Masaryk University
Faculty of Informatics

Web Content Cleaning

Master's thesis

Vít Baisa

Brno, May 2009

Declaration

Hereby I declare that this thesis is my original writing worked out by my own. All sources and literature used in this thesis are cited and listed properly.

Adamov, May 21, 2009

Advisor: Mgr. Pavel Rychlý, Ph.D.

Acknowledgement

I would like to thank my advisor Pavel Rychlý for his valuable advice, my parents for their helpfulness which allowed me to work on my thesis unmolested, my girlfriend who mitigated my nerves whenever needed and J. S. Bach for his music which has been accompanying me during writing this thesis.

Abstract

In this thesis we deal with web content cleaning, i.e. on the one hand extracting useful text parts from web pages and on the other hand removing so-called boilerplate and other irrelevant text material. An output of this procedure serves as a source of data for further processing in building text corpora. We use DOM (Document Object Model) representation of HTML (HyperText Markup Language) structure of web pages and to be more versatile, we do not use any language-dependent method (for example n-grams, language models etc.).

Keywords

Cleaneval, web as corpus, WWW, web content cleaning, boilerplate removal, DOM, HTML

Contents

Introduction

Till lately, biggest text corpora included millions and tens of millions words – for one almost unconceivable amount of text (especially for those who were to prepare data for these corpora manually). Text data were often taken from periodicals, books, newspapers and printed media in general. It took months of hard work to build such corpora.

In an ideal world we would have each text (which has been ever written[1]) and speech or dialog (which has been ever spoken) in digital textual form and on our harddisk. But the world is not so merciful (otherwise, Guttenberg would have invented XML (eXtensible Markup Language) instead of the typography). *The Gutenberg Galaxy* [15] is very large and there is no hope to be completely digitalized in next few years.

A fact that the more data we have the more precise information we can get from it obviously holds and so we turn our attention from printed media to World Wide Web (WWW) since it is unbearable to prepare data for very big corpora manually.

Main advantage of using WWW as a source of text data is that we suddenly have millions of volunteers who prepare texts for us – without knowing that!. Other advantage is that we have instant access to documents in many languages. It has not been possible before. Try to find at least one book, physical one of course in Mongolian in Czech Republic... But with Google, it is matter of a few mouse-clicks to find some text in Mongolian.

WWW looks at first sight like ideal source of data but with the mentioned advantages we get also a few disadvantages. The first and probably the most serious is diversity of web pages: blogs, articles in online magazines, descriptions of goods in internet shops, discussion forums, lists of links etc. No common presentation of texts does exist. Inspite of standards (HTML, CSS[2]), a necessity of visual presentation predominates over logically structured presentation of data.

And here comes web content cleaning (WCC). Its goal is to find and

[1] And which will be even written in future!

[2] Cascade Style Sheets – for further information see [16].

extract valuable text data[3] from web pages.

Several tools for WCC were developed so why to make another one? First of all, most of these tools are language dependent – we present a tool working independently on language of document. All these tools also clean a single document. We introduce a method exploiting comparing of similar documents from a single website.

[3]Using the term *valuable text data* we mean texts consisting typically of whole sentences, paragraphs – as they can be found in books, magazines etc.

1. Web content cleaning

Special Offer!
A copy of Internet on 100 DVDs.
Or on 2 DVDs (without porn)!

Web pages consist of (aside from text) hypertext links, pictures, tables, lists, applications. All these objects together are called web content.

When building a corpus we need consistent texts – whole sentences, paragraphs. In real situations, the relevant text is often crouching in a corner (or rather in the middle) of web pages meanwhile blinking banners, flash advertisements, menus, footers and graphics are occupying major part of web pages. These predominant elements are called *boilerplate*. Web content cleaning equals removing the boilerplate and extracting only consistent and valuable text material from web pages.

Little more formally

As input of WCC, web pages in HTML format[1] are taken. We impose these requirements on an output (based on Cleaneval Guidelines [17]):

- ▷ *Code removal*: An output should not contain any code (a script, a style sheet, ...).
- ▷ *Boilerplate removal*: Any extraneous item as for example navigation, menu, list of links, footers, copyright notices, advertisements, ... should be removed too.
- ▷ *Structural annotation*: The output text should preserve some structural information of original document. Each list item should start with `<l>`, each heading with `<h>` and each paragraph with `<p>`. Additional tag `<doc>` can be used for separating various cleaned documents within one file.

You can see an example of a web page on figure 1. A possible manually cleaned output of this web page follows:

[1] In fact only small portion of them follows the standard strictly.

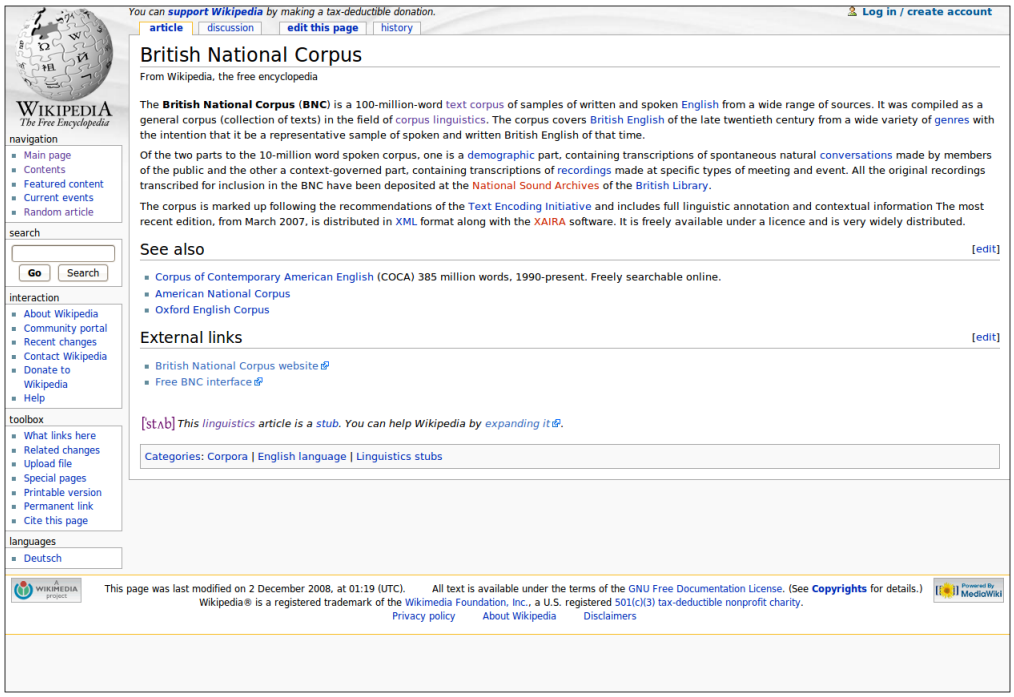

Figure 1.1: Wikipedia page

```
<h>British National Corpus
<p>The British National Corpus (BNC) is a  100-million-word ...
<p>Of the two parts to the 10-million word spoken corpus ...
<p>The corpus is marked up following the recommendations ...
<h>See also
<l>Corpus of Contemporary American English (COCA) 385 ...
<l>American National Corpus
<l>Oxford English Corpus
<h>External links
<l>British National Corpus website
<l>Free BNC interface
```

This example is quite simple and straightforward. There are usually much more complicated web pages and several various cleanings are possible.

1.1 Cleaneval

"Cleaneval is a shared task and competitive evaluation on the topic of cleaning arbitrary web pages, with the goal of preparing web data for use as a cor-

pus, for linguistic and language technology research and development." [19]

Participants of Cleaneval contributed with their tools considerably to WCC. We shall describe their various approaches in following text. We also add description of BTE (Body Text Extraction) algorithm due to its smart and simple principle.

1.2 Brief survey of WCC tools

FIASCO – *Filtering the Internet by Automatic Subtree Classification* [3]

FIASCO uses DOM[2] trees for representation of HTML files, Lynx for converting HTML into plain text and Support Vector Machine for automatic recognizing of relevant ("clean") and irrelevant ("dirty") nodes of DOM tree (= HTML elements).

Since Gold standard supported by Cleaneval had been insufficient, FIASCO Gold standard was manually gathered. About 300 web pages were downloaded through Google queries, then only pages including at least one paragraph were manually filtered (yielding 158 pages). These files were converted into plain text and two annotators then manually cleaned data and added tags according to Cleaneval guidelines. Finally, their results were compared and combined into FIASCO Gold standard.

The whole process can be divided into a training phase and a cleaning phase. The first one consists of manual choosing *target nodes* (nodes including "text block-like elements"), labelling them as "dirty" or "clear" and building support vector machine (SVM) model based on this training data.

Following heuristics was used for finding of target nodes: a node in DOM tree was accepted as a target node if 10 % of its textual content were contained in text nodes among its immediate children.

Then, *feature vectors* were computed for each target node: *linguistic features* (total text length of target node, number of word tokens and word types, number of sentences and average sentence length, frequency of keywords (those words strongly prevailing in dirty nodes), ...), *structural features* (depth of target node in DOM tree, whether target node is heading, whether TN is paragraph (`<p>`) and proportion of `<a>` and `<img>` tags in TN) and *visual features* (they were extracted from rendered web pages in browser – shape of blocks, width of blocks and so on). Manually annotated training data was passed to SVM and a classifier was trained yielding SVM model.

In the second phase, first of all, preprocessing is done: validating of HTML by TagSoup, converting document into UTF-8 encoding, removing Javascript

[2]This representation is described in detail on page 15.

(`<script>` tags), CSS (`<style>` tags), comments... and changing particular tags for better block distinction (for example `<br>` into paragraph `<p>` and so on).

Finally the SVM model classifies dirty and clear nodes, and clear nodes are extracted and converted into appropriate format.

StupidOS – *A high-precision approach to boilerplate removal* [4]

StupidOS is written in Perl and therefore it runs quite fast (in comparison with the other tools). It uses n-gram language models for targeting "dirty" text (boilerplate). It operates with plain text only.

Within preprocessing phase, it removes images, comments (text within `<!--` and `-->`). It converts `<br>` tags (line break) into paragraphs. For converting HTML into plain text it uses Lynx text-dump function.

Precleaning of text follows: for example it searches for footers using regular expression (strings with repeated symbol "|").

Core part of StupidOS method is computing n-gram language models (trained on FIASCO Gold standard). StupidOS implements 3-grams and conditional probability.

In cleaning phase, StupidOS uses two n-gram language models – for boilerplate and for clear text segments. It distinguishes between these two cases based on model's results – boilerplate is removed yielding cleaned text.

GenieKnows – *web page cleaning system* [5]

The main purpose of this tool is preparing data for search engine indexing and that's why it slightly differs from Cleaneval's requirements. The tool prefers preservation of important content instead of completely removing boilerplate. Used method is also worth mentioning, anyway.

The tool divides HTML file into *semantic blocks*: each block represents a group of content that is semantically related and a building unit in the Web page layout (headers, footers, navigation menus, copyright notices, articles, images, forms, ...). It is done with help of DOM tree representation and several visual features (based on rendered web pages). Then various features are computed for each semantic block – block positioning on page, font size of block text, relative (with respect to whole document) length of text in the block, number of links in the block, relative number of sentences in the block, ...

Relevance of each block based on weighted average of features of blocks is computed and, then, only blocks with high relevance are chosen. Relation

(distance between blocks, equal width of blocks, etc.) between particular blocks is taken into account during computing relevance, too.

Some postprocessing is done to meet Cleaneval's requirements: a Perl script converts tags (`<hx>` into `<h>`), a cell of a table is considered as an item of a list and so on.

Htmcleaner – *Extracting Relevant Text from the Web Pages* [6]

The algorithm is based on the following observations: length of sentences in a relevant section is higher than in an irrelevant section, number of links in the irrelevant section of the text is higher and, finally, number of function words in the irrelevant part is lower.

In preprocessing phase Htmcleaner discards invisible parts of HTML file (comments, scripts, styles, ...), non-textual objects (images, form items, applets) and non-standard tags (those not included in HTML 4.0 standard). For parsing HTML file, Jericho (Java library) open source HTML parser is used. Formatting tags are removed in this phase, too.

In cleaning phase Htmcleaner looks for sequences of sentences fulfilling these conditions: each sequence starts and ends with *good sentence*. Good sentence is a sentence with length above average, with the first letter capitalized and with punctuation mark at the end. Htmcleaner then computes nA – number of characters within links and nFW – number of characters within function words[3].

The core of the algorithm follows: Htmcleaner looks for sequences with the highest difference $nFW - nA$. These sequences are treated as clean text and are extracted from the document.

Web Content Cleaning *using Content and Structure* [7]

The tool implements several experimental methods – baseline, using heuristics, decision trees, language models and genetic algorithms.

Baseline method uses HTML parser to extract block elements (*p, div, li, h1..6*) and plain text. Some postprocessing is done: converting `<br>` into paragraph break and UTF-8 encoding. This method does not distinguish headlines (`<h>`), list items (`<l>`) and converts each element into `<p>`.

Heuristic method simply takes only those lines (of converted plain text) with sufficient length (experimental length is 3 characters): lines with more than 10 words and with average length of word above 3 characters are treated as paragraphs. The other lines are considered to be headlines. This method

[3]Words with little lexical meaning, often expressing rather grammatical relationships with other words in sentence.

does not take list items into account. Authors also prepared a little more sophisticated heuristics but it was considerably slower and therefore not used in Cleaneval.

In the next method decision trees are used for classifying block level elements into four classes: *h* (heading), *l* (list item), *p* (paragraph) and *n* (not included). Cleaneval Gold standard was adapted to serve as training data for decision trees – blocks from source files were matched with texts in their manually cleaned counterparts. Annotations (*h, l, p, n*) were inserted according to manually assigned tags. Flag *n* was used when matching element could not be found. Following features were used: number of words, parent tag type, ratio of words to capitalized words, ratio of words to punctuation, ... Then tree classifier was trained on adapted Gold standard and used for automatic classifying of blocks in files.

The language model method is an experimental attempt and was not submitted into Cleaneval. It is based on the method used in Information retrieval – with the presumption that user's query could have been generated by one (or more) documents in the document set, we can compute the language model of the query and each document and, comparing them, we can choose the document most appropriate to the query. In our case, the query is obtained from titles included in the web page. The language model for the query and for each element of the page is calculated and appropriate elements are chosen as relevant.

The last method is an experiment with genetic algorithm and it was not submitted to Cleaneval. A genetic algorithm is used for evolving a regular expression which extracts relevant text parts of HTML files. Population consists of regular expressions coded as trees (relevant nodes, irrelevant nodes (to be removed), "OR" nodes and literals in leaves (simple RE, for example `\w*?`, `<.*?>`, ...)). In each generation candidates are evaluated with scoring function. This function gives 2 points to candidate regular expression for each preserved word (in comparison with manually cleaned file) and takes 1 point away for each extra word (supposed to be boilerplate). Candidates for next generation are chosen and crossover and mutation are performed.

Kimatu *a tool for cleaning non-content text parts from HTML documents* [9]

The tool, first of all, detects blocks: block is a group of HTML tags sharing some features (tags, classes) and which are close to each other. Then, ratios of features between them are calculated. There features are: relative block length (with respect to the longest block of document), relative average block length, relative number of punctuation in block and relative number of links in block. Candidates are chosen based on ratio threshold. Repetitive function

blocks ("posted by x") and quotations (a candidate block is compared with consequent small blocks) are removed. At the end it searches for relevant blocks in close surroundings of candidate blocks (possible headlines with low ratio) and it assigns proper tags to each candidate block (based on included tags, punctuation and block length).

Web Page Cleaning *with Conditional Random Fields* [8]

The input file is, first of all, validated with HTML Tidy to obtain parsable HTML document. Removing of unusable parts (scripts, style definitions, etc.) follows.

In the next step, text in HTML file is chopped into blocks respecting tags: `<p>Hello <b>world</b>!</p>` is chopped into three blocks `Hello|world|!`. Then following features for each block are extracted:

- ▷ *Markup-based features*: a feature *container.*{*p, a, u, img*} of a block is set to 1 if the block is enclosed by the appropriate tag, a feature *container.class-*{*bold, italic, list, form*} serves for classifying similar blocks.
- ▷ *Content-based features*: represent absolute and relative numbers of characters (distinguishing letter, numerical character, punctuation mark, whitespaces) and of tokens (words), number of sentences, average length of sentences in the block, number of URLs, dates, times (based on regular expressions), ...
- ▷ *Document-related features*: relative position of the block in the document, number of words, sentences and blocks in the document...

A training phase follows – each content block (to be extracted) in files of Cleaneval Gold Standard is manually labelled as *header, paragraph, list, continuation* (of preceding content block of any type) and *other* (to be deleted). Manually labelled data are used for training of Conditional Random Fields model.

The input HTML file is also validated, chopped into blocks. Features are extracted and the file is passed through CRF, resulting in an automatically labelled file – blocks labelled as header, paragraph and list are marked with appropriate tags (according to Cleaneval annotation guidelines) and exported. The other blocks are omitted.

Body Text Extraction [20]

In its basic form, BTE does not take types of tags into account. It converts an HTML file into a sequence of "0" and "1". "0" corresponds to a word and "1" to a tag. BTE algorithm computes such i and j which maximise the

expression:

$$T_{i,j} = \sum_{n=0}^{i-1} B_n + \sum_{n=i}^{j} (1 - B_n) + \sum_{n=j+1}^{N-1} B_n.$$

These "indices" split the HTML file into three parts, and the second one (with the minimum amount of tags inside) is supposed to include the desired text.

Implementation of this algorithm is currently used in Centre for Natural Language Processing at Faculty of Informatics for building BiWeC (Big Web Corpus)

Summary

Obviously, FIASCO, StupidOS and Htmcleaner are dependent on language of documents. BTE, GenieKnows, WPC (partially), and Kimatu are language independent.

Evaluation of Cleaneval's participants and comparison of BTE and our approach are given in the third chapter.

1.3 Our approach

We can see that there are many possible ways how to clean a web content. Prior to describing our methods, we introduce several observations on which our methods are based.

- ▷ A relevant part of a web page forms coherent and continuous text – with headings, paragraphs and lists.
- ▷ Conversely, the other parts are usually scattered throughout the web page and form small "enclaves".
- ▷ There exists a single HTML element that tightly encloses the overwhelming majority of the relevant part. This always holds since, in the worst case, it is the `<body>` element.
- ▷ Finally, this "target" element excludes boilerplate and unwanted parts from itself.

When gathering text data from the Internet we do not download just one page per website. On the contrary a crawler often comes back and, in fact, it downloads thousands of HTML files from one website. In this case, these facts can be observed:

- ▷ Menus, footers, headers, etc. are very similar (or equal) within one website.

- ▷ It would be unbearable to maintain these similar (or equal) parts manually and therefore this material is generated mostly by scripts or derived from templates.
- ▷ Previous observation implies that these parts have equal (having been generated automatically) HTML structure and so we can detect them quite easily.

These two groups of observations lead us to two methods of WCC. The first one deals with cleaning a single page – in the same way as Cleaneval's participants and BTE do. The second one tries to exploit facts from the second group of observations. Therefore it requires similar web pages from a single website.

In the next chapter we shall describe our approach to web content cleaning in more detail.

2. Implementation of the tool

A working program is the one that has only unobserved bugs.

We have implemented two methods – one for single file cleaning and the other for multiple file cleaning, and we named the whole tool *WC Cleaner*[1] with regard to its unenviable job.

First of all, we will describe their common steps.

2.1 Preprocessing

Precleaning

In this phase the tool removes invisible parts of HTML: scripts, style definitions and comments. Since HTML includes corresponding tags (`<script>`, `<style>` and `<!-- -->` respectively), it is quite easy to get rid of them. A straightforward solution is to use regular expressions for that. The only problem arises – we must pay attention and use non-greedy variants of `*` element of regular expressions to match as little as possible. For example, if a web page included two scripts – at the beginning and at the end – the regular expression `<script>.*</script>` would swallow the whole page.

Validating

At the beginning, we tested WC Cleaner on HTML files "as they are" – rough, downloaded from the Internet. Since the tool uses Python library `xml.dom` and its parser which requires valid HTML, only about 4 files (of 100 files from the data set) were flawlessly parsed. It means that there are very few pages being completely valid. Amongst several types of errors let us mention these two (the most frequent):

[1] Web Content Cleaner

- ▷ Non standard tags – those not defined in DTD[2] of HTML.
- ▷ Incorrect encoding of HTML file – declared encoding in HTML file differed from actual encoding of the file.

We experimented with HTML Tidy [22] which was used by several participants of Cleaneval, but it proved to be too little robust. It was not able to repair non-standard tags. That's why we have subsequently tried TagSoup open source tool [14]. After validating by TagSoup, all files were parsed without problems.

Encoding/decoding

It is a nontrivial task to detect encoding of a (HTML) document properly. The task is easier if we know the language of the document. Since there already exist several tools dealing with this task, we did not implement this phase.

The developed data set (will be described later) consists only of documents in English (with one exception), and therefore it was not necessary to implement it. If necessary, this phase can be maintained using for example *Universal Encoding Detector* [23]. Actual converting into desired encoding (UTF-8) can be done with `iconv` tool. Finally, it suffices to change encoding declaration in HTML header.

Sorting

This preprocessing phase is specific for multiple file cleaning. As prefigured in section 1.3, HTML files are downloaded in bulk, but crawlers usually do not download two files from one website consecutively. It would overload server traffic. Instead, crawlers download files from many servers with (usually) several-seconds gap between particular accesses to a single server.

Downloaded files are stored within a single file in a special format – ARC (Archive File Format) [24]. Documents are separated by headers with information on length, server from which they have been downloaded, etc.

As mentioned, we need files from one website but files in ARC are stored in rather arbitrary order. Therefore we sort them. Since typical ARC files are about 100 MB and consist of thousands of HTML files, it is impractical to just extract files and save them into directories – it may cause problems with a file system – being forced to maintain millions of relatively small files. Therefore sorted files are stored in ARC file again.

[2]Document Type Definition, it specifies grammar of markup language and also a set of elements.

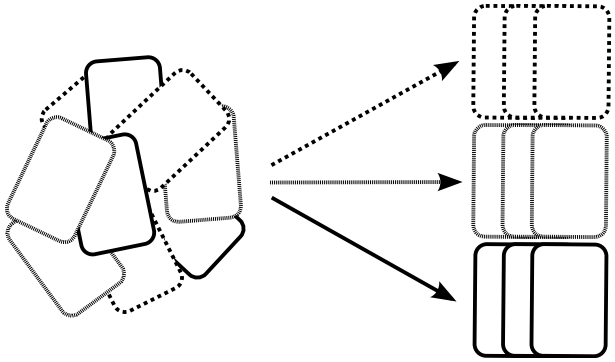

Figure 2.1: Sorting of files.

We have implemented simple Python script (`sorting.py`) which takes an ARC file and a name of output directory as parameters, extracts all files from the ARC file, and saves them to appropriate ARC file. The script reads URL (Uniform Resource Locator) address from which a HTML file has been downloaded. Then it modifies the address to obtain names of target directory and target file as follows:

First of all the script gets rid of beginning and end of the address:

`http://www.abc.co.uk/documents/index.html` → `abc.co.uk`

This simplified address serves as filename of ARC file to which all files from the respective address (`abc.co.uk`) are saved. There are many addresses where files are downloaded from and if we saved all files in one directory, it might cause problems, as mentioned before. Therefore, we use a simple method to group ARC files in directories with counterbalanced amount of files. The script takes the first and the last character of the third-level domain:

`abc.co.uk` → `ac`

This heuristics yields good results. We have also tried to take the second-level domain (as the name of the directory) and the first and the second character of third-level domain (as a name of directory) resulting in too few (too many) directories with too many (too few) files inside in both cases respectively.

2.2 Parsing input – DOM

The tool uses Document Object Model (DOM) [25] for representing the HTML file. It is a standardized, platform- and language-independent interface for accessing and updating content, structure and style of documents.

We use Python library `xml.dom.minidom` which implements a light-weight API (Application programming interface) for DOM including XML (HTML) parser and necessary classes and their objects and methods.

We chose DOM for its ability to represent a whole document within one object and to offer instant access to all its components.

DOM represents an XML document as a tree and includes these interfaces (Python classes of Python DOM API): `Node`, `NodeList`, `DocumentType`, `Document`, `Element`, `Attr`, `Comment`, `Text`, ... The most important are:

- ▷ `Node` is the most general class (all of components of DOM are subclasses of `Node`) and it represents actual node of DOM tree.
- ▷ `Document` is an interface representing the whole document. It is a subclass of `Node`.
- ▷ `Element` is equivalent to a tag of HTML document.
- ▷ `Text` object bears actual textual content of a tag.

You can see an example of simple DOM tree on figure 2.2 which has been built by parsing of following sample of HTML code:

```
<body>
<p>This is highly random <b>sample</b> of text
with arbitrary <a href="www.random.com">link</a>
And eventually end of paragraph.</p>
<p>Another paragraph with quite boring content.</p>
</body>
```

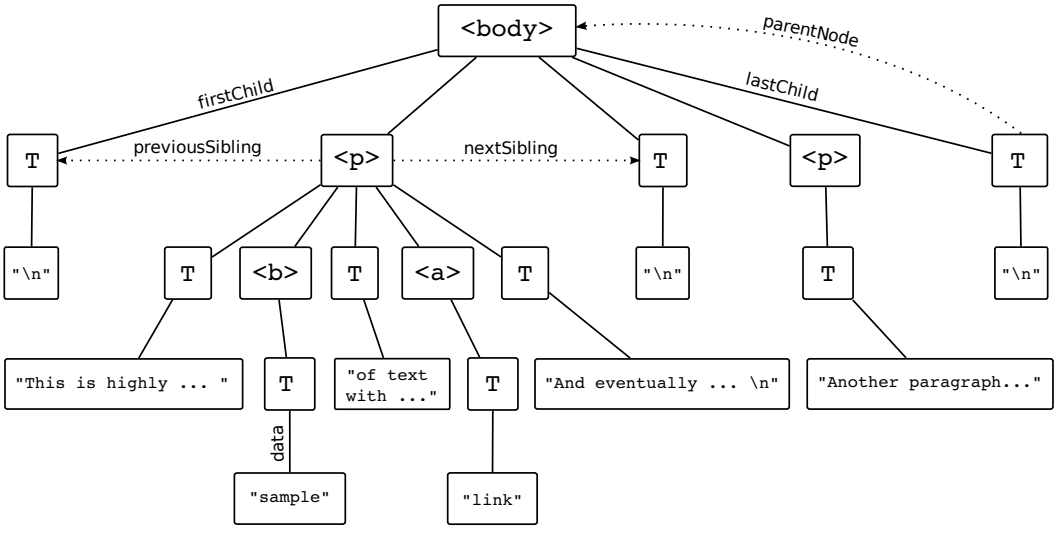

Figure 2.2: Very simple example of portion of a DOM tree.

`Node` class includes these objects: `nodeType`, `parentNode`, `nextSibling`, `childNodes` and other treelike methods and objects – all of them with obvious meaning (see the labelling on the figure 2.2). The most important object of `Document` is `documentElement` – it represents the only root element of document (`<html>` tag in HTML document).

The whole process of parsing is maintained by class `DOMSTree` – within initialization of class instance. Its name is an abbreviation of DOM Statistical Tree.

2.3 Single file cleaning

When cleaning a single file, simple heuristics (based on our observations described in previous chapter) is used. The tool looks for a node in DOM tree with these properties:

- ▷ The node has a sufficiently high ratio

$$\frac{\text{number of all characters}}{\text{number of all } \textit{structural elements}}$$

  inside itself – we call it *density*.
- ▷ The node includes sufficient number of characters in comparison with number of characters of whole document.

To find a HTML element with these properties, we need to add some information into DOM nodes.

Tweaking DOM

Once a HTML document is parsed and represented by a DOM tree, we add some statistical information into it. The class creates new attributes of `Element`:

- ▷ `chars`: number of characters in a subtree (the element is its root node),
- ▷ `tags`: number of all tags (`Element` nodes) in a subtree and
- ▷ `s_tags`: number of *structural tags*[3].

These attributes are computed by recursive function `addStatToDOMTree`. It goes through the DOM tree, like BFS (Breadth First Search), and for each text node it computes the number of characters of the node. For each element node it computes the number of characters of all nodes in the subtree. In the example of a DOM tree on figure 2.2 we can see that DOM includes many text nodes with just one character (`"\n"`). Browsers always omit more than

[3]Structural tags correspond with block elements of HTML and will be listed later on.

one whitespace in source HTML code but the parser from `xml.dom.minidom` creates many extra text nodes with this unwanted material. Therefore the function counts characters of only those strings of text nodes including at least one printable character. It also trims whitespace at the beginning and the end of the string.

After that, the function `getTargetNode` looks for a node with mentioned properties. The function is, again, recursive and can be expressed by the following pseudocode (see Algorithm 1).

Algorithm 1 Pseudocode of `getTargetNode(`*node*`)`

```
nodeDensity ← density of node
nodeChars ← number of characters of node
if node has child nodes then
  list ← list of all element child nodes of node
  if list is nonempty then
    maxDensityChild ← take the child node with the highest density
    if # of characters of maxDensityChild > 0.5 × nodeChars then
      return getTargetNode(maxDensityChild)
    end if
  end if
end if
return node
```

The algorithm goes through (from `<body>` element node) the DOM tree and on each level it chooses a node with the highest density. If such a node exists and contains at least half the characters of its parent node, the algorithm continues recursively into the node.

Single file cleaning can be simply figured by diagram on figure 2.3.

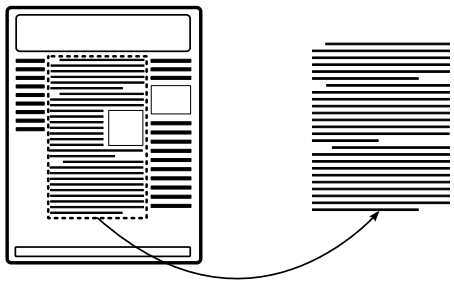

Figure 2.3: Single file cleaning – principle.

Converting the target node is common to both single and multiple file cleaning – it will be described later.

2.4 Multiple file cleaning

The principle of multiple file cleaning is outlined on figure 2.4. The basic idea behind it is that if we take two web pages from a particular website, they likely include equal parts (menus, headers, ...) which are generated automatically and that is why these parts have the same HTML structure.

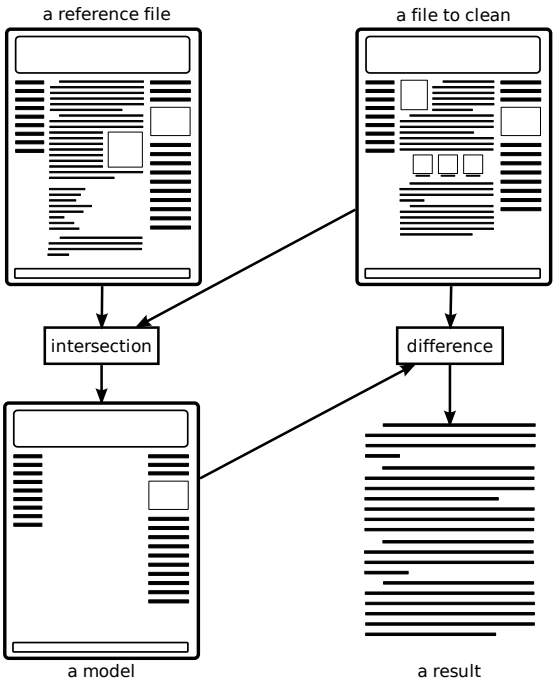

Figure 2.4: Multiple file cleaning – a principle.

Simply put, when we make intersection of these two pages we obtain a model – a set of HTML elements which are common to these two pages. They can be deleted because (i) if they were generated they can not hold any valuable text data and (or) (ii) if we kept them we would have duplicate text in the output.

As you can see on figure 2.4, the process of cleaning is divided into two parts: *intersection* and *difference*. In fact, the implemented algorithm works slightly differently. The appropriate function (`diffCleanNode`) is also recursive. It goes through (also in BFS manner) DOM trees of a reference file and a file intended for cleaning simultaneously – level by level, deeper and deeper – and looks for distinct levels (comparing numbers and types of child nodes). Two levels are compared within function `isEqualLevel`. Since there are many text nodes in HTML files bearing no valuable data (see the third

child node of `<body>` tag on figure 2.2), the function considers only element and nonempty text nodes.

We have said that `diffCleanNode` looks for distinct levels – if it finds equal levels, it goes down DOM trees (also simultaneously) and calls itself recursively. If it encounters mutually distinct levels, it calls the method `diffCleanLowestLevel`. These levels likely contain relevant text – according to our observations.

The function `diffCleanLowestLevel`, given two lists of element nodes (`refList` and `docList`), deletes common parts of text. Algorithm 2 simply presents how the function works.

Algorithm 2 Pseudocode of `diffCleanLowestLevel(`*refList, docList*`)`

```
i ← 0, j ← 0, k ← 0
while i < len(docList) do
  k ← j
  while k < len(refList) do
    if docList(i) and refList(k) stand for the same tag or they both are
    text nodes then
      if docList(i) and refList(k) share any parts of text then
        delete these parts
        j ← k + 1
        break
      end if
    end if
    k ← k + 1
  end while
  i ← i + 1
end while
```

Function `diffCleanLowestLevel` tries to map elements of `refList` onto elements of `docList`. It deletes equal (with regard to textual content) elements.

Typically, there is more than just one occurrence of such distinct levels between two files. One for main texts (for example articles on a web portal) and another for menus because menus can slightly differ page from page. But it is likely not critical since alteration of menu corresponds with actual page content (an extra item in the menu may stand for a subcategory of the website). The tool can also pick out target node as in the case of single file cleaning and try to run `diffCleanNode` with two target nodes of two processed documents.

Once we removed all common text, we can convert the resulting node

(parent node of `docList`'s items) into desired format (there may be more than one such node). This process is common to both single and multiple file cleaning and will be described in the next section.

Searching reference file

Not all files from a single website are similar. Usually, there are various "kinds" of pages, apart from ordinary articles, ranging from help, map of the site to rather technical pages, for example PHP[4] info page, 404 error[5] page... Since we need a reference file similar to as many files from the website as possible, we have invented a simple heuristic method which tries to look for the reference file.

We assume that all regular files from a website have similar size. In other words, that the other files considerably differ from them in size. Therefore, if we sort a list of all files by their size, irregular files (in accordance with our assumption) get near to borders of the sorted list. Then, if we take a file from the middle of the list, it is likely the reference file similar to the most of files from list.

Of course, a website may consist of several groups of mutually similar pages, for example, one group for each subcategory of website. Even in this case, the solution would be quite easy and straightforward.

2.5 Converting DOM

Described algorithms yield nodes supposed to bear relevant data – single file cleaning returns the target node of document's DOM, multiple file cleaning may return a few cleaned nodes. In both cases we must deal with DOM tree(s) which is (are) to be converted into required format.

We face two distinct problems. The first (and easier) one emerges from the fact that we convert tree (recursive) structure of DOM into a simple (sequential) format. It is solved by recursive function `convertNode` which takes 2 parameters – one stands for a node (to be converted) and the other determines the last exported tag.

The second one lies in fact that the required output format of the document permits only three tags whereas HTML files include many tags. It is the more complicated task. As we could see in the previous chapter, Cleaneval's

[4]Hypertext Preprocessor

[5]HyperText Transfer Protocol error – file not found

participants had solved it in various ways – with help of regular expressions, heuristic rules or browser rendering of HTML files.

Since (i) we consider regular expressions too general and (ii) we don not have a style sheet (usually stored in external file) at our disposal, we try to exploit semantic information of HTML tags intended by HTML standard designers[6].

We put together these four lists:

- ▷ `structureTags` – tags splitting web content into logical units. These tags are scattered throughout the web page and they often hold less textual data in its irrelevant parts. Therefore, their number is used in computing density of nodes.
- ▷ `formattingTags` – tags affecting especially typeface (`<b>`, `<i>`, `<span>`, `<em>`, ...) plus `<a>` tag standing for hyperlink and several other usually inline[7] tags (`<abbr>`, `<acronym>`, ...).
- ▷ `headlineTags` – tags for headings: `<h1>` ... `<h6>`.
- ▷ `paragraphTags` is the most problematic list because we do not know how content of the node will look like in advance. We chose tags which usually form a text block: `<p>` (paragraph), `<td>` (a cell of a table), `<blockquote>` (quotation) and so on.

A line break (`<br>`) is treated as a structure tag not as a paragraph tag. There is the only tag standing for a list item in the standard – `<li>`. It occurs both in unordered (`<ul>`) and ordered (`<ol>`) lists. Therefore we convert all `<li>` tags (and no others) into `<l>` tags.

A comprehensive list of all HTML tags can be found in [1].

Tweaking output

As mentioned in subsection *Sorting* on page 14, it is impractical to save immense amount of small files on a harddisk. Instead, in the process of multiple file cleaning, we save all results into a single file sequentially and label each result with additional tag `<doc>` to separate it and to be able to split individual documents effectively in future.

[6] And, unfortunately, often ignored by web coders.

[7] Those which do not cause a line break

3. Evaluation of the tool

Statistika nuda je,
má však cenné údaje.
Neklesejte na mysli,
ona vám to vyčíslí.[1]

Since we implemented two methods – single file and multiple file cleaning, evaluation of the tool will be divided into two parts. In both cases we have used Python script `cleaneval.py` for evaluation of WCC (improved version of official Cleaneval evaluating script) from [26].

3.1 Cleaneval evaluating method

The evaluation script `cleaneval.py` takes two parameters – (i) a name of the directory containing automatically cleaned files and (ii) a name of the directory containing manually cleaned files (referential files). It assumes concordance of filenames in both directories. The algorithm then goes through directories and for each couple of corresponding files computes:

- *Precision* value is defined as

$$\frac{\text{amount of relevant retrieved text}}{\text{amount of all retrieved text.}}$$

  Within WCC, precision stands for accuracy of boilerplate removing. Computation of the value is based on Levenhstein edit distance between automatically and manually cleaned documents: the smallest number of "insert word" and "delete word" steps to get from the one text to the other ([21]).

[1] a popular song from a Czech fairy movie. Its inartistic translation: Statistics is boring thing (but) / she can bring stuff interesting. / So let us be of good cheer (cause) / with her everything is clear!

▷ *Recall* value is defined as

$$\frac{\text{amount of relevant retrieved text}}{\text{amount of all relevant text.}}$$

In our case, this values measures successfulness of tag matching.

▷ *F-score* is a weighted average of these two values:

$$\text{F-score} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.$$

The script writes results into the standard output in the form of a table. Since it produces much extra information, we link `cleaneval.py`'s standard output (by pipe) with a simple script `evaluate.py` which computes average values for all files in the directory and writes them into the standard output.

3.2 Silver standard

We were forced to prepare our own dataset because of our distinct approach which requires a set of pages from a single website.

Collecting

Files in Silver standard were downloaded manually. It was possible to pick out files randomly from immense amount of crawled data but we wanted to ensure diversity of types of HTML pages.

The dataset consists of 100 (50 couples) HTML files. Each file (`xxa.html`) has its own counterpart (`xxb.html`) which comes from the same website.

Here is a survey of archetypes of webpages included in Silver standard:

- ▷ *equally structured pages, differing slightly in menus*: Wikipedia, The New York Times, CNN news, blogs, ...,
- ▷ *complicatedly structured pages with rather poor valuable content*: Amazon, ebay.com, discussion forums, ...,
- ▷ *sparse pages without continuous text* – tables, "no results" pages, ... and
- ▷ *simply formatted pages* without graphical layout.

In ideal case, two pages (being cleaned) have absolutely the same structure and differ only in the desired, valuable part. In this case multiple file method would yield absolutely correct result. But it would be kind of cheating to include only such files into Silver standard. Instead, it includes several non-corresponding couples of files.

Manual cleaning

Two questions emerge immediately: (i) what to consider as boilerplate and "clear" text? and (ii) what is a heading, a list and a paragraph?[2]

Both questions are subject to our own personal (subjective) consideration. An answer to the first question is probably more complicated.

Let us discuss an example of a page corresponding with the second archetype: Amazon product page (concretely file `03a.html` from Silver standard). It is straightforward to take the product name as a heading, but is "Canon Pixma MP610 Photo All-In-One Inkjet Printer (2180B002)" valuable for us? Yes, it may be. Unfortunately, it is the easiest step of cleaning because plenty of words and phrases follow. The overwhelming majority of them may be considered as repetitive (and thence boilerplate): "In stock.", "See larger image", "There is a newer model of this item:", "Frequently Bought Together", ... And we continue till heading "Product Description". No doubt, it is unique and to be cut out as valuable text. We omit the offer of related accessories since it also contains only repetitive phrases: "In stock. Processing takes an additional 4 to 5 days...", ... We skip following sections up to Customer Reviews. Ecce – unique information! After deleting some repetitive phrases ("93 of 97 people found the following review helpful", "Published 15 days ago by Mimi", ...) we obtain quite a fine text. Despite several sections remain, we omit them not being relevant and, eventually, reach the end of the page.

An answer to the second question is a little bit clearer. Given any webpage we are to chose – for each part of relevant text – from one of the three tags: `<h>`, `<l>`, `<p>`.

We use `<h>` for headings which are, fortunately, easily distinguishable. They often differ in colour, size and positioning on a webpage. They also capture content (or meaning) of the following portion of text. But this information is available only to us – human beings – not to algorithms, so we have an advantage.

Tag `<l>` is used for items of a list. It is often very easy to distinguish list items in web pages. Both their graphical formatting and their content help us.

Every other part of text in web page is treated as `<p>`.

[2] In fact, Gold standard was prepared by several volunteers and the final output was averaged from their particular results.

3.3 Single file cleaning

Cleaneval dataset – Gold standard

Gold standard (provided by Cleaneval) consists of 58 manually chosen HTML files and of 58 corresponding, manually cleaned and annotated text files.

We can compare our results with Cleaneval's participants approximately[3] – see table 3.1. Results of the winner of Cleaneval (WCP) are taken from [21]. We also can compare our approach with BTE[4] algorithm.

	precision	recall	F-score
WCP	84.10	65.30	—
BTE	87.28	**85.40**	**83.42**
WCC[1]	**90.83**	79.76	81.24
WCC[2]	89.17	81.57	80.05

Table 3.1: Evaluation results on Gold standard.

As for WCP, according to [21], precision in table 3.1 means "text-only" cleaning and recall means "text and markup" cleaning. Therefore this precision is more or less comparable with our and BTE's results while recall is not.

WCC[1] is our WC Cleaner run in single file mode. WCC[2] is run in mode that omits sparse documents (see Appendix). We can see that BTE is better in tag matching while WCC[1] is better in precision, thus, in boilerplate removing. Results show that the omitting of sparse documents is not convenient. It is also obvious that decrease of precision causes increase of recall – compare results of WCC[1] and WCC[2].

Silver standard

We also evaluated WC Cleaner on our own dataset – Silver standard. For these results see table 3.2.

WCC[1] and WCC[2] have the same meaning as before. This evaluation is similar to the previous one.

[3] Actually, we do not have test set used on Cleaneval at our disposal.

[4] An implementation by Jan Pomikálek was used.

	precision	recall	f-score
BTE	83.64	**72.96**	**68.78**
WCC^1	**92.09**	58.13	62.53
WCC^2	88.41	67.23	67.16

Table 3.2: Evaluation results on Silver standard.

3.4 Multiple file cleaning

To be able to compare results of BTE and WC Cleaner run in multiple file mode, Silver standard must be processed two times – WC Cleaner needs a reference file for each cleaning. We have at our disposal couples of corresponding[5] files (`xxa.html` and `xxb.html` in Silver standard so we run WC Cleaner on `*a.html` files using `*b.html` files as reference files and vice versa.

	precision	recall	f-score
BTE	83.64	72.96	68.78
$WCC^{\emptyset}$	85.07	69.46	66.14
WCC^{F}	83.37	70.60	66.05
WCC^{P}	**85.16**	68.86	65.68
WCC^{R}	82.84	**76.09**	**68.63**
WCC^{PR}	82.93	75.50	67.92
WCC^{PF}	83.46	70.36	65.91

Table 3.3: Evaluation results on Silver standard.

Here, $WCC^{\emptyset}$ stands for the multiple file mode, WCC^{F} for the mode in which the only target node of document is cleaned (against the only target node of reference document), WCC^{P} for the mode in which file differing structurally from an appropriate reference file is cleaned in single file mode and WCC^{R} for the same mode as WCC^2 within single file mode. WCC^{PR} and WCC^{PF} are combinations of previous modes.

Now, BTE wins clearly. But results of WCC are pretty comparable too. WCC^{P} gets considerably closer to BTE's precision but it is caused by the fact that during this run, several files are cleaned in single file mode.

The highlighted recall (the highest in this column) of WCC^{R} is not so significant – notice that WCC^{R} has also the lowest precision.

[5]Downloaded from the same website.

3.5 Time consumption

Since we need to clean thousands and millions of files, time consumption is a crucial point. A survey of particular procedures and their time consumptions is listed in table 3.4. All benchmarks (except of sorting) were done on PC with Pentium Dual Core 2.5GHz processor and 2 GB RAM.

	GS	**SS**
cleaning.py	1.85	3.67
TagSoup	21.6	37
BTE	8.37	17.31
WCC^{1}	9.8	17.42
WCC$^{\emptyset}$	—	27.61
WCC$^{\mathrm{R}}$	—	27.11

Table 3.4: Time consumption (in seconds) of particular procedures.

We also tested `arc_extract.py` in batch sorting of 85 ARC files (each file about 100 MB) which took approximately 185 minutes. This benchmark was done on a server in NLP laboratory with a quad-core AMD Opteron 2.5GHz processor and 32 GB RAM.

3.6 Conclusion

We can see that simpler methods yield better results – in average. In a few cases, multiple file method gives very good results but, in general, it rather struggles with wild forests full of DOM trees. This method has good results on well structured web pages but in these cases also simple methods (single file mode, BTE) give satisfying results.

In spite of the fact that WC Cleaner wins (especially in the case of single file method) in precision, BTE is always better in recall. But it was already mentioned that precision is more important for us. We need mainly relevant text data without boilerplate. Their structure is subsidiary.

Appendix – Brief documentation

`arc_extract.py` script

The script takes two arguments – the first one stands for name of an ARC file and the second one for the directory where to put resulting ARC files with sorted files. It save these ARC files into appropriate subdirectories. The script uses an adapted ARC file parser by RNDr. Jan Pomikálek.

`cleaning.py` script

This script serves for removing unwanted elements (tags) from HTML file. It tries to remove scripts (`<script>`), styles (`<style>`) and HTML comments (`<!-- -->`). It uses regular expressions from module `re`. The script reads the standard input and writes its output in the standard output.

`tagsoup` tool

Tagsoup tool is used for validating HTML files. In our case just one parameter sufficed: `--nobogons` triggers off removing of unknown HTML elements.

The tool takes the filename of the file to be validated as a parameter and writes its output in the standard output. For more detailed information and documentation, see [14].

`DOMSTree` class

This class consists of methods working directly with DOM structure. Here is a brief list of them:

- `__init__(self, html_body)`
  Variable `html_body` stands for a string – whole content of HTML file. This method initializes the instance of `DOMSTree` class. It parses a HTML file, creates new attributes of DOM tree which will be computed in future (`chars`, `tags`, `s_tags`).

- ▷ `addStatToDOMTree(self, actualNode)`
  recursively goes from `actualNode` into its child nodes, computes the attributes for every element node and saves them into DOM structure.
- ▷ `getMainHeadline(self)`
  tries to extract general title of whole document: it looks at all `<hX>` tags and takes content of the first occurrence of them. If it does not succeed, it looks at tag `<title>`. It returns textual content of one of these tags or empty string.
- ▷ `getTargetNode(self, node)`
  Given a node of DOM tree, this method looks for the target node. This procedure is described on page 18.
- ▷ `convertNode(self, node, withinTag)`
  converts DOM tree into the format required by Cleaneval annotation guidelines (see [17]). Heading tags (`<hX>`) are converted into `<h>` tag, items of lists (`<li>`) are converted into `<l>` tag. The other situations are described on page 22.
- ▷ `diffClean(self, docNode, refNode)`
  Given two nodes, this method simply calls another recursive method `diffCleanNode` and returns its output.
- ▷ `diffCleanNode(self, docNode, refNode)`
  steps through DOM subtrees of both nodes simultaneously. It looks for two nodes (one from docNode DOM subtree, the other from refNode DOM subtree) which differ mutually. In such case it calls method `diffCleanLowestLevel` and passes these two nodes on.
- ▷ `isEqualLevel(self, docNode, refNode)`
  compares lists of child nodes (only element and non-empty text child nodes) of `docNode` and `refNode`, respectively. If these lists are equal, the method returns *true*.
- ▷ `deleteEqualNodes(self, docNode, refNode)`
  takes two element nodes and steps recursively through their DOM subtrees. If it finds two equal text parts, those from `docNode` are removed. The method returns number of all removed characters in `docNode` DOM subtree.
- ▷ `diffCleanLowestLevel(self, docNode, refNode)`
  `docNode` and `refNode` are those elements which differ in structure. They may contain valuable text data but they can include some common or repetitive parts. This method tries to find these common parts.
- ▷ `isVisibleItem(self, node), isElementNode(self, node)`
  have obvious meanings. A node is a "visible item" if it is an element node or a non-empty text node.

- ▷ `getChars(self, node), getDensity(self, node)`
  return number of chars in `node` and density of element `node`, respectively.
- ▷ `DOM2String(self, node)`
  simply converts recursively a DOM subtree into a string. Tag names of element nodes are not taken into account.
- ▷ `isSimilarByChildCount(self, node1, node2, level)`
  compares two DOM subtrees according to their structure – number of child nodes.

All methods are profusely commented in the source code.

`wc.py` – main script

The script `wc.py` manages the whole process of cleaning. It has several options and their complete survey follows:

- ▷ `-i INPUTFILE | --input=INPUTFILE`
  a filename of the input file
- ▷ `-d DIRECTORYNAME | --directory=DIRECTORYNAME`
  If you want to clean whole directory instead of a single file, this option stands for the directory name. The directory must contain only (valid) HTML files.
- ▷ `-s | --singlemode`
  Multiple file cleaning mode is implicit and single file cleaning mode must be turned on with this option.
- ▷ `-r | --sparse`
  You can omit sparse files (consisting only of tables, lists, ...) with this option. The tool decides according to density.
- ▷ `-e | --reference`
  Running in multiple file mode, the algorithm requires a reference file. This option specifies its filename. It is an obligatory option for multiple file mode.
- ▷ `-t | --addtitle`
  Sometimes it is useful to add a title at the beginning of the document.
- ▷ `-p | --prune`
  Multiple file cleaning requires similarly structured documents. This option causes omitting differently structured documents (according to a reference file). These documents are subsequently cleaned in single file mode.
- ▷ `-v | --verbose`
  With this option, information about the process is sent in the standard error output.

- ▷ `-f | --fromtarget`
  Running in multiple file mode, you can specify that the algorithm will start to clean a document from its target node.
- ▷ `-c | -- clave`
  When you store many cleaned documents into a single file, you may need to separate them. This option adds `<doc>` tag at the beginning of each document.

`wclib.py`

This script includes additional functions and constants. Here are the most important of them:

- ▷ `SPARSE_DENSITY = 10`
  Crucial constant for omitting sparse documents.
- ▷ `exportDoc(doc, string, addTitle, separate)`
  Makes final touches on exported text. Adds optional title, `<doc>` delimiter etc.
- ▷ `findCommonParts(docStr, refStr)`
  Web pages usually include repetitive parts, e.g. “posted by xyz”, “send mail” etc. This function tries to find these common parts of texts. `"posted by Jack F."` (`docStr`) and `"posted by anonymous"` result in `"Jack F."`. This function returns only appropriate indexes from `docStr`. The string is cut off within further progress of multiple file cleaning.
- ▷ Lists described on page 22.

Bibliography

[1] MIKLE, Pavol. *XDHTML – úplná přesná referenční příručka HTML DHTML XML*. 1. ed. Brno : Zoner Press, 2004.

[2] FAIRON, Céderick – NAETS, Hubert – KILGARRIFF, Adam – SCHRYVER, Gilles-Maurice de. *Building and Exploring Web Corpora* – Proceedings of the 3rd Web as Corpus workshop, incorporating Cleaneval. Louvain : Presses universitaires de Louvain, 2007.

[3] BAUER, Daniel – DEGEN, Judith – DENG, Xiaoye et al. *FIASCO: Filtering the Internet by Automatic Subtree Classification*. In Cahiers du Cental, no. 4, 2007. p. 111–121.

[4] EVERT, Stefan. *StupidOS: A high-precision approach to boilerplate removal*. In Cahiers du Cental, no. 4, 2007. p. 123–133.

[5] GAO, Weizheng – ABOU-ASALEH, Tony. *GenieKnows Web Page Cleaning System*. In Cahiers du Cental, no. 4, 2007. p. 135–139.

[6] GIRARDI, Christian. *Htmcleaner: Extracting the Relevant Text from the Web Pages*. In Cahiers du Cental, no. 4, 2007. p. 141–143.

[7] HOFMANN, Katja – WEERKAMP, Wouter. *Web Corpus Cleaning using Content and Structure*. In Cahiers du Cental, no. 4, 2007. p. 145–154.

[8] MAREK, Michal – PECINA, Pavel – SPOUSTA, Miroslav. *Web Page Cleaning with Conditional Random Fields*. In Cahiers du Cental, no. 4, 2007. p. 155–162.

[9] SARALEGI, Xabier – LETURIA, Igor. *Kimatu, a tool for cleaning non-content text parts from HTML docs*. In Cahiers du Cental, no. 4, 2007. p. 163–167.

[10] MERTZ, Paul. *Text processing in Python*. 1. ed. Boston : Addison-Wesley, 2003.

[11] RYBIČKA, Jiří. *LaTeXpro začátečníky*. 3. ed. Brno : Konvoj, 2003.

[12] HORNBY, A S. *Oxford Advanced Learner's Dictionary of Current English*. 7. ed. Oxford : Oxford University Press, 2005.

[13] SWAN, Michael. *Practical English Usage*. 3. ed. Oxford : Oxford University Press, 2005.

[14] *TagSoup home page* [online] 2002-2008. <http://home.ccil.org/~cowan/XML/tagsoup> [accessed 18 May 2009]

[15] *The Gutenberg Galaxy*. Wikipedia, The Free Encyclopedia. [online] <http://en.wikipedia.org/w/index.php?title=The_Gutenberg_Galaxy&oldid=285820849> [accessed 18 May 2009]

[16] *Cascading Style Sheets*. Wikipedia, The Free Encyclopedia. [online] <http://en.wikipedia.org/w/index.php?title=Cascading_Style_Sheets&oldid=289451348> [accessed 18 May 2009]

[17] BARONI, Marco – SHAROFF, Serge – HARTLEY, Tony – KILGARRIFF, Adam. *[Sigwac] CLEANEVAL annotation guidelines: feedback request*. <http://liste.sslmit.unibo.it/pipermail/sigwac/2006-October/000020.html> [accessed 18 May 2009]

[18] POMIKÁLEK, Jan – RYCHLÝ, Pavel – KILGARRIFF, Adam. *Scaling to Billion-plus Word Corpora.* In Advances in Computational Linguistics, Mexiko : Instituto Politécnico Nacional, 41, 2009, p. 3-13.

[19] *CLEANEVAL home page* [online] 2007. <http://cleaneval.sigwac.org.uk/> [accessed 18 May 2009]

[20] FINN, A – KUSHMERICK, N. – Smyth, B. *Fact or fiction: Content classification for digital libraries.* In DELOS Workshop: Personalisation and Recommender Systems in Digital Libraries, 2001

[21] BARONI, Marco – CHANTREE, Francis – KILGARRIFF, Adam, SHAROFF, Serge. *CleanEval: a competition for cleaning webpages* <http://www.kilgarriff.co.uk/Publications/2008-BaroniChantreeKilgSharoff-LREC-cleaneval.pdf [accessed 20 May 2009]

[22] *HTML Tidy Project Page* [online] 2005. <http://tidy.sourceforge.net> [accessed 18 May 2009]

[23] PILGRIM, Mark. *Universal Encoding Detector: character encoding auto-detection in Python* [online] 2006-2008 <http://chardet.feedparser.org> [accessed 19 May 2009]

[24] BURNER, Mike – BREWSTER, Kahle. *Internet Archive: Research Access – Arc File Format* [online] 1996 <http://www.archive.org/web/researcher/ArcFileFormat.php> [accessed 19 May 2008]

[25] *W3C Document Object Model* [online] 2009. <http://www.w3.org/DOM> [accessed 20 May 2009]

[26] *Web as Corpus: Software & Data Sets: Software* [online] <http://webascorpus.sourceforge.net/PHITE.php?sitesig=FILES&page=FILES_10_Software> [accessed 20 May 2009]