DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Evaluating tools and techniques for web scraping

EMIL PERSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluating tools and techniques for web scraping

EMIL PERSSON

Master in Computer Science
Date: December 15, 2019
Supervisor: Richard Glassey
Examiner: Örjan Ekeberg
School of Electrical Engineering and Computer Science
Host company: Trustly
Swedish title: Utvärdering av verktyg för webbskrapning

Abstract

The purpose of this thesis is to evaluate state of the art web scraping tools. To support the process, an evaluation framework to compare web scraping tools is developed and utilised, based on previous work and established software comparison metrics. Twelve tools from different programming languages are initially considered. These twelve tools are then reduced to six, based on factors such as similarity and popularity. Nightmare.js, Puppeteer, Selenium, Scrapy, HtmlUnit and rvest are kept and then evaluated. The evaluation framework includes performance, features, reliability and ease of use. Performance is measured in terms of run time, CPU usage and memory usage. The feature evaluation is based on implementing and completing tasks, with each feature in mind. In order to reason about reliability, statistics regarding code quality and GitHub repository statistics are used. The ease of use evaluation considers the installation process, official tutorials and the documentation.

While all tools are useful and viable, results showed that Puppeteer is the most complete tool. It had the best ease of use and feature results, while staying among the top in terms of performance and reliability. If speed is of the essence, HtmlUnit is the fastest. It does however use the most overall resources. Selenium with Java is the slowest and uses the most amount of memory, but is the second best performer in terms of features. Selenium with Python uses the least amount of memory and the second least CPU power. If JavaScript pages are to be accessed, Nightmare.js, Puppeteer, Selenium and HtmlUnit can be used.

Sammanfattning

Syftet med detta examensarbete är att utvärdera moderna vertkyg som ligger i framkant inom webbskrapning. Ett ramverk för att jämföra verktygen kommer att utvecklas och användas och är baserat på tidigare forskning samt etablerade värden som används för att karaktärisera mjukvara. Från början övervägs tolv verktyg från olika programmeringsspråk. Dessa tolv reduceras sedan till sex baserat på deras likheter och populäritet. De sex verktyg som blev kvar för utvärdering är Nightmare.js, Puppeteer, Selenium, Scrapy, HtmlUnit och rvest. Ramverket för utvärdering involverar prestanda, funktioner, pålitlighet och användarvänlighet. Prestandan mäts och utvärderas i körtid, CPU, samt minnes-användning. Funktionaliteten utvärderas genom implementering av olika uppgifter, korresponderande till de olika funktionerna. För att resonera kring pålitlighet används statistik gällande kodkvalitét samt statistik tagen från vertkygens GitHub-repositories. Bedömning av användarvänlighet inkluderar utvärdering av installations-processen, officiella handledningssidor samt dokumentationen.

Samtliga verktyg visade sig användbara, men Puppeteer klassas som det mest kompletta. Det hade de bästa resultaten gällande användarvänlighet samt funktionalitet men höll sig ändå i toppen i både prestanda och tillförlitlighet. HtmlUnit visade sig vara det snabbaste, men använder även mest resurser. Selenium körandes Java är det långsammaste samt använder mest minnesresurser, men är näst bäst när det kommer till funktionalitet. Selenium körandes Python använde minst minne och näst minst CPU kraft. Om sidor som laddas med hjälp av JavaScript är ett krav så fungerar Nightmare.js, Puppeteer, Selenium och HtmlUnit.

Contents

Chapter 1

Introduction

The World Wide Web is constantly growing and is currently the largest data source in the history of mankind [6]. The process of publishing data on the World Wide Web has become easier with time, and thus caused the World Wide Web to grow immensely. Most data published on the World Wide Web is meant to be accessed and consumed by humans, and is thus in an unstructured state. Web scraping is a way for creating automated processes that can interact and make use of the vast amount of World Wide Web data.

There are typically three phases that a web scraping errand goes through. First, data is fetched. This is done by utilising the HTTP protocol. Once the data is fetched, the desired data is extracted. This can be done in different ways, but common ones include XPath, CSS-Selectors and regular expressions. Once the data wanted has been extracted, the transformation phase is entered. Here the data is transformed into whatever structure wanted, for example JSON.

There are many different types of web scraping tools and techniques. For the layman, desktop based web scraping tools may be of preference. These often come with a graphical interface, allowing the user to point-and-click the data to be fetched and usually doesn't require programming skills. A more technical approach is to use libraries, in a similar vein to how software is developed. By combining libraries in the three different phases, a web scraper can be built. Additionally, web scraping frameworks can be used. These take care of each step in the process, removing the need to integrate several libraries. In this thesis, web scraping frameworks are compared.

Web scraping has been utilised in multiple areas, including retailing, property and housing markets, stock market, analysing social media feeds, biomedical and psychology research, and many more.

The purpose of this thesis was to compare web scraping solutions. Four key areas has been investigated. Once the framework for comparison was developed, it was applied to several state of the art tools. This is meant to aid in choosing the right tool for a given project.

The thesis is structured as follows. Chapter 2, the background chapter, explains what web scraping is and its workflow. This chapter also includes previous use cases, challenges with web scraping, different tools and techniques for doing web scraping and previous research. Twelve state of the art web scraping tools are presented, working as the basis of tools to be ran through the framework developed In Chapter 3, the method chapter. First, Chapter 3 presents an initial selection task, meant to reduce the twelve state of the art web scraping tools to an amount more suitable for the thesis time and resource limitations. Factors such as similarity and maintenance was used in the selection process. Once the amount of tools had been reduced, the evaluation framework was developed. The framework is split into four categories: performance, features, reliability and ease of use. Chapter 4 presents the results from running the different web scraping tools through the framework. It also includes a discussion section for each category, to make it easier for the reader to reason about the specific category one at a time. Finally, a conclusion chapter is presented in Chapter 5. This includes recommendations based on the framework results, limitations, a list of contributions and where this type of work can go next.

First, however, the research question is defined.

1.1 Research question

The research question to be answered is

> *With respect to established metrics of software evaluation, what is the best state of the art web scraping tool?*

Chapter 2

Background

In this chapter, relevant web scraping background is presented. The chapter is split up into four sections. First, web scraping as a concept is introduced, along with its workflow. The second section presents a survey of the available state of the art web scraping tools. The third section presents established ways to evaluate software. This includes both traditional, standardised metrics, but also more recent ways used. Finally, the fourth section presents previous work on web scraping and how evaluation on web scraping tools has been done previously, with respect to the metrics presented in the third section.

2.1 Web scraping

The World Wide Web is currently the largest data source in the history of mankind [6] and consists of mostly unstructured data, which can be hard to collect. Extracting the World Wide Webs unstructured data can be done with traditional copy-and-paste, as some websites provide protection against an automated machine accessing the website. However, this is a highly inefficient approach for larger projects [19]. Sometimes websites or web services offer APIs to fetch or interact with the data. However, it is not uncommon that APIs are absent or that the available solutions does not cover the users needs. Using APIs also require some programming skill [20]. If APIs are not available or are insufficient for the task, a technique known as web scraping can be applied. Web scraping, also known as web data extraction, web data scraping, web harvesting or screen scraping [24] can be defined as

> *"A web scraping tool is a technology solution to extract data from web sites in a quick, efficient and automated manner, offering data in a more structured and easier to use format" [6].*

In essence, web scraping is used to fetch unstructured data from web pages and transform it to a structured presentation, or for storage in an external database. It is also considered an efficient technique for collecting big data, where gathering large amounts of data is important [29].

Search engines use web scraping in conjunction with web crawling to index the World Wide Web, with the purpose of making the vast amount of pages searchable. The crawlers, also called spiders, follow every link that they can find and store them in their databases. On every website metadata and site contents are scraped to allow for determining which site best fit the users search terms. One example of a way to "rank" the pages is by an algorithm called PageRank [1]. PageRank looks at how many links are outgoing from a website, and how often the website is linked from elsewhere.

Three different phases build up web scraping:

Fetching phase

First, in what is commonly called the fetching phase, the desired web site that contains the relevant data has to be accessed. This is done via the HTTP protocol, an Internet protocol used to send requests and receive responses from a web server. This is the same techniques used by web browsers to access web page content. Libraries such as `curl` [2] and `wget` [3] can be used in this phase by sending an HTTP GET request to the desired location (URL), getting the HTML document sent back in the response [12].

Extraction phase

Once the HTML document is fetched, the data of interest needs to be extracted. This phase is called the extraction phase, and the technologies used are regular expressions, HTML parsing libraries or XPath queries. XPath stands for XML Path Language and is used to find information in documents [19]. This is considered the second phase.

Transformation phase

Now that only the data of interest is left it can be transformed into a structured version, either for storage or presentation [12]. The process described above can be summarised in Figure 2.1.

[1] `https://www.google.com/intl/ALL/search/howsearchworks/`

[2] `https://curl.haxx.se/`

[3] `https://www.gnu.org/software/wget/`

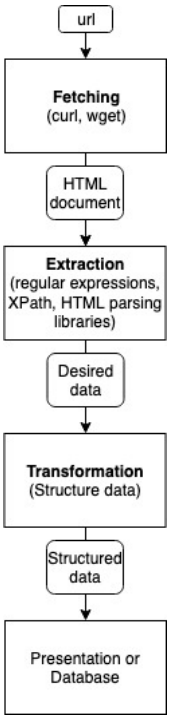


Figure 2.1: Web scraping process

2.1.1 Usage

Web scraping is used in many areas and for several purposes. It is commonly used in online price comparisons where data is gathered from several retailers, to then be presented in once place. One example of this is Shopzilla, a half billion dollar company that is built on web scraping. Shopzilla gathers pricing information from many retailers, offering a way for users to search for products and find alternatives on where to buy the product from [20]. In the Netherlands web scraping was used to compare the manual way of collecting flight ticket prices and web scraped prices, showing that there were commonalities between the two. Similar research on flight tickets has also been conducted in Italy, where the authors compared the time it took between an automated web scraper and manually downloading price quotes for flights in and out of Italy. Time could be saved by using a web scraping approach [23].

Web scraping has also been used for official statistics in the Netherlands. In one example the Dutch property market was examined. During a span of two years, five different housing websites were monitored with about 250.000 observations per week. This data was later used in market statistics. In the

clothing sector web scraping was deployed to monitor 10 different clothing websites. Around 500.000 price observations were gathered each day and then used in the production of Dutch CPI (Consumer price index) [27].

The rental housing market in the US had lacking information sources, especially for large apartment complexes. Web scraping was used to close this gap by analysing Craigslist rental housing data. Exposing the web scraping technique within the housing market was a second purpose of the investigation. 11 million rental posts were collected between May and July 2014 [3].

Web scraping has been used in the stock market to present a visualization of how price changes over time, and Social media feeds has been scraped to investigate public opinions on opinion leaders [29].

When searching for grey literature, web scraping can be used to make for a more efficient way of spanning the searching over multiple different websites. It also increased search activity transparency. Many academic databases that hold information on research papers can often be lacking in specific details, web scraping has been used to solve this issue by gathering that extra information elsewhere [13].

For biomedical research, accessing web resources is of major importance and many different are being used every day. Nowadays APIs are the standard way in which one fetches data from these biomedical web resources. There are still scenarios when web scraping is useful: if the published APIs do not offer the usage of a desired tool or access to wanted data, or if the costs of learning and using the APIs are not justified (if the data source is to only be accessed once, for example) [12].

In Psychology research, web scraping was used to gather data from a public depression discussion board, Healthboards.com. The purpose was to examine correlations between gender and the number of received responses, and the responders gender [16].

Due to cumbersome bureaucratic processes to gather weather data in South Sumatran, a web scraping was investigated. The purpose is to gather data for further statistical analysis. Targeting two websites offering real time weather information. Data were gathered every hour, periodically [9].

2.1.2 Challenges

There are some challenges that arise when doing web scraping. As the Internet is a dynamic source of data, things may change. This includes site structure, the technology used and whether the website follows standards or not. When a website changes it is likely that a built web scraper needs to be altered as

well [27]. Other challenges include unreliable information and ungrammatical language. Even if the information exists and is scrapable, it might not actually be correct. Grammar and spelling can be an issue in the parsing phase, as information might be missed or falsely gathered [19].

While there has been constant evolvement in development of web scraping tools, the legalities concerning web scraping are somewhat unexplored. Some legal frameworks can be applied, a websites terms of use can state that programmatic access should be denied and can lead to a breach of contract if broken. A problem with this however is that the website user needs to explicitly agree to these terms and unless the web scraper does this, the breaching may not be able to lead to a prosecution. If prohibited material is obtained and used can lead to prosecution as well as overloading the website can lead to charges [15].

The ethical side is for the most part ignored. For example, a research project that involves collecting large amounts of data via web scraping might accidentally compromise the privacy of a person. A researcher could match the data collected with another source of data and thus reveal the identity of the person who created the data. Companies and organizations privacy can be unintentionally revealed via trade secrets or other confidential information through web scraping. It can also harm websites that rely on ads for income, since utilizing a web scraper causes ads to be viewed by software and not a human [15].

2.1.3 Tools and techniques

There are several approaches one can take when implementing a web scraper. A common path is to use libraries. Using this approach the web scraper is developed in the same vein as a software program using a programming language of choice. Popular programming languages for building web scrapers include Java, Python, Ruby or JavaScript, in the framework Node.js [6]. Programming languages usually offer libraries to use the HTTP protocol to fetch the HTML from a web page. Popular libraries for using the HTTP protocol include `curl` and `wget`. After this process regular expressions or other libraries can be used to parse the HTML [12].

Using a web scraping framework is an alternative solution. These frameworks usually take care of each step in the web scraping process, removing the need to integrate several libraries. For example, a Python framework called Scrapy allows defining web scrapers as inheriting from a BaseSpider class that provides functions for each step in the process. Some popular libraries for im-

plementing web scrapers include Jsoup, jARVEST, Web-Harvest and Scrapy [12].

If programming skills are absent, desktop-based environments can be used. These allow for creating web scrapers with minimal technical skills, often providing a GUI-based usage with an integrated browser. The user can then navigate to the desired web page and simply point-and-click on data that should be fetched. This process avoids the use of XPath queries or regular expressions for picking out data [12].

Web scraping tools can be split up into two subgroups, partial and complete tools. Partial tools are often focused on scraping one specific element type and can come in the form of a browser extension. It is more light weight and requires less configuration. Complete tools are more powerful and offer services such as a GUI, visual picking of elements to scrape, data caching and storage [6].

XPath

XPath [4], which stands for XML Path Language, is used to access different elements of XML documents. It can be used to navigate HTML document as HTML is an XML-like language and shares the XML structure. XPath is commonly used in web scraping to extract data from HTML documents and uses the same notation used in URLs for navigating through the structure.

Some examples are shown using the file in Listing 2.1, describing a small book store taken from the Microsoft .NET XML Guide [5].

```
<?xml version='1.0'?>
<bookstore xmlns="urn:newbooks-schema">
<book genre="novel" style="hardcover">
       <title>The Handmaid's Tale</title>
       <author>
              <first-name>Margaret</first-name>
              <last-name>Atwood</last-name>
       </author>
       <price>19.95</price>
</book>
<book genre="novel" style="other">
       <title>The Poisonwood Bible</title>
       <author>
              <first-name>Barbara</first-name>
              <last-name>Kingsolver</last-name>
```

[4] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors

[5] https://docs.microsoft.com/en-us/dotnet/standard/data/xml/select-nodes-using-xpath-navigation

```
        </author>
        <price>11.99</price>
</book>
<book genre="novel" style="paperback">
        <title>The Bean Trees</title>
        <author>
                <first-name>Barbara</first-name>
                <last-name>Kingsolver</last-name>
        </author>
        <price>5.99</price>
</book>
</bookstore>
```

Listing 2.1: An XML file describing a book store

Following examples are ran using the command line tool `xpath` on the file shown in Listing 1.

Running the command `//book` will extract all the book elements in the document, as shown in Listing 2.2.

```
<book genre="novel" style="hardcover">
        <title>The Handmaid's Tale</title>
        <author>
                <first-name>Margaret</first-name>
                <last-name>Atwood</last-name>
        </author>
        <price>19.95</price>
</book>

<book genre="novel" style="other">
        <title>The Poisonwood Bible</title>
        <author>
                <first-name>Barbara</first-name>
                <last-name>Kingsolver</last-name>
        </author>
        <price>11.99</price>
</book>

<book genre="novel" style="paperback">
        <title>The Bean Trees</title>
        <author>
                <first-name>Barbara</first-name>
                <last-name>Kingsolver</last-name>
        </author>
        <price>5.99</price>
</book>
```

Listing 2.2: The resulting XML when fetching all book elements

The two // indicates that XPath will look everywhere in the document, i.e it will look for all book elements.

If only the book element with the title The Poisonwood Bible is of interest, it can be extracted by `//book[title = "The Poisonwood Bible"]`. The result is shown in Listing 2.3.

```
<book genre="novel" style="other">
        <title>The Poisonwood Bible</title>
        <author>
                <first-name>Barbara</first-name>
                <last-name>Kingsolver</last-name>
        </author>
        <price>11.99</price>
</book>
```

Listing 2.3: The resulting XML when only fetching books with the title "The Poisonwood Bible"

Once again the book elements need to be reached. Then `[title = "The Poisonwood Bible"]` will look for book elements with the title property set to "The Poisonwood Bible".

If the property lies in the element itself, @ can be used to indicate that. The command `//book[@style = "paperback"]` will extract all paperback books, as shown in Listing 2.4.

```
<book genre="novel" style="paperback">
        <title>The Bean Trees</title>
        <author>
                <first-name>Barbara</first-name>
                <last-name>Kingsolver</last-name>
        </author>
        <price>5.99</price>
</book>
```

Listing 2.4: The resulting XML when fetching paperback books

While the examples shown are relatively basic, there are more complex ways to use XPath for extracting data from XML-like documents.

CSS Selectors

A second popular method of extracting data from HTML documents within web scraping tools is by using CSS Selectors. CSS (Cascading Style Sheets) is the language used to style HTML documents, more generally it describes the rendering of structured documents such as HTML and XML. CSS Selectors

[6] are patterns used to match and extract HTML elements based on their CSS properties.

There are a few different selector syntaxes that correspond to the way CSS is structured. One example for each selector is presented with the html code at the top and then the CSS selector code under it.

The **type selector** is simple, `input` will match any element of type <input>. An example is shown in Listing 2.5.

```
<h4>Hello</h4>

// using the type selector to set the h4
// fields text color to white
h4 {
        color: white;
}
```

Listing 2.5: Setting h4 fields text color using the CSS type selector

Matching CSS classes is done via the **class selector**, by prepending a dot. For example, `.firstclass` will match any element that has a class of firstclass. Listing 2.6 presents an example using the class selector.

```
<div class="fat-bordered">Element of class fat-bordered</div>

// using the class selector to set a border to elements
// of class fat-bordered
.fat-bordered {
        border: 5px solid black;
}
```

Listing 2.6: Setting a border to elements of class fat-bordered using the CSS class selector

Furthermore, if only one element is to be targeted, the **ID selector** is used, as every ID should be unique. This is done by prepending a #, `#text` will match the element with id text. An example using the ID selector is shown in Listing 2.7.

```
<div id="1">I have id 1</div>

// using the ID selector to add a top margin
// to the div with id 1
#1 {
        margin-top: 100px;
```

[6] https://www.w3.org/TR/selectors-4/

```
}
```

Listing 2.7: Setting a top margin to the element with id 1 using the CSS id selector

Attributes with different values can be matched via the **attribute selector** by surrounding the expression in [braces]. `[autoplay=false]` will match every element that not only has the autoplay attribute present, but has it set to false. The attribute selector is shown in Listing 2.8.

```
<input type="number" name="amount" placeholder="How many?" />

// using the attribute selector to set the input
// fields background color to blue
[name="amount"] {
       background-color: blue;
}
```

Listing 2.8: Setting the background color of elements with the name attribute set to "amount" using the CSS attribute selector

Headless browsers

Many popular browsers offer a way to run in headless mode. This means that the browser is launched without running its graphical interface. A common use case of utilizing a headless mode browser is for automated testing, but it can also be used for web scraping. Benefits of using a headless browser include faster performance, as the CSS, HTML and JavaScript code needed to render the web page can be avoided. One drawback that can be encountered with a browser running in headless mode is that websites may be configured to try preventing automatic access of its content. If a non-headless browser is utilized, this is not a problem, it will behave as if a human is controlling it.

2.2 Survey of Web Scraping tools

In this section twelve state of the art tools used for web scraping are presented. They have been found through searching the web or having heard about them due to their popularity. These twelve tools are considered for the evaluation moving forward.

Scrapy

Scrapy is an open source Python framework originally designed solely for web scraping, but now also supports web crawling and extracting data via APIs. XPath or CSS selectors is the built in way to do the data extraction, but external libraries such as BeautifulSoup and lxml can be imported and used. It allows for storing data in the cloud [7].

Selenium

A way of automating and simulating a human browsing with a web browser can be accomplished by using a tool called Selenium. It is primarily used and intended for testing of web applications, but is a relevant choice for web scraping. Using the Selenium WebDriver API in conjunction with a browser driver (such as ChromeDriver for the Google Chrome browser) will act the same way as if a user manually opened up the browser to do the desired actions. Because of this, loading and scraping web pages that makes use of JavaScript to update the DOM is not a problem. The Selenium WebDriver can be used in Java, Python, C#, JavaScript, Haskell, Ruby and more [8].

Jaunt

Jaunt is a web scraping library for Java that, compared to Selenium, uses a headless browser. It allows for controlling HTTP headers and accessing the DOM, but does not support JavaScript. Because of lacking the ability to work with JavaScript it is more lightweight, faster and can be easily scaled for larger projects. If JavaScript is a must, a crossover between Jaunt and Selenium called Jauntium [9] can be used. The idea with Jauntium is to overcome both the limitations of Jaunt and Selenium and keep the design and naming philosophy similar as with Jaunt, making for an easy transition over if needed. If JavaScript is not needed however, regular Jaunt is recommended [10].

jsoup

jsoup is an open source, Java based tool for manipulating and extracting HTML data. It implements the WHATWG HTML5 [11] standard and offers DOM meth-

[7] https://scrapy.org/

[8] https://www.seleniumhq.org/

[9] https://jauntium.com/

[10] https://jaunt-api.com/

[11] https://html.spec.whatwg.org/multipage/

ods for selecting HTML elements as well as CSS Selectors and jQuery-like extraction methods. It is designed to deal with any type of HTML, malformed or not [12].

HtmlUnit

HtmlUnit is a headless Java browser allowing commonly used browser functionality such as following links, filling out forms and more. Similarly to other headless browsers it is typically used for testing web applications but can also be used for web scraping purposes. The goal is to simulate a "real" browser, and thus HtmlUnit includes support for JavaScript, AJAX and usage of cookies [13]. HtmlUnit code is written inside JUnit [14] tests, which is a popular framework for Java unit testing.

PhantomJS

PhantomJS is a headless web browser that is controlled with JavaScript code. It is open source and is commonly used to run browser based tests but can be used for any type of automated browser interaction. QtWebKit is used as the back end and supports fast, native web standards such as CSS selection, DOM handling, JSON, Canvas and SVG. As of recently, PhantomJS is no longer being maintained [15].

Puppeteer

Puppeteer is an open source tool maintained by the Google Chrome DevTools team, used to control a headless Chromium [16] (or Chrome, if configured) instance. It is used as a Node.js module and is thus written and configured by JavaScript code. The intention of Puppeteer is to present a browser automation tool that is fast, secure, stable and easy to use [17].

CasperJS

CasperJS is a JavaScript tool for web automation, navigation, web scraping and testing. It is used as a Node.js module and uses a headless browser. CSS

[12] https://jsoup.org/
[13] http://htmlunit.sourceforge.net/
[14] https://junit.org/junit5/
[15] http://phantomjs.org/
[16] https://www.chromium.org/Home
[17] https://pptr.dev/

Selectors and XPath queries are used for content extraction. However, as of now, it is no longer being maintained [18].

Nightmare.js

Nightmare.js, a JavaScript tool, was originally designed to perform tasks on web sites that do not have APIs but has evolved into a tool that is often used for UI testing and web scraping. Exposing just a few simple methods (`goto`, `type` and `click`) makes for a simple yet powerful API [19]. Under the hood it uses Electron [20] as the browser.

BeautifulSoup

BeautifulSoup is a Python HTML extracting library. While not being a complete web scraping tool, it can be used in conjunction with the `requests` [21] package, which allows for doing HTTP calls in Python. BeautifulSoup is a simple, pythonic way to navigate, search and modify parse trees, such as an HTML tree. It is intended to be easy to use and provides traversal functionality such as finding all links or all tables matching some condition [22].

MechanicalSoup

MechanicalSoup is a combined solution of BeautifulSoup in conjunction with the `requests` Python package. It is written and configured by Python code. It does not support JavaScript loaded content but can automatically send cookies, follow browser redirections and links and supports form submissions [23].

rvest

rvest is a web scraping tool for the statistic-focused programming language R. Its intention is to be easy to use and claims to be inspired by Python web scraping tools such as BeautifulSoup and RoboBrowser. It allows usage of both CSS Selectors and XPath queries for element extraction [24].

[18] `http://casperjs.org/`
[19] `http://www.nightmarejs.org/`
[20] `https://electronjs.org/`
[21] `http://docs.python-requests.org/en/master/`
[22] `https://www.crummy.com/software/BeautifulSoup/`
[23] `https://mechanicalsoup.readthedocs.io/en/stable/`
[24] `https://cran.r-project.org/web/packages/rvest/README.html`

2.3 Evaluating software

With the ever evolving field of Computer Science and its technologies, deciding on what tool or framework to use for a desired project or goal can be difficult. When examining related work done on web scraping, many papers revolve around comparing or testing out different tools. This implies that the selection process is not always clear and in most cases has to be evaluated and examined thoroughly.

In this section, aspects and metrics for evaluating software are presented.

Evaluating or measuring software is a way in which different attributes are quantified. By then comparing the same types of quantified attributes between different software tools, conclusions may be drawn regarding the software quality. This can also shine a light on parts of a tool that might need improvements. A software metric is defined as a part of the software system, documentation or development process that can be measured in an objective way. This can include metrics such as the lines of code, cyclomatic complexity and the time taken for a process to finish. In Table 2.1, some common quantitative parts of software quality is listed and briefly described.

Quantitative metrics	**Description**
Time	The time taken for a process to be completed
Resources	How many resources (CPU power, memory) is required for a process
Cyclomatic complexity	The number of independent paths through a piece of code
Lines of code	The amount of code (lines) in a program, file or function
Depth of inheritance	How many levels of inheritance that are used
Number of error messages	The amount of error messages thrown
Features	What features a system or tool offers to its users

Table 2.1: Qualitative parts of software quality

While some areas are hard to objectively assess, judgements on these areas can still be measured by assuming that quantitative attributes such as size or cyclomatic complexity that can be objectively measured are related to qualitative aspects. These qualitative attributes are in the end subjective, but account

for a large part of software quality. Even if a software product's functionality is unexpected, workarounds to still fulfil required tasks are often available. This is not the case for if the software product is, for example, unreliable. There wont be possible for any software product to be perfectly optimised for all these aspects. For example, improving security may hinder performance. Thus, the aspects of most importance should be focused upon. The qualitative parts of software quality can be split up into 15 metrics [25]. These are listed and described briefly in Table 2.2.

Qualitative metrics	**Description**
Safety	Software operates without high consequence failure
Security	Software is confidential, integral and available
Reliability	Probability of failure free operation over a specified time, environment and purpose
Resilience	How well a software can maintain services despite events such as failure and cyberattacks
Robustness	System should be able to recover from individual processes failures
Understandability	Ease of understanding the source code and its structure
Testability	Requirements should be designed to allow for testing
Adaptability	Processes should be designed flexible and not, for example, require specific order of operations
Modularity	Related parts of the software are grouped together and avoids redundancy
Complexity	Number of relationships between components in a system
Portability	Does not require a specific environment to use
Usability	How easy a software is to use
Reusability	How easy and effective reuse of the software is
Efficiency	Does not waste system resources, such as CPU power and memory
Learnability	How easy learning to implement or use the software is

Table 2.2: Qualitative parts of software quality

2.3.1 Other metrics

While the more established metrics are presented above, more recent methods have been used to draw conclusions on software quality. Some of these methods encountered are presented below.

Machine learning

Work within software analytics has shown that, given enough data, machine learning and data mining can be used to analyse software repositories and draw conclusions without having to assume relation between software quality and collected data [28]. Machine learning has been used to predict error rates in software based on existing data history, based on open source repositories for Java projects. It was shown to be able to predict bug locations, based on a projects software metrics. These software metrics include Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Number of other classes that reference a class (FANin), Number of other classes referenced by a class (FANout), Lack of Cohesion of Methods (LCOM), Number of Children (NOC), Number of Attributes (NOA), Lines of Code (LOC), Number of Methods (NOM), Response for Class (RFC) and Weighted method count (WMC) [14].

GitHub repository statistics

The code itself is not the only thing occupying a GitHub repository, statistics and social tools regarding the repository is also present. This includes the amount of Stars, Watches, Forks, Contributors, Open/Closed issues and the amount of commits within the last year. *Stars* is a way to bookmark a repository to gain easy access to it from a GitHub account, GitHub may also look at what a user has starred [25] and base recommendations and the news feed content upon it. *Watching* a repository is like subscribing to it; a user watching [26] a repository will be notified when a new pull request or a new issue is created. *Forking* [27] is one of the ideas that urges open source work: it creates a copy of the repository and allows the user to work on the code without interfering on the main repository. If for example a feature is to be added, the user can fork the repository, implement the features and then create a pull request with the changes. *Contributors* is simply the total amount of people (unique GitHub

[25] `https://help.github.com/en/articles/about-stars`

[26] `https://help.github.com/en/articles/watching-and-unwatching-repositories`

[27] `https://help.github.com/en/articles/fork-a-repo`

users) that have contributed to the repository. An *issue* [28] is a way to present and organize tasks needed to be done within a repository, this includes bugs and user feedback. Issues can be paired with pull requests so that whenever a pull request is accepted the accompanying issue is closed as well.

Research has shown that information about a projects GitHub repository has implications on how the project quality is seen from developers. The amount of activity, especially the amount of recent commits, indicate the level of commitment and investment to the project. A project with many people *watching* show that there is interest, and would indicate a high quality project. The same results were shown for projects with a high *fork* count, it would indicate that the community had interest in the project, with the likely reasoning that the state of the project is good [7]. In another research paper, developers were surveyed on the popularity impact of the amount of *stars*, *forks* and *watchers* for a GitHub software project. Results showed that stars was viewed as the most useful measure, but all three were impactful. A second survey was conducted on why people starred GitHub repositories and whether the amount of stars impact how the users interact with a repository. Other research has also shown that the amount of stars correlate with other popularity metrics [4]. It was shown that people mostly star repositories for showing appreciation (52.5%) and for using it as a bookmark (51.5%), but also that they are current of previous users of the project (36.7%). When asked if the amount of stars is considered before using or contributing to a project, most answered yes (73.0%). A large dataset of repositories (5000) were then examined. It showed that there are correlations in the amount of stars and the primary programming language used in a project. JavaScript projects generally had a higher amount of stars, while non-web related projects had the lowest median amount of stars. There were a moderate correlation shown between the amount of stars and the amount of forks and contributors. No correlation between stars and repository age were shown [5].

Documentation

When looking at project documentation, research has shown that popular ones had a consistently evolving documentation, which in turn attracted more collaborators. A dataset with 90 GitHub projects were used and the popularity was defined by GitHub repository statistics as the number of Stars + Forks + Pulls. As for the changes in documentation, the size and frequency of the change were utilised. The conclusion was that a project with consistent popu-

[28] `https://help.github.com/en/articles/about-issues`

larity in turn consistently evolves and improves its documentation [2].

Similarly, research has shown that documentation is a key factor to the reusability and understandability of a component. This is considered increasingly important in open source projects, due to the nature of having contributions from many different developers. Analysing code is also considered difficult without documentation [8].

2.4 Previous work

In this section relevant research papers or theses are presented to gain an insight to what has already been done. In the end, a section will summarize the different evaluation methodology used for comparing web scraping tools.

2.4.1 Web scraping - A Tool Evaluation

In a master thesis called Web scraping - A Tool Evaluation, the author evaluated a few web crawling tools to find the one providing the highest value for a company's future projects. It was done by defining a value-based tool selection process: first identify what benchmark features are of value to the company. Then map these benchmark features with the results of evaluating different web crawlers, to then decide if an existing crawler should be used or if a new one needs to be developed. The benchmark features includes: Configuration, Error Handling, Crawling Standards Navigation and Crawling, Selection, Extraction, File (how the data can be saved), Usability and Efficiency. These features included sub-features that were considered and used as a check list, indicating whether the tool fulfils said feature. In total 8 different web crawlers were investigated:

- Lixto Visual Developer 4.2.4
- RoboSuite 6.2
- Visual Web Task 5.0
- Web Content Extractor 3.1
- Web-Harvest 1.0
- WebSundew Pro 3
- Web scraper Plus+ 5.5.1
- WebQL Studio 4.1

Results

RoboSuite had the highest rating overall out of the 8 tools compared, closely followed by Lixto Visual Developer and WebQL. Visual Web Task, WebSundew Pro and Web Content Extractor were the lowest scorers. WebQL had the highest rating in the configuration category. The code, written in the programming language WebQL, is reusable and can be automated. The fact that it is configured via code makes the configuration faster when websites crawled are similar [19].

2.4.2 Evaluation of webscraping tools

A thesis that evaluates three Java tools used for web scraping, jArvest, Jaunt and Selenium WebDriver. Only Java compatible tools were chosen because of having to integrate with a Java framework called Play. The purpose is to gather data from different web resources about horse trotting, including data about trainers, horses and race tracks. The gathering process is divided into five steps. First the base URL is retrieved and a search form is to be filled and submitted. Then the correct table for the horse trainer in the resulting view should be found and the right hyperlink followed as this will redirect to the trainers list of current active horses. The fourth step is to extract each horses individual document and then in the final step extract a table from these documents. The tools were evaluated on functionality, solution complexity and ease of implementation. It does not consider performance benchmarking as the tools differ much in their implementation and are thus not using the same approach for the case implementation. The author argues for using headless browsers as a platform for web scraping, as web browsers have been constantly developed throughout the years, keeping them relevant as extraction methods.

Results

The best matching framework for the desired implementation turned out to be Jaunt. While both Jaunt and Selenium provided a sufficient solution to the implementation, jArvest failed due to not being able to use session cookies, which proved to be a necessity. Jaunt offers observing and editing HTTP headers and provides an API for producing header requests which can allow for getting batches of specifically requested data. Selenium instead has to do it as a browser would by first requesting a document, filling a form and then posting it for each batch, making it less efficient. If JavaScript and AJAX is used to load dynamic data on the website, Selenium is to prefer, as Jaunt is

unable handle DOM elements updated this way [21].

2.4.3 Web scraping the Easy Way

Web scraping the Easy Way is a paper that investigates and explores the usage of web scraping tools. It starts off by the author being asked to program a web scraping software for a client. The client wants to gather items from several retailers, to present and sell them in one place. Very few of the retailers offered APIs and thus web scraping software had to be developed. After spending some time developing the scrapers, the author found out about the already existing low cost and effort tools for doing web scraping. This revelation was the papers basis, to expose the easy to use tools that exist on the market. In the end, a tool called Data Toolbar [29] was used. Data Toolbar is a browser extension available for Internet Explorer (now called Microsoft Edge), Mozilla Firefox and Google Chrome and does not require any specific technical skills. It has a free and a paid version, the only difference being that the free version is limited to a 100 row output. Scraping can be done in the background and does not interrupt other applications. Selecting elements with the mouse and keyboard points out what should be extracted from the page. Then the form of output can then be chosen. In conclusion, Data Toolbar is a good tool for small, simple extractions usable by the average computer user [20].

2.4.4 A Primer on Theory-Driven Web Scraping

In a psychological research paper, the use of web scraping was utilised to gather data. The data gathered consists of posts from a discussion board where people can seek depression help. The purpose was to find information on interaction and gender correlation. The hypothesis leading into the project was defined as
Women engage in more support-seeking coping behaviors than men. Healthboards.com [30], containing approximately 120.000 posts across 20.000 threads was the discussion board chosen for the project. By using this data and the initial hypothesis, two research questions were formed:

- Are there gender differences in the number of responses to these support-seeking coping behaviors?

[29] `http://datatoolbar.com/`
[30] `https://www.healthboards.com/`

- Is there an interaction between support-seeking gender and respondent gender such that men respond more often to men, women respond more often to women, women respond more often to men, or men respond more often to women than would be expected from the main effects alone?

To be able to web scrape the posts, the authors had to first learn Python, via a 13 hour free course. Then Scrapy, the web scraping framework used had to be learned. Learning Scrapy took about 5-20 hours per person. With the knowledge gained, the web scraper was written in about four hours. In total it was ran for about 20 hours, gathering 165.527 posts in total. After removing duplicate posts, 153.040 remained. After examining the data it was found that reporting gender when signing up was introduced around 2004, 4 years after the first gathered post appeared. Thus, some of the data had lacking information. To solve this, posts prior to November 2005 were discarded and 66.387 posts were left with 99.2% of posters having their gender set. Based on this dataset 10001 cases were defined, one case per thread containing all of the replies and their genders.

Results

Out of all the threads examined, authors of the posts were 70.5% female and 29.5% male. The total amount of replies from women were 69.32% thus supporting the hypothesis. 67% of the people responding to male authored posts were female while 70% of replies to female posts were female. The most significant lesson learned from the project is said to be the value of a data source theory: only after gathering the data and examining it unexpected results were revealed. For performing web scraping projects in psychology research, knowledge in Python or R is considered critical, while technical and time requirements are considered low. Many technical resources needed are common or easily obtained, such as computer hardware, internet access (preferably of high speed) and web scraping software.

A four step process is recommended when including web scraping in a psychology research project. First, identify information sources and form a data source theory to explain why the data exists and for what informational purpose. Secondly a system for the dataset layout needs to be designed. When the data source is known and the desired dataset layout has been defined the scraper can be coded. R and Python are recommended programming languages as they are commonly used in psychology and within the skill set of many psychologists. Finally, once the data is fetched it should be cleaned and

checked for false data that can indicate errors in previous steps. Now that the data is cleaned and correct, it is ready to be used [16].

2.4.5 Methodology summary

Out of the four papers examined, two are based on comparing web scraping tools. Web scraping - A Tool Evaluation [19] and Evaluation of webscraping tools for creating an embedded webwrapper [21]. In the first-named, main focus is on comparing functionality for tools in general. It consists of a thorough process that goes looks at 9 different feature areas and breaks them into sub areas. In total, 54 features are considered. Some of these features include performance related topics such as memory and CPU usage, but also topics regarding documentation, installation and support [19]. In the second-named, functionality is also the main evaluation point, but it also takes solution complexity and ease of implementation in to account. This includes factors such as IDE support, additional dependencies and general ease of implementation [21].

The different areas of evaluation used in previous research are summarized in Table 2.3.

Area
Features
Functionality (Navigation, selection, filling forms)
Technologies supported
Performance (Memory, CPU, Disk space)
Ease & efficiency of use
Documentation
Installation
Platform independence
Dependencies

Table 2.3: Evaluation areas used in previous methodology

Methodology critique

While many of the aspects used are part of established software evaluation metrics, some are missing. Performance, in terms of run time, is neglected.

This is a vital part of software evaluation [25] and should be included. For web scraping specifically, having a fast tool enables utilization of the ever increasing bandwidth amount. When running large, reoccurring web scraping tasks, which is common for example in businesses revolving around web scraping many different retailers, run time becomes increasingly important and beneficial. For use cases that uses web scraping for one time, data gathering, the run time may not be as important, but given a faster tool, scraping tasks can be re ran more often in the case of implementation faults, where the data gathered may not be the desired data or when the implementation needs to be altered in other ways.

None of the more recent methodology used is present. Using machine learning and data mining as methods for evaluating software quality [28] [14] could be utilised. This way, there might be less need for evaluating the software quality aspects manually, which is prone to contain interpretation errors. A machine learning solution could also, given the tools source code, find new patterns that implies indication of software quality.

No statistics regarding the projects repository is taken into account, which has shown to impact the way outside software developers view the project quality [7] [5] [4]. This does however require the projects to be open source, which could have been an issue. As there does not seem to be a clear cut, most popular or widely used web scraping tool, this is a factor that could eventually be a deciding factor for such a tool. All of the tools presented in the tool survey are open source, indicating that this is the common way of developing a web scraping tool. As such, these statistics is an indication for the outside developers view on the project quality, making it more likely for a project with good repository statistics to get further open source development.

As for the documentation, more focus could have been put on the actual content rather than simply looking at whether it exists and is searchable. Good documentation helps understanding and analysing code [8]. For web scraping, being certain that the tasks are performed the way desired is of high importance. Not only to be sure that the scraper performs any unnecessary actions that could put a heavy load on the web page, but also to avoid any unwanted access.

Chapter 3

Method

This chapter first presents an initial evaluation, with the purpose of reducing the amount of tools for the evaluation process. The twelve tools presented in the background section are considered. Once the amount of tools have been reduced, the evaluation framework is presented. The model is split up into four sections and areas: performance, functionality, reliability and ease of use.

3.1 Selection of tools

The twelve tools described in the background section will perform a simple web scraping task, to further decide which tools to investigate in the evaluation process. Tools that are similar are grouped together and reduced to one representative. The reasoning is that twelve tools is considered too many given the time restrictions for the thesis. A second purpose of the task is to be introduced to how web scraping is done using these tools, as greater understanding is needed for the evaluation.

3.1.1 Selection task

The initial web scraping task is based on the sandbox environment at `https://scrapethissite.com/pages/simple/`. 250 different countries and information on their capital, population and area are presented in one page, shown in Figure 3.1.

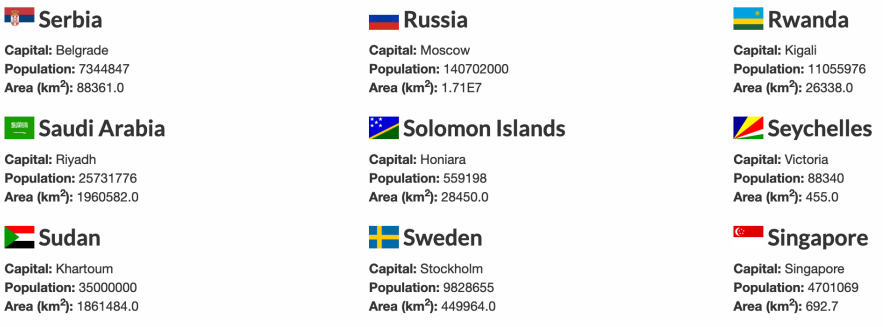


Figure 3.1: Example subset of data to be scraped

The task is simply to scrape each country and their capital and print them out to the screen. An example output is shown in Listing 3.1:

```
...
Country: Serbia - Capital: Belgrade
Country: Russia - Capital: Moscow
Country: Rwanda - Capital: Kigali
Country: Saudi Arabia - Capital: Riyadh
Country: Solomon Islands - Capital: Honiara
Country: Seychelles - Capital: Victoria
Country: Sudan - Capital: Khartoum
Country: Sweden - Capital: Stockholm
...
```

Listing 3.1: Example output from the initial scraping task

Note that this task is not part of the evaluation framework, but exists solely to reduce the amount of tools to compare using the evaluation framework.

3.1.2 Selection results

Implementation of the twelve tools posed no real complications, every tool managed to scrape the countries and their capitals with low effort and code amount. There were implementation similarities of a few tools. The code for the PhantomJS and Puppeteer versions look almost identical, while Nightmare.js and CasperJS are not as similar. Both CasperJS and PhantomJS are no longer actively maintained. BeautifulSoup with requests and the MechanicalSoup side are also very similar. This is to be expected, as MechanicalSoup [1]

[1] https://mechanicalsoup.readthedocs.io/en/stable/

is built on using BeautifulSoup along with the requests library As for the Java tools, Jaunt, Jsoup and HtmlUnit, are also similar in structure. The Selenium case is special, as it has bindings for multiple languages. In this case a Python and Java version were implemented.

Based on these findings, Puppeteer and Nightmare.js are kept from the JavaScript group. This is mostly due to PhantomJS and CasperJS no longer being maintained, but also the similarities mentioned. Scrapy is kept from the Python solutions as it appears to be the more maintained project with over ten times the amount of commits compared to MechanicalSoup (6.879 versus 505). As the three Java versions are similar the level of maintaining had to be investigated. HtmlUnit is actively kept maintained with over 15.000 commits (when this is written) and the most recent commit being a few hours ago. Jaunt recently had a new release (the 13th February, 2019) and is thus considered being well maintained. However, Jaunt does not support JavaScript. Jsoup had its last commit 2 months ago. Based on these observations, HtmlUnit is kept from the Java group. Selenium is a popular solution and has been around since 2004, proving its longevity. Because of this, both a Java and a Python version using Selenium is kept. rvest, being the only tool considered for the R language is also kept further.

Out of the twelve tools considered, the six kept to be examined in the evaluation process is shown in Table 3.1.

Name	**Released**	**Language**
Puppeteer	2017	JavaScript
Nightmare.js	2014	JavaScript
Selenium	2004	Multiple
HtmlUnit	2002	Java
rvest	2014	R
Scrapy	2008	Python

Table 3.1: The six tools to be evaluated, their first release date and programming language

3.2 Evaluation framework

For comparing the tools, a framework for evaluation is developed. These are the four areas that are of interest, based on previous work in general software evaluation and previously used areas in web scraping research [25] [12]:

- Performance
- Features
- Reliability
- Ease-of-use

3.2.1 Hardware

All of the evaluation processes were ran on the following hardware:

- Intel Core i7 2,2 GHz processor
- 16 GB 2400 MHz DDR4 RAM
- Intel UHD Graphics 630 1536 MB Graphics card

3.2.2 Performance

In this section the performance related task will be presented. First, the different aspects that are measured are presented. These include time, CPU, real and physical memory usage, which are common metrics used in software evaluation [25]. In the following section, the task used to measure these aspects is presented.

Measurements

An established part of software performance measurement is how long a process takes to run [25], therefore all the different tools were timed when evaluating performance tasks. The python tools were timed using the `time` package [2], Java tools via `System.currentTimeMillis` [3] and rvest by `Sys.time`

[2] `https://docs.python.org/3/library/time.html`

[3] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html`

[4]. For JavaScript, `Date.now` [5] was used. To not get one-off results, each tool was ran 100 times on the same task and then an average time was calculated. The amount of resources used is also an interesting aspect to measure. As such, memory and CPU activity will also be recorded while performing the task. A tool called `psrecord` [6] is used. It allows for starting and monitoring a specific process to get its CPU and memory information. More specifically, the CPU (%), virtual memory (MB) and real memory (MB) are measured. A time interval (between measurements) can be specified to get the precision wanted. While `psrecord` is told to be an experimental tool, it utilizes `psutil` [7] to record the CPU and memory activity. `psutil` is used by both Google and Facebook, and is because of this considered a reliable tool. Seeing as the CPU and memory activity data from `psrecord` is fetched by using `psutil`, `psrecord` is considered reliable enough to be used in the thesis. The CPU and memory measurement terms are explained further below.

CPU (%)

The first value that `psrecord` records is CPU (%). This value is the total amount of CPU that the process consumes. It does allow values above 100%, which indicates that the process is ran on multiple threads on different CPU cores. The machine that ran the tests has a total of 6 cores, thus 600% would mean that all cores work to their full extent on the process.

Real memory

Real memory, also known as physical memory or RAM, is the second metric recorded. It simply measures the amount of physical memory the process is using. As a reminder, the machine the evaluation is performed on has a total of 16GB physical memory.

Virtual memory

Virtual memory is the final value that is recorded. A process may require more physical memory capacity than available. In this case, the technique known as virtual memory is utilized. Only the currently used part of the process or program is kept in physical memory, the rest is put on disk. Parts can then

[4] https://stat.ethz.ch/R-manual/R-devel/library/base/html/Sys.time.html

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now

[6] https://github.com/astrofrog/psrecord

[7] https://github.com/giampaolo/psutil/

be swapped back and forth from physical memory and disk [26]. The value measured by `psrecord` is simply the amount of virtual memory that the process uses.

Book scraping task

The performance task consists of web scraping book information from the sandbox website `books.toscrape.com`. In total 1000 books are displayed, spanning over 50 pages (20 books per page). Each entry contains the book cover, a randomly assigned star rating (1 to 5), a title, a random price and a button to put it in the basket. The task is to gather the title and star rating for each book. As the books are distributed over multiple pages, navigating the site is required to finish the task. While running the task, time was taken and memory and CPU usage were recorded. An image of the first page is shown in Figure 3.2.

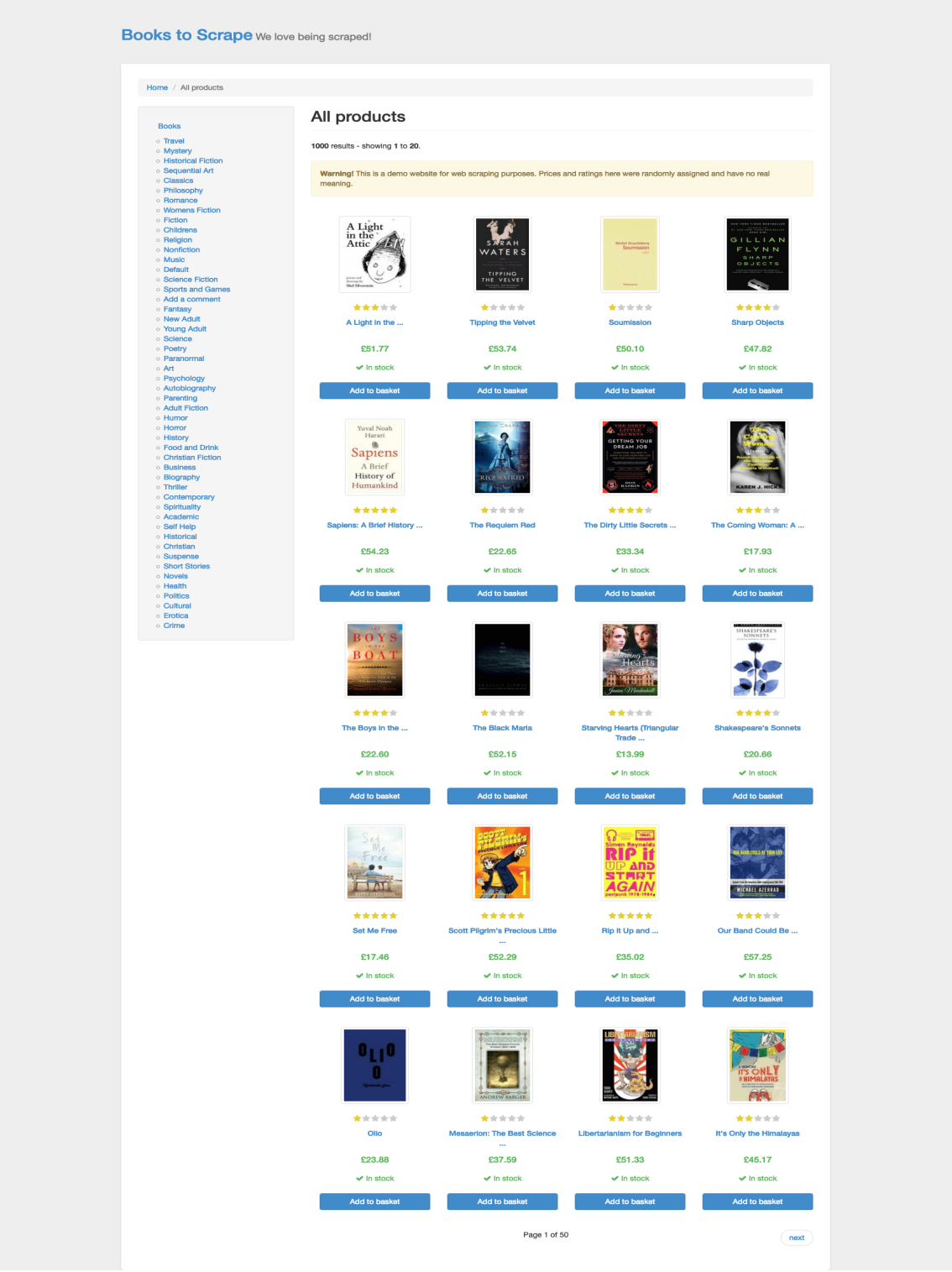


Figure 3.2: The page to be scraped by each tool to evaluate performance

3.2.3 Features

In this section different evaluation tasks regarding tool features is presented. Features are a core part of a tools area of usage [25], and has been used previously in web scraping evaluation research [19] [21]. Seeing as Selenium offer the same functionality indifferent of programming language, the Java and Python versions are paired up, as the results would have been the same. The result of passing the task or not and the implementation is presented in the result chapter.

Capability to handle forms and CSRF tokens

Websites may require searching via a form or other form actions such as logging in. Therefore one feature evaluation is based on using a login form. This includes filling a username, password and submitting the form. A version of this use case can be found on `http://quotes.toscrape.com/login` which is a similar sandbox environment for web scraping as the one used in the book scraping task. This page is also guarded by a hidden CSRF (Cross-Site Request Forgery) token. A CSRF token is a security measure for preventing Cross-Site Request Forgery, a common vulnerability on web sites. In what is known as a CSRF attack, a malicious website tries to get the users browser to send requests to a regular, honest site. For example, Gmail had a CSRF vulnerability that got exploited in 2007. A user visited a malicious site that would generate a request to Gmail, but making it look like it was part of the already established session the user had with Gmail (the regular, correct session). The attacker then managed to add a filter to the users email account that would forward all emails to the attacker. The users personal website used email authentication, thus allowing the attacker with the confiscated emails to take control of the website. A popular protection measure to CSRF vulnerability is to include a hidden token with each request (each time the form is submitted, a new token is passed along with the form fields) to validate that the form submission is tied to the users session [1]. This is called a CSRF token and is present on the sandbox website where the evaluation task is to take place. The form fields that are to be filled and submitted by the tools are shown in Figure 3.3, followed by the view after a successful login in Figure 3.4.

Figure 3.3: The page to be used to test form capability for each tool, before successful login

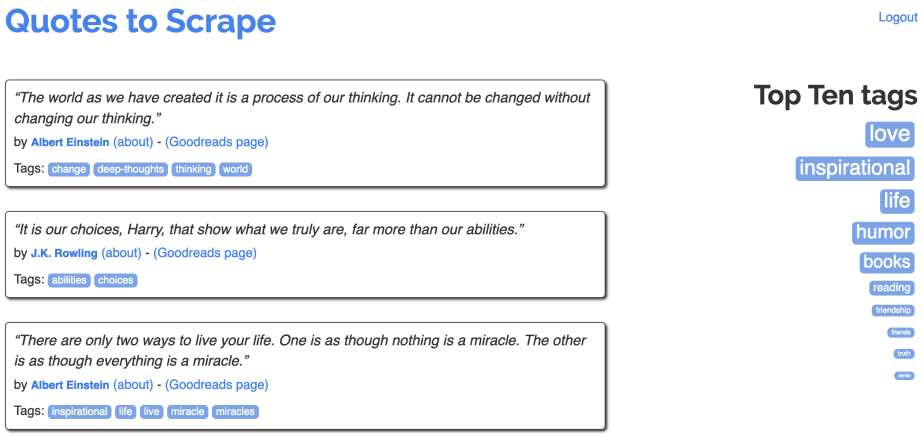


Figure 3.4: The page to be used to test form capability for each tool, after successful login

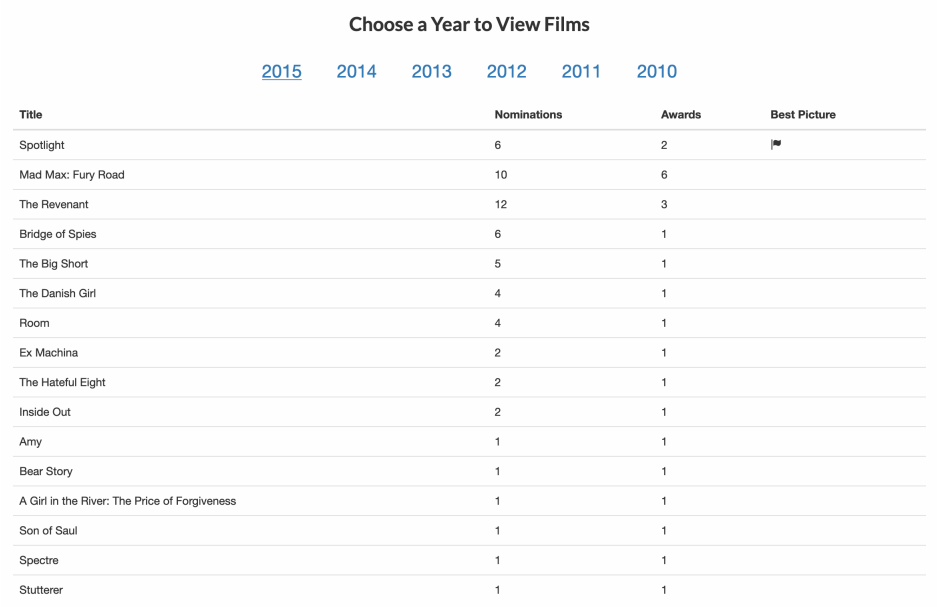
Once logged in, the URL has redirected to `http://quotes.toscrape.com/` and the text "Login" has changed to "Logout". This can be seen in Figure 3.4. Extracting and checking the Login button state was used to confirm the tasks success or not. If the username field or the CSRF token was not supplied the page will stay at `http://quotes.toscrape.com/`. showing error message "Error while logging in: please, provide your username." or "Error while logging in: invalid CRSF token."

AJAX and JavaScript support

There are cases where website content is loaded dynamically via JavaScript code. Having web scrapers support loading and scraping these sort of websites may be of importance. A sandbox website for testing this feature out exists at `https://scrapethissite.com/pages/ajax-javascript/` where information on recent years Oscar nominated films can be accessed. The initial view, before the data has loaded, is shown in Figure 3.5.

Figure 3.5: The page to be used to test JavaScript capability for each tool, before data is loaded

To get the film information a year has to be chosen. When one of these links are triggered a table of the selected years Oscar nominated films is loaded dynamically through JavaScript, presenting a table consisting of the film title, the amount of nominations, how many of these nominations were won and which film that won best picture. The view after the data has been loaded can be seen in Figure 3.6.



Choose a Year to View Films

2015 2014 2013 2012 2011 2010

Title	Nominations	Awards	Best Picture
Spotlight	6	2	⚑
Mad Max: Fury Road	10	6	
The Revenant	12	3	
Bridge of Spies	6	1	
The Big Short	5	1	
The Danish Girl	4	1	
Room	4	1	
Ex Machina	2	1	
The Hateful Eight	2	1	
Inside Out	2	1	
Amy	1	1	
Bear Story	1	1	
A Girl in the River: The Price of Forgiveness	1	1	
Son of Saul	1	1	
Spectre	1	1	
Stutterer	1	1	

Figure 3.6: The page to be used to test JavaScript capability for each tool, after data is loaded

The task is to go from the first view, display the 2015 information and scrape each title and the amount of nominations for each title. If the tool manages to do the task out of the box, without additional dependencies, it is considered a pass.

Capability to alter header values

Some sites will try preventing automated software from accessing its content by looking at the HTTP request to find out who is sending it. A header field known as `User-Agent` tells the receiver who (what browser, what OS the machine runs) is making the request. The site may then prohibit requests that do not seem to comply to a real user using a real web browser. This test checks the ability to bypass these restrictions. A sandbox environment that can be used to test these properties exist at `https://scrapethissite.com/pages/advanced/?gotcha=headers` and is simply a page that prints out a success message, if it seems to stem from a real browser. A successful attempt is shown in Figure 3.7.

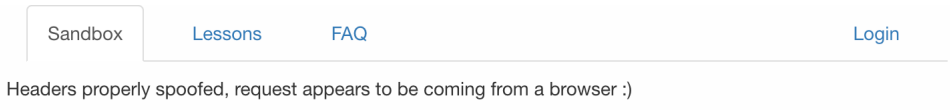


Figure 3.7: The page used to test header spoofing for each tool, showing a successful attempt

If something is wrong, an error message is printed. For example, if the `User-Agent` header field is not properly set, the webpage will let the user know. This can be seen in Figure 3.8.

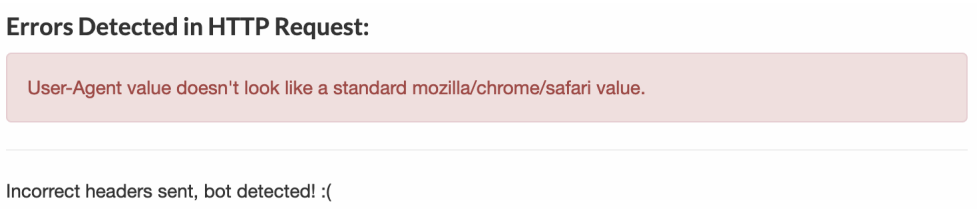


Figure 3.8: The page used to test header spoofing for each tool, showing incorrect User-Agent header value

If the `Accept` header field, that tells the receiver what type of data can be sent back to the client (text/html would be the value in this case), is incorrect, a different message is presented. Figure 3.9 shows the message.

Figure 3.9: The page used to test header spoofing for each tool, showing incorrect Accept header value

The task is simply to get the correct message output, indicating that the web scraper has bypassed the header checks.

Capability to take screenshots

A way to visualize the state of the web scraper is by taking a screenshot. This can be useful when debugging crashes or faulty operations. The tools are examined on whether they allow for taking a screenshot of the page, without extra dependencies. The task is to navigate to `http://example.com/` and take a screenshot of the page. This is shown in Figure 3.10.

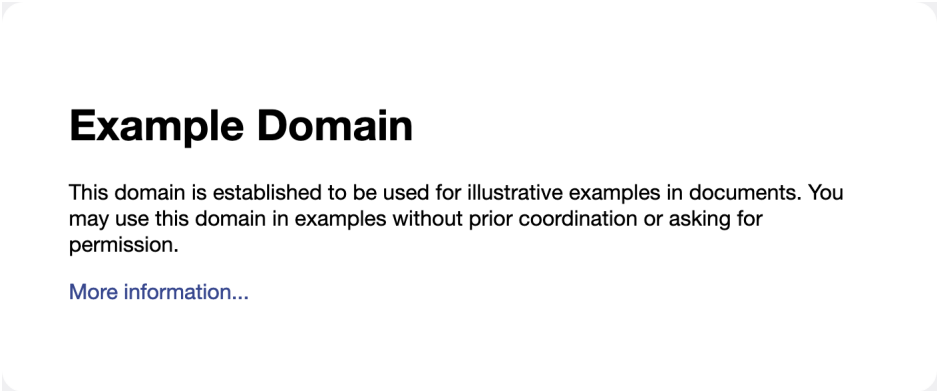


Figure 3.10: The page used to test screenshot capability for each tool

Capability to handle Proxy Servers

By utilising a proxy server the web scraper can appear to be from a different location. This can be useful both bypassing location restrictions but also for masking the fact that it is indeed a web scraper accessing the site. The task is to first implement proxy server integration and then visit `http://httpbin.org/ip`, that returns a JSON-object consisting of the clients ip address. By looking at the ip address returned it can be determined whether the tool successfully went through the proxy server or not.

Capability to utilise browser tabs

Instead of having to open a new browser instance for each errand, a new tab in an existing browser window can be utilized. This is tested by fetching data from one tab and then using the data in another tab. Two pages are to be opened in separate tabs. First, a static webpage that is hosted locally, consisting of two paragraphs, a username and password. This webpage is shown in Figure 3.11.

admin

12345

Figure 3.11: A webpage with a static username and password

Then, a sandbox login testing environment, `http://testing-ground.scraping.pro/login`, that only accepts the username (admin) and password (12345) taken from the static website. This webpage is shown in Figure 3.12.

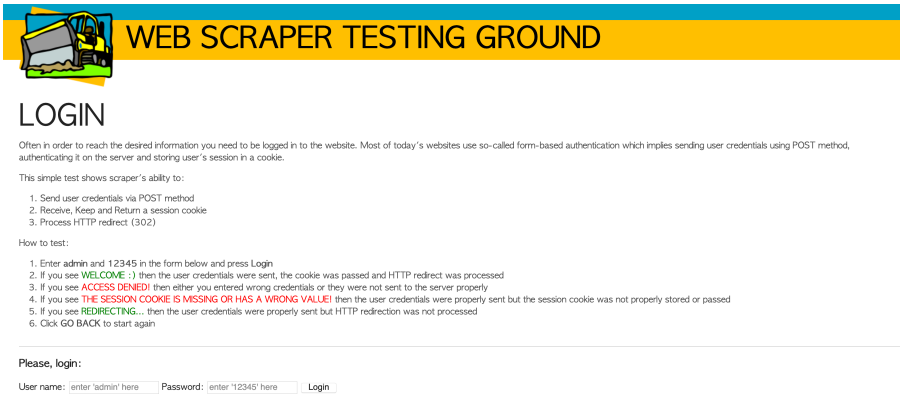


Figure 3.12: A login form to test tab capability

Whether the tool manages to login with the correct username and password, taken from the first tab, will decide if the task is passed or not.

3.2.4 Reliability

This section presents the reliability part of the evaluation framework. While reliability is considered a qualitative aspect of a software project, quantitative aspects can be used to reason about reliability. Such aspects include cyclomatic complexity [25], which is one of the metrics that was used in this part of the evaluation. Other metrics include GitHub repository statistics, which has shown to be a factor on how developers view the projects quality [7] [4] [5]. First, the terms software reliability and technical debt are explained. Different metrics regarding code quality of the tools code base and GitHub repositories statistics are to be gathered. Thus, two tools used for the gathering process are presented. Finally some of these metrics and previous research involving them are discussed.

Software reliability

Software reliability is defined as "the probability of failure-free software operation for a specified period of time in a specified envirnoment" [22]. Reliability is included among other areas in what is called Software quality [10]. The work to achieve a higher software quality is increased by having something called a high technical debt. Essentially, the more technical debt a project is considered to have, the harder it is to maintain, update and the software becomes more likely to encounter errors. This in turn counteracts the software reliability. Previous research has shown that three factors are the driving force for high technical debt within software projects. These include architectural, complex code and lack of documentation [17]. A tool called CBRI-Calculation [8], which is presented in the paper [17], offer information regarding these metrics. It was developed especially for investigating public GitHub repositories reliability and maintainability. CBRI-Calculation uses Understand [9], to gather information regarding the code complexity. Understand is a software tool which is used for statically analysing code. All of the tools examined have public GitHub repositories, and is thus able to be examined. However, CBRI-Calculation does not support R and also showed to not fully support code complexity metrics for JavaScript. As such, a second tool called `gitinspector` [10], is used to gather information regarding the repositories cyclomatic complexity. There is considered a strong link between cyclomatic complexity and code reliability.

CBRI-Calculation also fetches information and statistics regarding the GitHub repository itself. This includes the amount of stars, collaborators, watches, forks and recent commit activity. These terms will be explained further. Result data is used as a base to reason about the tools reliability and maintainability, as this is the purpose of the CBRI-Calculation tool [17]. The metrics used are:

- Propagation cost
- Architecture type
- Core size
- Lines of code
- Comment to code ratio

[8] https://github.com/StottlerHenkeAssociates/CBRI-Calculation

[9] https://scitools.com/features//

[10] https://github.com/ejwa/gitinspector

- Classes
- Files
- Median lines of code per file
- Files >200 lines
- Functions >200 lines
- Cyclomatic complexity
- GitHub repository statistics

While some of these metrics are self explanatory, the more complex terms are explained below.

Propagation cost

A measurement that looks at the amount of files that are directly or indirectly linked to each other. If a file is linked to many other files, a change in it will likely cause changes needed in the other files. A low number indicates that the files are not linked, or dependant, on each other.

Architecture type

An algorithm is used to divide the architectural structure into a type. It has four different choices; core-periphery, borderline core-periphery, multi-core, and hierarchical.

Core size

When calculating the core size, each class or file is classified into one out of five groups: core, shared, control, peripheral, or isolate. It is based on the number of links from the class or file. The core size consists of the largest group of classes or files that are linked to each other. Core files have been shown to contain more defects and become harder to maintain [17].

Cyclomatic complexity

In short, the cyclomatic complexity value amount to the number of independent paths through a method. The cyclomatic complexity value is achieved

by first representing the code as a directed graph with unique starting and end nodes, and then applying the following formula:

$$E - N + 2P$$

N is the number of nodes, which represent a block of sequential code that does not change the "direction" of the program. E is the number of edges that corresponds to the amount of times the program transfers control, for example by branching out in the form of an if statement (where it can go in one or more direction). P represents the number of connected components, for example if a subroutine is called within the program, P would be incremented by 1 [18].

Below is a small example program, its graph representation and its cyclomatic complexity value:

```
def test(num):
        if(num == 2):
                print("num is 2!");
        else:
                print("num is not 2");
```

Listing 3.2: Example Python code

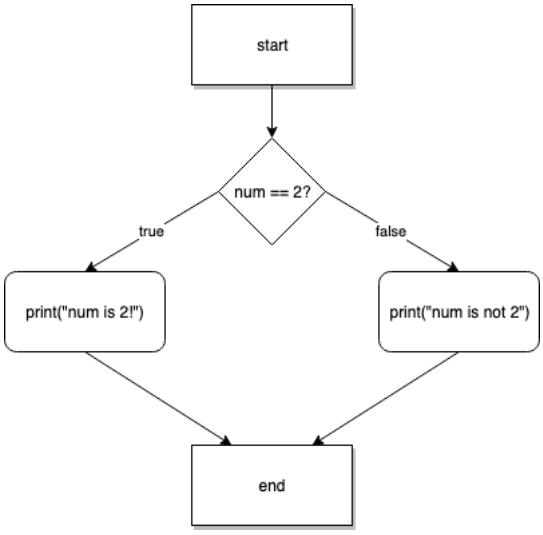


Figure 3.13: Graph representation of code in Listing 3.2

Filling in the formula with the corresponding values, as we have 5 edges

$E = 5$, 5 nodes $N = 5$ and no external component is connected $P = 1$:

$$5 - 5 + 2 * 1 = 2$$

Thus, we have a cyclomatic complexity value of 2.

Three different metrics involving cyclomatic complexity were measured. First, the amount of files with cyclomatic complexity over 50 and then the average cyclomatic complexity, among all files within the project. Finally, the average cyclomatic complexity density was measured for each project.

Cyclomatic complexity density

Another way of measuring software complexity is by using cyclomatic complexity density. This method combines the cyclomatic complexity metric with lines of code, two metrics that was shown to be highly correlated for code complexity. Cyclomatic complexity density has been shown to be an indicator on the maintainability of a software project [11]. The value is calculated, for each file, as:

$$CCD = CC/LOC$$

Where CCD is the cyclomatic complexity density, CC is the cyclomatic complexity, and LOC the total lines of code for the file. The average cyclomatic complexity density will then be presented for each tool.

GitHub repository statistics

While the code itself is analysed and measured on the statistics mentioned above, information regarding the repository itself was also gathered. This includes the amount of Stars, Watches, Forks, Contributors, Open/Closed issues and the amount of commits within the last year. As mentioned in the background, these statistics impact the project quality seen from developers [7] [5], which is crucial for open source projects that rely on outside contributions. As such, these metrics were gathered from each projects GitHub repository.

3.2.5 Ease of use

In this section, dependencies, installation, official tutorials and the documentation were examined to draw conclusions regarding ease of use. Good documentation has shown to increase re-usability and ease of understanding and analysing code [8]. Evaluation regarding installation and tutorials has been

present in previous web scraping evaluation research [21]. Concrete, quantitative results are hard to achieve within these areas and as such most of the results are subjective. Installation, tutorials and documentation are rated on a scale from 1 to 15, which was suggested by the host company. For these areas, a 5 point Likert scale survey is produced and performed, with the purpose of making the subjective results gathered more comparable. Each area consist of three Likert statements. Dependencies was simply collected and listed in a table.

- Dependencies
- Installation
- Tutorial
- Documentation

Dependencies

The dependencies is defined as the required extra software needed to run the tool. This includes the programming language and its environment, as well other external libraries or software. Dependencies that are fetched and installed automatically, when installing the main tool, are not considered.

Installation

An evaluation of the installation process. The installation process is defined as the steps required to install the tools and how to import it to the code. Three statements were evaluated for each of the tools and their installation process:

Q1: The installation method is easy to find

Q2: The installation process is simple to follow

Q3: No other resources are needed

Tutorials

A tutorial acts as a introduction and a way to get started. The better the tutorial, the faster a developer can start working with a new technology or framework. All tools had its official tutorial evaluated on the following statements:

Q1: The get started section is easy to follow

Q2: There are examples showcasing different use cases

Q3: There are links or references to external resources

Documentation

The general quality of the documentation. When having decided on a tool or technology to use, a developer will likely interact with the documentation frequently. Therefore it is of importance that it serves its purpose and is pleasant to use. The official documentation for each tool was evaluated on the following statements:

Q1: API references are well described

Q2: The documentation contain examples

Q3: The documentation is easy to navigate

Chapter 4

Results and discussion

In this chapter, results from the defined evaluation process is presented. As the results are many, spanning over four areas, they are also be discussed at the chapters end. This is done to prevent the reader from losing details that might have otherwise been forgotten between chapters. First up is the performance section, where the performance task results is presented. This is followed by the feature section, where the results regarding the different feature evaluations is presented. Each tools implementation is also shown, including code snippets. The reliability section presents results gathered from the tools code base and GitHub repositories. Dependencies, the installation process, tutorials and documentation results is shown in the ease of use section. Finally, a discussion section will conclude the chapter. This section will look at the results and discuss them. It will also present which tool is considered the best, general quality of the framework developed, its impacts on theory and practice, and where this type of work can go next.

4.1 Performance

First, the average time taken to perform the performance book scraping task is shown in a table. Then, the time of the individual runs for each tool is presented in bar and box graphs. Finally the CPU, virtual memory and physical memory usage for a single run are shown. A discussion section concludes the performance section.

4.1.1 Book scraping

All selected tools managed to scrape the 1000 book titles and their respective ratings. The average time taken (n=100) to perform the task are presented in Table 4.1. Figure 4.1 and Figure 4.2 show the run time for all 100 runs in two different forms, to better illustrate and compare the different results. The CPU, Virtual memory and Real memory used during the runtime is summarized in Table 4.2, where the average values are shown. Figure 4.3 presents the amount of CPU power (%) used each second while performing the task. Similarly, Figure 4.4 shows the Virtual memory (MB) used. Finally, Figure 4.5 shows the Real memory (MB) needed for each tool.

Name	**Headless (ms)**	**Non headless (ms)**
Puppeteer	7604	9732
Nightmare.js	-	20834
Scrapy	8397	-
Selenium w/ ChromeDriver (Python)	-	29013
Selenium w/ FirefoxDriver (Python)	-	26041
Selenium w/ ChromeDriver (Java)	-	29827
Selenium w/ FirefoxDriver (Java)	-	30316
HtmlUnit	2940	-
rvest	3698	-

Table 4.1: The average run time for each tool over 100 runs

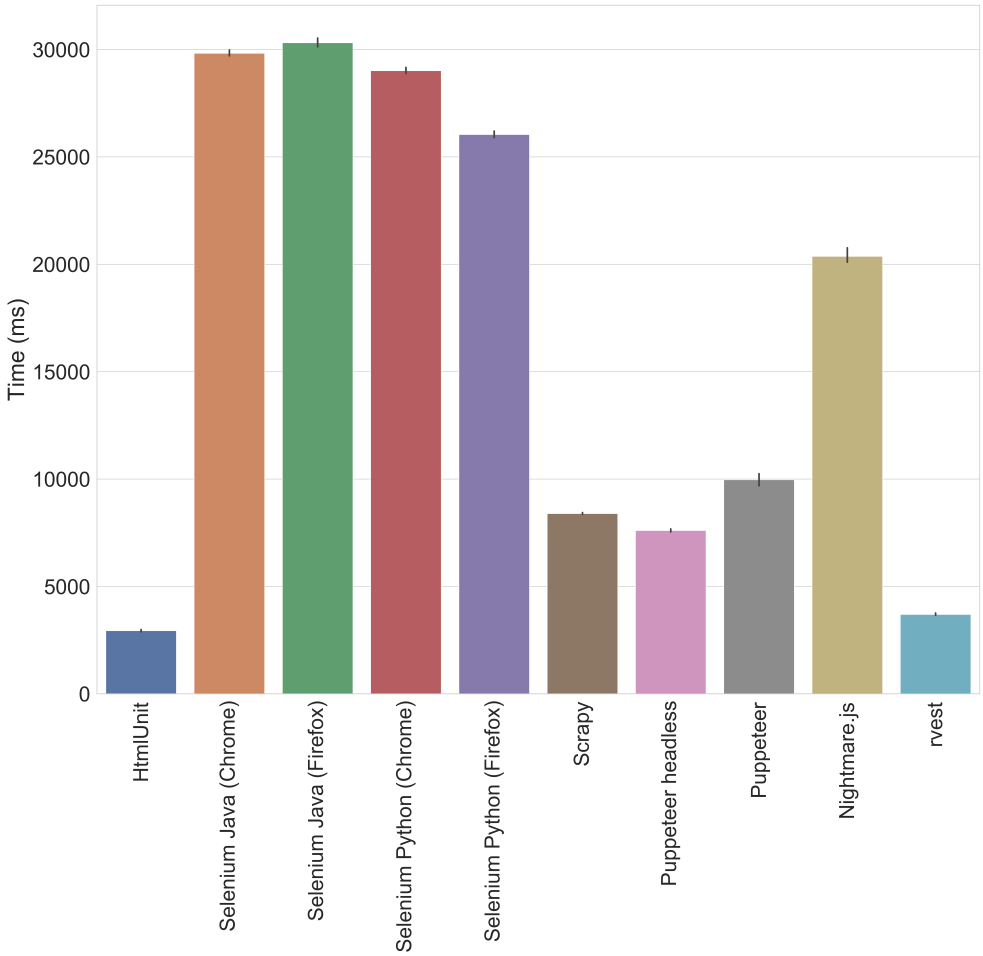


Figure 4.1: Bar graph detailing the run time for all 100 runs, for each tool

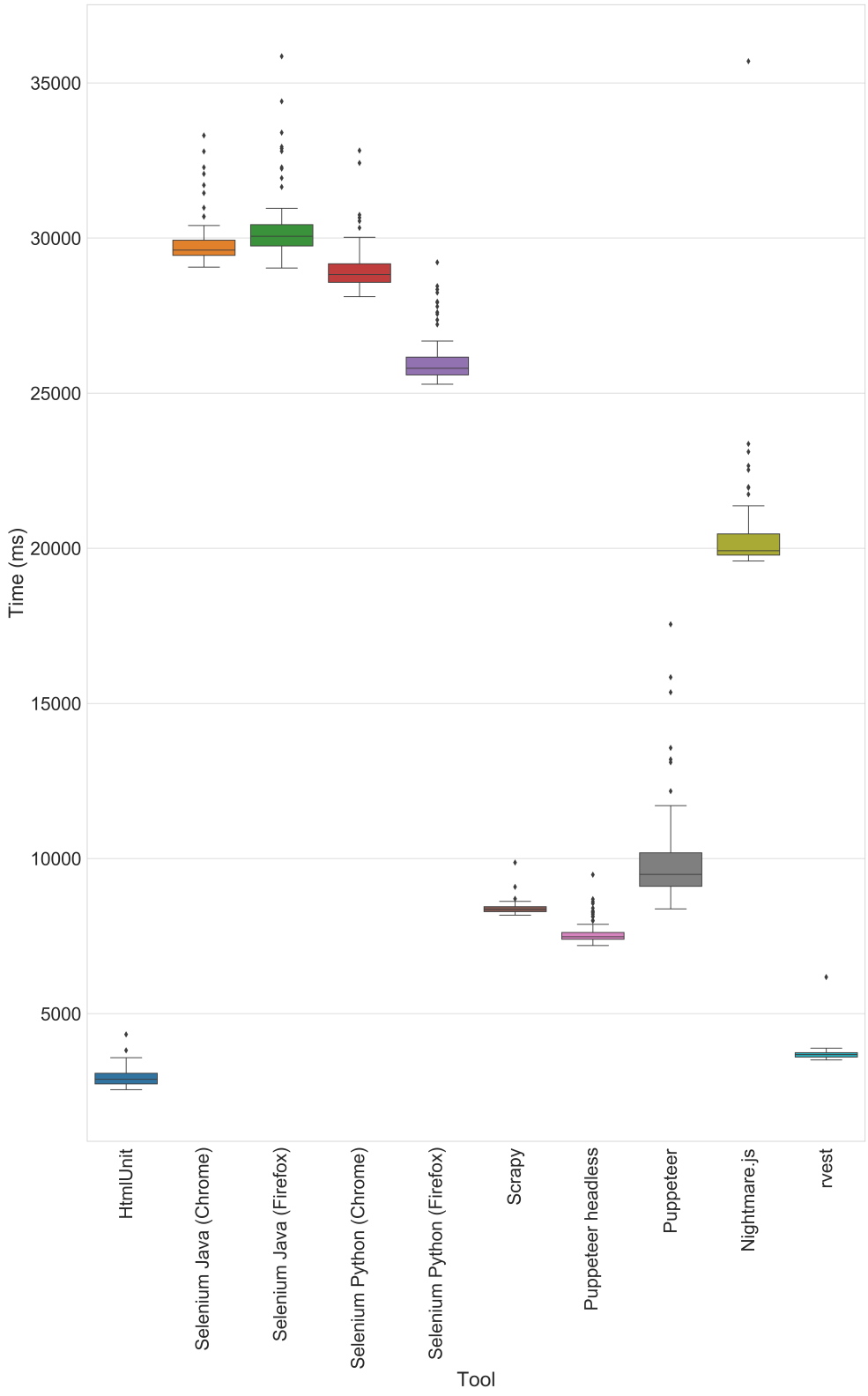


Figure 4.2: Box graph detailing the run time for all 100 runs, for each tool

Name	CPU (%)	Virtual memory (MB)	Real memory (MB)
Puppeteer	11.89	4520.11	36.49
Puppeteer headless	11.59	4521.48	37.34
Nightmare.js	1.87	4477.14	24.09
rvest	51.65	4346.32	100.81
scrapy	20.07	313.54	70.25
Selenium Python (Firefox)	5.33	140.68	18.44
Selenium Python (Chrome)	4.67	144.45	18.60
Selenium Java (Firefox)	24.43	10037.37	211.67
Selenium Java (Chrome)	25.42	10041.35	213.56
HtmlUnit	259.53	10098.01	204.11

Table 4.2: Table summarizing average performance statistics for each tool

Below are plots presenting CPU, virtual and real memory usage spent over time:

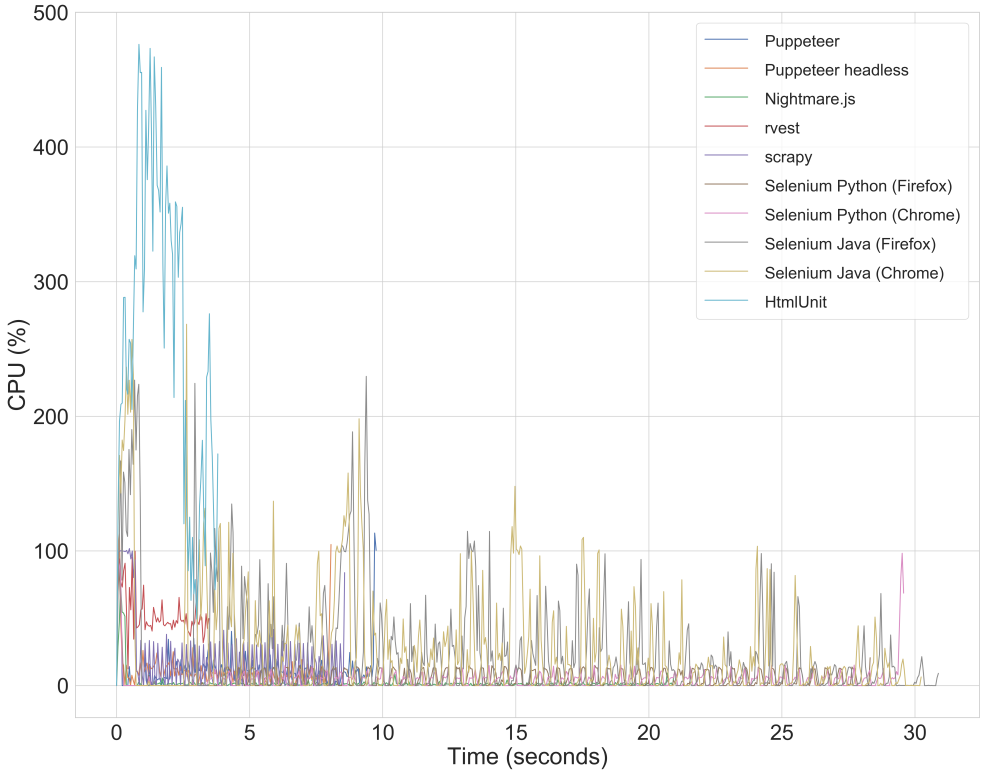


Figure 4.3: CPU (%) used for the book scraping task for each tool

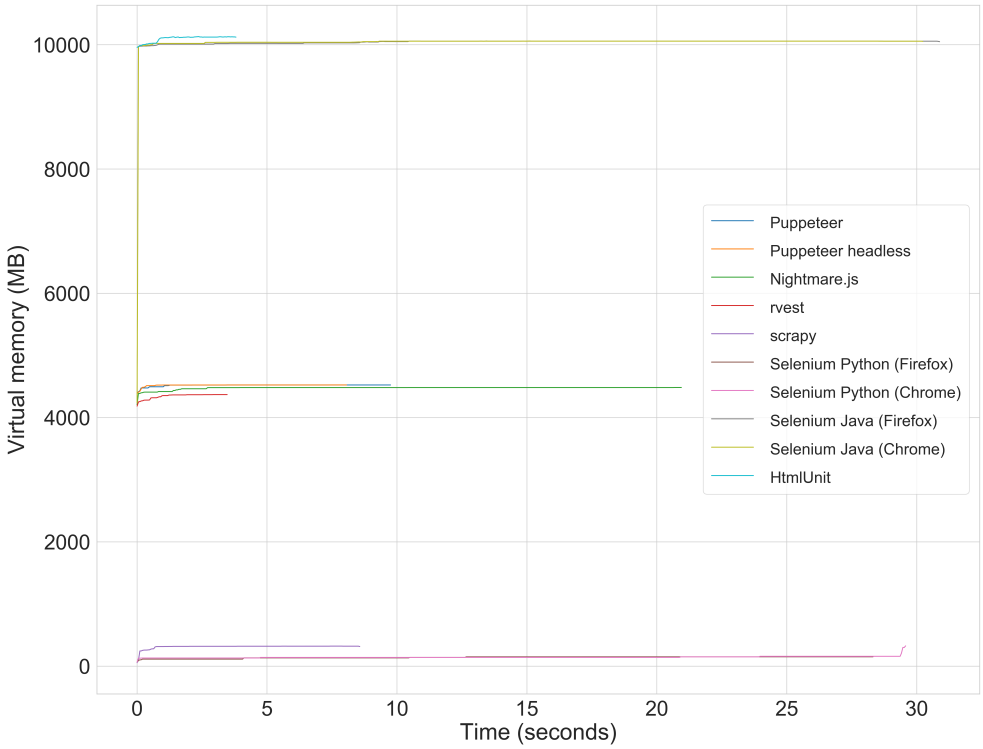


Figure 4.4: Virtual memory (MB) used for the book scraping task for each tool

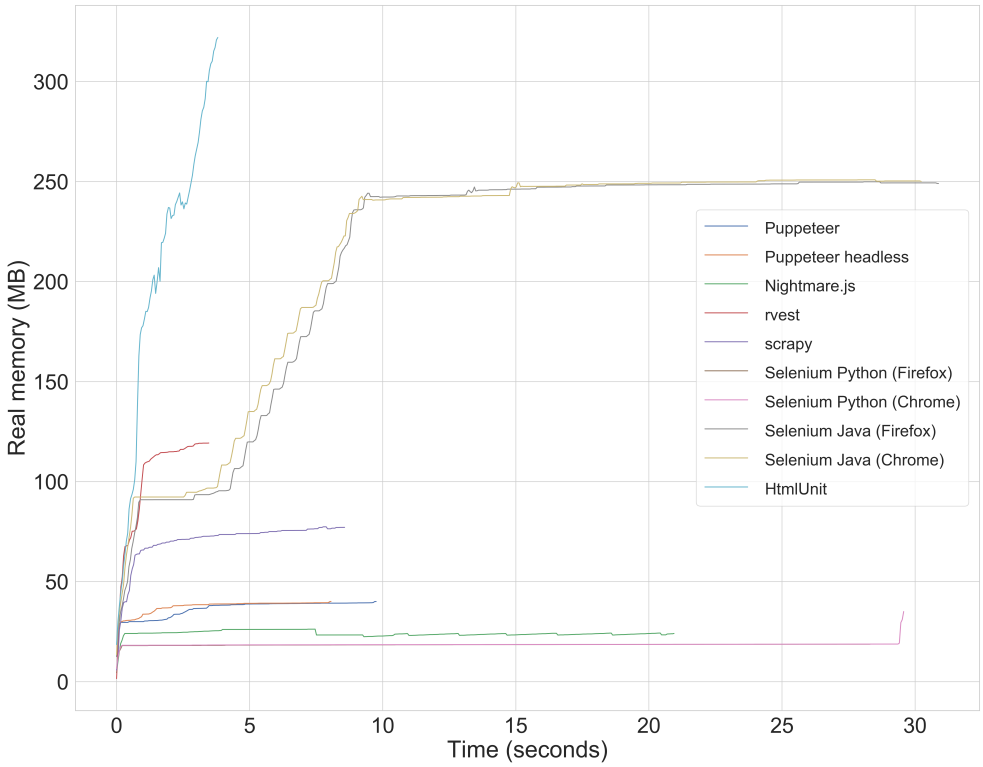


Figure 4.5: Real memory (MB) used for the book scraping task for each tool

4.2 Features

The following sections presents the different feature evaluations for each tool. First, a table summarizes whether the tasks were passed or not (Yes or No). Then, the implementations for each tool are discussed briefly. As a Python and Java version were implemented with Selenium, the syntax differs. Whenever a Selenium API function is used, both the Java and Python version are presented. A summary of the feature results is shown in Table 4.3.

Name	Capability to handle forms and CSRF tokens	AJAX and JavaScript support	Capability to alter header values	Capability to take screenshots	Capability to handle Proxy Servers	Capability to utilise browser tabs
Nightmare.js	Yes	Yes	Yes	Yes	Yes	No
Puppeteer	Yes	Yes	Yes	Yes	Yes	Yes
Selenium	Yes	Yes	Yes*	Yes	Yes	Yes
Scrapy	Yes	No	Yes	No	Yes	No
HtmlUnit	Yes	Yes	Yes	No	Yes	No
rvest	Yes	No	Yes	No	Yes	No

Table 4.3: Feature results for each tool. Yes indicates a task passed, No a task failed

* While Selenium managed to pass the header spoofing task, no way of altering headers currently exists in Selenium.

4.2.1 Capability to handle forms and CSRF tokens

The task was to fill a login form and successfully login. This included passing a CSRF token check. The implementations will now be discussed.

Nightmare.js

Filling forms was done via the `.type(selector, text)` function, that takes a CSS-selector and the text that the element matched should be filled

with. `.click(selector)` is then used to submit the form, once again taking a CSS-selector to find the button. `.wait(selector)` tells Nightmare to wait until an element is visible. This was utilized to wait for the page to reload after the submission.

Puppeteer

`.click(selector)` was used to click the form input fields, and then `keyboard.type(text)` to fill the text. Once the form has been submitted, `.waitForNavigation()` waits for the page change to happen.

Selenium

Finding the forms to fill was done by using `find_element_by_id(id)` (Python) or `findElementById(id)` (Java). Once the form fields are fetched, they are filled by calling `field.send_keys(text)` (Python) or `field.sendKeys(text)` (Java). The button is fetched and then clicked by `button.click()`.

Scrapy

Scrapy has a class dedicated to working with forms called `FormRequest`. In this case, the function `form_response` was used. The data is passed using a Python dictionary, with the key used to indicate which element to fill, and the value as the form text.

HtmlUnit

Similar to the Selenium version, the form fields were fetched by using method `getElementById(id)`. These are stored as `HtmlInput` objects, and can then be filled by calling `field.setValueAttribute(text)`. The button is fetched in similar fashion, but once clicked it returns a new `HtmlPage` object, which consists of the new page. As HtmlUnit is used within JUnit tests, an assertion was made to ensure that the login was successful, by checking the Login/Logout button state.

rvest

By using the rvest function `html_form` to store the form, it can then be filled by calling `set_values`. One then supplies name-value pairs. Once the form

is filled properly, `submit_form` is called. This function takes a webpage and a filled form object, and returns the webpage after submission.

4.2.2 AJAX and JavaScript support

For testing AJAX and JavaScript, each tool visited a webpage that load information dynamically, through JavaScript. In this case, information is regarding prior years Oscar nominees [1]. The task was to fetch each 2015 film title and its nominations. This involved first clicking a link for showing the 2015 results, which populate a table of information dynamically. Each implementation will now be presented briefly.

Nightmare.js

First, the link was clicked. Then, to wait for the data to populate, `.wait(selector)` tells Nightmare to wait until the element matching the selector is visible. To gather and return data, the `.evaluate()` function is used. It takes a function as the parameter. This function should return the data that is to be gathered. For example, the code snippet in Appendix A, Listing 1 is used to fetch the film titles.

Nightmare uses Promises [2], which represents an asynchronous operation that will eventually complete or fail. While it does not know the value upon creation, it promises that a value will be there in the future. The example in the Appendix returns a Promise, which then needs to be handled. To handle a Promise (get the result of the Promise), `.then(resFn)` is called, where `resFn` is a function that holds the value contained in the Promise (a list of film titles, in this case) as its argument. If a second value needs to be returned it can be done within the first Promise `.then` body. This is the case now; the list of nomination amounts also need to be gathered. This is shown in Appendix A, Listing 2. The whole Nightmare.js code snippet is shown in Appendix A, Listing 3.

[1] `https://scrapethissite.com/pages/ajax-javascript/`

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`

Puppeteer

Puppeteer is also based on using Promises. However, await [3] is used to handle the Promises. Using the `await` keyword before a Promise causes the program to wait until the Promise is ready and resolved before moving on to the next operation. Note that it is not required to use `await`, the Promises can be handled the same way as in the Nightmare.js version. However, all the examples and documentation is using `await`, and it was thus considered the natural way of usage.

Apart from using `await`, the implementation is not that different from Nightmare.js. The full code example is shown in Appendix A, Listing 4.

Selenium

First, the Java version is discussed. The page is loaded and the button is found and clicked by calling `findElementByClassName(name).click()`. To wait for the JavaScript to load data, WebDriverWait [4] is used. It takes as parameters the driver instance and an integer, indicating how long it should wait before throwing an exception. The `.until(res)` takes a function that tells the WebDriverWait what to wait for. If the element is found before the specified time, operation continues. Once the wait is over and the data is available, it is fetched and processed. The code snippet is shown in Appendix A, Listing 5.

The Python version is written in similar fashion. The difference is the way the data was processed. Instead of keeping the titles and nominations in separate lists, a Python dictionary was used to combine them. A WebDriverWait instance is used in the same way as the Java version. The code snippet is shown in Appendix A, Listing 6.

Scrapy

Scrapy is unable to handle JavaScript pages by default. However, there are alternatives. Scrapy and Splash [5] allows working with JavaScript. Splash [6] is a JavaScript rendering service.

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await`

[4] `https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/WebDriverWait.html`

[5] `https://github.com/scrapy-plugins/scrapy-splash`

[6] `https://github.com/scrapinghub/splash`

HtmlUnit

HtmlUnit needs to be explicitly told to handle JavaScript. This is done by `.setJavaScriptEnabled(true)`. It also needs an Ajax controller, which is set by `.setAjaxController(new NicelyResynchronizingAjaxController())`. The `waitForBackgroundJavaScript(10000)` [7] row tells HtmlUnit to wait 10 seconds for background JavaScript calls to load. If 10 seconds have passed and there are still JavaScript tasks running, the amount of said tasks are returned. When fetching the titles and nominations XPath is used, as CSS-Selectors have been mostly used in the other implementations. The code used is shown in Appendix A, Listing 7.

rvest

By default, rvest is unable to handle JavaScript pages. As with Scrapy, there are ways to accomplish this by including other libraries. RSelenium [8] is a way to use Selenium in R, which would allow loading JavaScript pages. A second alternative is using V8 [9], an embedded JavaScript engine for R.

4.2.3 Capability to alter header values

The task was to visit `https://scrapethissite.com/pages/advanced/?gotcha=headers`, which looks at the request and will reject it if it seems to stem from a machine and not a regular browser. Even if a tool passes the check without any configuration, ways to change headers are to be explored. While the task was simply to bypass the check, changing header values is the main evaluation goal.

Nightmare.js

Nightmare.js allows for adding custom headers in the `.goto()` function, by passing a JSON-object. The keys are the header field names and the values what they should be set to.

[7] `http://htmlunit.sourceforge.net/apidocs/com/gargoylesoftware/htmlunit/WebClient.html#waitForBackgroundJavaScript`

[8] `https://github.com/ropensci/RSelenium`

[9] `https://cran.r-project.org/web/packages/V8/index.html`

Puppeteer

To alter or set headers in Puppeteer, `page.setExtraHTTPHeaders()` is used. In the same way as Nightmare.js, It also takes a JSON-object.

Selenium

As was shown in the summarizing table, Selenium got a Yes* result. This means that it did pass the test by default, but Selenium offers no way to alter header values.

Scrapy

In Scrapy, headers can be changed in the `scrapy.Request()` function. By supplying a key-value dictionary with the header field names as key, headers can be altered.

HtmlUnit

The WebClient [10] object has a method, `addRequestHeader(header, val)`, that takes two strings as input. The first argument is the header name, and the second is its value.

rvest

In rvest, headers can be changed by supplying values to the initial `html_session()` [11] function. For example, the call shown in Listing 8 will alter the User-Agent field to look like it originates from a Mac running Firefox. This does require the library httr [12] to be imported, but seeing as it is a standard package included in a default R installation, it is considered valid. An example is shown in Appendix A, Listing 8.

4.2.4 Screenshot

The task was to navigate to `http://example.com/` and take a screenshot.

[10] `http://htmlunit.sourceforge.net/apidocs/com/gargoylesoftware/htmlunit/WebClient.html`

[11] `https://rdrr.io/cran/rvest/man/html_session.html`

[12] `https://cran.r-project.org/web/packages/httr/index.html`

Nightmare.js

Nightmare.js offer a `.screenshot()` function, allowing to save an image of the current view to file, given a file path.

Puppeteer

Puppeteer also offer a `.screenshot` function. It takes as argument a JSON-object describing its options. This includes ability to specify the page region to screenshot, type (jpeg or png), path, quality and more.

Selenium

For taking a screenshot in Selenium the `save_screenshot(path)` (Python) or `getScreenshotAs(type)` (Java) function is used. The Python version takes a file path as argument while the Java version takes a type [13] to save as. Types include base64, bytes and to file.

4.2.5 Capability to handle Proxy Servers

Testing proxy capability was done by configuring the tools so that the connection goes through a proxy server. Then, to confirm, `http://httpbin.org/ip` is visited to verify that the IP address has changed.

Nightmare.js

When instantiating Nightmare through its constructor, a JSON-object called switches can be supplied to include a proxy server. The key `proxy-server` is used to define the IP and port as value. This is shown in Appendix A, Listing 9

Puppeteer

For Puppeteer, a list of arguments can be passed to the `.launch()` function. This includes adding a proxy server. The code is shown in Appendix A, Listing 10.

[13] `https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/OutputType.html`

Selenium

There seem to be multiple ways to include proxies when working with Selenium. The methods used in this thesis are simply the ones that managed to work first.

Python

Selenium adds proxies differently depending on which WebDriver is used. For Python and Chrome, proxy settings can be added to `ChromeOptions`. Firefox, however, uses `DesiredCapabilities` for adding proxies. In Appendix A, Listing 11, the code for the Chrome version is shown. The same version using Firefox is shown in Appendix A, Listing 12.

Java

For Java, the usage is similar. Using ChromeDriver, `ChromeOptions` is also utilised. The code is shown in Appendix A, Listing 13. For using a proxy with Firefox, a Proxy object is instantiated and then added to `DesiredCapabilities`, as shown in Appendix A, Listing 14.

Scrapy

As with altering headers in Scrapy, proxies can also be added in the `scrapy.Request()` function. By passing a dictionary to the meta argument, the requests will go through the proxy server. This is shown in Appendix A, Listing 15.

HtmlUnit

A ProxyConfig [14] is used to set up proxies in HtmlUnit. It takes an IP and a port as arguments. The code is shown in Appendix A, Listing 16.

rvest

rvest allows adding proxies through the httr package, by calling `httr::set_config()`. A small yet functioning example is shown in Appendix A, Listing 17.

[14] `http://htmlunit.sourceforge.net/apidocs/com/gargoylesoftware/htmlunit/ProxyConfig.html`

4.2.6 Capability to utilise browser tabs

Working with tabs was tested by opening up two tabs. First, a static website that is hosted locally, consisting of two paragraphs: a username and password. The second tab then navigates to `http://testing-ground.scraping.pro/login`, where the username and password from the first tab was to be entered.

Puppeteer

Opening up a new tab in Puppeteer is simply done by calling `browser.newPage()` and storing it in a variable. This variable then works as the tab reference. The code snippet is shown in Appendix A, Listing 18.

Selenium

When looking for ways to open up tabs in Selenium, a few ways were tested. The first method that worked was telling the driver to execute a script, telling it to open a new window, which in this case is actually a tab. References to all active windows are then accessible through `driver.window_handles`. The WebDriverWait is used to make sure that the `driver.switch_to.window()` call is not done before the tab has loaded. Then the username and password is filled and submitted. The Python version is shown in Appendix A, Listing 19.

Java is done in similar fashion, handling the window opening, switching and waiting in the same way. The only real difference is the syntax. A code example is shown in Appendix A, Listing 20.

4.3 Reliability

In Table 4.4, the reliability statistics generated from the public GitHub repositories for each tool is presented. Because R is not a supported language, its data had to be manually collected. This is the reason for the rows missing; not everything was able to be manually gathered. Where possible however, different techniques were used. For the GitHub statistics, the GitHub repository API [15] was used. The cyclomatic complexity was calculated using `cyclocomp` [16]. Remaining data filled in the rvest column came from using various bash commands and Python scripts.

[15] `https://developer.github.com/v3/repos/`

[16] `https://cran.r-project.org/web/packages/cyclocomp/README.html`

Field	Nightmare.js	Puppeteer	Selenium	Scrapy	HtmlUnit	rvest
creation date	2014-04-05	2017-05-09	2013-01-14	2010-02-22	2018-09-02	2014-07-23
stars	17052	47193	13794	32015	112	1045
watches	367	1086	1273	1840	9	97
forks	1042	4170	4701	7584	20	265
contributors	111	196	423	308	5	20
languages	JavaScript	JavaScript	Java	Python	Java	R
open issues	127	323	541	770	9	19
closed issues	1407	3869	6458	2925	13	170
last year commit #	0	547	1447	218	757	
Propagation Cost	3.2	0.5	8.0	8.9	38.5	
Architecture Type	Core-Periphery	Hierarchical	Hierarchical	Hierarchical	Core-Periphery	
Core Size	7.5%	0.0%	2.1%	3.4%	37.5%	
Lines of Code (LOC)	4277	20869	97756	26403	310193	1680
Comment/Code Ratio	17%	25%	38%	13%	53%	141%
Classes	9	130	2130	774	2325	
Files	40	239	1451	293	1927	12
Median LOC per File	3	24	34	53	34	51
Files >200 LOC	4	35	106	32	309	3
Functions >200 LOC	5	16	0	0	2	0
Cyclomatic complexity >50	4	30	490	26	504	0
Avg cyclomatic complexity	49.3	36.15	44.95	78.3	77.7	12.8
Avg cycl. comp. density	0.82	0.23	0.19	0.19	0.39	0.29

Table 4.4: Reliability metrics for each tool gathered with `CBRI-Calculation` and `gitinspector`

4.4 Ease of use

In this section, results regarding ease of use are presented. First, the Likert scale results are listed. Then, the dependencies, installation process, official tutorials and documentation are presented. In Table 4.5 the different Likert survey results are presented. The results go from 1, Strongly Disagree to 5, Strongly Agree. Figure 4.6 show results in the form of a bar graph, where each category score is summarized for easy comparison between the different tools.

Statement	Nightmare.js	Puppeteer	Selenium	Scrapy	HtmlUnit	rvest
The installation method is easy to find	5	5	5	5	5	5
The installation process is simple to follow	5	5	5	5	3	2
No other resources were needed	5	5	4	5	5	4
The get started section is easy to follow	5	5	5	5	5	4
There are examples showcasing different use cases	4	5	4	5	5	4
There are links or references to external resources	5	5	5	2	1	1
API references are well described	5	5	4	5	5	5
The documentation contain examples	5	5	5	5	5	5
The documentation is easy to navigate	4	5	3	5	4	5

Table 4.5: Ease of use evaluation results for each tool, based on the Likert survey

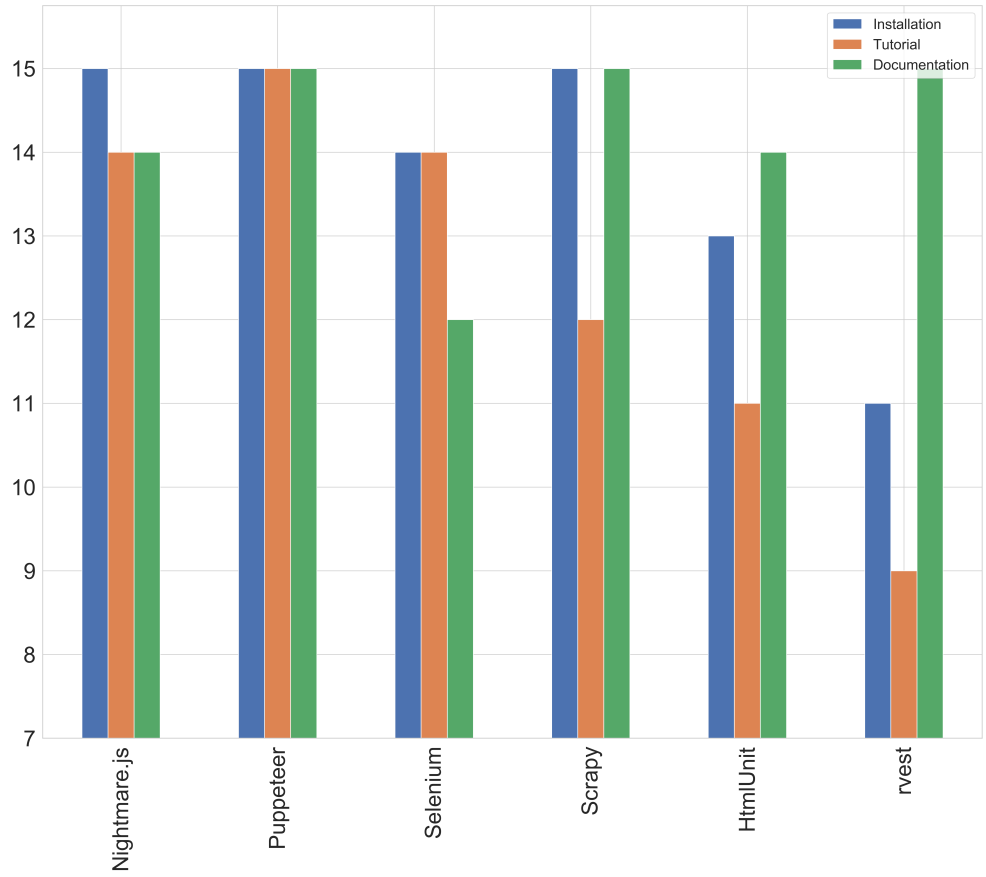


Figure 4.6: Bar graph presenting the ease of use evaluation results for each tool

4.4.1 Dependencies

Table 4.6 presents the dependencies required for each tool to run.

Name	Dependencies
Nightmare.js	Node.js
Puppeteer	Node.js
Selenium	WebDriver, Python/Java
Scrapy	Python
HtmlUnit	JUnit, Java
rvest	R

Table 4.6: Each tool and their dependencies

4.4.2 Installation

In this section the different installation processes are presented. This includes the steps required to install the tools and how to import the tool to code. The results are shown in Table 4.5. As a reminder, these are the survey statements:

Q1: The installation method is easy to find

Q2: The installation process is simple to follow

Q3: No other resources were needed

Nightmare.js

Nightmare.js, along with other JavaScript libraries, offer installation through the Node.js package manager `npm` [17]. When installing, navigate to the folder the project is to be used in and run `npm install nightmare`. This command will fetch Nightmare.js and its dependencies. Once installed, Nightmare.js can be accessed by importing it. It is done by adding the following line:

```
const Nightmare = require('nightmare')
```

The Nightmare API can now be accessed and utilized via the `Nightmare` variable. It uses an instance of `Electron` as the browser, which is installed automatically.

[17] `https://www.npmjs.com/`

Puppeteer

Similar to Nightmare.js, Puppeteer is also installed by using `npm`. `npm install puppeteer` will fetch and install Puppeteer, along with a version of Chromium, which is the browser that Puppeteer utilizes. Once downloaded, it is imported in code by:

```
const puppeteer = require('puppeteer');
```

The `puppeteer` variable is now used to access the API.

Selenium (Python)

`pip` [18], the package manager for Python, can be used to install Selenium. By running the command `pip install selenium` Selenium is downloaded. However, it does not come with a browser to control, and thus requires downloading a WebDriver. For this thesis, the Firefox geckodriver and Google Chromes Chromedriver were used, but options such as Safari [19] and Edge [20] also exist.
The code below shows how to import the Selenium API:

```
from selenium import webdriver
```

To instantiate the webdriver it needs to know which browser and where to find the WebDriver. In the following example, the Firefox geckodriver is used, but it can be swapped out for any of the browsers mentioned earlier. In the Firefox constructor the executable path to the geckodriver is passed:

```
browser = webdriver.Firefox(
executable_path='/Users/emilpersson/Downloads/geckodriver')
```

The `browser` variable now expose the Selenium API.

Selenium (Java)

There are more than one way to install Selenium when working with Java. There are also multiple ways of working with Java in general. In this case, Intellij [21] and Maven was used. Maven [22] is a dependency manager for Java projects. Installing Selenium was done by including the following code to the Maven file, `pom.xml`:

[18] https://pypi.org/project/pip/
[19] https://www.apple.com/safari/
[20] https://www.microsoft.com/en-gb/windows/microsoft-edge
[21] https://www.jetbrains.com/idea/
[22] https://maven.apache.org/

```
<dependency>
        <groupId>org.seleniumhq.selenium</groupId>
                <artifactId>selenium-java</artifactId>
        <version>3.141.59</version>
</dependency>
```

Intellij then notices the change and prompts to import the changes, downloading and adding Selenium as a dependency. Importing and using Selenium is similar to the Python version. First, the desired WebDriver is imported

```
import org.openqa.selenium.firefox.FirefoxDriver;
```

Then the system needs to know where it can find the driver (Firefox geckodriver, in this example)

```
System.setProperty("webdriver.gecko.driver",
"/Users/emilpersson/Downloads/geckodriver");
```

Finally, a driver can be instantiated

```
FirefoxDriver driver = new FirefoxDriver();
```

Scrapy

Scrapy can be installed by using `pip` with the command `pip install Scrapy`. Once installed, a Scrapy project can be generated by using the `scrapy startproject <name>`, which will create a project and generate a standard file structure:

```
<name>
├── __init__.py
├── items.py
├── pipelines.py
├── settings.py
├── middlewares.py
└── spiders
    └── <define spiders here>
```

In Scrapy, scrapers (or crawlers, as it also supports web crawling) are called spiders. These are defined in the spiders folder and are given unique names, to enable running a specific spider.

HtmlUnit

The official installation section is short, detailing two ways to install HtmlUnit. The first way links to all the required jar files and tells the user to put them on

the class path. The process is not detailed further, and is the reasoning behind the Q2 score. The second method is by using maven. This was the method used for this thesis and worked the same way as the installation for Selenium. Once installed, the WebClient used to control the scraping procedure can be imported:

```
import com.gargoylesoftware.htmlunit.WebClient;
```

and then instantiated:

```
WebClient client = new WebClient();
```

rvest

Installing rvest is done by opening an interactive R shell in a terminal and running the command:

```
install.packages("rvest")
```

Once installed and, rvest can be imported inside the code by:

```
library('rvest')
```

External resources had to be looked up, as the first attempt, when following the official documentation, failed. This is likely due to the authors inexperience with R as a language; it was not stated that the `install.packages("rvest")` call was to be done in an interactive R shell.

4.4.3 Tutorials

In this section the results from evaluating the official tutorial or get started sections are presented. The following statements have been evaluated:

Q1: The get started section is easy to follow

Q2: There are examples showcasing different use cases

Q3: There are links or references to external resources

Nightmare.js

A Usage section describes the workflow from installing to running a small example. There are multiple links to external resources.

Puppeteer

Also contains a Usage section that shows how to install and run a few examples. Links to more examples and an external list of community resources are present.

Selenium

The Selenium introduction jumps straight into showcasing an example. There are small code snippet examples showcasing basic functionality such as how to fetch elements, filling forms, handling cookies and more. Every code snippet presented can be toggled between Java, C#, Python, Ruby, Perl and JavaScript. Links to external resources exist.

Scrapy

Scrapy contains the most in depth and thorough tutorial section yet. It includes the installation, creating a spider, extracting and storing scraped data, following links and more. Multiple examples are shown. The only external link that was found pointed to a GitHub repository, where two different Scrapy spiders are defined. One spider is showcasing the use of CSS-Selectors and the other XPath expressions. The repository's purpose is to give the user something to experiment and play around with.

HtmlUnit

Showcases a get started section with multiple examples. Contains sections on different areas including working with the keyboard, tables, frames, JavaScript and more. No external resources are found.

rvest

rvest presents an overview including one example, how to install and brief introductions of the core functions. There are three examples in the official GitHub repository, but these are not described or introduced in the tutorial.

4.4.4 Documentation

Q1: API reference are well described

Q2: The documentation contain examples

Q3: The documentation is easy to navigate

Nightmare.js

For Nightmare.js, the documentation is located in the GitHub repository readme file [23]. API references are described well and contain examples where necessary. As the documentation is simply listed in a markdown file, it does not offer search functionality. However, searching for keywords using the browsers default search functionality works fine.

Puppeteer

A well formed documentation [24], every parameter and return value is described well. Code examples are present for most API references. For navigation, there is a search bar that can be used, including suggestion drop-down functionality.

Selenium

The Java [25] and Python [26] of Selenium have different documentations, however, they are very similar in structure. The Java documentation page is generated by javadoc [27], a tool used to build documentation source code. Arguments and return values are defined for the core parts of the API, but many methods are not explained at all. The same goes for showing examples; many methods are left out of the API documentation. Examples and common use cases are instead presented at the Selenium webpage [28], which is considered part of the documentation. Only the Python API reference offers search functionality.

Scrapy

The Scrapy documentation [29] is very thorough, with each method and class described well. Every argument and return value is sufficiently documented. There are many examples showcasing the different methods, classes and key concepts. A search form can be used to navigate the documentation.

[23] https://github.com/segmentio/nightmare/blob/master/Readme.md
[24] https://pptr.dev/
[25] https://seleniumhq.github.io/selenium/docs/api/java/
[26] https://seleniumhq.github.io/selenium/docs/api/py/
[27] https://www.oracle.com/technetwork/java/javase/tech/index-137868.html
[28] https://www.seleniumhq.org/docs/
[29] https://docs.scrapy.org/en/latest/

HtmlUnit

The HtmlUnit API reference is, as with Selenium, generated by javadoc. Every argument and return value is explained briefly. While the API reference does not show examples, the HtmlUnit webpage [30] offer examples on many different use cases. There is no search functionality for neither the API reference nor the webpage, but the webpage is relatively easy to navigate using the sidebar.

rvest

rvest has a well formed, concise, yet sufficient documentation [31]. The functions can be searched by text and each function is presented in terms of usage, arguments, return value and examples. It even allows for posting your own code examples, that would land under the functions "Community examples" section.

4.5 Discussion

In this section, results from all four evaluation areas are discussed.

4.5.1 Performance

When looking at the performance results, there were some large differences. In terms of run time, HtmlUnit and rvest are the fastest, averaging 2.9 and 3.7 seconds respectively. This is a large difference to the slower Selenium and Nightmare.js versions, whose operation took around 30 and 20 seconds. Puppeteer and Scrapy are relatively close. Puppeteer is unique in the sense that it allows for hiding the browser, which showed to reduce the time taken by around 22% (0.2186) compared to running it with the browser shown. Selenium versions were all relatively similar in run time, but a Python version using the Firefox driver is the fastest out of the four. Headless versions are, as expected, faster. This is likely due to the avoidance of having a web browser render HTML, CSS and JavaScript. The CPU usage varies quite a lot. Nightmare.js only uses 1.87% while HtmlUnit utilizes multiple cores, totalling 259.53%. This is likely a factor to HtmlUnit being the fastest. Selenium cases show that Java uses more resources than its Python counterpart: 5 times more CPU power and almost 100 times more virtual and physical memory. All of the Java

[30] `http://htmlunit.sourceforge.net/`

[31] `https://www.rdocumentation.org/packages/rvest/versions/0.3.3`

tools (Selenium and HtmlUnit) use similar amounts of memory. However, the Python tools (Selenium and Scrapy) differ: Scrapy uses around 4 times the CPU power, and more virtual and real memory. rvest uses the second most CPU power but the second least virtual memory. Putting Puppeteer in headless mode had no significant reduction in either CPU or memory area.

4.5.2 Features

Puppeteer is the winner in terms of features, passing them all while providing smooth implementations. rvest and Scrapy were the worst performers, managing to pass half of the feature tasks. No tool claimed to support a feature that then turned out unsupported. There are clear differences in the amount of code required to perform the tasks. This is likely due to the difference in programming languages, as the Java based tools required more code compared to the tools based on the other lannguages. Puppeteer and Nightmare.js are both based on using JavaScript Promises. They do however treat them differently. Puppeteer promotes using async / await, for a more linear code flow, avoiding the use of nested `.then()` statements that do occur in Nightmare.js. When comparing Scrapy to Selenium using Python, Scrapy consists of a much more structured file and code framework. This can be a drawback when wanting to implement something quick, as it requires more time for understanding and navigating the different files and modules. As such, the feature tests performed does not do Scrapy any favours, as they are one time implementations. An important point when comparing implementations is the tab feature. Both Selenium and Puppeteer managed to pass the task, but the Selenium version feels less stable and no official way of working with tabs was found. The Internet had to be searched quite a bit to find a functioning way. Puppeteer on the other hand took a very simple and straight forward approach.

4.5.3 Reliability

When looking at the reliability results, Puppeteer is the most starred repository, followed by Scrapy. This is despite Puppeteer being a relatively young project (created in 2017 compared to Scrapy, created in 2010). While the first release of HtmlUnit happened 2002, it was just recently opened up as a public repository in 2018. This is likely the reason for its low amount of GitHub metrics (stars, watches, forks, contributors and issues); it has primarily been developed outside of GitHub and probably uses some other form of version tracking system. As for the code quality metrics, HtmlUnit is by far the largest

project, having over 200 000 lines of code more than Selenium, which is the second largest. As can be expected of these older, larger projects, they also have the most files and classes. rvest has more comments than code (141%) but is a very small project, utilizing already existing R libraries. As such, it also has a very low average cyclomatic complexity and median lines of code per file. Scrapy and HtmlUnit have the highest average cyclomatic complexity values. If instead the cyclomatic complexity density is looked at, which helps level the playing field for projects with large files, Nightmare.js is the clear leader. HtmlUnit has the highest core size as well as the highest propagation cost, which has shown correlated to maintenance difficulty.

4.5.4 Ease of use

As can be seen from the ease of use results, Puppeteer received a perfect score. JavaScript and Python tools are generally installed through the languages package manager. For JavaScript this is node and for Python it is pip. These package managers provide a very quick and easy installation process. This can be seen from the results: Nightmare.js, Puppeteer and Scrapy received a perfect score for installation. Selenium would also have gotten a perfect score if only the Python version was evaluated. The more recent tools generally provide what feels like a more modern and efficient documentation with good examples, compared to the documentation generated by tools such as javadoc. As mentioned in the installation section for rvest, it was the only process that actually posed some issues. The issue however is likely due to the authors inexperience when it comes to the R language and its ecosystem. This needs to be taken into account, it is likely that a person previously exposed to R would have managed to install it without any problems.

While the evaluation results clearly indicate Puppeteer being the best, such conclusions are hard to draw.

4.5.5 Evaluation framework quality

Overall, the evaluation framework is considered to be of good quality. The criteria used in the evaluation framework are performance, features, reliability and ease of use. Initially, it was suggested to also include evaluation on future roadmap and maintenance. Future roadmap was considered to be overlapping enough with reliability, and also hard to evaluate. If a tool is considered reliable, it is likely that it will continue improving and growing as time passes. Maintenance was omitted due to the likely difficult way of evaluating

it. One would have to either develop two versions of a simulated webpage, where one version is altered and then try to adapt the tool implementation to work with the new version of the webpage. Not only would it take a lot of time to develop these webpages, but no reasonable way of getting concrete results that would fit the time span was found. The four criteria that ended up being used are considered as relevant for web scraping tools. Performance is a key aspect of any software. In the world of web scraping there are various types of tasks. This is where the different features distinguish what a tool is capable of. For example, a very important feature that could make or break a web scraper is the support of scraping JavaScript pages. Even though a webpage might not rely on JavaScript dynamically loading its data at the time of development, it is possible that it might migrate to this form in the future. If a scraper supports JavaScript, it could still be possible to change the scraper to a working version. If it does not support JavaScript at all, a brand new, different, web scraper would have to be developed. Reliability is thought of as a way of providing stability and longevity, something that is considered important when investing time and money in a tool to use. Ease of use is also considered important not only as a good way to get started quickly with a tool, but also having a enjoyable work flow, as the tool documentation is likely to be frequently visited.

However, evaluating software is not an easy task. It is considered an impossibility to objectively conclude whether a software tool meets desired specifications or goals, as these may be interpreted in different ways. Qualitative parts account for a large part of software quality, and these are considered impossible to measure directly. While certain quantitative aspects may indicate the level of a qualitative aspect, they cannot be specified in an unambiguous way [25]. When performing some of the evaluation, mainly the ease of use par, a lot of the results were based on the authors interpretation. For example, the documentation evaluation is just based on personal preference, and is thus highly subjective.

If a minimal set is to be suggested, to save time, it would include features and ease of use. The reliability part is somewhat up for interpretation. Factors such as code complexity are not easily distinguishable, especially between different programming languages. They might offer the possibility about reasoning and comparing software projects reliability, but these factors are simply an indication, and might not actually be as important. Similarly, performance may not be a factor that is worth basing the tool choice around, as even large operations are relatively quick (scraping information on 1000 books took at most about 30 seconds). The feature set however could be vital to support the

task at hand, and could thus allow or disallow the use of a specific tool based on the task. Ease of use is considered not only important in getting started, but the documentation is likely going to be used throughout the projects life span.

4.5.6 Theoretical impact

The framework's use of the code quality statistics in conjunction with GitHub repository statistics could be a way to theorize about software reliability, not only for web scraping tools. Other types of software are built in similar fashion. However, taking this approach does require that the participant projects are open source and has public GitHub repositories. The code quality statistics reason about the state of the actual code, while the GitHub statistics allows for an indication on popularity, developer satisfaction, longevity and usage. If the code quality was the sole resource, a project could be very well built in terms of code quality, but have no interest from actual developers and users.

Similar could be argued for the use of both quantitative and qualitative aspects over all. The quantitative aspect makes for comparable results that are easy to distinguish. The qualitative are by definition more subjective to the user, but ignoring them completely would miss out on results that could be very beneficial. For example, imagine one of the tool having good performance and feature results but a non existing (or lacking) documentation. While the functionality and speed is there, having to learn the tool without a good documentation could be tedious and time consuming, and while a good documentation is subjective, it could still function as an indication on its overall quality.

4.5.7 Practical implications

By utilising the framework developed, web scraping tools can be compared. This provides utility for others that are interested in comparing web scraping tools, and could impact on a better decision on what tool to use in a given project, which in turn can save time and money investment. One of the research papers presented in the previous work section revolved around using web scraping to gather data for psychology research. In this project, the authors chose to use Scrapy as their web scraping tool. However, it indicates as if they had no previous experience with using Python, as they all had to first take a Python course. Given this information, the evaluation framework results could have been useful. For example, rvest could have been a more suitable choice based on its performance speed, as the task is a one time data gather-

ing task. The authors mention that R is commonly used by psychologists, and perhaps some knowledge existed within the group [16].

4.5.8 Future work

A suggestion on future work is to run the unselected tools (the tools that were discarded in the initial selection process) through the framework. This would not only cover more web scraping tools, but also provide a way to validate the framework when used by others.

Using Machine Learning to analyse software repositories would have been beneficial [28] [14] and is thus recommended for future work. This could eliminate interpretation errors, where some aspects might indicate certain conclusions, which may be untrue. Because of time restriction and the authors lack of knowledge in the area, it was not utilised in this thesis.

A different approach for a step forward would be to investigate the different browser drivers and compare them. Some comparisons can be made from the results in this thesis, as two different browser drivers are used in the Selenium cases. A project solely focused on comparing the most popular browser drivers in terms of performance could prove important. These browser drivers are ways to control the active web browsers used today, which are constantly being improved and developed, and is likely to remain a popular way of accessing the internet in the future.

Chapter 5

Conclusions

The purpose with this thesis was to evaluate state of the art web scraping tools based on established software evaluation metrics. A recommendation based on state of the art web scraping tools can then be presented, with these different software metrics in mind. As a reminder, the research question is defined as follows

> *With respect to established metrics of software evaluation, what is the best state of the art web scraping tool?*

Regardless of how specific the methodology would have been, there is no way to conclude a definite 'best' tool. Many aspects are highly subjective, and can be interpreted in different ways [25].

However, some recommendations can be made. These are based on bias in terms of desired programming language, whether the user has more experience or preference within a certain language. For example, in one of the papers presented in the previous work section, psychologists used web scraping to gather data for research. The authors claim that proficiency with Python and R is common in the psychology research field [16], and thus recommending a tool that uses, for example, Java, would not make sense, especially for a one-time web scraping scenario. The time spent learning Java and investigating Java related bugs in the code would out weigh the performance gain from using a slightly slower tool in a known language.

One more aspect worth discussing is how different developers may value different aspects. For example, developer A may not need or want to use the documentation as frequently as developer B, but instead needs support for a feature that developer A does not. As such, if there are specific requirements or preferences, looking at the different results related to those is likely to be more beneficial.

5.1 Recommendations

Based on the results gathered from the evaluation framework, Puppeteer is considered the most complete tool.

If language preference is absent, or JavaScript is the preferred language, Puppeteer is recommended.

If Python is preferred and speed is a priority, and JavaScript support is not of importance, Scrapy is suggested. If any of these factors are not true, Selenium is recommended.

For Java, if speed is important, HtmlUnit is recommended. If not, Selenium is suggested.

If R is the programming language of choice, rvest is a good choice, unless JavaScript pages are to be accessed.

Selenium supports multiple languages, and is thus a good choice if one of the other supported languages is of preference. These include: C#, Haskell, Objective-C, Perl, PHP, Ruby, Dart, Tcl and Elixir [1]. Note that a few of these are not developed by Selenium themselves.

5.2 Limitations

The main limitations for this thesis are the subjective parts of the results. Even though the author has deep dived into the world of web scraping, thing such as rating the documentation, tutorial and installation are highly subjective tasks. To get a better, more quantifiable result in these areas could involve preforming an interview survey of some sort. Instead of just the author using and rating these parts, having multiple people with different backgrounds involved would have been beneficial. For example, these people could have installed and implemented a web scraping task with the different tools, using the official tutorials and documentation. The same Likert scale survey could then be used to gather multiple results to analyse. This would generate better and more general results.

The book scraping task already performed is basically a data-gathering only problem, with a little bit of navigation. A second, more extensive and realistic performance task would have been useful. Something more advanced, that consist of characteristics that might appear in a real world project involving web scraping. This would be a good addition to the evaluation framework.

[1] `https://www.seleniumhq.org/download/#thirdPartyLanguageBindings`

5.3 List of contributions

- A survey of state of the art web scraping tools
- Applying the evaluation framework on web scraping tools, providing recommendations

Bibliography

[1] John C. Mitchell Adam Barth, Collin Jackson. Robust defenses for cross-site request forgery. pages 75–88, 01 2008.

[2] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. Co-evolution of project documentation and popularity within github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 360–363, New York, NY, USA, 2014. ACM.

[3] Geoff Boeing and Paul Waddell. New insights into rental housing markets across the united states: Web scraping and analyzing craigslist rental listings. *Journal of Planning Education and Research*, 37(4):457–476, 2017.

[4] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, Oct 2016.

[5] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *CoRR*, abs/1811.07643, 2018.

[6] Osmar Castrillo-Fernández. Web scraping:applications and tools. European Public Sector Information Platform, 2015.

[7] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[8] Fazal e Amin and Alan Oxley Ahmad Kamil Mahmood. Reusability assessment of open source components for software product lines. 2011.

[9] Fatmasari Fatmasari, Yesi Kunang, and Susan Purnamasari. Web scraping techniques to collect weather data in south sumatera. 12 2018.

[10] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[11] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, Dec 1991.

[12] Daniel Glez-Peña, Anália Lourenco, Hugo López-Fernández, Miguel Reboiro-Jato, and Florentino Fdez-Riverola. Web scraping technologies in an api world. pages 788–796, 2013.

[13] Neal Haddaway. The use of web-scraping software in searching for grey literature. *Grey Journal*, 11:186–190, 10 2015.

[14] Stanislav Kasianenko. *Predicting Software Defectiveness by Mining Software Repositories*. PhD thesis, 2018.

[15] Vlad Krotov and Leiser Silva. Legality and ethics of web scraping. 09 2018.

[16] Richard Landers, Robbie Brusso, Katelyn Cavanaugh, and Andrew Collmus. A primer on theory-driven web scraping: Automatic extraction of big data from the internet for use in psychological research. *Psychological Methods*, 21, 05 2016.

[17] J. Ludwig, S. Xu, and F. Webber. Compiling static software metrics for reliability and maintainability from github repositories. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 5–9, Oct 2017.

[18] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[19] Andreas Mehlführer. Web scraping: A tool evaluation. Technische Universität Wien, 2009.

[20] Yolande Neil. Web scraping the easy way. Georgia Southern University, 2016.

[21] Joacim Olofsson. Evaluation of webscraping tools for creating an embedded webwrapper. page 44. KTH, School of Computer Science and Communication (CSC), 2016.

[22] Jiantao Pan. Software reliability. Carnegie Mellon University, 1999.

[23] Federico Polidoro, Riccardo Giannini, Rosanna Lo Conte, Stefano Mosca, and Francesca Romana Rossetti. Web scraping techniques to collect data on consumer electronics and airfares for italian hicp compilation. 2015.

[24] S. Sirisuriya. A comparative study on web scraping. *International Research Conference, KDU*, 8:135–139, 11 2015.

[25] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.

[26] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[27] Olav ten Bosch. Uses of web scraping for official statistics. Statistics Netherlands.

[28] D. Zhang, S. Han, Y. Dang, J. Lou, H. Zhang, and T. Xie. Software analytics in practice. *IEEE Software*, 30(5):30–37, Sep. 2013.

[29] Bo Zhao. Web scraping. pages 1–2. Oregon State University, 2017.

Appendix A

Code examples

A.0.1 AJAX and JavaScript support

Nightmare.js

```
.evaluate(() => {
  const titles =  [...document.querySelectorAll('.film-title')]
    .map(element => element.textContent)
    .map(element => element.trim())
  return titles;
})
```

Listing 1: Code for fetching film titles from a table in Nightmare.js

```
.then((titles) => {
  nightmare.evaluate(() => {
    const nominations =
      [...document.querySelectorAll('.film-nominations')]
      .map(element => element.textContent)
    return nominations
  })
.end()
.then((nominations) => {
  console.log(titles)
  console.log(nominations)
  })
})
```

Listing 2: Code for fetching film nominations from a table in Nightmare.js

```
nightmare
  .goto('https://scrapethissite.com/pages/ajax-javascript/')
  .click('.year-link')
  .wait('.film-title')
  .evaluate(() => {
    const titles =
      [...document.querySelectorAll('.film-title')]
        .map(element => element.textContent)
        .map(element => element.trim())
    return titles;
  })
  .then((titles) => {
    nightmare.evaluate(() => {
      const nominations =
        [...document.querySelectorAll('.film-nominations')]
          .map(element => element.textContent)
      return nominations
  })
  .end()
    .then((nominations) => {
      console.log(titles)
      console.log(nominations)
    })
})
```

Listing 3: Full code example for fetching JavaScript loaded data in Nightmare.js

Puppeteer

```
const browser = await puppeteer.launch({ headless: false });
const page = await browser.newPage();
await page.goto(
'https://scrapethissite.com/pages/ajax-javascript/');
await page.click('.year-link')
await page.waitForSelector('.film-title');
await page.waitForSelector('.film-nominations');

const titles = await page.evaluate((sel) => {
  return [...document.querySelectorAll('.film-title')]
    .map(e => e.textContent)
    .map(e => e.trim())
});

const nominations = await page.evaluate((sel) => {
  return [...document.querySelectorAll('.film-nominations')]
    .map(e => e.textContent)
});

console.log(titles)
console.log(nominations)

await browser.close();
```

Listing 4: Full code example for fetching JavaScript loaded data in Puppeteer

```
System.setProperty("webdriver.chrome.driver",
"/Users/emilpersson/Downloads/chromedriver");
ChromeDriver driver = new ChromeDriver();
driver.get(
"https://scrapethissite.com/pages/ajax-javascript/");

driver.findElementByClassName("year-link").click();
WebDriverWait wdw = new WebDriverWait(driver, 10);
wdw.until((d) -> d.findElement(By.className("film-title")));

final List<WebElement> titles =
driver.findElementsByClassName("film-title");
final List<WebElement> nominations =
driver.findElementsByClassName("film-nominations");

final List<String> resTitles = titles.stream()
  .map(WebElement::getText)
  .collect(Collectors.toList());

final List<String> resNominations = nominations.stream()
  .map(WebElement::getText)
  .collect(Collectors.toList());

resTitles.forEach(System.out::println);
resNominations.forEach(System.out::println);
driver.quit();
```

Listing 5: Full code example for fetching JavaScript loaded data in Selenium, with Java

```
browser = webdriver.Chrome(
executable_path='/Users/emilpersson/Downloads/chromedriver')
browser.get(
"https://scrapethissite.com/pages/ajax-javascript/")
browser.find_element_by_class_name("year-link").click()

def get_data():
  d = []
  titles_all = browser.find_elements_by_class_name(
  "film-title")
  nominations_all = browser.find_elements_by_class_name(
  "film-nominations")
  titles = [x.text.strip() for x in titles_all]
  nominations = [x.text for x in nominations_all]
  for i in range(len(titles)):
  d.append(dict([(titles[i], nominations[i])]))
return d

wait = WebDriverWait(browser, 10)
wait.until(lambda d: d.find_element_by_class_name('film-title'))
d = get_data()
print(d)
browser.quit()
```

Listing 6: Full code example for fetching JavaScript loaded data in Selenium, with Python

```
WebClient client = new WebClient();
client.getOptions().setJavaScriptEnabled(true);
client.setAjaxController(new NicelyResynchronizingAjaxController());
String baseUrl = "https://scrapethissite.com/pages/ajax-javascript/";
HtmlPage page = client.getPage(baseUrl);

HtmlAnchor btn = (HtmlAnchor) page.getElementById("2015");
page = btn.click();
client.waitForBackgroundJavaScript(10000);

List<HtmlTableDataCell> titles =
  page.getByXPath("//td[@class='film-title']");
List<String> resTitles = titles.stream()
  .map(DomNode::getTextContent)
  .collect(Collectors.toList());
List<HtmlTableDataCell> nominations =
  page.getByXPath("//td[@class='film-nominations']");
List<String> resNominations = nominations.stream()
  .map(DomNode::getTextContent)
  .collect(Collectors.toList());

resTitles.forEach(System.out::println);
resNominations.forEach(System.out::println);
```

Listing 7: Full code example for fetching JavaScript loaded data in HtmlUnit

A.0.2 Capability to alter header values

rvest

```
webpage <- html\_session(url,
user_agent(
'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:65.0)
Gecko/20100101 Firefox/65.0'))
```

Listing 8: Altering header values in rvest

A.0.3 Capability to handle Proxy Servers

Nightmare.js

```
let nightmare = Nightmare({
switches: {
  'proxy-server': 'proxy.geoproxies.com:1080'
  }
})
```

Listing 9: Code for setting up a proxy server in Nightmare.js

Puppeteer

```
const browser = await puppeteer.launch({headless: true,
args: ['--proxy-server=proxy.geoproxies.com:1080']});
```

Listing 10: Code for setting up proxy server in Puppeteer

Selenium

```
# Chrome
PROXY = "proxy.geoproxies.com:1080"
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument('--proxy-server=%s' % PROXY)
driver = webdriver.Chrome(options=chrome_options,
executable_path='/Users/emilpersson/Downloads/chromedriver')
driver.get("http://httpbin.org/ip")
print(driver.page_source)
```

Listing 11: Code for setting up a proxy server in Selenium Python, using ChromeDriver

```
# Firefox
PROXY = "proxy.geoproxies.com:1080"
desired_capability = webdriver.DesiredCapabilities.FIREFOX
desired_capability['proxy'] = {
  "proxyType": "manual",
  "httpProxy": PROXY,
  "ftpProxy": PROXY,
  "sslProxy": PROXY
}
driver = webdriver.Firefox(
executable_path='/Users/emilpersson/Downloads/geckodriver,
capabilities=desired_capability)

driver.get("http://httpbin.org/ip")
print(driver.page_source)
```

Listing 12: Code for setting up a proxy server in Selenium Python, using FirefoxDriver

```
// Chrome
System.setProperty("webdriver.chrome.driver",
"/Users/emilpersson/Downloads/chromedriver");
ChromeOptions options = new ChromeOptions()
  .addArguments("--proxy-server=http://" + proxyName);
ChromeDriver driver = new ChromeDriver(options);
driver.get("http://httpbin.org/ip")
```

Listing 13: Code for setting up a proxy server in Selenium Java, using ChromeDriver

```
// Firefox
System.setProperty("webdriver.gecko.driver",
"/Users/emilpersson/Downloads/geckodriver");
Proxy proxy = new Proxy();
proxy.setHttpProxy(proxyName)
  .setFtpProxy(proxyName)
  .setSslProxy(proxyName);
DesiredCapabilities desiredCapabilities =
  DesiredCapabilities.firefox();
desiredCapabilities.setCapability(
  CapabilityType.PROXY, proxy);
FirefoxDriver driver =
  new FirefoxDriver(desiredCapabilities);
driver.get("http://httpbin.org/ip");
```

Listing 14: Code for setting up a proxy server in Selenium Java, using FirefoxDriver

Scrapy

```
class ProxySpider(scrapy.Spider):
name = "proxy"
custom_settings = {
  'HTTPPROXY_ENABLED': True
}
def start_requests(self):
urls = [
  'http://httpbin.org/ip',
]
proxy = "http://proxy.geoproxies.com:1080"

for url in urls:
  yield scrapy.Request(
  url=url, callback=self.parse, meta={'proxy': proxy})

def parse(self, response):
  print(response.text)
```

Listing 15: Code for setting up a proxy server in Scrapy

HtmlUnit

```
WebClient client = new WebClient();
client.getOptions().setCssEnabled(false);
client.getOptions().setJavaScriptEnabled(false);
ProxyConfig proxyConfig = new ProxyConfig(
  "proxy.geoproxies.com", 1080);
client.getOptions().setProxyConfig(proxyConfig);
Page page = client.getPage("http://httpbin.org/ip");
System.out.println(page.getWebResponse().getContentAsString());
```

Listing 16: Code for setting up a proxy server in HtmlUnit

rvest

```
library('httr')
url <- "http://httpbin.org/ip"
httr::set_config(httr::use_proxy("proxy.geoproxies.com:1080"))
webpage <- GET(url)
print(webpage)
```

Listing 17: Code for setting up a proxy server in R

A.0.4 Capability to utilise browser tabs

Puppeteer

```
const browser = await puppeteer.launch({
headless: false });
const page = await browser.newPage();
await page.goto('http://localhost:8000/');

const username = await page.evaluate(() => {
  return document.querySelector(".username").innerHTML;
});
const password = await page.evaluate(() => {
  return document.querySelector(".password").innerHTML;
});

const loginpage = await browser.newPage();

await loginpage.goto(
'http://testing-ground.scraping.pro/login')
await loginpage.click("#usr")
await loginpage.keyboard.type(username);
await loginpage.click("#pwd")
await loginpage.keyboard.type(password);
await loginpage.click(
"#case_login > form:nth-child(2) > input:nth-child(5)")

const success = await loginpage.evaluate(() => {
  return document.querySelector("h3.success").innerHTML;
});
console.log(success)
await browser.close();
```

Listing 18: Code for using two tabs in Puppeteer

Selenium

```
driver = webdriver.Firefox(
executable_path='/Users/emilpersson/Downloads/geckodriver')
driver.get("http://localhost:8000")

username = driver.find_element_by_class_name("username").text
password = driver.find_element_by_class_name("password").text

driver.execute_script(
"window.open('http://testing-ground.scraping.pro/login')")
driver.switch_to.window(driver.window_handles[1])

wait = WebDriverWait(driver, 120)
wait.until(EC.visibility_of_element_located(
(By.CSS_SELECTOR, '#usr')))

usr = driver.find_element_by_css_selector("#usr")
pwd = driver.find_element_by_css_selector("#pwd")
usr.send_keys(username)
pwd.send_keys(password)

btn = driver.find_element_by_css_selector(
"#case_login > form:nth-child(2) > input:nth-child(5)")
btn.click();

msg = driver.find_element_by_css_selector("h3.success").text
print(msg)
driver.quit()
```

Listing 19: Code for using two tabs in Selenium Python

```
System.setProperty("webdriver.gecko.driver",
"/Users/emilpersson/Downloads/geckodriver");
FirefoxDriver driver = new FirefoxDriver();
driver.get("http://localhost:8000");

String username =
driver.findElementByClassName("username").getText();
String password =
driver.findElementByClassName("password").getText();

driver.executeScript(
"window.open('http://testing-ground.scraping.pro/login')");
driver.switchTo().window(
driver.getWindowHandles().toArray()[1].toString());

WebDriverWait wait = new WebDriverWait(driver, 120);
wait.until((d) -> d.findElement(By.cssSelector("#usr")));

WebElement usr = driver.findElementByCssSelector("#usr");
WebElement pwd = driver.findElementByCssSelector("#pwd");
usr.sendKeys(username);
pwd.sendKeys(password);

WebElement btn = driver.findElementByCssSelector(
  "#case_login > form:nth-child(2) > input:nth-child(5)");
btn.click();

String msg = driver.findElementByCssSelector(
  "h3.success").getText();
System.out.println(msg);
driver.quit();
```

Listing 20: Code for using two tabs in Selenium Java

Appendix B

Ease of use survey

Q1: The installation method is easy to find

Q2: The installation process is simple to follow

Q3: No other resources are needed

	Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
Q1	☐	☐	☐	☐	☐
Q2	☐	☐	☐	☐	☐
Q3	☐	☐	☐	☐	☐

Q1: The get started section is easy to follow

Q2: There are examples showcasing different use cases

Q3: There are links or references to external resources

	Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
Q1	☐	☐	☐	☐	☐
Q2	☐	☐	☐	☐	☐
Q3	☐	☐	☐	☐	☐

Q1: API references are well described

Q2: The documentation contain examples

Q3: The documentation is easy to navigate

	Strongly Disagree	Disagree	Undecided	Agree	Strongly Agree
Q1	☐	☐	☐	☐	☐
Q2	☐	☐	☐	☐	☐
Q3	☐	☐	☐	☐	☐

TRITA EECS-EX-2019:834