Article

Price discrimination in the online airline market: an empirical study - Additional file 1

Version September 9, 2021 submitted to J. Theor. Appl. Electron. Commer. Res.

Tools and software architecture

In this document the external software tool Selenium that is used for the pricing data acquisition is described. Additionally, more details about the software architecture and its development are illustrated.

Selenium

When a website is accessed, the content shown can be generated either statically or dynamically. On a static website, the elements that are being displayed do not depend on the user interaction. No matter who visits the website, the content is always the same. On the contrary, a dynamic website offers the possibility of user interaction. Hence, the displayed content depends on how the user interacts with the website. This is achieved by executing scripts on the server-side using scripting languages such as Javascript and PHP. Given that every user can have different requests, most airline websites rely on dynamic content generation to display flight search results. Standard scraping techniques offered by tools such as BeautifulSoup and Scrapy do not allow dynamic interaction with the websites and are hence inadequate to collect this type of data. A better tool for the intended goal is Selenium. Primarily, it is used for automating web applications for testing purposes, but it can also be used to collect dynamically-generated data by simulating user-interaction on websites. Selenium automates browser interaction by using a webdriver , which is an interface responsible for the communication between Selenium and a browser (see figure 1). The interface provides a variety of functions that allow the remote control and behaviour of the browser.
Furthermore, Selenium offers the possibility to set the identity of the user by providing a user agent

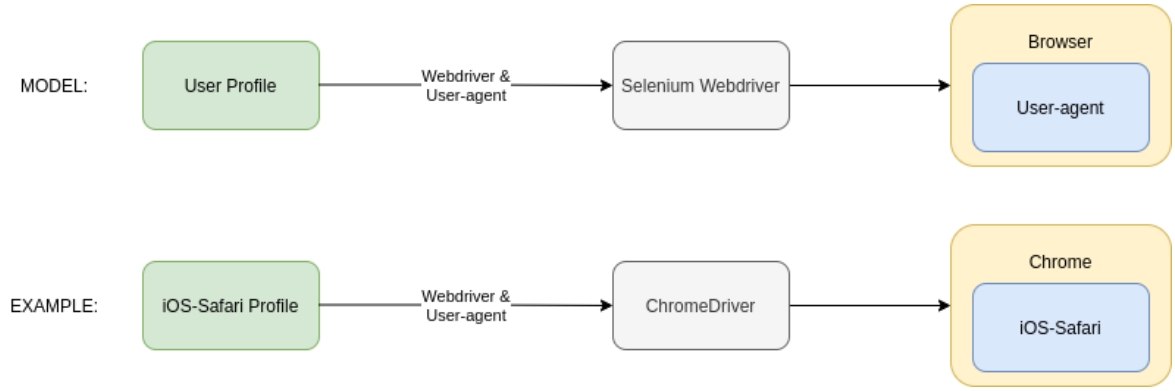

Figure 1. Selenium Webdriver

string. User agent strings include relevant details about the user that makes a request to a website, such as i) the device (i.e. vendor, model and type), ii) the browser (i.e. name and version), and iii) the operating system (i.e. name and version). Every time the Selenium webdriver sends out a request to a website server, it includes the user agent string in the user agent request header. This is then read by

the website to extract all the user details.
For this research, the ChromeDriver has been used to scrape the airline websites. For each user profile, the ChromeDriver has been provided with a specific user agent that reflects the characteristics of the defined users shown in table 1.

USER PROFILE	USER AGENT STRINGS
Windows-Chrome	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Safari/537.36
Android-Chrome	Mozilla/5.0 (Linux; Android 10; M2007J3SG) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Mobile Safari/537.36
macOS-Safari	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.2 Safari/605.1.15
iOS-Safari	Mozilla/5.0 (iPhone; CPU iPhone OS 14_3 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.0.2 Mobile/15E148 Safari/604.1

Table 1. User Agent Strings

Software design

The *template method design pattern* was chosen to design the scraper in order to permit scalability. This is achieved by abstracting the steps of the search flight process in a template class and by deferring their implementation to any derived class. Another advantage of using this design pattern is that any air carrier-agnostic functionality is included in the template class, thus reducing code duplication. In Python 3 the *template method design pattern* is implemented by relying on the Python Standard Library package ABC, which stands for *Abstract Base Class*. When a class inherits from ABC, it can use the method decorator *@abstractmethod*. This decorator forces any subclass to override the method upon which it is applied. The system design is depicted in figure 2.

The modules of the software

The implemented software consists of four main categories of modules, the scraper, the individual air carrier, the configuration, and the support modules, which are described in the following subsections.

The Scraper Class

The *scraper.py* module contains the *Scraper* class, which is abstract and therefore cannot be instantiated. Its constructor is invoked directly by its subclasses. The *_load_configuration* method instantiates the Selenium webdriver, which for this research is the ChromeDriver. The *_load_cookies* method is in charge of loading the cookies from previous sessions into the ChromeDriver. These cookies are handled with the Python Standard Library package *Pickle*. This package allows serializing and de-serializing Python object structures into byte-streams. At the end of each scraping session, the *save_cookies* method is called. The cookies are then serialized and saved into the *cookie_jars* folder using .pkl files. The *cookie_jars* folder contains all the cookies for each user profile and each airline that has been scraped (see figure 3).
Once the setup is complete, the *scrape* method dictates the following steps. At first, the ChromeDriver connects with the airline website. Here, the itinerary parameters are set by calling the *get_availability* method. This method uses the *@abstractmethod* decorator because the exact interaction with the HTML

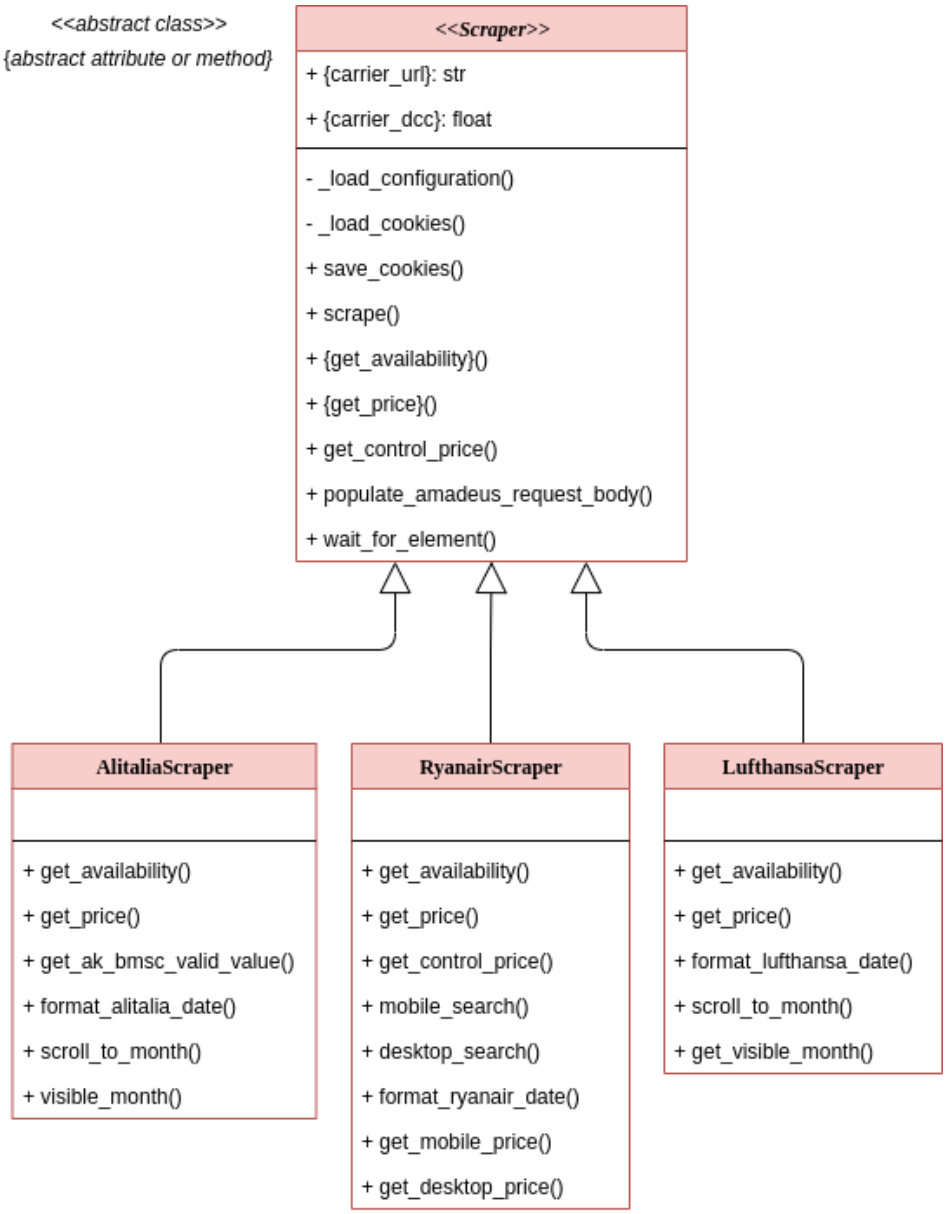

Figure 2. System Design

varies between websites. Therefore, this method is empty and its implementation is deferred to its derived classes. All the same applies to the *get_price* method which is used to select the flight time and offer.

To collect the control data, the *get_control_price* method is called. This method establishes a connection with the Amadeus APIs and collects the required control data. The body of the API request is generated in the *populate_amadeus_request_body* method.

As reported in the manuscript, the Amadeus APIs cannot be used to get the control data for Ryanair. For this reason, the Ryanair scraper overrides the *get_control_price* method with its own implementation, which will be discussed later on.

Once the airline data is scraped and the control data is collected, the *scrape* method calls the *export_to_csv* method. This method is a supporting method contained in the *tools* folder. It relies on the 3rd-party package *Openpyxl* to parse through the collected data and export it into the *raw_data.xlsx* contained in the *output* folder.

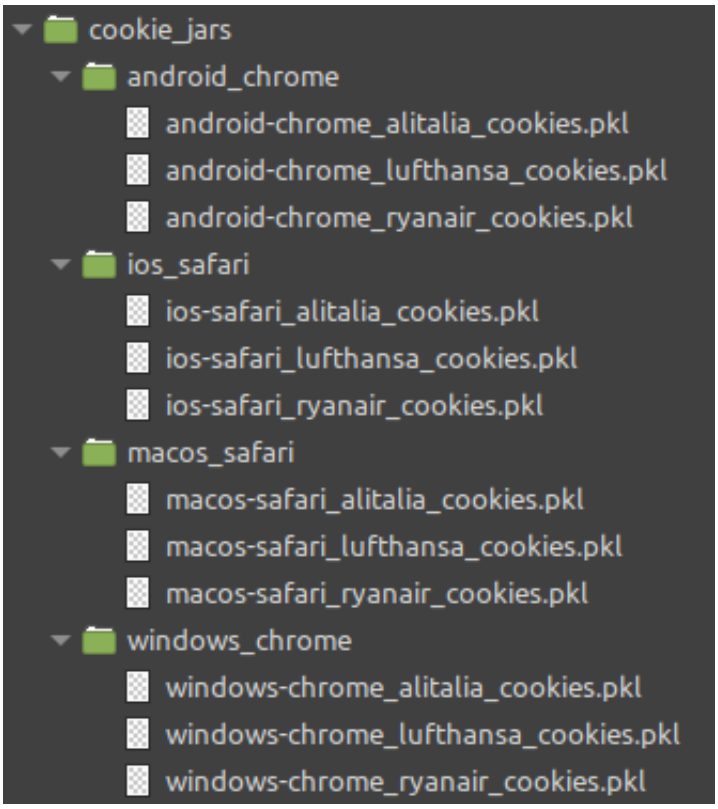

Figure 3. Cookie Jars

The Individual Carrier Classes

The individual carrier classes can be found in the respective module located in the *scrapers* folder. They all inherit from the *Scraper* class and have hence access to all the parent methods and attributes. As already mentioned in section , their main purpose is to define how the ChromeDriver needs to interact with the HTML elements to generate the correct flight offer. This is achieved by overriding the *get_availability* and *get_price* methods. Furthermore, each airline scraper class can define more methods required to adapt to the specifics of the carrier, such as locale or security checks. The three scrapers all have a *carrier_url* class attribute, which is used to store the airline website address.

The AlitaliaScraper Class

The Alitalia website handles desktop and mobile searches in the same way for the most part. The only difference relies on how the departure and return date are passed over: in a desktop request, the date can be typed in, whereas in a mobile search the date must be selected within an interactive widget. Therefore, both desktop and mobile search can be handled in the same *get_availability* method with a simple switch case for the sole date selection. When the desktop flag is triggered, the scraper sends the date in a string format. Instead, when the mobile flag is triggered, the scraper interacts with the widget with the support of the *format_alitalia_date*, *scroll_to_month* and *visible_month* methods. Finally, the *AlitaliaScraper* class introduces the *get_ak_bmsc_valid_value* method. This method is used to get a valid *ak_bmsc* cookie, which is used by Alitalia to grant permission to its website.

The RyanairScraper Class

The Ryanair website handles desktop and mobile searches in two complete separate ways. For this reason the *get_availability* method uses the desktop/mobile flag to make a call to either the *desktop_search* or *mobile_search* methods. Same goes for the *get_price* method which uses the same flag to call either *get_desktop_price* or *get_mobile_price* methods. Due to the Amadeus API limitations presented in the manuscript, the *RyanairScraper* class overrides the *get_control_data* method. To collect control data, the *RyanairScraper* creates a new instance of itself skipping over the cookie loading process. This renders

the control scraper a cookie-less scraper, hence replicating the search that a normal user would perform while in incognito mode.

The LufthansaScraper Class

The Lufthansa website handles desktop and mobile searches in the exact same way. For this reason the *get_availability* and *get_price* methods can easily handle either type of search. The departure and return date must be selected in an interactive widget, therefore as for Alitalia, the scraper relies on three methods, namely the *format_lufthansa_date*, *scroll_to_month* and *get_visible_month* methods.

The Configuration Modules

The configuration modules hold the necessary information for the scraper to execute the flight searches. These are the modules that will need to be modified in the future to extend the research to more air carriers, itineraries and user profiles.

itineraries.py

This module contains the details of the itineraries and air carriers. For each carrier, one or more itineraries can be defined. For the scope of this research, for each air carrier only one itinerary was used. This module condenses all the itinerary details in the variable *ITINERARIES*, which is imported by the entry point module to configure the scrapers. See listing 1 for a compact overview.

user_profiles.py

This module, depicted in listing 2, contains the details of the user profiles. This is where the user agents string, mentioned in section 1, are defined and assigned. Also, the user unique IP address is assigned. This is achieved by associating the profile with a specific VPN server which has a unique IP address registered in Italy. Finally, the path to the cookie jars, reported in section , is provided. This module condenses all the user profile details in the variable *USER_LIST*, which is imported by the entry point module to configure the scrapers.

```
ALITALIA_ITINERARIES = [{'carrier': 'Alitalia',
                         'fare_brand': 'Economy Light',
                         'origin': 'FCO',
                         'destination': 'CTA',
                         'departure_date': '16/07/2021',
                         'departure_time': '17:00',
                         'return_date': '18/07/2021',
                         'return_time': '20:20'}]
RYANAIR_ITINERARIES = [{'carrier': 'Ryanair',
                        'fare_brand': 'Regular',
                        'origin': 'FCO',
                        'destination': 'CTA',
                        'departure_date': '16/07/2021',
                        'departure_time': '17:50',
                        'return_date': '18/07/2021',
                        'return_time': '20:10'}]
LUFTHANSA_ITINERARIES = [{'carrier': 'Lufthansa',
                          'fare_brand': 'Economy Light',
                          'origin': 'FCO',
                          'destination': 'MUC',
                          'departure_date': '23/07/2021',
                          'departure_time': '19:15',
                          'return_date': '25/07/2021',
                          'return_time': '16:55'}]
ITINERARIES = {'Alitalia': ALITALIA_ITINERARIES,
               'Ryanair': RYANAIR_ITINERARIES,
               'Lufthansa': LUFTHANSA_ITINERARIES}
```

Listing 1: itineraries.py

```
WINDOWS_CHROME = {
    'user': 'Windows-Chrome',
    'os': 'Windows 10', 'browser': 'Chrome 87',
    'user_agent': WINDOWS_CHROME_UA,
    'vpn_server': 'it170',
    'ip_address': '185.183.105.28',
    'cookie_jar': os.path.join('cookie_jars', 'windows_chrome') }
ANDROID_CHROME = {
    'user': 'Android-Chrome',
    'os': 'Android 10', 'browser': 'Chrome 87',
    'user_agent': ANDROID_CHROME_UA,
    'vpn_server': 'it175',
    'ip_address': '82.102.21.68',
    'cookie_jar': os.path.join('cookie_jars', 'android_chrome') }
MACOS_SAFARI = {
    'user': 'MacOS-Safari',
    'os': 'Mac OS 10.15', 'browser': 'Safari 14.0',
    'user_agent': MACOS_SAFARI_UA,
    'vpn_server': 'it180',
    'ip_address': '192.145.127.236',
    'cookie_jar': os.path.join('cookie_jars', 'macos_safari') }
IOS_SAFARI = {
    'user': 'iOS-Safari',
    'os': 'iOS 14.3', 'browser': 'Safari 14.0',
    'user_agent': IOS_SAFARI_UA,
    'vpn_server': 'it190',
    'ip_address': '37.120.201.244',
    'cookie_jar': os.path.join('cookie_jars', 'ios_safari') }
USER_LIST = [WINDOWS_CHROME, ANDROID_CHROME, MACOS_SAFARI, IOS_SAFARI]
```

Listing 2: user_profiles.py

The Support Modules

logger_tool.py
The *logger_tool.py* module instantiates the logger. The logger output is saved in the *logbook.log* in the *output* folder. The contents of this file have been used to identify the errors occurred while scraping. How these errors have been handled will be presented later on.

spreadsheet_tool.py
The *spreadsheet_tool.py* module allows to export the collected data into a spreadsheet. It relies on the *Openpyxl* package and it was designed to allow quick adaptation to display the desired data. This is achieved by providing a scraper instance to the *generate_data* function. The data contained in the scraper is mapped to the cell headers, which are represented as keys of an ordered dictionary. This type of implementation gives a simple way to add, remove or modify the spreadsheet headers by modifying the keys and mapping them with the corresponding value. The Lufthansa surcharges

present in Amadeus are here subtracted from the control price retrieved (see line 20 below).

```
{
'os': scraper.user['os'],
'browser': scraper.user['browser'],
'ip_address': os.popen('curl -s ifconfig.me').read(),
'search_date': str(datetime.now().strftime("%m-%d-%Y")),
'search_time': str(datetime.now().strftime("%H:%M:%S")),
'carrier': scraper.carrier,
'origin': scraper.itinerary['origin'],
'destination': scraper.itinerary['destination'],
'fare_brand': scraper.itinerary['fare_brand'],
'departure_date': to_date(scraper.itinerary['departure_date']),
'departure_time': to_time(scraper.itinerary['departure_time']),
'departure_flight': scraper.itinerary['departure_flight'],
'departure_price': to_float(scraper.itinerary['departure_price']),
'return_date': to_date(scraper.itinerary['return_date']),
'return_time': to_time(scraper.itinerary['return_time']),
'return_flight': scraper.itinerary['return_flight'],
'return_price': to_float(scraper.itinerary['return_price']),
'total_price': to_float(scraper.itinerary['total_price']),
'control_price': to_float(scraper.itinerary['control_price']) - scraper.carrier_dcc,
'dep_fare_basis': scraper.itinerary['dep_fare_basis'],
'dep_control_fare_basis': scraper.itinerary['dep_control_fare_basis'],
'ret_fare_basis': scraper.itinerary['ret_fare_basis'],
'ret_control_fare_basis': scraper.itinerary['ret_control_fare_basis'],
'seats_left': scraper.itinerary['seats_left']
}
```

Listing 3: spreadsheet_tool.py

The Entry Point Module

The *run_scrapers.py* module functions as the entry point for the scraper. At first, the air carrier scraping classes, user profile list and itineraries are imported. Then the module uses a nested iteration to call every combination of carrier and user profile. If any error occurs during one of the iterations, it is recorded in the *logbook* file. The code fragment shown below is a simplified version of the actual scraping algorithm.

```
CARRIER_SCRAPERS = {'Alitalia': AlitaliaScraper,
                    'Ryanair': RyanairScraper,
                    'Lufthansa': LufthansaScraper}

ITINERARIES = {'Alitalia': ALITALIA_ITINERARIES,
               'Ryanair': RYANAIR_ITINERARIES,
               'Lufthansa': LUFTHANSA_ITINERARIES}

USER_LIST = [WINDOWS_CHROME, ANDROID_CHROME, MACOS_SAFARI, IOS_SAFARI]

for carrier, scraper_class in CARRIER_SCRAPERS.items():
    for user in USER_LIST:
        try:
            run(['vpn', 'connect', f'{user["vpn_server"]}'])
            scraper = scraper_class(user=user,
                                    selenium_browser='Chrome',
                                    itinerary=ITINERARIES[carrier])
        except Exception as e:
            logger.error(f'Error: {e}')
```

Listing 4: run_scrapers.py