charge. Moreover, we would maybe need to request permission from Comparis.ch, as their copyright states "[...] the user will refrain from copying, publishing or otherwise reproducing accessible data in any form, including the Internet". Our project would need to seek legal advice before rendering such platform open source.

2) Scalability: While we concentrated our research and prototype only in collecting data in Lausanne, our platform is theoretically scalable to the size of Switzerland. However, we've found that it took 80 minutes to collect only 800 data points (10 data points per minute). If we'd want to pull the whole dataset of all listed properties in Switzerland, we'd have to collect around 37'000 data points per day, which would take 3'700 minutes per day (around 62 hours a day), which not possible. To improve in scalability, we might need to either find ways to make our code more efficient, test whether more processing power or a better internet connection would improve scrapping time or use another programming language (compiled language such as C++ or COBOL).

VII. Conclusion

Despite Switzerland being one of the leading countries in term of Real Estate Transparency, we have identified imbalances of information between players in the Swiss housing market. This information asymmetries, where sellers and Agents may possess more information than buyers, contributing to skewing the price of houses. 2001 Nobel Prize winners "for their analyses of markets with asymmetric information", G. Akerlof [5], M. Spence [13] and J. Stiglitz [14] have warned that such asymmetries may lead to adverse selection and cause entire markets to collapse. Furthermore, studies by J.N Gordon have found that there is a positive correlation between real estate transparency and attractive business environments. While we believe that Switzerland's housing market is in a "healthy" shape with sustainable growth since 1999 (Swiss National Bank), we believe that improvement was possible, as well as necessary. Therefore, we aimed to reduce this information asymmetry by providing an open-source, free and easily accessible platform for all economic

actors to use. The platform may be used by buyers to better understand the average price of listings per location (NPAs), size (Sqm), number of rooms, construction year and amenities. This platform may be also used by other economic actors, such as sellers, actuaries, Agents, and researchers. Our research led to constructing the prototype of the platform, where despite many hurdles, proved sucessful. Our methodology consisted of first building a web scrapper, which collected the necessary data and organized it, followed by translating the data into a user-friendly GUI. Though the prototype only collected data from Lausanne, it is scalable to the size of the whole country. However, some adjustments, such as increasing computing power, faster internet access or changing programming language is necessary before scaling the platform country-wide, due to the long time needed to scrape the data. Finally, while our goal is to publish our platform as an open-source software for all to use, we would need to verify that our platform and our methodology is compliant to local laws, copyright of Comparis.ch, as well as EU GDPR.

REFERENCES

[1] *Global Real Estate Transparency Index 2020*. https://www.jll.de/en/trends-and-insights/research/global-real-estate-transparency-index.

[2] *Real Estate Transparency — Zell/Lurie Real Estate Center*.

[3] *Scrapy Tutorial — Scrapy 2.6.1 documentation*. https://docs.scrapy.org/en/latest/intro/tutorial.html.

[4] *IBM Documentation*. https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/cics-ts/5.2?topic=concepts-components-url, Mar. 2021.

[5] G. A. AKERLOF, *The Market for "Lemons": Quality Uncertainty and the Market Mechanism*, The Quarterly Journal of Economics, 84 (1970), pp. 488–500.

[6] V. CONRAD, *Web data collection by Swiss actors in a data protection perspective*, Jusletter IT, (2019).

[7] S. E, *Web Scraping For Beginners Beautifulsoup,Scrapy,Selenium & Twitter API*. https://towardsdatascience.com/web-scraping-for-beginners-beautifulsoup-scrapy-selenium-twitter-api-f5a6d0589ea6, May 2019.

[8] JOHN WATSON ROONEY, *Always Check for the Hidden API when Web Scraping*, Aug. 2021.

[9] S. D. LEVITT AND C. SYVERSON, *Market Distortions when Agents are Better Informed: The Value of Information in Real Estate Transactions*, Working Paper 11053, National Bureau of Economic Research, Jan. 2005.

[10] A. REUSOVA, *Web Scraping: A Less Brief Overview of Scrapy and Selenium, Part II*. https://towardsdatascience.com/web-scraping-a-less-brief-overview-of-scrapy-and-selenium-part-ii-3ad290ce7ba1, Apr. 2019.

[11] R. C. RUTHERFORD, T. M. SPRINGER, AND A. YAVAS, *Conflicts between principals and agents: Evidence from residential brokerage*, Journal of Financial Economics, 76 (2005), pp. 627–665.

[12] A. S, *Demystifying Scrapy Item Loaders*. https://towardsdatascience.com/demystifying-scrapy-item-loaders-ffbc119d592a, July 2020.

[13] M. SPENCE, *Informational Aspects of Market Structure: An Introduction*, The Quarterly Journal of Economics, 90 (1976), pp. 591–597.

[14] J. E. STIGLITZ AND A. WEISS, *Asymmetric Information in Credit Markets and Its Implications for Macro-Economics*, Oxford Economic Papers, 44 (1992), pp. 694–724.