Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

June 20, 2017

WebDistiller

Integrating a Summarizer into a Holistic Recommender System

Lucas Pennati

Abstract

Developers often rely on web resources, such as documentation, Q&A websites, and tutorials to solve the task at hand. Once the information has been gathered, developers need to manually sort through the collected documents, and find the relevant parts to help them complete the task. Due to the high amount of information available, developers tend to experience information overload, which can be mitigated with the use of summarization.

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. By using the algorithm behind LIBRA [13], a holistic recommender system, we allow the developer to obtain summaries which are pertinent to the previously browsed documents. The project also allows developers to generate a summary of every web page they visited, giving an overview of the documents that have been consulted, 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

List of Figures

1 Introduction

To solve the task at hand, developers often rely on web resources. The knowledge that a developer may have is usually not enough, requiring further exploration of documentation, Q&A sites, and tutorials [20]. Developers are then faced with a large availability of material, which has to be filtered by either using a few keywords provided by the search engine, or by consulting the results deemed more important. Developers then collect all of the information found and filter it to find the relevant sections needed 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. Summarization in software engineering is not new, and has been attempted before. Existing approaches include the summarization of email threads [14] to generate extractive summaries of bug reports as well as exploiting PageRank [9] to create general summarization approaches for bug reports [6].

The common limitation in the aforementioned approaches is the way artifacts are treated. In both cases, the techniques do not differentiate between purely text-based artifacts from code-based artifacts, but treat both as text-based artifacts. This limits the amount of information that may be extracted from a resource.

Ponzanelli *et al.* [12] propose a novel approach where the heterogeneous nature of artifacts is considered when performing summarization. The proposed approach augments LexRank [3] to deal with the heteregenous and multidimensional nature of complex artifacts, by providing a new similarity function for heterogeneous entities such as code samples.

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 integrating it with LIBRA [13], a holistic recommender system, through the usage of HoliRank [13], an algorithm which builds on the foundations of PageRank[9], the heterogeneous nature of artifacts is considered when determining the prominence of a certain section of a document. We use multiple sources, to provide the most complete extractive summary to support developers in filtering and choosing the most relevant parts of different documents.

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 and discuss 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

In the past years there have been a few attempts at summarization for software engineering. Rastkar *et al.* [14] base their approach on pre-existing techniques generally used to summarize the contents of email threads, and apply the same principles to generate summaries of bug reports. Lotufo *et al.* [6] also summarize bug reports, but base their approach on PageRank [9].

Mani *et al.* [7] proposes an implementation based on different techniques (e.g. Grasshopper, DivRank, Centroid) to generate summaries in an unsupervised manner. All of these approaches only consider plain text artifacts, whereas information retrieval techniques have been used to summarize source code. Such techniques include Vector Space Model and Latent Semantic Indexing, which were used to summarize code samples [4] [18].

The above examples all generate summaries to reduce information overload. Other techniques include the usage of recommender systems for software engineering [17], which suggest relevant artifacts to the developer and tend to harness different sources. The sources can include APIs as presented by Rigby *et al.* [15] [16], or by extracting information from Q&A sites, as presented by [10], or by analyzing existing code bases [2] [1] [5] [8].

Ponzanelli *et al.* [13] present LIBRA, a holistic recommender based on a meta-information system which is capable of dealing with the heterogeneous nature of resources, as well as considering the current context.

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 developer 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 HTML 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.

3.2 HoliRank

HoliRank [13] is an algorithm that builds on top of LexRank [3], which itself is based on PageRank [9]. Before we go any further, we describe how both LexRank and PageRank work from a high level perspective. PageRank 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, 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 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.

In theory, LexRank could have been used “as is” to perform summarization, as most of the documents developers will consult are mainly composed of text. Issues arise when the page being viewed contains artifacts which are not purely text-based, such as code snippets. The difference between the data can cause issues when LexRank determines the similarity between the artifacts, since it does so by considering the pure textual similarity of the artifacts, which hinders the whole information provided by source code. The approach followed by HoliRank is to consider the heterogeneous nature of information in software engineering, i.e. the possibility of an artifact to be composed of either text or code. By performing this distinction, the algorithm is able to access the multiple layers of information provided by the different artifacts, since the way similarity is calculated varies depending on the type of the artifact.

4 WebDistiller

The following 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 the client side components.

4.1 Overview

A graph is a collection of nodes, which form a network. If we consider the world wide web, we can represent each page as a node inside of that network. When a connection exists from a page to another, for example with a link, we can say that the two pages are connected and an edge between them is created.

The same reasoning can be applied to a document. Each part of the document becomes a node, and the edges between the different nodes in the network represent the similarity between the parts of the document. To provide a relationship between the nodes and the data displayed, 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.

In our approach, an information unit is a paragraph where the content can either be plain text, or code. The construction of the edges is the same as [12], and relies on the H-AST nodes composing the information units.

To create the extractive summary, a possible approach consists in using HoliRank to calculate the degree of centrality of each information unit in the graph, and then select the most central ones according to a threshold set by the user. An example of this process can be seen in Figure 1.

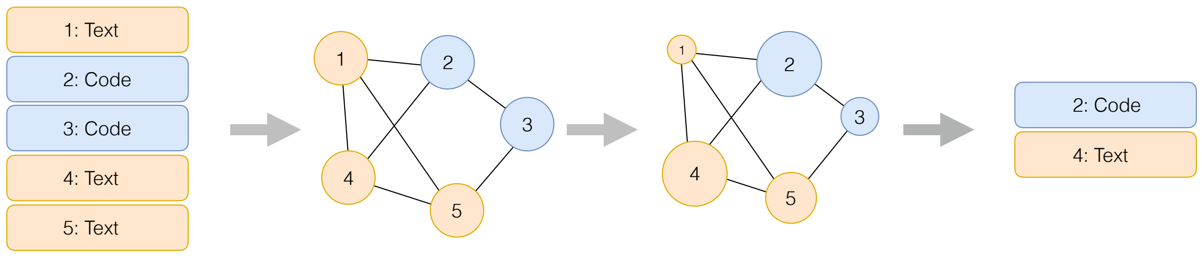


Figure 1. From website, to graph, to summary

A major issue with this approach is the lack of context when creating the summary. In other words, the knowledge acquired by the user through the previously browsed pages is not considered while selecting the most relevant units.

In our implementation, we consider the history visited by a developer. We create a Context Graph ad-hoc for each developer which aggregates information units from different sources. When the developer visits a new page we extract the units from it, and add them to the graph. An example is shown in Figure 2.

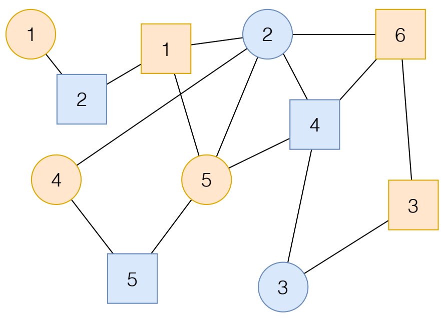


Figure 2. Context graph

In Figure 2, different sources are represented by different shapes (Squares represent document A, circles represent document B). When calculating the degree of centrality of a certain unit, the result is not based only on the content of the page currently being viewed, but it depends on all of the other units, which represent the history of pages visited. This means that anytime a new node is added to the context graph, the degree of a specific unit changes, as the context has changed. Therefore, a summary of a web page will change depending on the web pages visited.

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, we developed a web service.

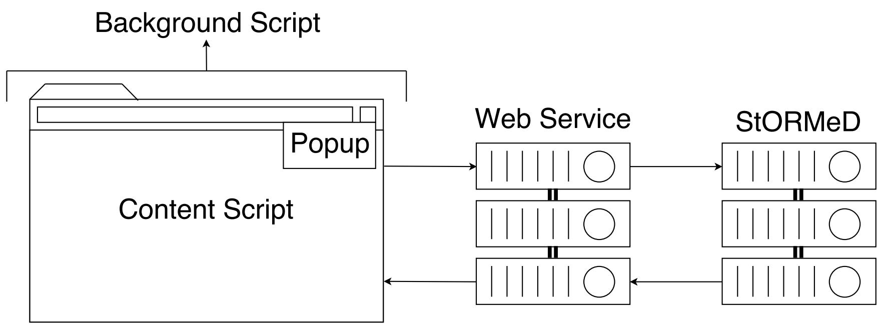


Figure 3. Architecture of WebDistiller

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

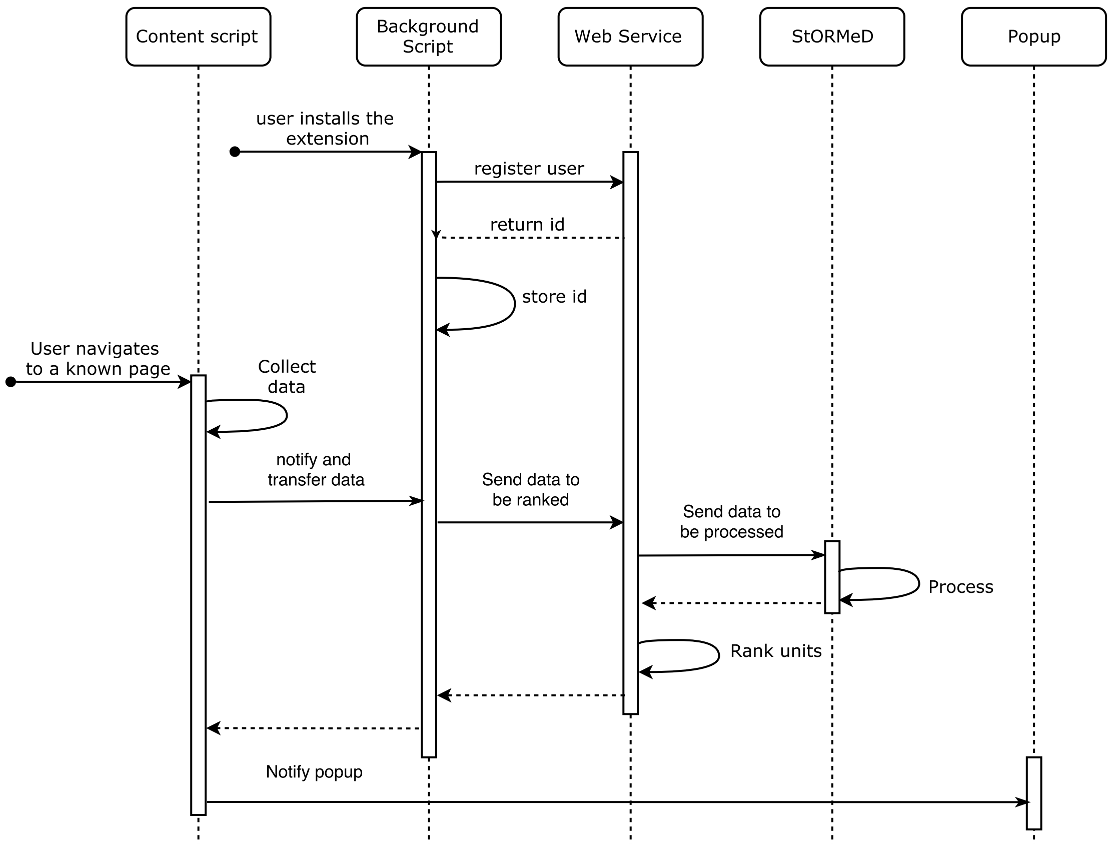


Figure 4. Sequence diagram

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

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 fadditional 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.mywebsite.io"
}
```

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 Heterogeneous Abstract Syntax Tree (H-AST)[13] for all units. An H-AST is a structure that is used to represent both textual fragments as well as code fractions of an artifact.

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

Listing 2. Sample response

The service will then add the nodes to the graph, and calculates the degree by using a library called Signal/Collect[19], which allows high performance processing of large graphs. 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 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 threshold chosen by the user. This is explained in details in the next section.

4.3.2 StORMeD

StORMeD[11] 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.

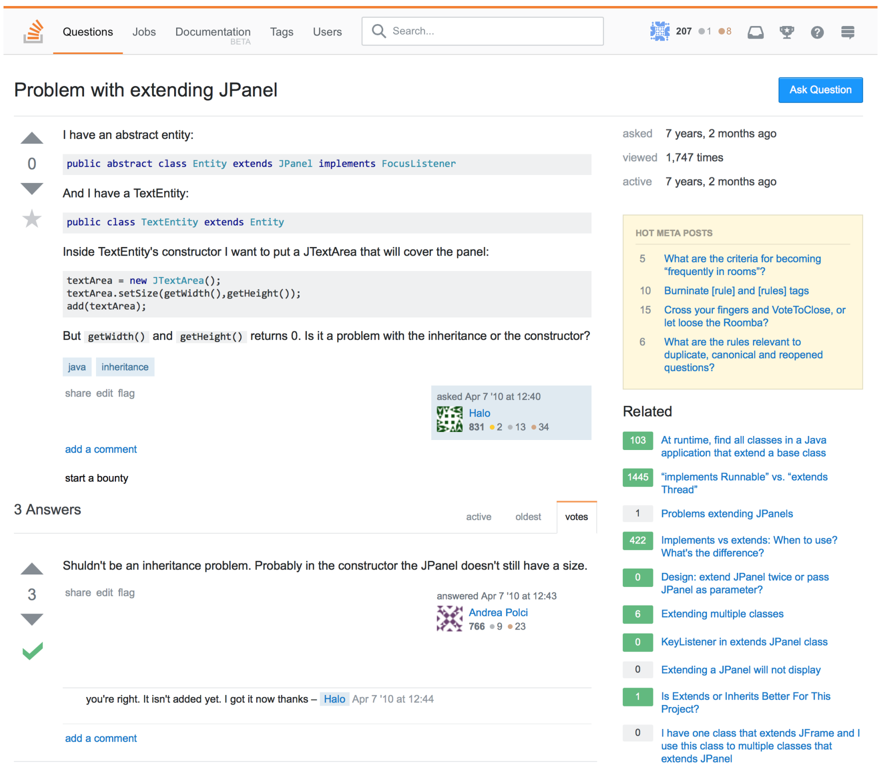
4.4 Client-side: Chrome extension

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 HTML constructs but widely differ in the implementation. This hurdle was solved by implementing a parser for each domain we target (i.e. Stack Overflow[1], Spring Documentation[2], DZone[3], Android Documentation[4]).

Consider a Stack Overflow discussion, as in Figure 5. A Stack Overflow discussion consists of a question, 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.

[1]http://stackoverflow.com/
[2]http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/
[3]http://dzone.com/
[4]https://developer.android.com/guide/index.html


Questions Jobs Documentation BETA Tags Users Search... 207 ●1 ●8

Problem with extending JPanel

Ask Question

I have an abstract entity:

```
public abstract class Entity extends JPanel implements FocusListener
```

And I have a TextEntity:

```
public class TextEntity extends Entity
```

Inside TextEntity's constructor I want to put a JTextArea that will cover the panel:

```
textArea = new JTextArea();
textArea.setSize(getWidth(),getHeight());
add(textArea);
```

But `getWidth()` and `getHeight()` returns 0. Is it a problem with the inheritance or the constructor?

java inheritance

share edit flag

asked Apr 7 '10 at 12:40
Halo
831 ●2 ●13 ●34

add a comment

start a bounty

3 Answers

active oldest votes

3

Shuldn't be an inheritance problem. Probably in the constructor the JPanel doesn't still have a size.

share edit flag

answered Apr 7 '10 at 12:43
Andrea Polci
766 ●9 ●23

you're right. It isn't added yet. I got it now thanks – Halo Apr 7 '10 at 12:44

add a comment

asked 7 years, 2 months ago
viewed 1,747 times
active 7 years, 2 months ago

HOT META POSTS

5	What are the criteria for becoming "frequently in rooms"?
10	Burninate [rule] and [rules] tags
15	Cross your fingers and VoteToClose, or let loose the Roomba?
6	What are the rules relevant to duplicate, canonical and reopened questions?

Related

103	At runtime, find all classes in a Java application that extend a base class
1445	"implements Runnable" vs. "extends Thread"
1	Problems extending JPanels
422	Implements vs extends: When to use? What's the difference?
0	Design: extend JPanel twice or pass JPanel as parameter?
6	Extending multiple classes
0	KeyListener in extends JPanel class
0	Extending a JPanel will not display
1	Is Extends or Inherits Better For This Project?
0	I have one class that extends JFrame and I use this class to multiple classes that extends JPanel

Figure 5. A typical Stack Overflow discussion

Since the structure of a Stack Overflow discussion differs from a tutorial from DZone[5] , we needed to write a parser for each website we wish to parse. To facilitate this situation, we created an an abstract parser, along with a few standard models. We used TypeScript[6], 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.

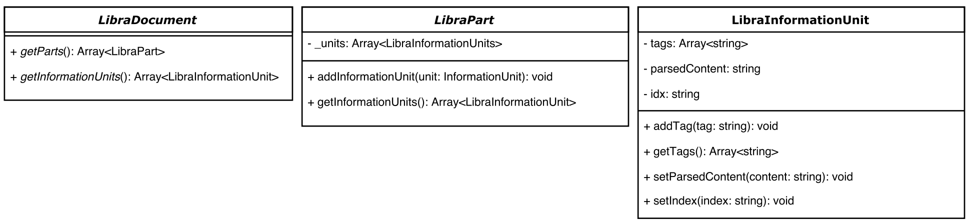

LibraDocument
+ getParts(): Array<LibraPart>
+ getInformationUnits(): Array<LibraInformationUnit>
LibraPart
- _units: Array<LibraInformationUnits>
+ addInformationUnit(unit: InformationUnit): void
+ getInformationUnits(): Array<LibraInformationUnit>
LibraInformationUnit
- tags: Array<string>
- parsedContent: string
- idx: string
+ addTag(tag: string): void
+ getTags(): Array<string>
+ setParsedContent(content: string): void
+ setIndex(index: string): void

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.

[5]`http://dzone.com/`
[6]`https://www.typescriptlang.org`

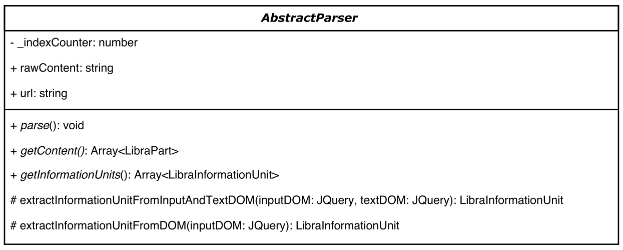

AbstractParser
- _indexCounter: number
+ rawContent: string
+ url: string
+ parse(): void
+ getContent(): Array<LibraPart>
+ getInformationUnits(): Array<LibraInformationUnit>
# extractInformationUnitFromInputAndTextDOM(inputDOM: JQuery, textDOM: JQuery): LibraInformationUnit
# extractInformationUnitFromDOM(inputDOM: JQuery): LibraInformationUnit

Figure 7. Abstract parser

These models are then used in the abstract parser (see diagram in Figure 7). This way the developer writing a new parser has to simply extend the different methods and models.

A remark has to be made about the `extractInformationUnitFromInputAndTextDOM` and `extractInformationUnitFromDOM` 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 artifact 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.

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

Listing 3. Index injection example

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 is small enough to be ignored.

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

A Chrome extension was created to parse the contents of a page, send it to the service, and then manipulate it. The typical structure of 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 interface of the extension. It allows the developer to create a user interface, which can be shown every time the icon of the extension is clicked. It is important to notice that the user interface 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.
- **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.
- **Popup**
  The popup is shown containing a range slider, which allows the user to interact with the information being currently displayed on the page, and to manipulate it.

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 local storage, 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.

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

Listing 4. Sort order tag injection

In the next section, we describe the resulting application, with an in depth explanation of the functionality.

5 User Interface

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 user interface for the popup component of the Chrome extension.

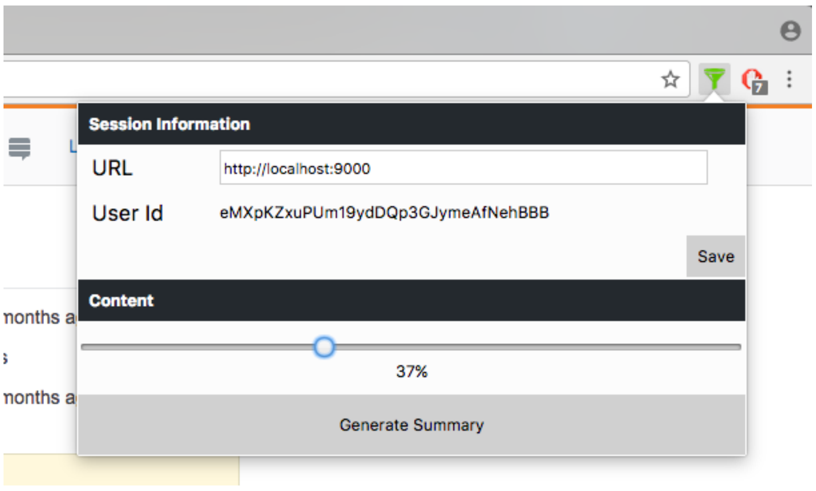


Figure 8. Interface of the Chrome extension

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 possible icons of the extension shown in Figure 9, which can represent any of the following states:

- **Inactive**
  Represented by a greyed out icon (Figure 9a), tells the user that the extension is not active in the current website.

- **Parsing**
  While the application is parsing, a yellow icon (Figure 9b) is shown to indicate that the Chrome extension is working, and the results will be shown shortly.

- **Ready**
  When the icon turns green (Figure 9c), 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 9d).

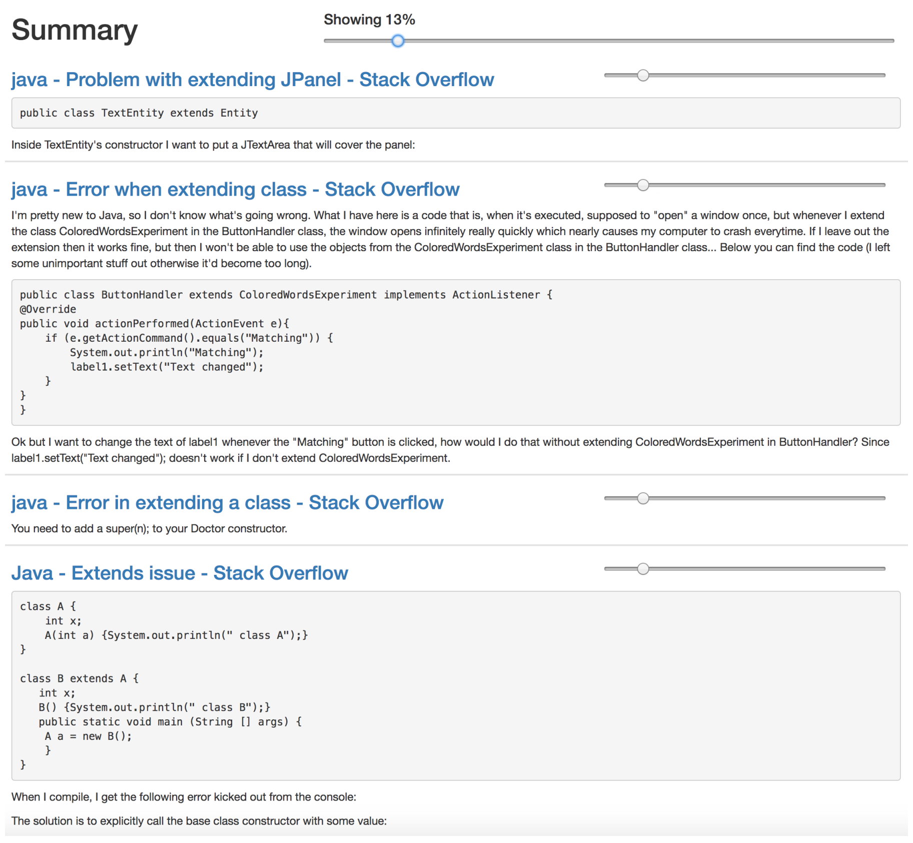
(a) Inactive **(b)** Parsing **(c)** Ready **(d)** Error

Figure 9. Chrome extension status icons

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.

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.



Summary

java - Problem with extending JPanel - Stack Overflow

```
public class TextEntity extends Entity
```

Inside TextEntity's constructor I want to put a JTextArea that will cover the panel:

java - Error when extending class - Stack Overflow

I'm pretty new to Java, so I don't know what's going wrong. What I have here is a code that is, when it's executed, supposed to "open" a window once, but whenever I extend the class ColoredWordsExperiment in the ButtonHandler class, the window opens infinitely really quickly which nearly causes my computer to crash everytime. If I leave out the extension then it works fine, but then I won't be able to use the objects from the ColoredWordsExperiment class in the ButtonHandler class... Below you can find the code (I left some unimportant stuff out otherwise it'd become too long).

```
public class ButtonHandler extends ColoredWordsExperiment implements ActionListener {
@Override
public void actionPerformed(ActionEvent e){
    if (e.getActionCommand().equals("Matching")) {
        System.out.println("Matching");
        label1.setText("Text changed");
    }
}
}
```

Ok but I want to change the text of label1 whenever the "Matching" button is clicked, how would I do that without extending ColoredWordsExperiment in ButtonHandler? Since label1.setText("Text changed"); doesn't work if I don't extend ColoredWordsExperiment.

java - Error in extending a class - Stack Overflow

You need to add a super(n); to your Doctor constructor.

Java - Extends issue - Stack Overflow

```
class A {
    int x;
    A(int a) {System.out.println(" class A");}
}

class B extends A {
   int x;
   B() {System.out.println(" class B");}
   public static void main (String [] args) {
    A a = new B();
    }
}
```

When I compile, I get the following error kicked out from the console:

The solution is to explicitly call the base class constructor with some value:

Figure 10. Generated Summary

6 Future Work

6.1 Granularity

Although this approach to summarization 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 summarization of the content. More specifically, what would be interesting to achieve is a summarization 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 summarizer understand exactly how to connect code and text may prove to be quite the challenge

6.2 Persistence

The current implementation of the web service uses in memory storage, 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.

6.3 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.

7 Conclusion

We presented WebDistiller, a novel approach to reduce information overload through extractive interactive summaries. We started by defining the possible challenges of the project, and how we may overcome them. We explained how both LexRank and PageRank work, before giving an overview of HoliRank, the algorithm behind LIBRA, which we used to perform summarization on documents. We discussed the general procedure, from the initial stage of data extraction, all the way to filtering. We described our approach more in depth, by explaining the context graph and how the development context may alter the resulting summary, even for a single document. The architecture of the project was next, we explored the different components such as the web service, StORMeD and the Chrome extension. We outlined the procedure of ranking the parts of a document and the criteria for filtering the elements based on a user set threshold. The user interface was presented, and an example of a multi-document summary was shown. Thus, we discussed the possible limitations of the project, and proposed possible solutions to mitigate those issues.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, New York, NY, USA, 2007. ACM.

[2] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[3] 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.

[4] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA, 2010. IEEE Computer Society.

[5] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 664–675, New York, NY, USA, 2014. ACM.

[6] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Softw. Engg.*, 20(2):516–548, Apr. 2015.

[7] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: Approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 11:1–11:11, New York, NY, USA, 2012. ACM.

[8] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can i use this method? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015. IEEE Press.

[9] 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.

[10] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*, pages 102–111. ACM, 2014.

[11] 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.

[12] 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.

[13] 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.

[14] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Trans. Softw. Eng.*, 40(4):366–380, Apr. 2014.

[15] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841, Piscataway, NJ, USA, 2013. IEEE Press.

[16] M. P. Robillard and Y. B. Chhetri. Recommending reference api documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[17] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[18] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 390–401, New York, NY, USA, 2014. ACM.

[19] 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.

[20] M. Umarji, S. E. Sim, and C. Lopes. *Archetypal Internet-Scale Source Code Searching*, pages 257–263. Springer US, Boston, MA, 2008.