Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

June 18, 2017

WebDistiller

Integrating a summariser into a Holistic recommender system

Lucas Pennati

Abstract

LIBRA[5] is a holistic recommender system that supports the developer in the process of searching and navigating the set of documents returned by a search engine like Google. Although LIBRA guides the user in finding pertinent material, it neither helps with reducing the information overload that may characterize resources that are too long and detailed for the task at hand, nor reduce the information overload occurring when multiple sources have to be aggregated and filtered.

We introduce WebDistiller, a summarizer that monitors the pages viewed by a user, extracts the content and creates interactive extractive summaries of a single page or multiple pages, that the user can interact with and decide the amount of filtration she may want to see. The project also allows the user to generate a summary of everything she has seen up to a certain moment, and once again allowing to extract the most relevant sections to solve the task at hand.

Advisor
Prof. Dr. Michele Lanza
Assistants
Dr. Andrea Mocci, Dr. Luca Ponzanelli

Advisor's approval (Prof. Dr. Michele Lanza): Date:

Contents

1 Introduction **2**

2 State of the art **2**

3 Project requirements and Analysis **3**
3.1 Challenges . . . 3
3.2 HoliRank . . . 3

4 WebDistiller **3**
4.1 Overview . . . 3
4.2 Architecture . . . 4
4.3 Server side . . . 5
4.3.1 Web Service . . . 5
4.3.2 StORMeD . . . 6
4.4 Client-side: Chrome extension . . . 7
4.4.1 Parser . . . 7
4.4.2 Chrome Extension components . . . 9

5 Results **10**

6 Implementation Issues **12**
6.1 Current Limitations . . . 12
6.1.1 Granularity . . . 12
6.1.2 Persistence . . . 12
6.1.3 Performance . . . 12
6.2 Future work . . . 12

7 Conclusion **12**

List of Figures

1 From website, to graph, to summary . . . 4
2 Multi document graph . . . 4
3 Architecture of WebDistiller . . . 4
4 Sequence diagram . . . 5
5 A typical StackOverflow discussion . . . 7
6 Abstract parser class diagram . . . 8
7 Abstract parser . . . 8
8 Interface of the Chrome extension . . . 10
9 Chrome extension status icons . . . 11
10 Generated Summary . . . 11

1 Introduction

In software engineering, the development of new products can be broken down into multiple phases: Requirements Gathering, Analysis and Specification, Implementation, Testing and Deployment. Where in the Analysis and specification phase a team picks the different frameworks and programming languages, in the Implementation phase these tools have to be used. In this phase the documentation plays an essential role, guiding developers through the execution of the idea. Developers tend to rely on web resources such as documentation, Q&A sites, as well as tutorials to solve the task at hand.

There are many reasons why developers rely on web resources aside from the documentation. In a perfect world, the documentation would be complete, maintained, and straight to the point, and would include everything needed to get started and productive. In reality, developers tend to rely on Q&A sites such as Stack Overflow[1] for help when dealing with problems. It is then the task of the developer to collect all of the information found and filter it to find the relevant sections to solve the task at hand. Not only is this process straining, but it can lead to information overload.

To mitigate this phenomenon, summarization can be used as an effective way to automate part of the process, presenting developers only with a subset of the collected data. In this project, we introduce WebDistiller, a tool which provides the ability to create interactive extractive summaries of a single page or set of pages. By building the tool using HoliRank[5], an algorithm which builds on the foundations of PageRank[2], the heterogeneous nature of artefacts is considered when determining the prominence of a certain section of a document.

To begin with, in section 2 we take a look at existing approaches for both summarization as well as recommendation systems for Software Engineering. In section 3 we analyze some of the challenges which we may be faced with, whereas in section 4 we go in depth an ddiscuss the implementation and design choices to create WebDistiller. In section 5 we analyze the issues and limitations with our implementation, and in section 6 we explore the results.

2 State of the art

Still a major TODO...

[1] http//stackoverflow.com

3 Project requirements and Analysis

3.1 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 constructs are the same, there are no strict rules in how they should be used.

- **Integrity of information**
  Users will mostly have to deal with documents that may contain code snippets. Many websites tend to include code, which helps in the explanation of the problem at hand. Due to this heterogeneous nature of the information, these section had to be kept intact while also

Someting type

3.2 HoliRank

HoliRank[5] is an algorithm that builds on top of LexRank[1], which itself is based on PageRank[2]. Before we go any further, we describe how both LexRank[1] and PageRank[2] work from a high level perspective. PageRank[2] simulates a random surfer, a user which randomly surfs the web clicking on links and never going back until it gets bored and starts from another random page. This random surfer is defined by the following formula:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{1}$$

where d is a damping factor (usually set to 0.85), $M(p_i)$ is the set of all pages that link to p_i, $L(p_j)$ is the number of outgoing links from p_I, and N is the total number of pages in the network and serves as a normalisation factor. In this approach, the probability that the random surfer may visit a page represents the degree of centrality of this page in a network of pages.

In order to move to LexRank[1], some modifications are needed. Instead of considering a network of pages, the approach considers a document as a collection of sentences that form a network. Then, PageRank[2] is applied to this network of sentences. This modification to the algorithm allows for sentences, which in theory have no proper link between them, to be connected and used.

HoliRank[5] is an algorithm that builds on LexRank[1], and considers the heterogeneous nature of information in software engineering. It applies the same idea as LexRank[1], meaning that a network of sentences is extracted from a document, with the difference that code artefacts are treated differently from purely text based artefacts.

4 WebDistiller

The upcoming sections will illustrate WebDistiller and all of its features. We also illustrate the decisions taken to be able to create this tool. We will start by explaining the general theory behind the approach, to then move onto the general architecture, the server side components, and then last but not least, the client side components.

4.1 Overview

HoliRank[3], like PageRank[2] and LexRank[1], calculates the degree of an entity inside a network of other entities. To represent this network, we can use graphs. We represent each entity as a separate vertex, with connections between them to indicate the similarity.

To encapsulate the information extracted from a a page, we introduce the notion of an information unit. An information unit is a piece of content extracted from a page, which can contain either code or plain text.

As previously stated, one of the challenges was the lack of structure in web pages. This is due to the fact that each website is structured in a different way, but uses the same constructs, such as paragraphs, sections, titles, etc. Since we needed to create separate information units for each piece of text, we started by analyzing a few websites. We decided that the best approach was to separate the content by paragraph where the content was plain text, whereas in the case of code, we kept it intact. Once these units are obtained, we add them as a vertex to a graph, and feed it to HoliRank[3] which calculates the prominence for each unit inside the graph.

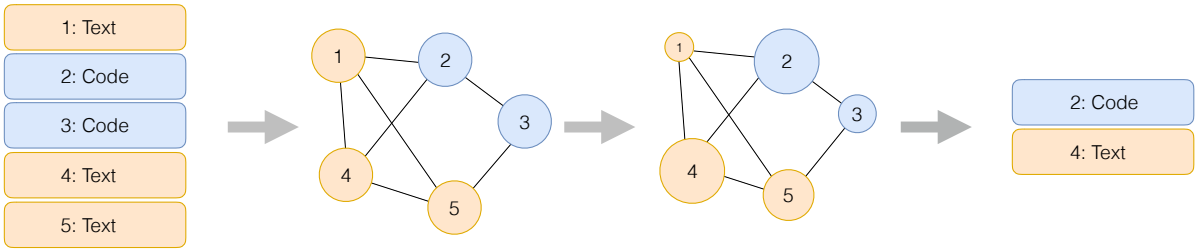


Figure 1. From website, to graph, to summary

In Figure 1 we can see the whole process. We start by splitting the page into sections, depending on whether they are text-based, or code-based. We then add these sections to the graph, calculate the degree of centrality for each vertex, and filter the content based on the prominence of the vertex. In the next sections, we are going to explore exactly how these actions are performed

In the case of multiple documents, we use the same principle as shown in Figure 1. The only difference is the origin of the content enclosed inside the vertex. The vertices are all added to the same graph, and the prominence of each is calculated in relationship to all vertices, regardless of the origin. This allows us to consider the user's history. An example is shown in Figure 2

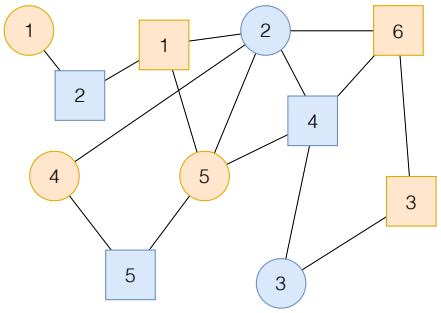


Figure 2. Multi document graph

4.2 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 importance of each section, a web service was developed.

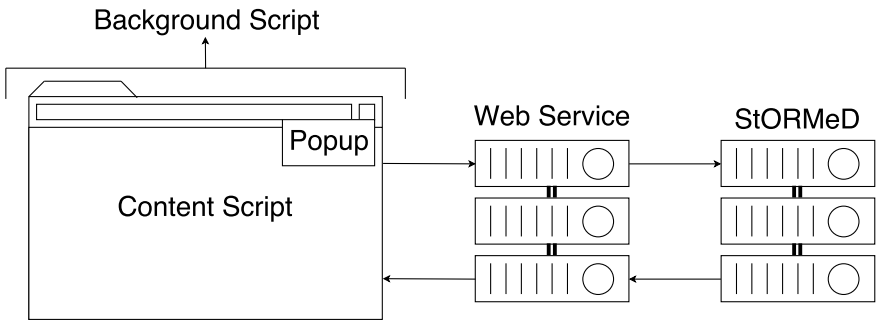


Figure 3. Architecture of WebDistiller

These two components interact with each other, in a typical client-server architecture, as shown in figure 3.

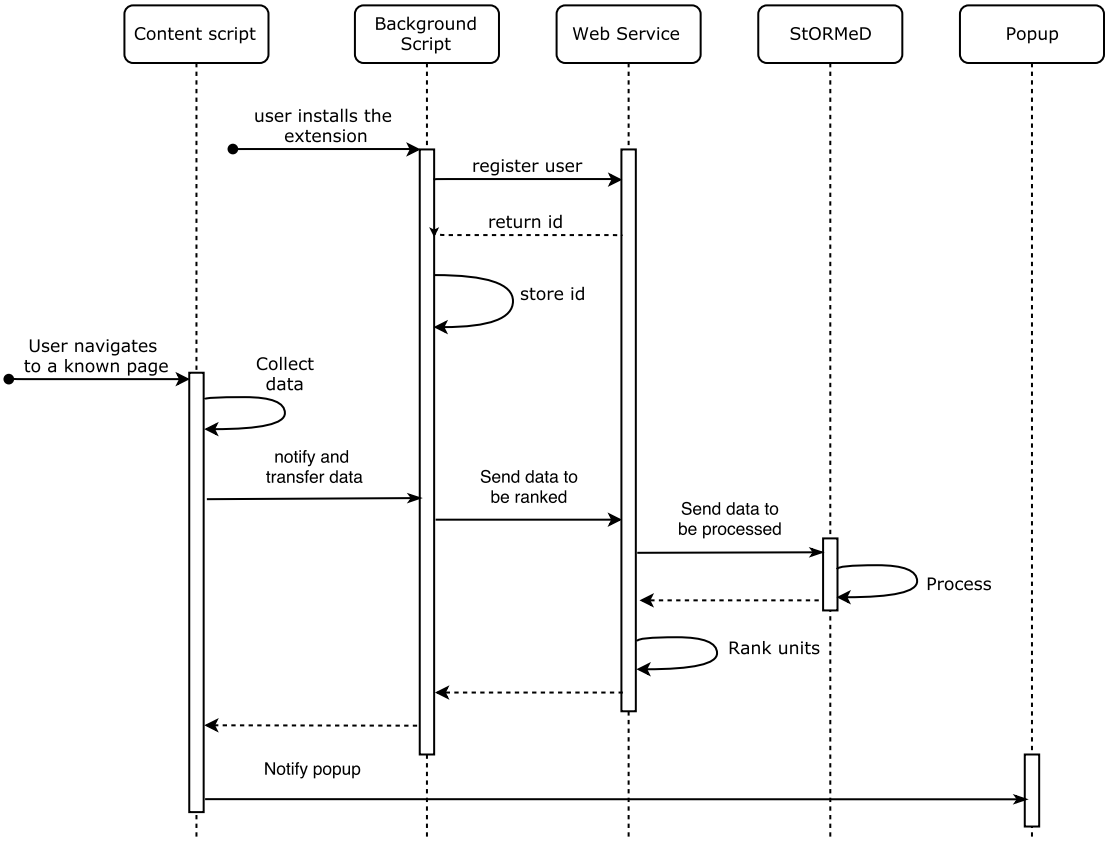


Figure 4. Sequence diagram

Figure 4 shows the sequence for both the registration procedure, as well as a normal request. For the registration, the procedure is as follows:

1. The user installs the extension, which triggers the registration request
2. The request is made to the service, which generates the user id and returns it to the client
3. The id is saved in in the local storage for the user

Whereas for the ranking request, the procedure is as follows:

1. The user navigates to a known page, which triggers the data collection phase
2. The data is sent to the web service to be ranked
3. Units are processed, and a request is made to the StORMeD service
4. Once the data returns, the units are added to the user's graph, and the prominence is calculated
5. Ranked units are returned to the Chrome extension, which now allows the user to manipulate the information available on the page

We are now going to explore each component in greater detail.

4.3 Server side

4.3.1 Web Service

The first 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, and has the following routes:

- `GET /register`
  Used to register the user to the service. This is needed to identify the user in all subsequent calls to the service. The response is a 32-character alphanumeric string, known as the user's id. 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.

To better understand how a request looks like, consider Listing 1. The user id that is created upon registration is attached to every request as a header as well as the URL of the page.

```
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"
}
```

Listing 1. Sample request from Chrome extension to web service

As shown in the sequence diagram in Figure 4, an external call is made to the StORMeD service, an island parser which we use to construct an H-AST for all units. An Heterogeneous Abstract Syntax Tree (H-AST)[**?**] is a structure that is used to represent both textual fragments as well as the language of an artefact.

The service will then add the nodes to the graph, and start calculating the degree by using a library called Signal/Collect[6], which allows the processing of large graphs in a really quick way. By describing the HoliRank[3] 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 as seen in Listing 2.

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.

4.3.2 StORMeD

StORMeD[4] is an island parser capable of building the Heterogeneous Abstract Syntax Tree (H-AST) for a piece of content. In out project, we use the service to analyze the code snippets which are parsed from the page. The web service makes a call to the StORMeD service, which returns the full H-AST that is fed into HoliRank[3].

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

Listing 2. Sample response

4.4 Client-side: Chrome extension

4.4.1 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 in section 3.1, 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.

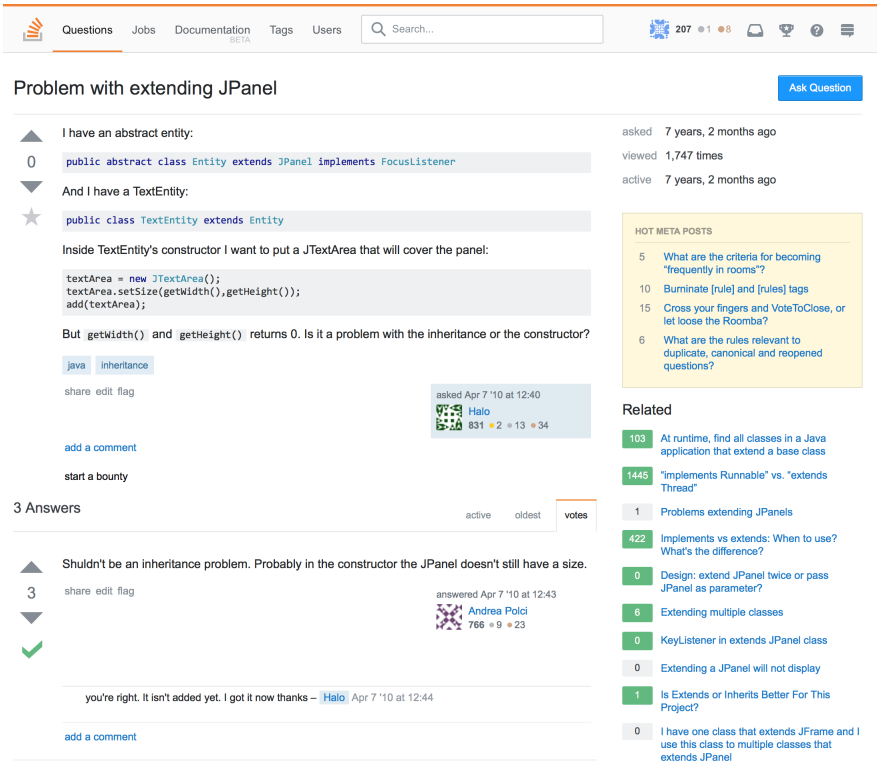


Figure 5. A typical StackOverflow discussion

Figure 5 represents the structure of a Stack Overflow[2] discussion. A Stack Overflow discussion consists of a question with the related comments, and a set of answers with their own set of comments. Therefore our parser needs to first obtain the question, more specifically the paragraphs and comments that make up such question, and then repeat the task for each of the possible answers.

[2] `http://stackoverlow.com/`

Since the structure of a Stack Overflow discussion is largely different from a tutorial from DZone[3] , we needed to write a parser for each website we wished to parse. To facilitate this situation, we created an an abstract parser, along with a few standard models. Due to the limitations of vanilla JavaScript, we decided to use TypeScript[4], 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. We started by defining the models that would be needed to parse a page, in a very general way. A document diagram can be found in Figure 6.

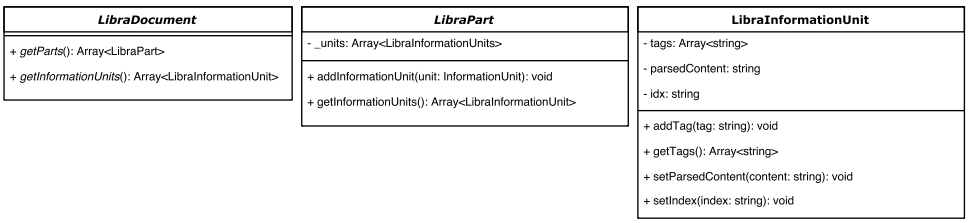


Figure 6. Abstract parser class diagram

The "root" model is called `LibraDocument` which provides the main methods called by the Chrome extension, namely `parse()` and `getInformationUnits()`.

In our implementation of the information unit, called `LibraInformationUnit`, we also store a few other useful parts such as the index of the unit and the tags attached to the DOM.

The last component is `LibraPart`, which separates the multiple parts of a document. As an example, consider the Stack Overflow discussion shown in Figure 5. We could divide the question and the answers in two separate parts, which in turn contain other parts (i.e. the possible comments to a question or answer).

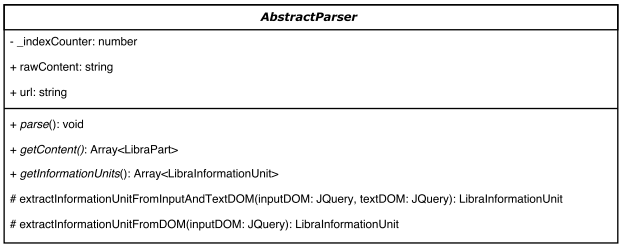


Figure 7. Abstract parser

These models are then used in the abstract parser, for which the diagram is show in Figure 7. 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 has to be made about the `extractInformationUnitFromDOM` and `extractInformationUnitFromInputAndTextDOM` methods. These two methods were created to keep the data extracted consistent among multiple parser implementations. The consistency is achieved by requiring the DOM element that contains the artefact to be extracted, and returning a ready to use `LibraInformationUnit`. What is hidden behind this implementation is a tagging procedure that allows the Chrome extension to connect the data returned by the service with the content being displayed. To provide a unique identifier for each of the elements on the page, we used a combination of the page's URL with a global counter. The structure of this index is as follows:

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

When extracting the content of the DOM, the two previously mentioned methods inject the index into the DOM of the page as seen in Listing 3

Then, when the service has returned the degree of centrality associated with a certain index, we can perform a simple search inside of the page and link the data to the related DOM.

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 `parse()` and `getInformationUnits()` 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.

[3] `http://dzone.com/`

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

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

Listing 3. Index injection example

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 the next section.

4.4.2 Chrome Extension components

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. Before we go any further, we need to understand the structure of a Chrome extension. A Chrome extension is divided into three components:

- **Background script**
  The background script runs in the background for the whole browser, and is persistent. The session is not unique to the tab, and therefore no data exclusive to a certain tab should be stored in here.

- **Popup**
  The only user facing component of the extension. It allows the developer to create a UI, which can be recalled every time the button on the right of the sidebar is clicked. It is important to notice that it is created every time the icon for the extension is clicked, and completely destroyed once the user is not interacting with it anymore.

- **Content script**
  The content script has access to the data currently being displayed. It is able to interact with the DOM, and its session is unique to each tab open.

Each of these components has a very distinct and precise task. In order to make our extension work, all three components were implemented.

Since each component of the extension has a different set of restrictions, the jobs have to be delegated to the component designed to perform it. Chrome uses message passing, a method where the different components can subscribe to receive certain types of messages, and have the ability to broadcast messages themselves to the other components. In our case, the three components were used in the following way:

- **Content Script**
  The content script is responsible of loading the available parsers, as well as using the correct one for the current page. Once this is completed, the data is handed over to the background script.

- **Popup**
  The popup is shown containing a range slider, as well as a few buttons, which allow the user to interact with the information being currently displayed on the page, and to manipulate it.

- **Background script**
  The background script is responsible of connecting the content script with the popup, as well as making the HTTP requests to the service.

As previously explained, message passing is the way components communicate with each other. This feature was used, for example, when 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 seen in Listing 4. In the next section, we are going to see the resulting application, with an in depth explanation of the functionality.

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

Listing 4. Sort order tag injection

5 Results

The goal of this project was to provide a simple way for developers to reduce information overload. It was very important that the user interface of the Chrome extension was as simple as possible to remove any possible hurdles. Figure 8 shows the UI for the popup component of the Chrome extension.

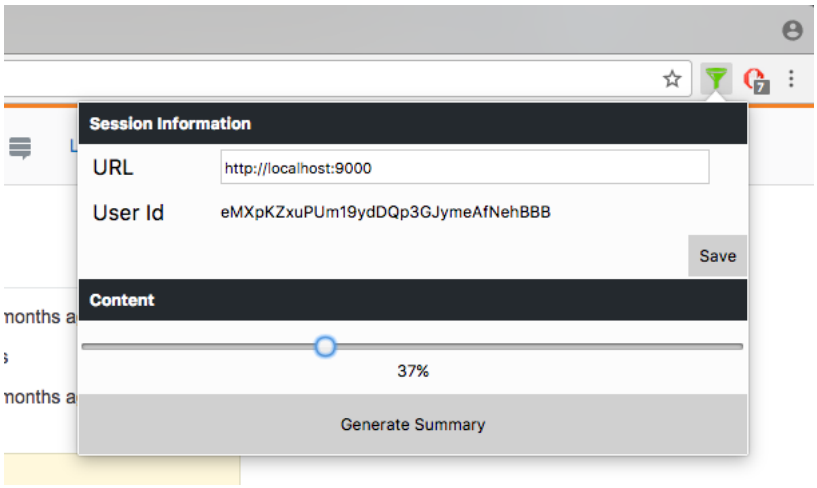


Figure 8. Interface of the Chrome extension

The interface consists of two main sections. The first section shows information about the current session. This was mostly used while debugging the application. The second section is used to interact with the user. The current view shows the slider, which means that the data gathering and ranking processes have completed successfully. In case of errors, or while the content script is parsing the page, the slider is hidden and messages are shown to inform the user of the current status.

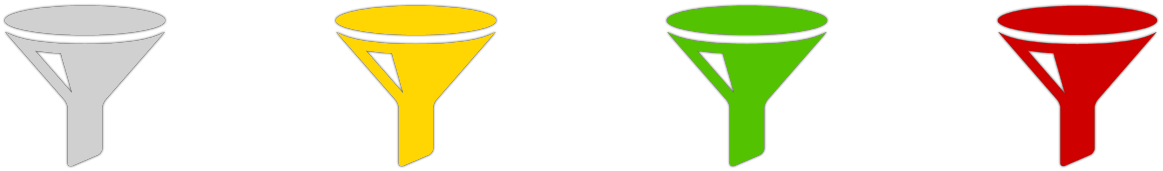
An additional visual feedback is provided by the icons of the extension shown in Figure **??**, which can represent any of the following states:

- **Inactive**
  Represented by a greyed out icon (Figure **??**), tells the user that the extension is not active in the current website.

- **Parsing**
  While the application is parsing, a yellow icon (Figure **??**) is shown to indicate that the Chrome extension is working, and the results will be shown shortly.

- **Ready**
  When the icon turns green (Figure **??**), the user has the ability to open the popup and start manipulating the content by dragging the slider.

- **Error**
  If an error occurs, a red icon is shown (Figure **??**).



(a) Inactive **(b)** Parsing **(c)** Ready **(d)** Error

Figure 9. Chrome extension status icons

I am not sure about this thing...looks toyish

When the Chrome extension has received the data back from the web service, the icon turns green and the slider is shown in the popup. The user has now the ability to choose the amount of filtration required, which updates the content of the page in real time. Any time the slider is dragged, the popup will fire an event caught by the content script, which will take care of hiding or showing the different units, depending on their sort order.

Another feature of WebDistiller is the ability to generate an overview of all the pages in a user's history. By clicking the Generate summary button from the popup (Figure 8), the extension will open a new page, containing the documents. An example is shown in Figure 10.

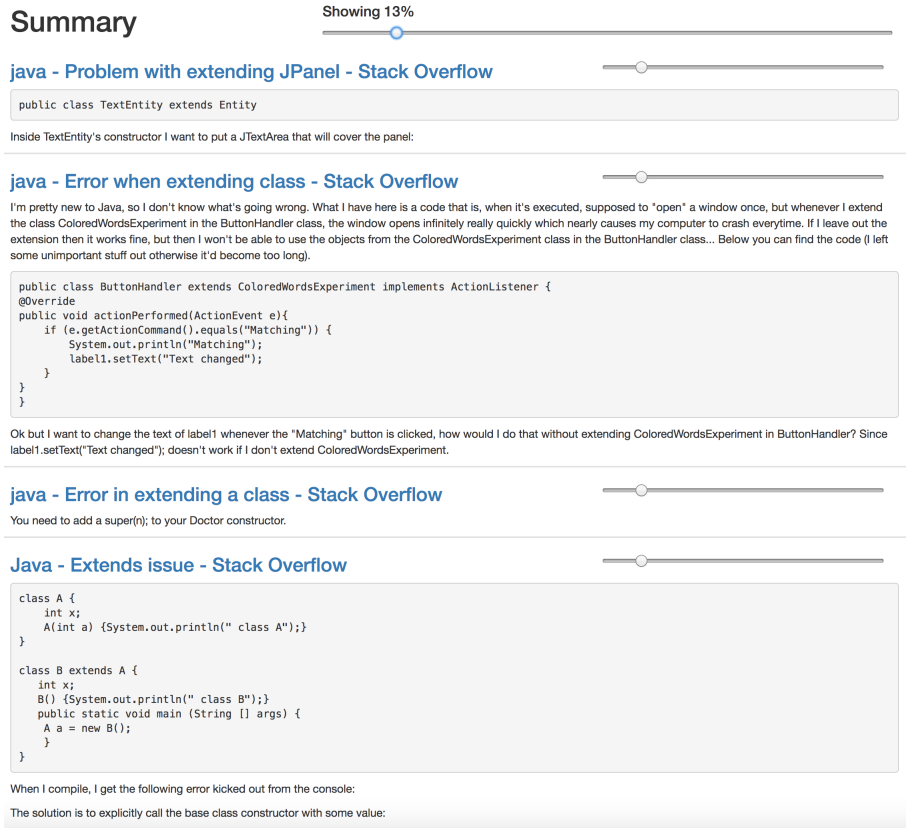


Figure 10. Generated Summary

For each page, a slider is available to once again filter the content independently from the other pages. An additional 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 its history, giving a very quick overview of multiple sites at the same time.

6 Implementation Issues

6.1 Current Limitations

6.1.1 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

6.1.2 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.

6.1.3 Performance

To implement the algorithm behind WebDistiller, 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.

6.2 Future work

7 Conclusion

We presented WebDistiller, a novel approach to reducing information overload. It is the combination of a holistic ranking approach, with a simple to use UI. Developers have now the ability to partially offload the task of filtering the content, allowing them to focus on the task at hand. By also providing a summary functionality, we provide the ability to see the previously seen pages, with an easy to use interface that allows them to filter the content once again, with the importance of each section based on the entire history, and not only the current document.

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. *Holistic Recommender Systems for Software Engineering*. PhD thesis, Università della Svizzera italiana, 3 2017.

[4] L. Ponzanelli, A. Mocci, and M. Lanza. Stormed: Stack overflow ready made data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, pages 474–477. ACM Press, 2015.

[5] 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.

[6] 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.