1 Introduction

From the advent of new languages, frameworks and methodologies, the way software is started, developed and maintained has dramatically shifted from what it was. Therefore, developers need to be able to keep up with the evolution, which requires consulting the documentation, understanding new principles, and in the end being able to be productive.

This process is not only straining, it is also time consuming. This is due to the fact that the developer has to consult the documentation to understand new concepts as well as watch out for breaking changes that may be introduced at a moments notice.

Luckily in the current age there are multiple ways to obtain information. From simple web searches, all the way to forums and Q&A websites such as Stack Overflow, the developer in question is able to quickly understand stay update on new technologies, languages, etc.

This availability of information can be a double edged sword. Where in the majority of cases a developer may be able to solve issues quickly because someone may already have encountered, there may be situations where the solution may be buried inside documents that do not directly treat the topic at hand. This causes the developer to be overloaded with information that is not pertinent to the task at hand, causing her to be less productive.

The goal of this project is to develop a tool that is able to summarise the contents of a page, and integrating it into the LIBRA[4] recommender. This will allow the developer to choose the amount of information she may like to see, going from the entire contents, all the way to seeing only one paragraph of the document. The strategy consists in taking an algorithm such as LexRank, augmenting it in order to support the heterogeneous nature of development artefacts, which may not only include text, but also code.

2 Project requirements and Analysis

2.1 Challenges

Add a small summary of the challenges

As previously stated, the goal of this project is to reduce the overload of information experienced by a user when searching for information online. While this is a challenge in itself, there are multiple other non-trivial challenges:

- **Lack of structure**
  In general, each website is implemented in a different structure. Although the constructors are the same, there are no strict rules in how they should be used. To overcome this challenge, ad-hoc parser were created for each website, aiding in understanding how to extract the relevant information, to then be sent the the service, as discussed in section . Add ref

- **Integrity of information**
  Being as the user will mostly have to deal with documents that may hold code snippets, it is crucial not to alter the contents, in such a way that the code may be used. One of the other challenges encountered was how to keep code intact. Many websites tend to include code, which aids in the explanation of the problem at hand, and often this code is spread among multiple lines, making a simple “split by newline” the wrong choice when parsing the content of the page. Therefore it was crucial for the data extraction phase not to modify the content, and therefore allow for the entire snippet to be treated as one.

2.2 Algorithm Overview

2.2.1 LexRank

LexRank[1] is an algorithm based on the principles of PageRank[2]. The way LexRank works, is by obtaining the separate sentences, and modelling a graph where each vertex is a sentence, and depending on the importance, a degree of centrality is assigned. In order to determine the degree of a node, the algorithm analyses the relationship between the different parts of a text, namely sentences, and determines what relationships they have between each other.

2.2.2 HoliRank

3 __PROJECT__NAME__

3.1 Overview

As described in section 1, the goal of this project is to reduce the amount of information presented to the developer, to reduce the overload that is introduced by the amount of information that is readily available.

In order to do so, we have developed __PROJECT__NAME__, which allows the user to choose the amount of information required in a page.

3.2 Architecture

The project has two main components:

- **Chrome Web Extension**
  Used to gather the information contained in a page by following the rules set up for a certain domain, as well as controlling the amount of information currently displayed

- **Web Service**
  Instead of relying on the end user's device to perform the calculations to determine the degree of centrality of each component, a web service was developed.

These two components interact with each other, in a typical client-server architecture, as shown in figure 1.

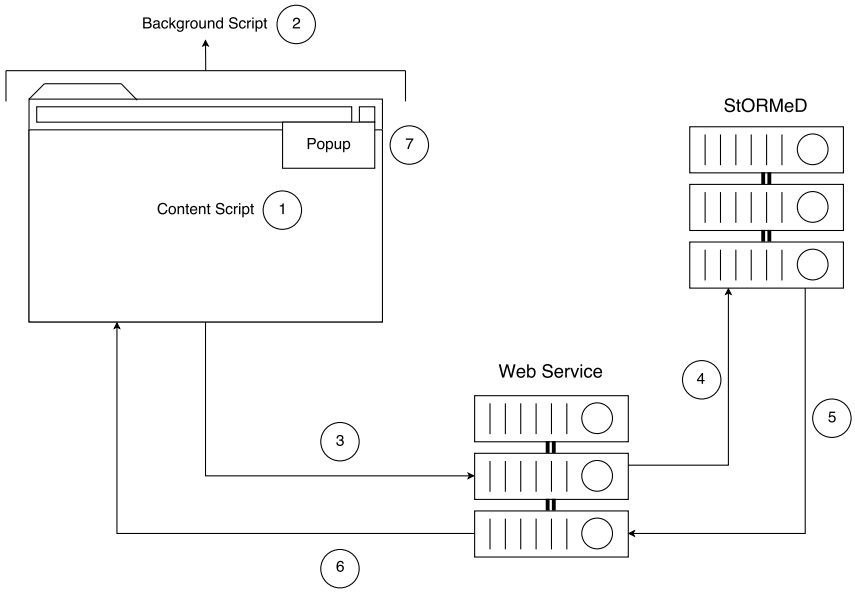

Figure 1. Architecture of __PROJECT__NAME__

Each of these numbers will be explained in a later section. In the next few sections, we will analyse each component, and its specific tasks.

3.2.1 Parser

The first step consists in collecting the information currently being displayed, which happens when the page has finished loading its content. As previously discussed, one of the main challenges of this project was to handle the wide variety of structures among webpages, being as each page tends to use the same constructs but widely differ in the implementation. This hurdle was solved by implementing a parser for each of the domains for which the information was needed.

Add refere

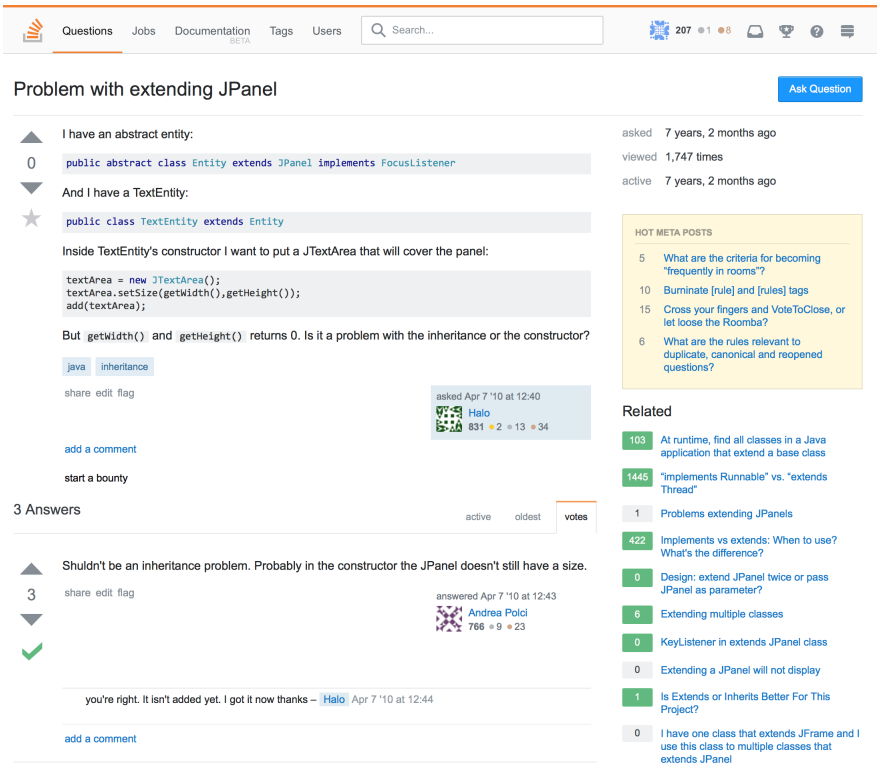

Figure 2. A typical StackOverflow conversation

Figure 2 represents the structure of a StackOverflow conversation. It is easy to see that there is a set structure which consists of a question, and 0 or n answers. Therefore our parser needs to first obtain the question, more specifically the paragraphs that make up such question, and then repeat the task for each of the possible answers.

In order to classify these pieces of information, we introduce the notion of an information unit. An information unit is a piece of content extracted from a page, which contains either text or code. The distinction has to be made when the content is extracted, in order to keep the integrity of code in such a way that it may be used later on by a simple copy and paste approach.

To ease the process of writing a parser, an abstract parser was created, with a few models to accompany the methods. Being as the parser has to live inside of the browser extension, the language sed to write such a program is JavaScript. Although a pretty complete programming language , the current version as of writting this document is ES5, which does not yet support features such as classes, inheritance, and more. Being as our implementation is supposed to be abstract, meaning that for each domain this parser can be extended, we needed a language that could support inheritance. After a bit of research, the choice ended up being TypeScript[1], a language developed by Microsoft which is a typed superset of JavaScript that compiles to plain JavaScript. This ensured that compatibility would not be an issue, while providing multiple useful features.

CRINGE

Sure?

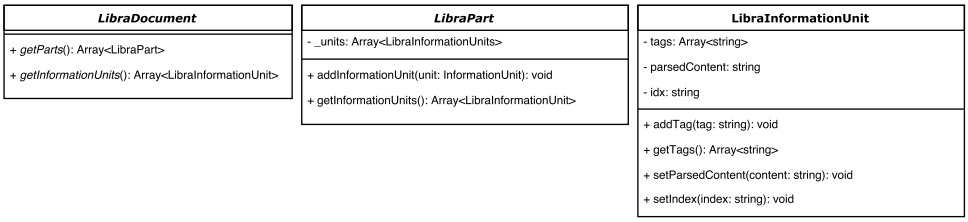

Figure 3. Abstract parser class diagram

We started by defining the models that would be needed to parse a page. A document diagram can be found in Figure 3.

The "root" model is called `LibraDocument` which provides the main methods that will be called by the extension, namely `parse()` and `getInformationUnits()`. Then, the information unit is defined, which encapsulates the data required to run the analysis. The last piece is `LibraPart`, which allows us to separate the multiple parts of a document.

[1] `https://www.typescriptlang.org`

As an example, consider the previously shown StackOverflow conversation. We could divide the question and the answers in two separate parts, which themselves contain other parts (i.e. the possible answers).

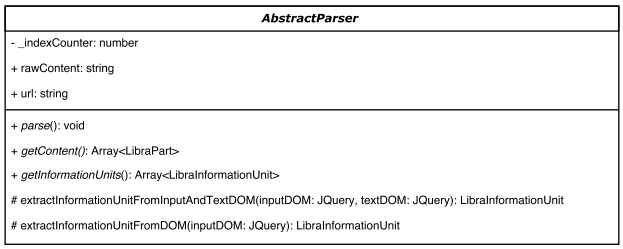

Figure 4. Abstract parser

These models are then used in the abstract parser, for which the diagram is show in figure 4. This way the developer writing a new parser has to simply extend the different methods and models, quickly creating a new parser which the extension is able to understand and use.

A remark ha to be made about the `extractInformationUnitFromDOM` and `extractInformationUnitFromInputAndTextDOM` methods. These two methods were created to keep the way that the data is extracted consistent among multiple parser implementations. The two methods are almost identical, and work in the following way:

1. A small marker, comprised of the URL hash combined with the current counter inside the document is injected into the HTML tags containing the information, like so:

```
<pre class="lang-java prettyprint prettyprinted" libra_idx="-656298628_0000000001">
    ...
</pre>
```

2. The counter is then increased
3. The type of the content, text or code, is identified and attached inside of the information unit
4. The information unit is returned to the developer

The reason why we are injecting the index is quite simple. In order to identify the unit to which a certain degree belongs to, we need to leave small traces such as this index, being as the service does not return the parsed text, but only the index and degree. This procedure is explained more in details in section . The index construction needed to abide to certain constraints, namely the ability of being unique inside a document, as well as when doing multiple document summarisation. In the end, the approach we chose was as follows:

Needs ref

```
Hash of url + "_" + 10 digit padded index
```

Although hashing may introduce cases where the produced hash is not unique, the probability of such an event occurring was small enough to be ignored.

By creating the two methods in `LibraDocument` as abstract, each parser can implement their own, while allowing the extension to call the same methods, regardless of the page being viewed (and hence parser being used), while effectively providing a common interface to the parser.

The execution of the parser is performed once the loading of the page is completed, which itself fires the request to the web service, discussed in this next section.

3.2.2 Web Service

The second main component of this project consists in a web service, that takes the information units given by the extension, and returns the degree of centrality for each of the information units. This service was written in Scala, using the Play framework. Most of the code is written in a functional way

Can this be said?

, which proved to be quite the challenge but very satisfying when things were working correctly.

The service has the following routes:

- `GET /register`
  Used to register the user to the service. This is needed as to identify the user in all subsequent calls to the service. The response is a 32-character string, which contains both numbers and letters. Once the user registers, a graph is created unique to this id, allowing for subsequent information units to be added, and therefore the ranking to be performed depending on the entire graph, not only the current units being analysed.

- `POST /rank`
  This path is used to rank the current units extracted from a document. By supplying the url of the webpage, the units, and the user id header, the service will return the units with a degree of centrality. `GET /all`
  Returns the entire graph for a certain user, with all units associated with their degree of centrality.

In figure 1 the entire architecture is shown. An example workflowis as follows:

This does good

1. User install the extension, opens it for the first time which prompts the registration procedure
2. The user id is stored in storage, for later use
3. The user navigates to a known page, which fires the content script's parsing process (Label 1)
4. Once parsing is completed, the background script takes over and prepares the data to be sent (Label 2)
5. The data is sent to the web service, which starts processing (Label 3)
6. The service sends a part of the data over to the island parser (Label 4)
7. Once the data is returned (Label 5), if it is the first request for that particular user, the graph is created
8. The nodes are added to the graph, and ranking starts
9. Once completed, the data is returned to the background script, which notifies the popup that the data is ready (label 6)
10. The popup prepares itself, and the user ca now interact with it (Label 7)

In this section we are going to focus on points 5, 8, and 9. When the extension makes the request, a few pieces of informations are needed. The following is an example of the request:

```
POST /rank
Content-Type: application/json
X-Libra-UserId: SsrktS2vrGPpHaAkYMsVDPo4qN6i38ei

{
  "units": [
    {
      "idx": "-656298628_0000000001",
      "parsedContent": "Creating an instance of a class:",
      "tags": [
        "plaintext"
      ]
    },
    {
      "idx": "-656298628_0000000002",
      "parsedContent": "MyObject myObject = new MyObject();",
      "tags": [
        "code"
      ]
    }
  ],
  "url": "http://www.MyAwesomeWebsite.rocks"
}
```

As we can see, the request requires the content type, the user id when registration happens, as well as the units with the url.

Once the units are received, the service will fire a request for each of the units to the StORMeD[3] service. This service is an island parser, which will allow us to construct the full AST from a certain unit.

The service will then add the nodes to the graph, and start calculating the degree by using a library called Signal/Collect[5], which allows the processing of large graphs in a really quick way. By describing the HoliRank algorithm using the provided syntax, the library handles the calculations returning the degree of centrality for a certain unit. The results are then returned to the user in the following format:

```
{
  "units": [
    {
      "idx": "-656298628_0000000001",
      "degree": 0.5,
      "url": "http://www.MyAwesomeWebsite.rocks"
    },
    {
      "idx": "-656298628_0000000002",
      "degree": 0.5,
      "url": "http://www.MyAwesomeWebsite.rocks"
    }
  ]
}
```

This information is then used by the extension to associate the index with its degree, which is then used to decide which units have to be hidden and which have to be shown depending on the value chosen by the user. This is explained in details in the next section.

3.3 Chrome Extension

As previously stated, in order to parse the content of a certain page, send it to the service, and then manipulate it, a Chrome extension was created. The extension encapsulates three main components:

- **Content Script**
  The content script has the task of loading the available parser, as well as using such parser. Once this is completed, the data is handed over to the background script.
- **Popup**
  The popup is the only piece of this project which a user can see. It is the main point of interaction between the user and the project. She has the ability to choose the amount of information that has to be displayed.
- **Background script**
  The background script has the task of connecting the content script with the popup.

Before we go any further, one must understand how a Chrome extension works. As with our implementation, an extension usually consists of three main components as described above. These components have different restrictions regarding what data they can access, how they are instantiated, and when they are destroyed.

The background script is persistent, meaning it stays alive for the duration of the session, and has the ability to make HTTP requests but has no access to the content on the page, or the content of the popup. On the other hand, the content script has the ability to access the content, but no access to the popup. Last but not least, the popup is recreated every time the user clicks on the icon, and has no access to the content. This means that the three components have to communicate to each other, by means of message passing.

Each of these components has the ability to send and receive messages, and react accordingly. This was useful when, for example, the user had just registered to the service. Each component has its own localStorage, but the user id had to be used by multiple components which created an issue regarding where to store it. In the end message passing was used and a common local storage was chosen, this being the background script's storage as this component was always listening and active.

As we have seen in the previous section, the service returns the payload containing the different degrees of each unit. It is the task of the extension as a whole to process this information, and make it usable to the user. This is

achieved by sorting the payload data, and adding in each of the units currently on the page a sort order index, as follows:

```
<pre class="lang-java prettyprint prettyprinted" libra_idx="-656298628_0000000001"
 ↪ sortorder="15">
    ...
</pre>
```

Once this manipulation is complete, the popup updates its view, and shows a range slider, allowing the end user to choose the percentage of the information that she may want to be displayed. Every time the value of the slider changes, the popup fires a message, calling into action the content script which will hide or show the units depending on their sort order. The interface is shown in figure 5

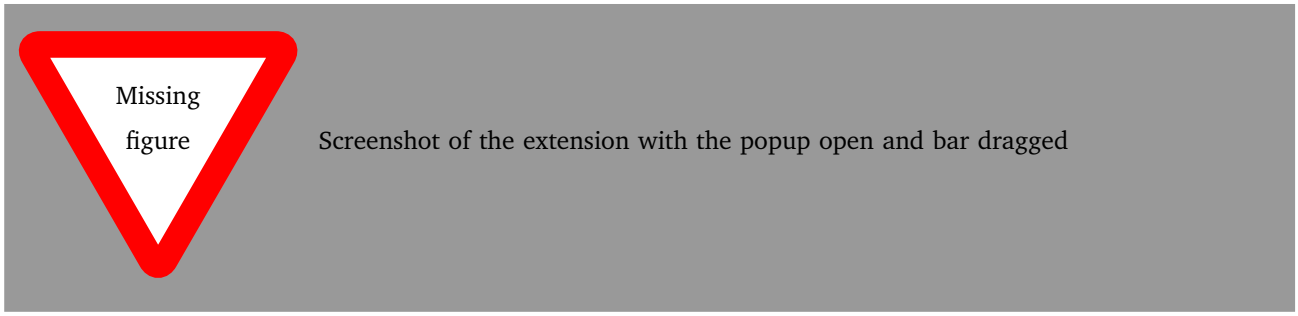

Figure 5. Interface of the Chrome extension

Another feature of the extension, is the ability of generating a page containing all the units that have been visited by a certain user. If the user clicks on the summary button, a new page is opened containing all of the units, and for each site a range slider is added to choose the amount of information displayed. An additional range slider is added at the top of the page, called a master slider, which controls the amount of information displayed by each site, regardless of the value chosen by each individual slider. This allows the user to consult only the highest ranked units in a website, giving a very quick overview of multiple sites at the same time. The result can be seen in figure 6

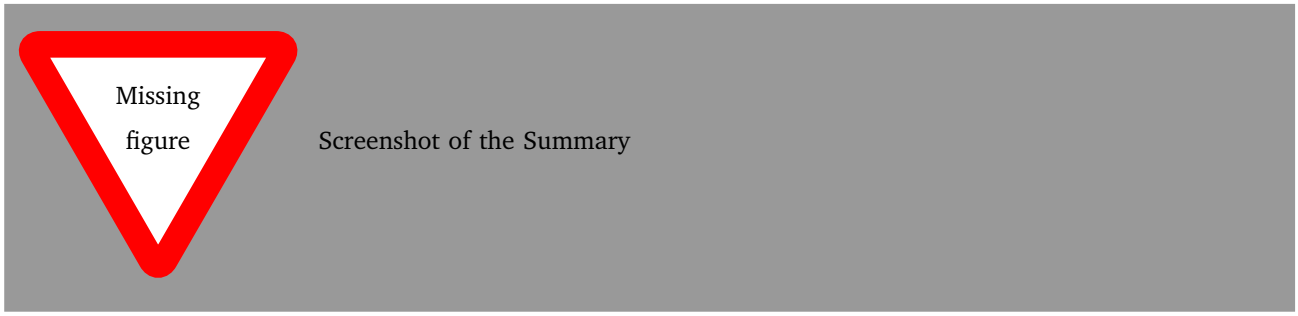

Figure 6. Generated Summary

4 Implementation Issues

4.1 Limitations of approach

4.1.1 Granularity

Although this approach to summarisation works, the biggest limitation is the granularity of it. As we have seen, the service is able to correctly identify the importance of each unit inside of a document or collection of documents, but what is still missing is a true summarisation of the content. More specifically, what would be interesting to achieve is a summarisation algorithm that is able to take a text, and recreate a new one by following grammar rules and keeping the context of such intact.

Although interesting, it would bring a new set of challenges, particularly when dealing with code that is embedded inside of a sentence, forcing it to be shown no matter what in the summary. Making the summariser understand exactly how to connect code and text may prove to be quite the challenge

4.1.2 Persistence

The current implementation of the web service uses an in memory store, meaning that there is no real database behind the service, and the data is kept in memory and destroyed once the service is shut down. Although not essential to the project, a database would be useful, allowing the user to retrieve data that is older than what is saved in current memory.

This brings the challenge of deciding what has to be kept, what has to be discarded and when. By keeping all of the units stored in a database, the user may see information that is too old or has nothing to do with the current context. A new way to tag the information would be required, tagging what the context was, when the information unit was added, and whether to discard it once new units are added to the current graph.

4.1.3 Performance

To implement the algorithm behind __PROJECT__NAME__, many libraries were used. These libraries tend to have their own way of doing things, which our code had to adapt to. One example is the `SimilarityParameters` present in the StORMeD devkit. These parameters have to be recalculated every time a new unit is added, which requires a lot of time. If we were only to consider small graphs, this would be no problem as there is almost no impact on the general performance. Once we start to consider larger graphs, the limitations start to show, and the performance of the whole system suffers. In order to fix such issue, the devkit would have to be rewritten from scratch, which is outside the scope of this project.

References

[1] G. Erkan and D. R. Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *J. Artif. Int. Res.*, 22(1):457–479, Dec. 2004.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[3] L. Ponzanelli, A. Mocci, and M. Lanza. Summarizing complex development artifacts by mining heterogenous data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, pages 401–405. ACM Press, 2015.

[4] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, M. D. Penta, R. Oliveto, and M. Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published, 2017.

[5] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.