CS 453: Project 2 - Data Preparation

Project Report

Creepy - Data Cleanser

Travis Hall
trvs.hll@gmail.com

Brittany Thompson
miller317@gmail.com

Bhadresh Patel
bhadresh@wsu.edu

Washington State University Vancouver
October 03, 2010

Abstract

The main goal of this project is to design and implement data preparation or data cleansing stage of the search engine. We already collected small collection of pages via the web crawler in our first project. We will be removing all unnecessary portions in the document collection through a combined process of (i) tag-stripping, (ii) tokenizing, (iii) stopping, and (iv) stemming. Additionally, we will be preparing a document graph for link analysis.

1 Overview

In this project, *Creepy* is designed to prepare the document collection that was crawled by the crawler for the further stages of the search engine *i.e.* cleanse documents. The documents crawled by the crawler are typically not in a plain text. Instead, most of them are in HTML format and can also be in other well know formats like Word, Powerpoint, PDF etc. These documents needs to be convert into text plus some metadata for the further stage of the search engine. In project 1, we tweaked our crawler to crawl documents that are in plain text or HTML format only. Conversion of other formats to plain text is rather complicated and is beyond the scope of this project.

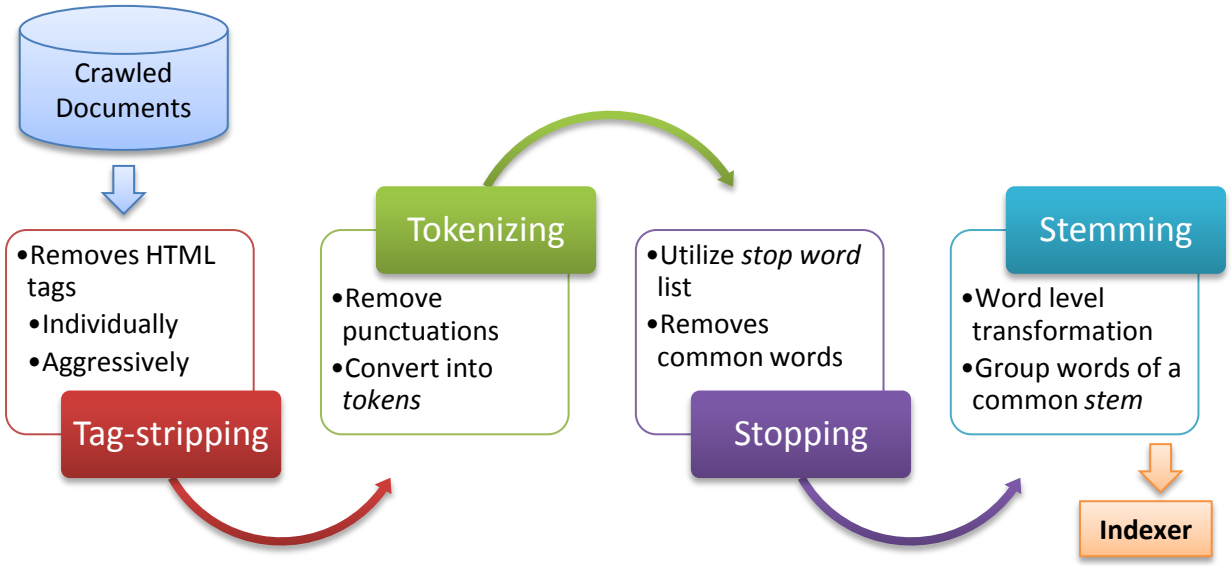

Figure 1: Data Cleansing

The task of data cleansing process is to take the collection of documents already crawled by the crawler and convert them into sequence of text *tokens* for the indexer component of the search engine. The cleansing process will generate the cleansed document through a combined process of (i) tag-stripping, (ii) tokenizing, (iii) stopping, and (iv) stemming as illustrated in the figure 1.

2 XML Document Graph

The first step in creating our XML document graph was to construct a document type definition (DTD) to base it off. The first few elements of the DTD were practically a given: (i) we need an element that will carry all of the documents as children, this we called `docstore`; and (ii) we need an element for storing the document information, fittingly, we named this one `doc`.

The design decisions after that were predominantly guided by the amount of information we needed. In order to create a graph structure, each document needed to have fields for tracking which documents linked to it and which documents are linked from it. However, we did not really need the full document descriptions of those links. Instead, all we needed were the document's URLs and their IDs (the filename on the local system). As such, we chose to create these elements as a list, where their children were very simple entities with a pair of attributes (URL and ID) and nothing more. As such, `to` and `from` contain a list of `page`s.

Creating the graph itself was not terribly complex. For this component, named *Rucksack*, I chose to use Ruby, being most comfortable with it and needing to complete it quickly. The majority of the work is done in the `DocStore` class, where it imports a map-file (`pid_map.dat`) and creates the structure of `Doc` and `Page` classes. In the interest of saving memory with larger map files, each structure is created only once and stored within a hash based on its URL. As the map is rigged together or as entities are fully defined, it simply updates the fields.

Finally, each class is given a `#to_xml` method that utilizes Ruby's *Builder* Gem (library). This means that each component can be converted directly into an XML description individually. The overall construction of the DTD-compliant structure is handled by `DocStore#to_xml`.

In the interest of verifying that components worked, there were two small testing scripts written for `Doc` and `DocStore`. While not terribly in-depth, they helped to smooth out several bugs with the map-rigging and the XML construction.

3 XML Validator

In order to make sure *Rucksack* created a valid tree based on our DTD, we wanted to write a XML-validator for doing so. Unfortunately, this process is extremely complex. While efforts were made to create a validator (which can be seen under `lib/validator/parser.rb`) the task was simply too large to complete in the remaining time. For comparison, *libxml2*, a standard and popular C library for handling XML, and also the library that most validators wind up utilizing (for example *libxml-ruby* and *libxml2* for Python, PHP, and Perl), took 15,000 lines just for validating the XML syntax alone (which included parsing DTD/Doctype declarations but not constructing the tree). The actual DTD validation added another 7,000 lines on top of that.

Our implementation in Ruby will validate a small portion of the XML syntax, however due to time constraints it was not possible to complete the full validator. Instead, so that we could still make sure *Rucksack* was constructing XML that was valid by our DTD, we leveraged *libxml* and a set of bindings for Ruby to complete the validation. This can be seen in *validate.rb* at the top level of the project.

In retrospect, starting the parser from the bottom-up would have made for a less daunting approach. Establishing the terminals and then moving upwards into the non-terminals would have allowed more flexibility in the code. Unfortunately, as my first parser, I did not have enough experience to recognize this.

For reference, the XML 1.0 language description used can be found here: `http://www.w3.org/TR/REC-xml/`

4 Tag-stripping

The first task in the data cleansing process is to recognize the structural elements of the document and remove them. This process is referred to as *tag-stripping*. Document structure is often specified by a markup language, for example HTML is used for specifying the structure of web pages. The tag-stripping component uses knowledge of the syntax of the HTML language to identify the structure of the document. In HTML, *tags* are used to define document elements. These tags does not add any value for the search engine and they are removed at this stage. There is two types of tag stripping (i) aggressively; and (ii) *individually*; that can be applied to the document collection. Tag-stripping is performed on the document collection that is crawled by the crawler.

4.1 Aggressive tag-stripping

Aggressive tag-stripping removes the whole HTML element *i.e.* removes tags and its content. There are certain tags in HTML page that is used only for structural purpose or for dynamic scripting only. These HTML elements can be removed fully since it does not contribute any relevant information for the search. For example, `<script type="text/javascript">functionCall();</script>` would be fully removed by the tag-stripping component. We selected following HTML tags that are aggressively removed from the document: script, style, object, embed, applet, noframes, and noscript.

4.2 Individual tag-stripping

Individual tag-stripping only removes the tags and its attributes from the HTML element and leaves any content intact that occurs between the opening and closing tag. For example,
`<h1 style="font-size:14px;">My page title</h1>` will become `My page title` after removing individual tags. After the aggressive tag-stripping is applied to the document, individual tag-stripping is applied to remove any tags appear in the document. After this stage, the resulting document will only contain plain text that does not have any structural elements of the document.

4.3 Implementation

The implementation of tag-stripping is done in Python. There are some Python modules available that allows stripping tags from the HTML document, for example, HTMLParser, htmllib, Beautiful Soup and sgmllib. In our initial experiments, we notice that most of these modules removes tags quite effectively when the HTML page is well-formed—syntactically valid HTML document. But they failed to remove some tags if the document is not well-formed. Since, web pages are inherently error prone document that does not necessarily follow the standards. So we were not able to use existing modules of Python for our tag-stripping component. We wrote our own module that uses regular expression to remove any tags. The regular expressions are written such that it does not care about well-formed HTML elements and still remove all the tags. The table 1 shows sample HTML document and after removing tags using the tag-stripping component the generated document is shown in table 2.

5 Tokenizing

Tokenizing is the process of forming term or token from the sequence of characters in a document. Thus, tokenizing converts most of the document content to search-able tokens. Tokenizer takes

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
  <head>
    <title>WSU Vancouver - Engineering and Computer Science</title>
    <link rel="shortcuticon" href="/sites/all/themes/departments/favicon.ico"
         type="image/xicon" />
    <script type="text/JavaScript" language="JavaScript">
        var id=3; functionCall(id);
    </script>
    <style type="text/css">
        .field-content {margin:4px;}
    </style>
  </head>
<body>
  <div class="fieldcontent">
    <p>The School of Engineering and Computer Science (ENCS) is an academic unit
    of the WSU College of Engineering and Architecture that houses the
    engineering and computer science programs located at
    <a href="http://www.vancouver.wsu.edu">WSU Vancouver</a>.
    The School offers ABET accredited Bachelor of Science degrees in computer
    science and mechanical engineering.</p>
  </div>
</body>
</html>
```

Table 1: Sample HTML document

the document that was generated by the tag-stripping component as input and generates a new document that contains only tokens. At this stage, we assume that the tag-stripping component removed any text that was structural part of the document and leaves text that is actual content of the document. The design of our tokenizer consist of three step process as described below.

Case folding This is the simplest step in which all uppercase letters are converted to lowercase.

Remove punctuations In this step we remove all the punctuation characters from the document. We considered `!"#$%&\'()*+,-./:;<=>?@[\\]^_‘{|}~` as the punctuation characters that are removed from the document.

Generate tokens The last step is to generate tokens, in this step multiple spaces, new line, tab etc.. are removed. The resulting document will contain all the terms separated by single space.

At the end of the three step process of tokenizer we save the generated tokens into new document that is used for further steps in data cleansing process. For example, the table 3 shows the tokens generated (each token is separated by sapce) by the tokenizer for the sample document produced by tag-stripping component as shown in the table 2. We note that, the design of the tokenizer is kept simple and flexible, the same rules would be applied to the search query in the query transformation

```
 WSU Vancouver - Engineering and Computer Science




 The School of Engineering and Computer Science (ENCS) is an academic unit
of the WSU College of Engineering and Architecture that houses the
engineering and computer science programs located at
 WSU Vancouver .
The School offers ABET accredited Bachelor of Science degrees in computer
science and mechanical engineering.
```

Table 2: Sample HTML document after tag-stripping

component. Hence, all the punctuations and tokens would be treated in same manner to the query and it should result in appropriate match to the index terms.

6 Stopping

7 Stemming

8 Roles

Travis Hall Link analysis graph and XML validator.

```
wsu vancouver engineering and computer science the school of engineering and
computer science encs is an academic unit of the wsu college of engineering
and architecture that houses the engineering and computer science programs
located at wsu vancouver the school offers abet accredited bachelor of
science degrees in computer science and mechanical engineering
```

Table 3: Tokens generated by tokenizer for the sample HTML document

Brittany Miller Stopping and stemming.

Bhadresh Patel Tag-stripping and tokenizing.

9 Test Environment

For testing/production purpose, we set up a machine instance on Amazon EC2. The instance id of the machine is `i-7d1e0d17`. The source code is checked out at `/home/ubuntu/creepy/`.

10 Usage Guide

The process for generating cleanse document is sequential as illustrated in the figure 1, so here are the steps for using the program in same sequence.

1. Crawl the web

   ```
   creepy$ python creepy.py -S storage -c seed.txt -T 2500 -N 300
   ```

2. XML Document Graph Rucksack requires two RubyGems be installed (both of which come standard with later versions of Ruby):

   ```
   $ gem install hpricot
   $ gem install builder
   ```

   For default options:

   ```
   $ ruby rucksack.rb pid_map.dat
   ```

   For the current production environment:

   ```
   $ ruby rucksack.rb -a crawled storage/pid_map.dat -o map.xml
   ```

3. Validation The validator requires one RubyGem be installed, as well as libxml and zlib. On OS X with MacPorts installed, libxml and zlib can be installed with:

   ```
   $ port install libxml2
   ```

   On Debian linux:

   ```
   $ apt-get install libxml2 libxml2-dev
   ```

   Installing libxml-ruby:

   ```
   $ gem install libxml-ruby
   ```

   Running:

   ```
   $ ruby validate.rb <dtd_file> <xml_file>
   ```

4. Tag-stripping

```
$ python creepy.py -S storage -s
```

5. Tokenizing

```
$ python creepy.py -S storage -t
```

6. Stopping
7. Stemming