# Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used


1st Given Name Surname
*dept. name of organization (of Aff.)*
*name of organization (of Aff.)*
City, Country
email address

2nd Given Name Surname
*dept. name of organization (of Aff.)*
*name of organization (of Aff.)*
City, Country
email address

3rd Given Name Surname
*dept. name of organization (of Aff.)*
*name of organization (of Aff.)*
City, Country
email address



***Abstract*—Last thing that will be written. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Mauris cursus mattis molestie a iaculis at erat pellentesque adipiscing. Vitae auctor eu augue ut lectus arcu bibendum. Dui id ornare arcu odio ut sem. Tellus at urna condimentum mattis. Sed euismod nisi porta lorem mollis aliquam. Orci eu lobortis elementum nibh tellus molestie. Posuere ac ut consequat semper viverra nam libero.**



***Index Terms*—swiss real estate, webscraping, user interface, transparency, economic research**


## I. Introduction

Transparency in the real estate market has long been associated with favorable business environments. While Switzerland is ranked 11th in the JLL "Global Real Estate Transparency Index", there are only a few platforms that may allow end-market investors to compare prices and assess the worth of a property. 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 inexperienced buyers to assess property value.

Therefore, buyers must rely on Agents, who have better knowledge of the housing market. Rutherford et al. (2005) show that Agents sell their own houses for a premium of approximately $4.5\%$, whereas Levitt and Syverson (2008) 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 4). Whilst this is already a solid foundation, there is no possibility to see the average cost per square meter in the 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 middlemen 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. These 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 (see 4), 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. The source code returned by `requests` for JavaScript heavy websites generally offers very limited insights. A common workaround is the scraping the "hidden API" cite. 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, scrapy 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 very 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. `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. However, even when using all the request headers that a normal browser sends to a website, the `Scrapy` spider's were constantly IP-blocked.

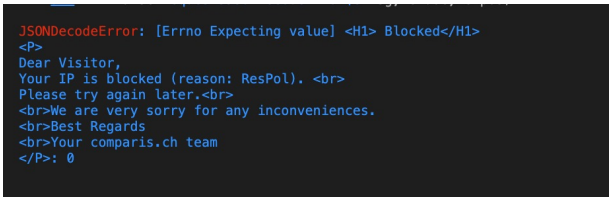



Fig. 2. IP block when running the spiders

### 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 scrapped 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 or having to go through the full scrapped 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.

### F. Structure

## 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
│   ├── cleaning_database.py
│   ├── zoes_version_1.py
│   ├── zoes_version_2.py
│   ├── zoes_version_3.py
│   └── zoes_version_4.py
├── 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 was built, 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 JavaScript's.

When a user runs a search on Comparis (e.g. 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.

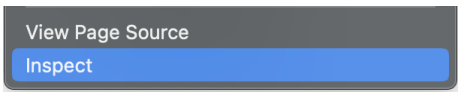



Fig. 3. Inspect Elements on Browser

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.

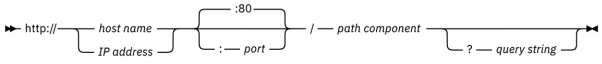



Fig. 4. Structure of a URL [2]

*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 [3] However, the given example had to be heavily adapted, especially since in this project, using the `ItemLoader` was of high importance. The `ItemLoader` allows for efficient input- and output processing of data obtained from scraping

HTML [1]. 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. Following the outline given in 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 JavaScript's. 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` in the `items.py` file. While writing the `property-scraper` spider, it was necessary to identify the fields 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 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. After gathering all the required XPATH's (they can be verified individually using "Inspect" on the website)

In `items.py` every field that is scraped is assigned to a variable which processes the input and output. As a consequence the `ItemLoader` is created. Whilst `Scrapy` provides the user with six different input processors such as `TakeFirst()` or `MapCompose()` . These work like normal functions [**?**] and custom functions can be added, which is exactly what was also done in this project. This was done on a trial and error basis and ensured that the scraped data came out as clean as possible.

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

### C. Implementation of the GUI

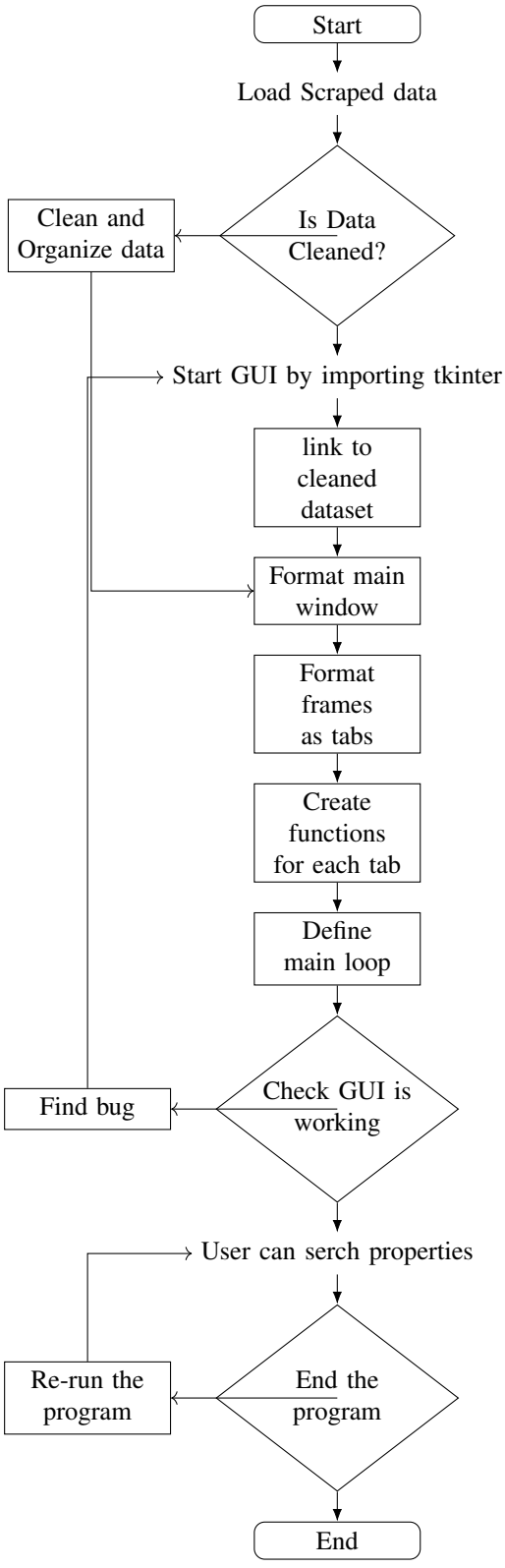
*1) Organizing the scraped data:* The first step to building the GUI is organizing the data that we obtain from the web scraping. We first analyze the dataset and then clean it for the GUI to function correctly. This is done in the program `cleaning_database.py` which we also use to do the data analysis in our parallel project.

In this cleaning program, we first load the dataset. Then we print the headings and the description to check the data and the types. We remove variables that are misspelled, incorrect or missing. 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 want to be able to search properties within a certain zip code. Finally we save the dataset. This dataset is the one that the GUI will use to return information to the user.

*2) Building the GUI:* To build the GUI we created a new program named main4.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. 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.

In the program 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. These are known as frames in tkinter. From there, we format the frames and add the labels and entries we wish to have in each tab. 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 hierarchical and tabular structure. Finally we define the fuctions that retrieve the data from the database to the interface. We also add a function to clear the searched information and start anew.

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 searches that are more complex, such as conditional searches with more than one characteristic.




*D. Maps?*

## V. Maintenance and Update

*A. Web Scraper*

*B. Dataset*

*C. GUI*

*D. Heatmap*

## VI. Results

*A. Subsection*

## VII. Conclusion

*A. Subsection*

## References

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

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

[3] 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.