International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume X, Number Y, pages 501–524, July 2017

Automated Dataset Construction from Web Resources with Tool Kayur

Alexander Kohan
Department of Mathematics and Informatics, Chiba University
Chiba, Japan

Mitsuharu Yamamoto
Department of Mathematics and Informatics, Chiba University
Chiba, Japan

and

Cyrille Artho
School of Computer Science and Communication, KTH Royal Institute of Technology
Stockholm, Sweden

Received: (received date)
Revised: (revised date)
Accepted: (accepted date)
Communicated by Editor's name

Abstract

Many text mining tools cannot be applied directly to documents available on web pages. There are tools for fetching and preprocessing of textual data, but combining them with the data processing tool into one working tool chain can be time consuming. The preprocessing task is even more labor-intensive if documents are located on multiple remote sources with different storage formats.

In this paper, we propose the simplification of data preparation process for cases when data come from wide range of web resources. We developed an open-source tool, called Kayur, that greatly minimizes time and effort required for routine data preprocessing steps, allowing to quickly proceed to the main task of data analysis. The datasets generated by the tool are ready to be loaded into a data mining workbench, such as WEKA or Carrot2, to perform classification, feature prediction, and other data mining tasks.

Keywords: Automation, Information extraction, Natural language processing, Web mining

1 Introduction

Textual information located on the Internet is usually designated for a human reader, but as the growth rate of available data is increasing, more and more automated tools are used to process such data to get insight about its properties or discover trends and patterns. Although some web resources are designed to facilitate machine processing by providing an Application Programming Interface (API) to export data in structured formats, such as XML, CSV, or JSON, on many

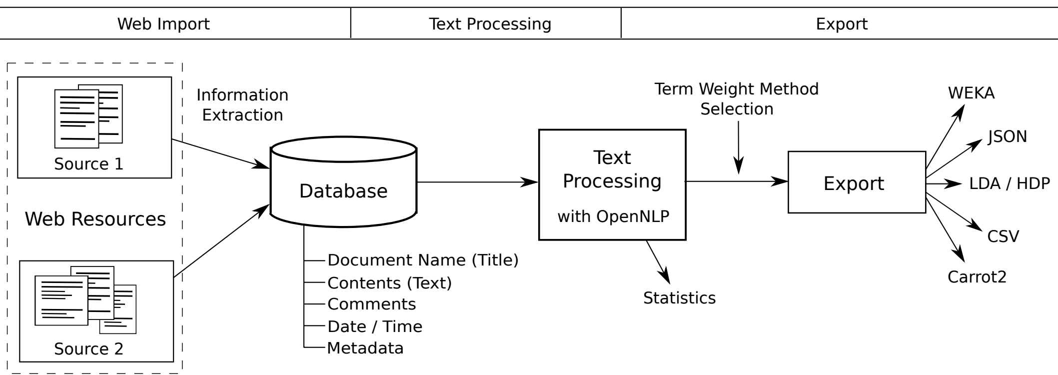


Figure 1: Workflow of Kayur

web resources useful information is still available only in HTML format. This makes automated processing more difficult, because, unlike XML, HTML tags do not describe the data they contain.

Web mining tools that extract data from HTML documents usually require a user to set up the extraction rules for each data field; the navigation rules to define transition between documents; and the integration rules to translate extracted data into the desired format. This configuration step may be labor-intensive, especially when dealing with multiple web resources of different structure. Partially because of that reason, there are web resources to which text mining has never been applied.

Although web resources are different, some common patterns can be found in structure of information they provide. If we leverage these similarities, we may arrive at substantial simplification of web mining application to a wide range of remote sources. In this work, we argue that by imposing certain restrictions on web resources and data, web mining process can be simplified, so that dataset construction step can be accomplished quickly and conveniently.

First, we suggest that data to be analyzed are a collection of texts written in a human language. Such data can be stored in the uniform format that comprises fields common to all textual documents (document's title, content, comments, publishing date, and metadata). Storing the data in the same format improves reusability, and makes possible to generate a dataset from subsets of documents that span over multiple web resources.

Second, we focus only on web resources that provide a search engine to find documents, and assign to each document a unique identifier. These conditions are met by many resources, because a web resource that provides to a user a collection of documents usually provides means to locate documents of interest as well. If these conditions are met, the access and navigation between documents can be automated, so that extraction and navigation rules can be simplified and their number can be substantially reduced.

We developed an open-source tool, called Kayur [11], to demonstrate these concepts. The tool can be applied to web resources that satisfy the mentioned conditions. It fetches documents from the Internet, translates them into the uniform format, and stores the results in a relational database. Textual data of stored documents are converted using underlying OpenNLP library [22] into a filtered list of normalized words that are then represented numerically according to the vector space model [31] with specified weight method. The resulting vectors are converted into an input format of a data mining tool (such as WEKA [35], Carrot2 [26], or LDA/HDP topic modeling tools by J. Chang and C. Wang [34]).

The tool demonstrates a shortcut to quickly generate a dataset for text mining needs from web resources that satisfy certain properties. Kayur requires minimum user input, has the most commonly used settings preconfigured, and provides the intuitive user interface, which makes it useful even for people unfamiliar with data preparation and preprocessing steps. By minimizing routine work, Kayur allows to quickly proceed to the most important step of data analysis. The tool itself and its source code is available on its home page [11].

This paper is an extended version from an earlier conference paper [12]. The earlier paper

included only small examples of possible applications of the tool; this extended paper shows how Kayur can be used in a larger context. We addressed this issue by including a large-scale application example, namely, the identification of the most frequent Android defect types by the semi-automatic analysis of a large bug report collection. Furthermore, we improved interoperability of the tool by adding dataset export to popular formats JSON and CSV, so that the tool can be used in conjunction with a wider range of software for data analysis.

The rest of this paper is structured as follows: Section 2 provides a brief theoretical background on text mining concepts; Section 3 explains Kayur's architecture and implementation; Section 4 presents small examples of the use of our tool and its evaluation; Section 5 demonstrates the application of the tool to a more complex and large-scaled task of identification of the most frequent Android defect types; Section 6 discusses the related projects; Section 7 summarizes the obtained results and presents directions for future work.

2 Background

2.1 Web mining and information extraction

Data mining is the process of automatic analysis of large stores of data to discover patterns and trends. Classification, clustering, association learning, and numeric prediction are the main types of information analysis in data mining [36]. The use of data mining techniques to extract and analyze information in web documents is known as *web mining*.

Because document structure greatly differs between web resources, one of the straightforward adopted approaches to extract information was to write a stand-alone program (script) for each resource using programming language such as Perl [5]. However, better methods were developed in *web information extraction* area, and such programs were superseded with *extractors* (wrappers) that allow working with multiple web resources by describing unique properties of sources with *extraction rules* [37].

However, not all the extracted data are immediately ready to be used by a data mining tool. Numeric values and various identifiers (names, locations) can be used "as is" most of the time, while texts written in human languages usually require additional processing using text mining methods.

2.2 Text mining and the vector space model

When data mining is applied to human-written texts, the term *text mining* is usually used instead. Text mining has many important applications, including analysis of stock reports, product manuals, business and normative documents [32]. It is also proved to be useful for bug report analysis, including bug report classification [4, 39], detection of duplicates [30], and prediction of certain properties of software flaws [13].

The standard approach for applying data mining techniques to textual data is to transfer the textual data into the *vector space model*. This model represents texts as algebraic vectors in a multidimensional space, to allow calculation of similarities between documents using linear algebra operations. The dimension of the vector space equals the number of distinct terms (words) in a document collection. Each document is represented as a vector with components reflecting the frequency of a particular term in the document.

The values assigned to vector components depend on the *term weighting method* being used. For the *Boolean* method, the value of a document vector component is zero if the corresponding term does not occur in the document, and one otherwise. For the *raw frequency* method, the value assigned to a component is the number of occurrences of the corresponding term in the document.

The *term frequency* method differs from the raw frequency method in that the assigned frequencies are normalized by dividing on the maximum number of occurrences of a term in the document. The most widely used *term frequency / inverted document frequency* (TF-IDF) method considers both term frequency in the current document and its frequency in the whole document collection. The TF-IDF method reduces the importance of terms that often occur in the collection, and em-

pathizes the unique terms of a document, which improves precision of grouping similar documents. There are several ways to calculate TF-IDF; Kayur uses the following formula [16]:

$$w_{ij} = tf_{ij} \times \log \frac{N}{df_i},$$

where tf_{ij} is the term frequency of the term t_i in the document d_j, df_i is the document frequency of the term t_i, and N is the number of documents in the whole collection.

3 Tool architecture and implementation

Kayur is an open-source cross-platform tool written in Java. It comprises: 1) the *information extraction* (web import) component to fetch data from the Internet, translate them into the uniform document format, and store the result in a database; 2) the *text processing* component to load documents from the database and process their contents; and 3) the *export* component to convert the results of text processing into formats of data mining workbenches. The complete workflow is shown on Figure 1.

In this section, we first describe the uniform document format, and then proceed to the description of each mentioned component.

3.1 The uniform document format

The structure of textual data to extract differs between web resources. For example, bug reports contain attributes such as "status" or "priority", while a firm catalog has fields "firm name", "location", and "phone". Because of that, data from different sources are not usually stored uniformly.

To simplify the configuration, storage, and access to documents obtained from the web, we propose the uniform format that describes a document as having the title, content, comments, date, and metadata. These fields are chosen to be appropriate for most textual documents on the Internet. The first three fields are designated to store textual data that need to be preprocessed before they can be used in a data mining tool. The metadata field is a storage for all other parameters as name/value pairs; it is designated for data that do not require processing and can be used as is. The proposed format thus generalizes special-purpose uniform formats of tools that work with textual data coming from a particular domain, such as bug reports or user messages in social networks.

The main benefit of such format is that configuration process is simplified by completely excluding the step of defining the structure of the output data. Therefore, a user only needs to specify the inputs in the form of extraction and navigation rules, as described in the following section. Another benefit is that documents from multiple web resources are stored uniformly, and hence they can be processed and analyzed together. Moreover, it becomes easier to share and reuse a selected set of subsets of documents from different resources, as it can be stored in a single consistent file.

3.2 Information extraction

The information extraction from a web resource is the first step towards dataset construction. This step often requires extensive configuration, as tools need to know the location and type of data to extract, how to navigate between documents, and what the structure of results is.

We propose a simplification of this process in the case when a web resource has two properties: a) every document in the collection has a Uniform Resource Locator (URL) that includes a unique identifier, and b) the web resource has a search engine to locate documents. Many web resources satisfy the second condition, as they are oriented for a human reader and hence provide convenient means to find documents of interest. The first property is usually satisfied for web resources that allow to view each document in a separate page.

If the mentioned conditions are satisfied, the number of extraction and navigation rules that user is required to specify can be significantly reduced. Moreover, the format of rules can also be simplified if the uniform format (discussed above) is being used.

3.2.1 Navigation rules

Suppose that the first condition is satisfied, so that the URL of each document has the form `http://...id...`, where `id` is a numeric or string identifier. For example, consider the following URLs:

- `https://code.google.com/p/Project/issues/detail?id=154684` (the identifier "154684" is numeric),
- `https://www.amazon.com/The-Name-Of-A-Commodity/dp/B00ZTK3JA0/` (the identifier "The-Name-Of-A-Commodity/dp/B00ZTK3JA0" is a string),
- `http://www.bbc.com/news/world-Region-Territory-38060919` (can use either a numeric identifier "38060919" or a string identifier "world-Region-Territory-38060919").

If an identifier is a number, then the navigation between documents can be simply defined by getting next the document with an identifier incremented by some predefined value (usually, just by one). No extraction rules are need to be provided, and a user only need to define one URL pattern that leads to a document.

When identifiers are strings, the list of available identifiers can be obtained using a search engine of the web resource. A typical search engine provides search results as a list of records that either lead to the desired documents or contain their identifiers. These identifiers (links) can then be collected automatically. The only parameters that user need to define is one extraction rule (to get an identifier (a link) from a search result), and three URL patterns: for a document, the initial search page, and a next page with search results. Table 1 summarizes the required user input for both cases.

Table 1: The minimum number of parameters to define navigation

Navigation Type	Extraction Rules	URL patterns
Incremental	0	1
Search-based	1	3

3.2.2 Extraction rules

Extraction rules specify which part of a web document is to be extracted. The format of extraction rules and the way they are set up varies between tools; the rules can be assigned manually or derived (semi-)automatically from a set of training pages by generalization of repetitive patterns [5].

In case of manual rule setup, the popular choice is the use of XPath-based expressions, which denote the full path inside the Document Object Model (DOM) tree of a document to a particular node. XPath notation is especially useful for structured XML documents, as the names of tags give hints on the type of data being stored; however, for generated HTML documents identification of data by HTML attributes, such as `id` and `class`, is more common. We propose to use the following notation that simplifies the manual input of extraction rules.

$$\begin{aligned} \textit{rule} &::= \text{`['}\ \textit{token}\ \text{`]'} \mid \text{`['}\ \textit{token}\ \text{`]'}\ \textit{rule} \\ \textit{token} &::= \textit{name} \mid \textit{name}\ \text{`,'}\ \textit{type} \mid \textit{name}\ \text{`,'}\ \textit{type}\ \text{`,'}\ \textit{index} \\ \textit{type} &::= \text{`id'} \mid \text{`cls'} \mid \text{`attr'} \mid \text{`tag'} \\ \textit{index} &::= -32,768\ ..\ 32,767 \mid \text{`*'} \end{aligned}$$

The data to extract is identified by a rule consisting of one or multiple *tokens*. One token is sufficient when the DOM element containing data has a unique property within the whole web document (for example, the "id" attribute). Otherwise, a pair of tokens can be specified, where the DOM element identified by the first token serves as a relative root for applying the second token. As an example, consider a trivial web page shown on Figure 2. Suppose we want to extract data

from the “p” tag located under the “div” element with the identifier “content”. The page contains many “p” tags, but the “p” tag of interest can be unmistakably identified by the two-token rule `[content][p,tag]`. If necessary, any number of tokens can be specified in such manner, where each token, except the last one, serves as the relative root for the subsequent token.

```
1  <body>
2    <p>Some text</p>
3    <div id="titleSection">
4      <p>Other text</p>
5      <p>Comment</p>
6    </div>
7    <div id="content">
8      <span>Some other data</span>
9      <p>Article...</p>
10   </div>
11 </body>
```

Figure 2: An example of an HTML page with multiple “p” tags

The *type* specifies how a DOM element is identified: by “id” attribute (*id*), by “class” attribute (*cls*), by some other attribute (*attr*), or by an HTML tag (*tag*). In the most frequent case of specifying a DOM element by its “id” attribute, the type can be omitted in the token. Therefore, the tokens `[name, id]` and `[name]` are equivalent.

Index specifies the position of a DOM element among siblings. The default value of zero can be omitted. A negative index specifies the position starting from the end of the sibling list. Thus, the token `[p,tag,-1]` specifies the last “p” tag in the document. Finally, the asterisk symbol indicates that the token denotes a set rather than a single element; all subsequent tokens are applied to every DOM element in the set.

To demonstrate the application of a token that captures multiple elements, consider the HTML page shown on Figure 3. The rule `[comment,cls,*][p,tag,1]` can be used to extract all comments from the page. The first token denotes all elements that have the attribute “class” with the value “comment”. There are three such elements, and each of them is set as a relative root for the second token `[p,tag,1]`. The second token denotes the second “p” tag, and is computed against each of the relative roots. As a result, the rule produces the list of three elements: “Comment 1”, “Comment 2”, “Comment 3”.

To extract the description at the bottom, we can use a simple rule `[descr]`. For a well-marked HTML page with identifiers, the set of extraction rules can be much cleaner and more compact comparing to the traditional XPath expressions. This simplifies the manual input, reduces the number of possible input errors, and facilitates the maintenance of the rules over time.

An extraction rule may be accompanied by a filtering expression that allows omitting unneeded part of the data extracted from a DOM element.

3.2.3 Information Extraction Algorithm

Once extraction rules are specified, the web import component fetches bug reports from a remote source in the following way:

1. If numeric identifiers are being used, the identifier of the first document to get is read from settings; otherwise, the list of all available identifiers is obtained by parsing search results.
2. The URL of the next document to fetch is determined by its identifier.
3. The web page with the given URL is parsed, and fields of a structure in the uniform document format are filled according to the extraction rules of the current module.
4. The structure holding the document is stored in the database.
5. The process is repeated until all available documents are processed, or a user-defined limit is reached.

```
<body>
  <div class="comment">
    <p>2016/10/09 12:16</p>
    <p>Comment 1</p>
  </div>
  <div class="comment">
    <p>2016/10/09 13:08</p>
    <p>Comment 2</p>
  </div>
  <img src="../ad/image.png"/>
  <div class="comment">
    <p>2016/10/09 18:44</p>
    <p>Comment 3</p>
  </div>
  <div>
    <div>
      <p id="descr">Description</p>
    </div>
  </div>
</body>
```

Figure 3: An example of an HTML page to demonstrate extraction rules

3.2.4 Example: Comparison of Information Extraction in Different Tools

In this subsection, we demonstrate the application of the described rules to an actual web resource. We show how the same task can also be accomplished in two other information extraction tools: Web Scraper [3] and Scrapy [1].

We selected a news portal as a typical web resource that serves textual documents and provides functionality to perform a custom search for the data of interest. The content to extract is 13 articles located on two pages. For each article, we extract its title, text, and the date of the publication. The web page containing the list of selected articles and its structure are shown on Figure 4.

Extraction with Kayur To extract the title, text, and date from each article, we need three extraction rules for data, and one navigation rule to obtain articles from the list of search results. Because the data of interest have unique class identifiers within an article, each extraction rule can be specified with only one token (see Table 2). In addition, we specify the date format to parse the date of the publication, and store it in the database as a time stamp.

We also need to set up three URLs to locate 1) an article, 2) the initial search page, and 3) the following search pages. The single navigation rule specifies the set of links to articles on each page containing search results. Table 2 summarizes the necessary configuration to extract data from this particular news portal.

Extraction with Web Scraper Web Scraper [3] is a browser extension for information extraction from web resources. The extraction rules are specified by visual selection of entries of interest from a web page. Navigation between pages is supported using pagination and other user interface elements for content loading.

To configure Web Scraper for obtaining data from the news portal we selected, three layers of element *selectors* are specified for: 1) navigation between pages containing search results, 2) selecting articles from search results, and 3) extracting the title, text, and date from a particular article. The latter selectors are shown on Figure 5.

Because the same HTML elements are selected, the extracted data is the same as when using Kayur.[1] Moreover, visual selection is easier than specifying the rules in the textual form, so for many web resources the configuration of data extraction can be quicker with Web Scraper. However,

[1]The minor difference is that the version of Web Scraper used for experiments concatenated some words on the boundaries of inner tags located inside the HTML element from which the text of an article is extracted. In such cases, Kayur correctly separates the data by whitespace.

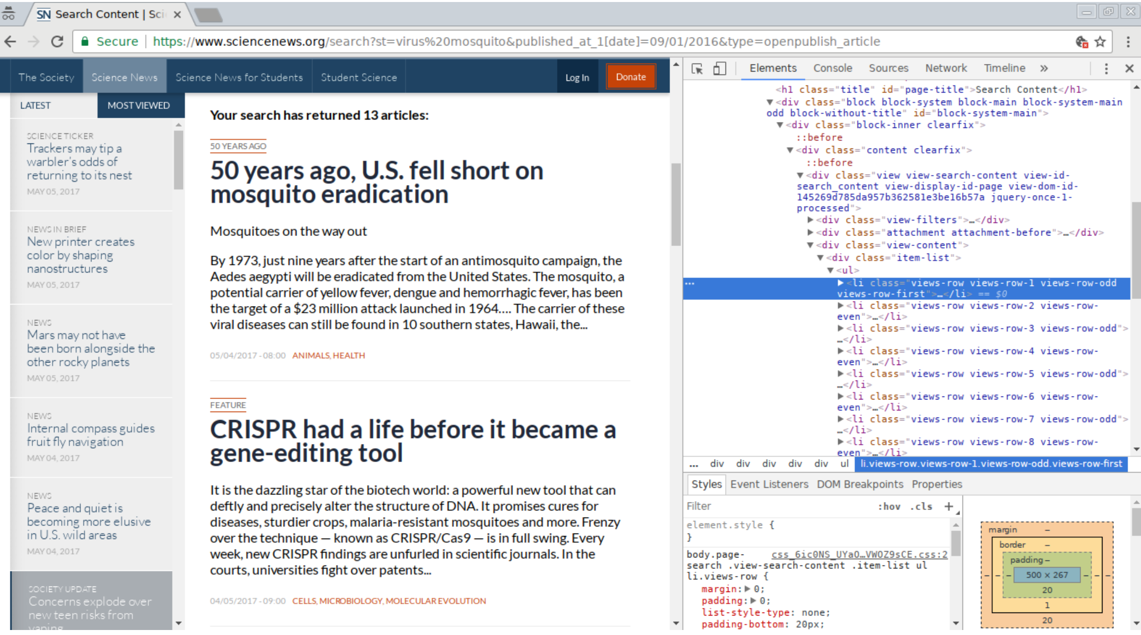
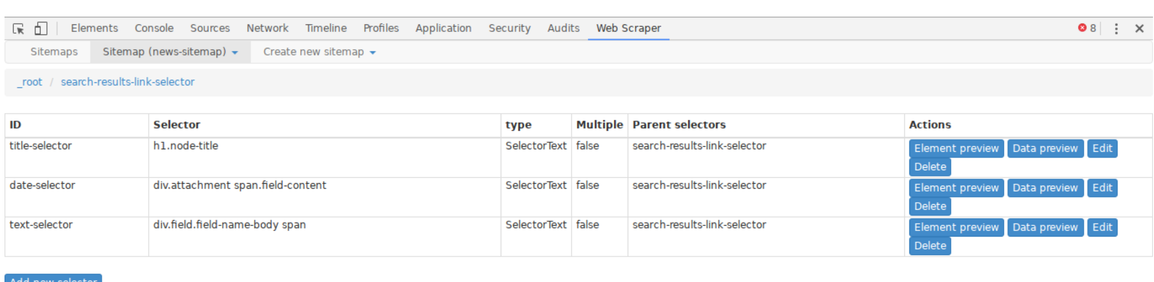


Figure 4: The page containing custom search results on a news portal, and its HTML structure.



ID	Selector	type	Multiple	Parent selectors	Actions
title-selector	h1.node-title	SelectorText	false	search-results-link-selector	Element preview Data preview Edit Delete
date-selector	div.attachment span.field-content	SelectorText	false	search-results-link-selector	Element preview Data preview Edit Delete
text-selector	div.field.field-name-body span	SelectorText	false	search-results-link-selector	Element preview Data preview Edit Delete

Figure 5: Page-level selectors (extraction rules) used by Web Scraper

Table 2: Configuration of the information extraction component for a news portal

Data	Value
Extraction	
Title	`[node-title,cls]`
Content	`[field-name-body,cls]`
Date and Time	`[view-display-id-attachment_1,cls]`
Date Format	HH:mma, MMM dd, yyyy
Document URL	https://www.sciencenews.org${id}
Navigation	
Result Links	`[view-search-content,cls][views-row,cls,*][a,tag][href,attr]`
Search URL	https://www.sciencenews.org/search?st=virus%20mosquito&published_at_1[date]=09/01/2016&type=openpublish_article
Search Page	https://www.sciencenews.org/search?st=virus%20mosquito&published_at_1[date]=09/01/2016&type=openpublish_article&page=${page}

visual selection is not possible for resources that provide Application Programming Interface (API) to export data in a structured format, such as XML, which is treated by Kayur the same way as HTML. The other difference with Kayur is that Web Scraper exports the data to files in CSV format, while Kayur stores the results in a centralized database, which makes easier to search for documents that match particular criteria, as well as select subsets that contain documents obtained from multiple resources.

Extraction with Scrapy Scrapy [1] is a Python library for information extraction. Obtaining data from a web resource is handled by a separate module, called a spider, which defines the extraction procedure in Python language. The extraction rules are specified using Cascading Style Sheets (CSS) and XPath selectors. The navigation is performed by first extracting links from a web page using extraction rules, and then adding the links to the list of the requests to process.

We have written a spider for the selected news resource, and verified that it produced the same results as Kayur and Web Scraper. Because the spider configuration is actually program code, it is extremely flexible, so that Scrapy can be configured to obtain data from web resources with non-standard navigation mechanisms. However, this approach requires knowledge of the Python programming language and CSS/XPath selectors to efficiently use the tool. The data is exported in JSON format, which has the same limitations as CSV, described above.

Summary Information extraction from web resources requires substantial effort to configure the extraction and navigation rules. The rules can be specified in a textual form (Kayur), using visual aids (Web Scraper), or in a programming language (Scrapy). The information extraction component of Kayur is designed to maximize usability by utilizing the straightforward syntax, so that knowledge of XPath expressions or programming languages is not required to use the tool. Moreover, the configuration is separated from the crawler's logic in an XML file, so that it can be easily reused and shared between different machines. However, use of stand-alone tools have the advantage of flexibility, allowing to fetch more types of data (images, video) from resources with different navigation mechanisms.

3.3 Storage

Documents obtained from different web resources are stored in the same database using the uniform format. This allows constructing datasets using arbitrary subsets of documents from multiple

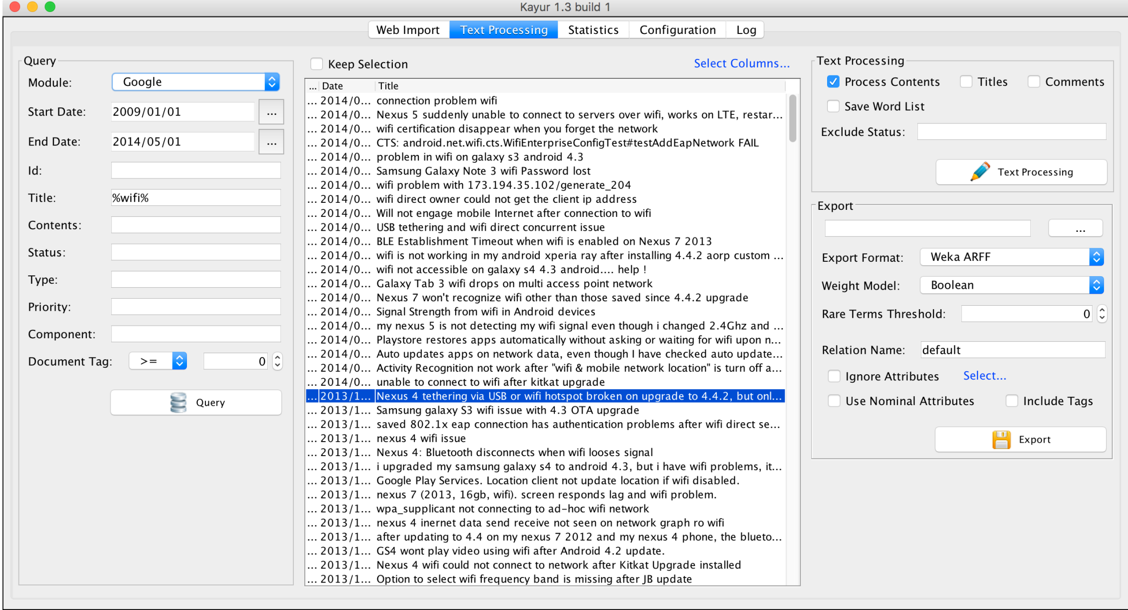


Figure 6: Screenshot of Kayur (Text Processing View)

sources. Documents from different sources can be selected for inclusion in a dataset based on a module identifier, date range, or metadata.

To simplify initial configuration, Kayur uses the embedded Derby[2] database, which is initialized on the first usage. The tool can be configured to use another database, such as MySQL, instead; the tables to store data will be generated automatically (provided that the database user specified in the settings has all necessary privileges).

3.4 Text processing

The text processing component performs a sequence of operations on the textual data of selected documents to remove irrelevant information and improve recognition of similar documents. Each operation is optional and customizable. The complete processing sequence includes:

1. Initial filtering.
2. Sentence detection.
3. Tokenization and conversion to lower case.
4. Part-of-Speech (word class) tagging and application of stemming routines.
5. Word filtering.
6. Stop-word removal.

The core operations of tokenization, sentence detection, and Part-of-Speech tagging are performed using OpenNLP [22]. This library relies on binary model files for English language that can be obtained from the website of the OpenNLP project [23].

Kayur includes two types of customizable filters in the format of Java regular expressions. The *initial filters* are applied to the whole text of a document to replace or remove structural elements (such as "Bug description:", "Steps to reproduce:" in bug reports) and non-alphanumeric symbols. The *word filters* do not have replacement functionality, and simply remove words that match a

[2]https://db.apache.org/derby/

predefined pattern. They are especially useful to remove measurement units, hexadecimal numbers, hyperlinks, and file names.

The component includes custom stemming routines that are applied only to tokens that are detected by Part-of-Speech tagger as:

- Noun, plural.
- Verb: third person singular present, gerund, present participle, past tense, or past participle.

The stemming is performed accordingly to English grammar rules; irregular verbs are converted to dictionary form by the (customizable) list of such forms. Because the tagger can mistakenly detect other parts of speech as verbs, the converted word is validated against the large list of all possible verbs from `libmind` library [15]. If the result is not found in the list, the conversion for this word is skipped.

The last step of text preprocessing operation is stop word removal. Stop words are common words such as articles or pronouns. The tool uses the initial stop word list from MALLET project [18]. The list can be further expanded to include high-frequency words of no particular importance (e. g., "bug", "problem", or "issue").

The tool builds a cache to speed up text processing. Each time a new word is processed, the cache stores the mapping between the word and its final form (or an empty string in case when the word is to be removed).

Once all text processing steps are finished, Kayur saves a corpus structure that contains processed documents, and displays a statistics window that presents a short summary of the data. The summary includes pie charts for metadata, the list of top keywords in the corpus, and length distribution of processed documents.

3.5 Export

The export component uses a corpus structure prepared during the text processing stage to create an output file in the selected format with a specified term weighting method. Kayur supports two integer weighting methods: Boolean model and raw frequency, and two floating point weighting methods: 1) term frequency and 2) term frequency / inverted document frequency (TF-IDF). Kayur supports the following output formats:

1. Attribute-relation file format (ARFF) of WEKA.
2. Plain text format of LDA/HDP topic modeling tools by J. Chang and C. Wang (only integer weights).
3. XML format of Carrot2 (only integer weights).[3]
4. CSV format, supported by wide range of applications, including Microsoft Excel and R [29]. An example of loading the exported file in R is shown on Figure 7.
5. JSON format, which is supported by web applications.

```
data <- read.csv("/path/to/exported/file/dataset.csv")
View(data)
```

Figure 7: Loading the exported CSV file in R

A user can also set a *rare term threshold* to exclude words that occur in less than a specified number of documents in the corpus. This is important for all text mining tasks, as it greatly reduces the dimension of the resulting vector space, increasing the performance of a data mining tool and reducing resource usage.

[3]Vector space model is not used when exporting to Carrot XML format, because Carrot2 works directly with textual data. The input file is created by composing documents from processed words.

3.6 User interface and configuration

Kayur supports both a graphical user interface (GUI) and a command line interface (CLI) designed as an interactive shell (read-eval-print loop). The capabilities of both interfaces are the same, except that the latter supports scripts comprised of valid Kayur commands.

The GUI consists of a single window with five tabs for: 1) the web import component, 2) the text processing component, 3) statistics, 4) program settings, and 5) the journal that contains log messages for the current session (see Figure 6). The GUI is especially useful for initial calibration, as it allows a user to quickly change and test different combinations of parameters.

Although the tool is preconfigured for typical usage scenarios, the GUI allows tuning almost every step of the processing chain to suit particular needs. Web import modules can be freely adjusted to produce desired results. The default internal Derby database can be replaced with a stand-alone database such as MySQL or PostgreSQL. Text processing routines can be disabled or their behavior can be changed by using an external library instead (a call to a library method must be wrapped by a class that implements the interface provided by the tool).

4 Case Studies and Evaluation

In this section, we show two more examples of using Kayur for information extraction. The examples cover two large classes of web resources: bug tracking systems and portals that store user opinions. We also analyze the performance of the tool for both information extraction and text processing stages.

4.1 Case Studies

4.1.1 Bug Tracking Systems

Bug reports are invaluable source of important information for software developers. Beyond a manual analysis of bug reports, there is also a need to process sets of bug reports as a whole using text mining techniques, such as clustering, to discover trends in software flaws.

Bug reports are usually available on web pages generated by bug tracking systems (BTS). While most of BTS support exporting the data into structured formats (XML, JSON), there are also resources that provide bug reports in HTML format only. In both cases, the pages containing bug reports are often well-structured, so that each bug report element, such as the title, text, priority, and status, are assigned unique identifiers or Cascading Style Sheets (CSS) classes. This greatly simplifies extraction rules, so that they can usually be written using only one token. Moreover, as BTS often use numeric identifiers for bug reports, the navigation rules can be omitted.

Table 3: Extraction rules for BTS based on Bugzilla

Data Field	Extraction Rule
Title	`[short_desc,tag]`
Content	`[long_desc,tag][thetext,tag]`
Comments	`[long_desc,tag,*][thetext,tag]`
Date	`[creation_ts,tag]`

For example, the settings shown in Table 3 are sufficient to generate a dataset from the Gnome bug tracker[4] based on a popular BTS Bugzilla. Each bug report is assigned a number, so that we can fetch a range of documents without the need for navigation rules. Furthermore, this BTS allows accessing bug report data in the XML format,[5] which simplifies the rules. In fact, because all BTS based on Bugzilla have the same structure, we included in Kayur the BTS wizard that can generate

[4]https://bugzilla.gnome.org/
[5]https://bugzilla.gnome.org/show_bug.cgi?ctype=xml&id={*identifier of the bug report*}

extraction rules only by the domain name of the BTS. The wizard also supports JIRA, which is another popular BTS.

4.1.2 Opinion Mining

Another popular topic that involves web mining is automatic analysis of user opinions. Such analysis is useful, for example, for companies to understand how their products and services are perceived [27]. Kayur makes it easy to create a dataset from web documents that contain user opinions, as they match the uniform format.

As an example of such a web resource, consider Amazon, which is one of the largest electronic shopping portals. The documents on this resource are descriptions of products from the catalog, which also contain the list of customer reviews. The main difference from regular documents is that the extraction rule for reviews should contain an asterisk symbol to specify that it returns a set of entries rather than a single piece of data. A sample configuration to obtain a product description with customer reviews from a single page on Amazon is shown in Table 4.

Table 4: Extraction rules for customer reviews at Amazon

Data Field	Extraction Rule
Title	`[productTitle]`
Content	`[productDescriptionWrapper,cls]`
Comments	`[revMHRL][a-section,cls,*]`
	`[a-spacing-small,cls][a-section,cls]`

4.2 Performance Evaluation

The performance of the tool in terms of processing speed is determined by download time and text preprocessing time. The download time depends on external conditions, such as network speed, bandwidth, and the responsiveness of web resources. Moreover, the tool cannot significantly reduce the download time by opening large number of connections to the same resource, as recommended by RFC 2616 [7]. Because the download time is the biggest factor affecting performance, which happens outside the tool, we do not compare the performance of Kayur to other web scraping tools described in Section 3.2.4.

Under normal conditions, for responsive resources and a network speed of 50 Mbps, the web document preprocessing rate is expected to be between 2 and 3.5 documents per second. The rate is significantly lower for resources that forbid automatic data fetching (such as Stack Overflow [2]), because a delay of up to a few seconds must be set between subsequent accesses.

The performance of the text processing component on a test machine,[6] with all processing steps enabled and default OpenNLP library plugin, is 50–60 documents per second. The main performance factor for text processing is the Natural Language Processing (NLP) plugin in use, which can be replaced if for a particular task the processing speed is not satisfactory. In addition, the word cache can be enabled to speed up stemming, which reduces the execution time on average by 20 %.

5 Application: Identification of the most frequent Android defects

Above, we have shown several small examples, demonstrating the use of the tool for processing the data from standalone web resources. In this section, we describe how Kayur can be used in a more complex task of identifying the most frequent Android defects, using multiple data sources.

This particular topic has been chosen, as the reliability of Android platform is an important issue due to its wide-spread use (86.8 % of the smartphone market share [9]) and the fact that it

[6]Intel Core i5-3210M CPU 2.50 GHz, 4.0 GB RAM, Windows 7 32-bit

operates on sensitive, confidential user data. The information about frequent defect types may hence be helpful to raise awareness among Android application developers, educators, and developers of software analysis and test generation tools.

5.1 Methodology

We determine the most frequent defect types on Android by the semi-automatic analysis of a large subset of bug reports maintained at Google Android Issue Tracker [8]. The Tracker is the main bug report repository for issues related to the Android platform itself and its core applications. Although the Tracker provides some metadata, such as bug status and priority, the most valuable information about the nature and origin of occurred problems is not directly available.

The size of the selected collection of bug reports makes manual analysis inefficient. To identify trends in Android defects, we use *unsupervised clustering*, a method to group similar bug reports when the topics are not known a priori. To achieve a meaningful and correct list of bug categories, we rely on better data preprocessing and selection of the most suitable clustering algorithm.

Better data preparation greatly affects the quality of obtained results [28]. In addition to the default Kayur settings, we improve the preprocessing process by configuring additional filters and adding stop words specific to the Tracker. The quality of filtering is assessed by monitoring the output list of words in the whole document collection.

The clustering algorithm to be used is determined by evaluation of all suitable clustering algorithms of WEKA and Carrot2. The evaluation is performed on another set of Android bug reports with known classification from US National Vulnerability Database [20]. We evaluate each algorithm on every supported term weighting method (see Section 3.5). We employ Kayur to quickly generate the necessary number of datasets in different formats.

When the optimal algorithm and term weighting method are chosen, we perform unsupervised clustering of the main set of reports from the Tracker. Each result cluster is thought of as representing a single issue. We infer the information about each issue using the list of the most descriptive keywords in the corresponding cluster. These keywords are words that appear often in a given cluster but rarely in others. We further confirm or correct the description of each cluster by analyzing topics of documents within the cluster, and, when necessary, by manual sampling of documents from the cluster.

5.2 Dataset Generation

For our purposes, we need a collection of small control datasets for clustering algorithm evaluation, and the main dataset for the analysis. As the main set of documents, we consider a subset of bug reports published on the Tracker from January 1, 2014 to January 1, 2015. Using Kayur's filtering by metadata, we exclude the following irrelevant documents: (i) labeled as “spam”, “duplicate”, “working as intended”, “user error”, or “declined”; (ii) related to Android Software Development Kit (SDK) tools (such as Android Debug Bridge or Android Virtual Device manager) and documentation; and (iii) questions and user suggestions. The remaining subset consists of 7,629 bug reports, and includes both solved and actual problems. We use only the actual content of bug reports, and omit comments, because for this particular dataset they often contain irrelevant information, such as off-topic talks or discussions about status of a bug report. The output of the preprocessing of this subset contains 6,878 non-empty documents with 6,034 distinct words, resulting in a sparse document-term matrix with density 0.45 %.

As the control (training) set, we use a collection of reports on Android vulnerability issues from the US National Vulnerability Database [20]. The collection consists of 1795 documents published from November 17, 2006 to February 22, 2015. The documents are divided into six groups according to Common Weakness Enumeration (CWE) classification. Each group contains documents of the same CWE family, and includes one or more particular CWE classes, as shown in Table 5.

We use a search-based Kayur fetching module to locate and obtain this collection of documents, and preprocess them using the default settings. Next, we generate 18 datasets in ARFF format to test each of the six WEKA algorithms (see below) with three different term weighting methods

(Boolean, raw term frequency, and term frequency / inverted document frequency (TF-IDF)). We also generate three datasets in Carrot XML format to evaluate the clustering algorithms of Carrot2.

5.3 Evaluation of clustering algorithms

We first perform a cross-validation of various clustering algorithms and settings on a labeled dataset containing vulnerabilities, and choose the best clustering algorithm to classify issues from the Android bug database. The most suitable algorithm is chosen based on the standard set of metrics, such as F-measure and accuracy, but the run time of the algorithm is also considered, as the unlabeled bug report data set is much larger than the training set.

5.3.1 Evaluation of WEKA Clustering Algorithms

We evaluate six WEKA clustering algorithms: Farthest First, Simple K-Means, Hierarchical Clusterer, Sequential Information Bottleneck (sIB), Expectation Maximization (EM), and XMeans. The remaining algorithms (CobWeb, CLOPE, DBScan, and OPTICS) are omitted as not being suitable for our purposes. Each clustering algorithm is executed with each input file, forming 18 possible combinations. Additionally, for some algorithms, such as Simple K-Means and Hierarchical Clusterer, experiments are performed with different input parameters. For all experiments, the number of clusters to generate is set to 6, equal to the number of groups in the control document collection.

The output of clustering is used to calculate accuracy, precision, recall, and F-measure of the algorithm. The calculated measures are presented in Table 6 (measures for individual classes are omitted).[7] In case when an algorithm is executed with different parameters, only the best acquired results are shown.

5.3.2 Evaluation of Carrot2 Clustering Algorithms

We evaluate all three Carrot2 clustering algorithms (Bisecting K-Means, Lingo, and Suffix Tree Clustering (STC)) using the same training set as for WEKA. However, Carrot2 algorithms work directly with text, so we use Kayur to export preprocessed data as text, but the term weights are calculated by Carrot2 itself.

[7] Three columns that contain precision are omitted for brevity. For Simple K-Means, the results are presented for the seed that minimizes sum of squared errors within cluster. Other WEKA algorithms were executed with default seeds.

Table 5: Input dataset for evaluation of clustering algorithms

Group	Documents		Family/Class Name	
	total	per class		
1	1412	1412	CWE-310	Cryptographic Issues
2	140	140	CWE-119	Buffer Errors
3	99	92	CWE-264	Permission, Privileges, etc.
		7	CWE-287	Authentication Issues
4	70	38	CWE-20	Input Validation
		13	CWE-22	Path Traversal
		11	CWE-79	Cross-Site Scripting
		8	CWE-94	Code Injection
5	46	46	CWE-200	Information Leak / Disclosure
6	28	28	CWE-189	Numeric Errors

Table 6: Evaluation of WEKA clustering algorithms on the training set

Algorithm	Time	Boolean Model			Raw Frequency			TF-IDF		
	(sec.)	Acc.	Recall	F_1	Acc.	Recall	F_1	Acc.	Recall	F_1
Farthest First	0.1	84.18 %	0.818	0.818	79.94 %	0.762	0.762	81.11 %	0.811	0.743
Simple K-Means	1.7	87.74 %	0.877	**0.880**	88.75 %	0.888	0.870	88.64 %	0.886	**0.877**
sIB	180	89.58 %	0.896	**0.884**	89.97 %	0.890	**0.893**	85.07 %	0.859	0.839
EM	16	87.41 %	0.874	**0.880**	81.06 %	0.811	0.829	78.38 %	0.784	0.819
XMeans	0.7	88.47 %	0.885	0.870	88.58 %	0.886	0.869	88.69 %	0.887	0.869
HC (Ward)	314	86.57 %	0.866	0.859	87.63 %	0.876	0.862	87.52 %	0.875	0.864

Table 7: Evaluation of Carrot2 clustering algorithms on the training set.

Algorithm	Accuracy	Precision	Recall	F-Measure
Lingo (average)	85.6 %	0.866	0.856	0.854
Lingo (max F_1)	86.2 %	0.878	0.862	0.864
Lingo (min F_1)	84.7 %	0.851	0.847	0.846
STC (average)	82.2 %	0.771	0.822	0.793
STC (max F_1)	83.2 %	0.791	0.832	0.809
STC (min F_1)	81.3 %	0.752	0.813	0.778
Bisecting K-Means	87.9 %	0.873	0.879	0.861

Measures calculation for Bisecting K-Means is straightforward, but Lingo and STC require modified approach for the following reasons. First, Lingo produces more clusters than requested, and we observed that small clusters are often contained entirely in larger clusters. To reduce the number of clusters closer to the number of classes (6), we remove all such "inner" clusters.

Second, both Lingo and STC produce overlapping clusters. We transform them to non-overlapping clusters by assigning every document that belongs to clusters $c_1, ..., c_N$ to only one randomly chosen cluster c_k, $1 \leq k \leq N$. Because measures depend on document assignments, we calculate maximum, minimum, and averages of values obtained in 1000 executions.

Unlike Bisecting K-Means and WEKA algorithms, Lingo and STC may leave some documents unassigned to a cluster. In particular, both algorithms leave 75 % of documents from the training set unassigned if term weights are not TF-IDF, so that we perform measure calculations for TF-IDF case only. The results of evaluation are presented in Table 7.

5.3.3 Results of Algorithms Evaluation

To achieve better results in clustering bug reports from the Google Android Issue Tracker, we are particularly interested in algorithms with high F-measure and accuracy. According to Table 6, for WEKA these are *sIB* and *Simple K-Means*. Considering the size of Google dataset, we choose Simple K-Means for performance reasons. As the difference in F-measure for Boolean and TF-IDF model is negligible for this algorithm, we will use TF-IDF as the more commonly used term weight method.

According to Table 7, we achieved relatively high F-measure for both Bisecting K-Means and Lingo (w.r.t. Simple K-Means, −0.019 and −0.026 respectively). We omit Bisecting K-Means, because it is an iterative clustering algorithm similar to Simple K-Means that is already selected for experiments. On the other hand, Lingo could be a good candidate for clustering bug reports from the Google dataset, because it is designed to produce descriptive cluster labels, so that the labeling step could be simplified [25]. However, it is recommended to limit the size of the input to be

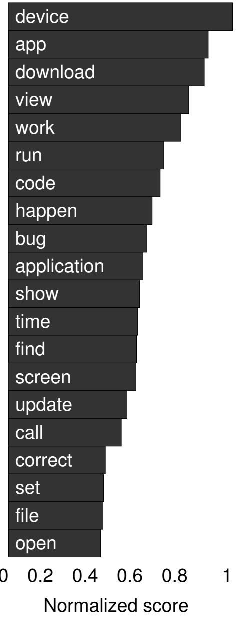


Figure 8: Top keywords among all clusters.

no more than 1000 documents to achieve meaningful results [26]. The preliminary tests confirmed the problem: on the Google dataset, Lingo leaves absolute majority of documents unassigned to any cluster. Although Lingo cannot be applied to Google dataset directly, we use it for cluster subdivision during the labeling phase to reveal inner structures of clusters.

5.4 Bug Reports Clustering

We used Kayur to export the processed Google bug report dataset using the chosen TF-IDF term weight model into WEKA's ARFF input format. The minimum document frequency of words was set to 3 (i.e. the words that occur in only one or two documents in the collection are ignored), as it offers a good compromise between quality and performance of clustering.

We loaded the generated file into WEKA, and executed Simple K-Means on the prepared dataset 100 times, varying the seed, to find partitioning such that sum of squared errors within cluster was minimum. Multiple executions were feasible due to relatively short run time of Simple K-Means on the dataset (approximately 30 minutes on our machine). The partitioning with minimum error comprised 22 clusters of size greater than 50, and 15 smaller clusters. For each of 22 clusters of medium and big size, we generated lists of the most frequent 500 keywords, and used them for cluster labeling.

To label the clusters, we first identified the cumulative score S_{kwd} for each keyword k among all N clusters $c_1, ..., c_N$:

$$S_{\text{kwd}}(k) = \sum_{i=1}^{N} S_{\text{kwd}}(k, c_i),$$

where $S_{\text{kwd}}(k, c_i)$ is the number of occurrences of keyword k in cluster c_i. This allowed us to see the top overall keywords (see Figure 8) and also the average score of each keyword across all clusters. In the figures, we normalized the score so the top score in each plot equals 1.

Within each cluster, we subtracted the average score of that keyword from each score, to see how *distinctive* a keyword is. Some keywords, like "app", occur frequently in many documents of most clusters. We are interested in keywords that occur frequently in a given cluster, compared to the average occurrence/score of a keyword across all clusters. We used normalized scores S_{norm} by setting the maximal keyword score in each cluster to 1, by dividing scores in each clusters by the

maximal raw score:

$$S_{\text{norm}}(k,c) = \frac{S_{\text{kwd}}(k,c)}{\max\limits_{\forall k \in \text{keywords}(c)} S_{\text{kwd}}(k,c)}$$

We used the relative frequency of keywords in each cluster so large clusters do not automatically have a high score for most keywords. Based on these normalized scores we calculated the difference in scores S_{diff} between each keyword in a given cluster c, compared to the average normalize scores in all clusters:

$$S_{\text{diff}}(k,c) = S_{\text{norm}}(k,c) - \frac{1}{N}\sum_{i=1}^{N} S_{\text{norm}}(k,c_i)$$

Using this formula, we obtained the most distinctive keywords for each cluster; in the plots, we again normalized the values for easier comparison (see Figures 9 – 10). We can see that the top keywords for the first cluster are quite generic, so about one third of the bugs cannot be classified easily. For all the other reports, we can assign each cluster to a topic relatively well based on the top keywords, although there is some overlap between smaller clusters.

A further analysis of each cluster is performed by its subdivision using Lingo and Bisecting K-Means algorithms from Carrot2 workbench. Additionally, adjustments to initial labels were made based on the title and sometimes full text of random samples. As shown in Table 8, we can obtain a meaningful label for each cluster except for the first, largest one. The largest cluster contains about 38 % of all issues, and it cannot be readily labeled because it contains bug reports related to many different issues.

5.5 Results

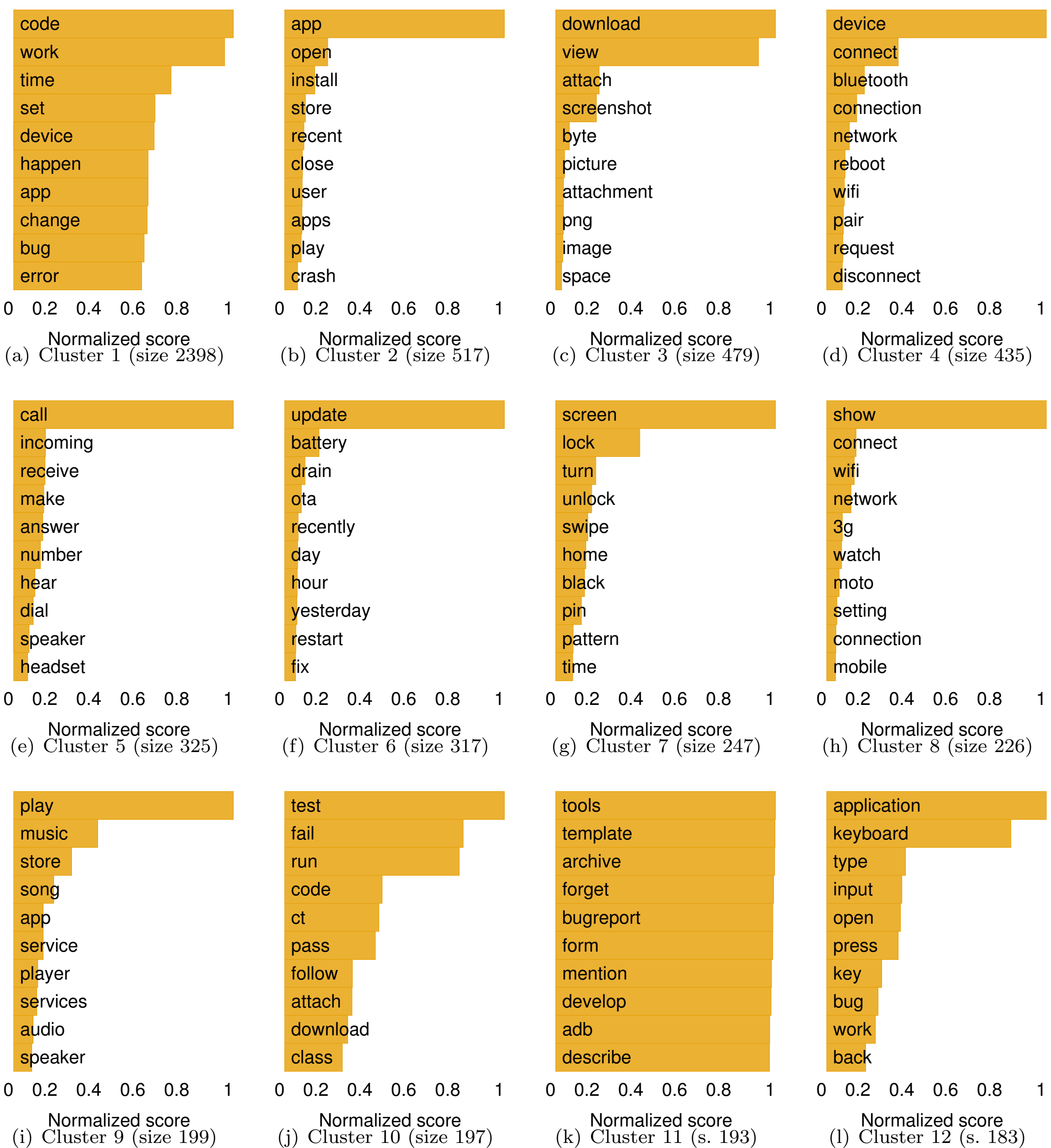
Using the initial labeling, we combined the reports into 12 groups of issues. On a high level, there are two kinds of issues: ten classes of problems affecting an application or end user directly, and two types of problems (Application Programming Interface (API) usage, testing) affecting developers (see Table 8). While user-related problems seem to make up the bulk of the bug database (54 % of all issues), problems in libraries tend to affect many applications, so that group should not be neglected solely based on its small size. Our results show that software updating and handling unreliable network connectivity are the two key concerns, which is not surprising as much research is dedicated to these difficult topics (for example, [14, 38]). We think these two issues deserve more attention in the context of the Android system.

The other important class of problems contains issues related to the user interface (UI). Subdivision and sampling from the corresponding cluster revealed that this class includes the following problems: overlapping, missing, or inactive UI elements; UI design flaws that adversely affect user experience; and improper or misleading translation for languages other than English.

Finally, issues related to phone calls handling and operation of screen and device buttons (Home, Back, Menu, Volume, Power) make the forth and fifth class of important Android issues. These problems are unique to mobile devices such as smartphones/tablets, and therefore require special treating, as traditional bug detection techniques for desktop applications cannot be applied directly.

5.6 Application: Summary

We demonstrated the use of the tool for a real-world example of identification of the most frequent Android bug types, which involved the large-scale document processing and unsupervised clustering. Kayur has been useful for automatic document fetching, as it allowed us to obtain a large set of bug reports; for document preprocessing, as it helped to improve the detection of similarities between documents by a clustering algorithm; and for generation of a large number of datasets in different formats. The quality of the preprocessing is further confirmed by obtaining the meaningful bug type categories.



(a) Cluster 1 (size 2398) (b) Cluster 2 (size 517) (c) Cluster 3 (size 479) (d) Cluster 4 (size 435)

(e) Cluster 5 (size 325) (f) Cluster 6 (size 317) (g) Cluster 7 (size 247) (h) Cluster 8 (size 226)

(i) Cluster 9 (size 199) (j) Cluster 10 (size 197) (k) Cluster 11 (s. 193) (l) Cluster 12 (s. 183)

Figure 9: Top distinctive keywords in clusters 1 – 12.

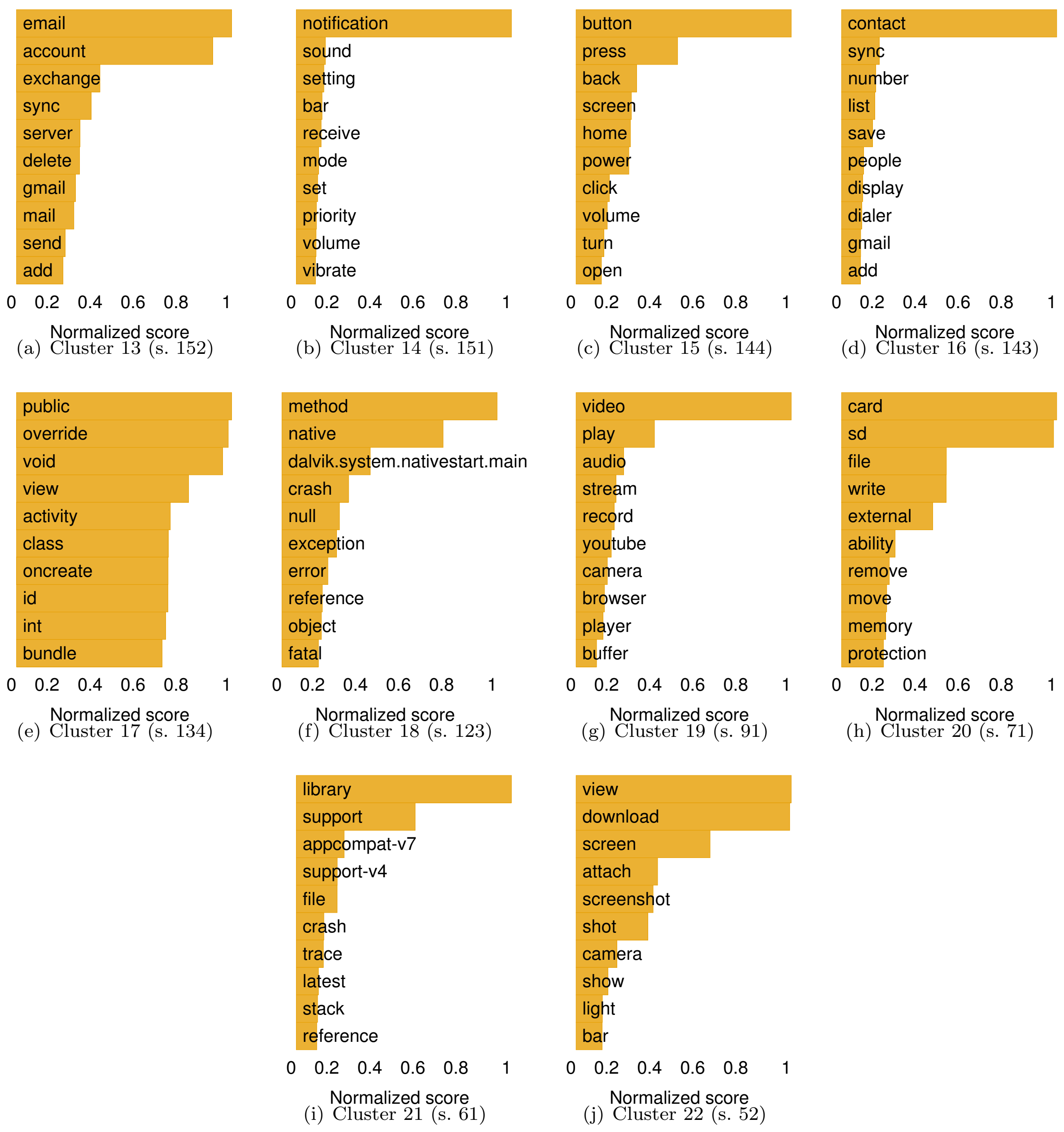


Figure 10: Top distinctive keywords in clusters 13 – 22.

Table 8: Summary of the 12 classes of Android bugs

	Topic	Clusters	Issues	Cluster labels of individual clusters
1	Software update	2, 6	834	Recently installed app crashes, software update
2	Networking	4, 8	661	Connectivity, networking
3	UI	3	479	
4	Phone calls	5, 16	468	Phone call handling, contact list
5	Lock/unlock, buttons	7, 15	391	Screen lock/unlock, hardware buttons
6	Media playback	9, 19	290	Music playback, video playback
7	Keyboard input	12	183	
8	E-mail	13	152	
9	Notifications	14	151	
10	SD cards	20	81	
a	API	2, 3, 17, 18	257	API usage, exception/crashes in API
b	Testing	10	197	
	Others	1, 11	2591	Miscellaneous; original report is removed

6 Related work

Nayrolles and Hamou-Lhadj point at the problem of automatic data extraction from diverse web resources. They introduce the BUMPER environment [19], which has different purpose than Kayur,[8] but also leverages the idea to store information from multiple web resources into a single database in the uniform format. Similar to Kayur, BUMPER provides the ability to export data from the database into JSON, XML, and CSV formats, but not in data mining tool's formats such as ARFF or Carrot XML. BUMPER exports the data as is, and does not perform any preprocessing, such as stemming or converting the data into the vector space model.

TraceLab [10] is a highly customizable general-purpose framework for setting up experiments in the form of a data processing tool chain composed of components that are either built-in or created by a user. Compared to Kayur, TraceLab offers richer text processing and visualization capabilities and is more flexible, as it allows to arrange the components in arbitrary way to obtain desired data flow. Although the functionality of TraceLab and Kayur overlap on the text processing stage, Kayur offers the following benefits: 1) uniform access to any web resources that store documents in HTML/XML formats; 2) a persistent storage for fetched documents; 3) a simpler user interface with predefined settings that minimizes time and effort for preparing a dataset from an arbitrary web resource, even if it does not provide API to get data in a structured format; 4) support of popular data mining workbench formats.

There is also a variety of stand-alone tools to perform subtasks corresponding to those of Kayur's components. For example, the task of information extraction can be handled by Web Scraper [3], Apache Nutch [21], or Scrapy [1]. The stand-alone tools provide greated flexibility to handle wider range of types of web resources; however, Kayur have the advantage in easier and more straightforward configuration, provision of the centralized data storage, and interconnection with text preprocessing routines that minimizes the time to generate datasets for analysis with data mining workbenches.

Text processing can also be performed by stand-alone tools based on powerful toolkits, such as NLTK [17] and GATE [6], or even directly in some data mining workbenches, including Carrot2, which implements tokenization, stemming, and stop-word and rare-term removal. Input files for WEKA and Carrot2 can be generated by conversion utilities from other formats such as CSV or XML. The mentioned tools can be arranged to work together to generate results similar to Kayur's, but this can be time-consuming, especially when dealing with multiple data sources.

[8]BUMPER is a platform for software developers that facilitates bug fixing and reasoning about software quality based on the information from bug report repositories and version control systems of open source projects.

7 Conclusion and future work

We developed our tool Kayur to speed up laborious fetching and preprocessing steps that are often necessary for raw data obtained from web resources before they can be used in data mining workbenches. The tool is aimed at a broad audience of data mining researchers, as it allows them to obtain real-world data sets relatively easily. It also can be useful for software maintainers that wish to analyze bug reports or user feedback using text mining. As far as we know, Kayur is the only tool that spans the whole sequence of steps needed for textual data processing, ranging from retrieving data from semi-structured documents, over processing it, to exporting it to data mining tools.

For future work, we plan to make extraction rule input easier by implementing semi-automatic rule generation based on sample documents. We are also working on extending the range of available export formats by supporting Orange's formats [24], and including more term weight methods (such as ConfWeight [33]). Another goal is to provide more ready templates for bug tracking systems and other resources, and examples of plugin usage to better demonstrate the functionality of the tool. We hope that Kayur's modular design and its scripting interface will inspire novel uses and extensions by its users as well.

References

[1] Scrapy – a fast and powerful scraping and web crawling framework. `https://scrapy.org/`, 2017. [Online; accessed 25-April-2017].

[2] Stack overflow. `http://stackoverflow.com/`, 2017. [Online; accessed 11-June-2017].

[3] Web Scraper. `http://webscraper.io/`, 2017. [Online; accessed 25-April-2017].

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[5] Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowl. and Data Eng.*, 18(10):1411–1428, October 2006.

[6] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, and Yorick Wilks. Experience of using GATE for NLP R&D. In *Proceedings of the COLING-2000 Workshop on Using Toolsets and Architectures To Build NLP Systems*, pages 1–8, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[7] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. `https://www.ietf.org/rfc/rfc2616.txt`, 1999. [Online; accessed 11-June-2017].

[8] Android open source project – issue tracker. `https://code.google.com/p/android/issues/list`, 2016. [Online; accessed 15-August-2016].

[9] Smartphone OS Market Share, Q3 2016. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2016. [Online; accessed 12-February-2016].

[10] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1375–1378, Piscataway, NJ, USA, 2012. IEEE Press.

[11] Alexander Kohan. Kayur. http://kayur.net, 2016. [Online; accessed 11-February-2017].

[12] Alexander Kohan, Mitsuharu Yamamoto, and Cyrille Artho. Automated dataset construction from web resources with tool Kayur. In *Proceedings of the Fourth International Symposium on Computing and Networking*, pages 98–104, 2016.

[13] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 249–258, Washington, DC, USA, 2011. IEEE Computer Society.

[14] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi. Modular software model checking for distributed systems. *IEEE Transactions on Software Engineering*, 40(5):483–501, 2014.

[15] Libmind library repository. https://github.com/neuromancer/libmind/blob/master/data/pos/verbs/, 2015. [Online; accessed 24-January-2015].

[16] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Edward Loper and Steven Bird. NLTK: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[18] Machine learning for language toolkit. http://mallet.cs.umass.edu/index.php, 2002. [Online; accessed 24-January-2015].

[19] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Bumper: A tool for coping with natural language searches of millions of bugs and fixes. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 649–652. IEEE, 2016.

[20] National vulnerability database. https://nvd.nist.gov/, 2015. [Online; accessed 22-February-2015].

[21] Apache Nutch. http://nutch.apache.org/, 2015. [Online; accessed 22-February-2015].

[22] Apache OpenNLP. https://opennlp.apache.org/index.html, 2015. [Online; accessed 24-January-2015].

[23] OpenNLP tools models. http://opennlp.sourceforge.net/models-1.5/, 2015. [Online; accessed 24-January-2015].

[24] Orange – loading and saving data. http://docs.orange.biolab.si/reference/rst/Orange.data.formats.html, 2015. [Online; accessed 24-January-2015].

[25] Stanislaw Osinski, Jerzy Stefanowski, and Dawid Weiss. Lingo: Search results clustering algorithm based on singular value decomposition. pages 359–368. Springer, 2004.

[26] Stanislaw Osinski and Dawid Weiss. Carrot2 – user and developer manual. http://download.carrot2.org/head/manual/, 2016. [Online; accessed 15-August-2016].

[27] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2(1-2):1–135, January 2008.

[28] Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[29] The R project for statistical computing. https://www.r-project.org/, 2016. [Online; accessed 2-November-2016].

[30] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[32] Guoliang Shi and Yanqing Kong. Advances in theories and applications of text mining. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, pages 4167–4170, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Pascal Soucy and Guy W. Mineau. Beyond tfidf weighting for text categorization in the vector space model. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 1130–1135, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[34] Chong Wang and David Blei. Topic modeling software. `https://www.cs.princeton.edu/~blei/topicmodeling.html`, 2015. [Online; accessed 22-February-2015].

[35] WEKA 3: Data mining software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`, 2015. [Online; accessed 22-February-2015].

[36] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[37] Tao Xie, Shengsheng Shi, Fuliang Quan, and Chunfeng Yuan. Research on complex structure-oriented accurate web information extraction rules. In *Proceedings of the 2010 IEEE International Conference on Progress in Informatics and Computing*, pages 312–316. IEEE, 2010.

[38] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. Continuous test suite augmentation in software product lines. In *Proc. 17th Int. Softw. Product Line Conf.*, SPLC '13, pages 52–61, Japan, 2013.

[39] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 311–320, Washington, DC, USA, 2014. IEEE Computer Society.