

Univerzita Karlova v Praze  
Matematicko-fyzikální fakulta

## BAKALÁŘSKÁ PRÁCE



Michal Marek

### **Automatické čištění HTML dokumentů** **Automatic cleaning of HTML documents**

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Pavel Pecina  
Studijní program: Informatika, programování

2007

Chtěl bych poděkovat svému vedoucímu Pavlovi Pecinovi za vypsání zajímavého tématu, rady k implementaci programu a výběr literatury. Jemu i jeho kolegovi Mirkovi Spoustovi také děkuji za připomínky ke článku a práci. Na závěr chci poděkovat své ženě Petře za pomoc při přípravě trénovacích dat a za podporu.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 8. srpna 2007

Michal Marek

# Obsah

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Related Work . . . . .	6
<b>2</b>	<b>User documentation</b>	<b>7</b>
2.1	Installation . . . . .	7
2.2	Configuration . . . . .	8
2.3	Usage . . . . .	8
<b>3</b>	<b>System Overview</b>	<b>12</b>
3.1	Standardizing HTML code . . . . .	12
3.2	Precleaning . . . . .	13
3.3	Parsing Precleaned HTML . . . . .	13
3.4	Feature extraction . . . . .	14
3.5	Learning And Cleaning . . . . .	15
<b>4</b>	<b>Features</b>	<b>17</b>
4.1	Markup-based Features . . . . .	17
4.2	Content-based Features . . . . .	18
4.3	Document-related features . . . . .	20
<b>5</b>	<b>Data and Evaluation</b>	<b>21</b>
<b>6</b>	<b>Experiments and Results</b>	<b>23</b>
<b>7</b>	<b>Conclusions</b>	<b>25</b>
7.1	Future Ideas . . . . .	26

Název práce: Automatické čištění HTML dokumentů  
Autor: Michal Marek  
Katedra (ústav): Ústav formální a aplikované lingvistiky  
Vedoucí bakalářské práce: Mgr. Pavel Pecina  
e-mail vedoucího: pecina@ufal.mff.cuni.cz

Abstrakt: Tato práce popisuje systém pro automatické čištění HTML dokumentů, který byl použit při účasti Univerzity Karlovy v soutěži CLEAN-EVAL 2007. CLEAN-EVAL je *sdílená úloha* (shared task) a soutěž automatických systémů pro čištění libovolných stránek s cílem použít webová data jako korpus v počítačové lingvistice a zpracování přirozeného jazyka. Tuto úlohu řešíme jako problém *značkování sekvencí* (sequence labeling) a náš experimentální systém je založen na algoritmu Conditional Random Fields, používajícím *vlastnosti* (features) bloků textu odvozené z textového obsahu a HTML struktury analyzovaných webových stránek.

Klíčová slova: Čištění webových stránek, Sequence labeling, Conditional random fields

Title: Automatic cleaning of HTML documents  
Author: Michal Marek  
Department: Institute of Formal and Applied Linguistics  
Supervisor: Mgr. Pavel Pecina  
Supervisor's e-mail address: pecina@ufal.mff.cuni.cz

Abstract: This paper describes a system for automatic cleaning of HTML documents, which was used in the participation of the Charles University in CLEAN-EVAL 2007. CLEAN-EVAL is a shared task and competitive evaluation of automatic systems for cleaning arbitrary web pages with the goal of preparing web data for use as a corpus in the area of computational linguistics and natural language processing. We try to solve this task as a sequence-labeling problem and our experimental system is based on Conditional Random Fields exploiting a set of features extracted from textual content and HTML structure of analyzed web pages for each block of text.  
Keywords: Web page cleaning, Sequence labeling, Conditional random fields

# Chapter 1

## Introduction

The web with its enormous amounts of textual materials is a rich and easily accessible source of linguistic data that can be used to create extremely large corpora with relatively low cost and within a short period of time (compared to the traditional way of building text corpora which is an expensive and time-consuming process). The idea of having a corpus “as large as the web” recently attracted attention of many researchers in computational linguistics, natural language processing, and related areas, who would greatly benefit from such amount of data. Creating such a corpus comprises two steps: a) *web crawling* - automatic browsing the web and keeping a copy of visited pages and b) *cleaning* the pages to be included in the corpus. In this work we focus on the latter.

Apart from the main content, a typical web page contains other material of no linguistic interest, such as navigation bars, panels, and frames, page headers and footers, copyright and privacy notices, advertisements and other data (often called *boilerplate*). The goal is to detect and remove such parts from an arbitrary web page.

Although this task might seem too heterogeneous to be appropriate for methods based on supervised training, we solve it as a sequence-labeling problem with Conditional Random Fields trained on a manually cleaned set of web pages. Our primary goal is an attempt to explore whether supervised training methods can perform well enough to be successfully used in this task.

## 1.1 Related Work

Although web page cleaning is a crucial step in the procedure of building a web corpus, only a relatively little work has been done in this area. Most of it originated in the area of web mining and search engines, e.g. [3] or [5]. In [1], a notion of pagelet determined by the number of hyperlinks in the HTML element is employed to segment a web page; pagelets whose frequency of hyperlinks exceeds a threshold are removed. [6] extract keywords from each block content to compute its entropy, and blocks with small entropy are identified and removed. In [8] and [7], a tree structure is introduced to capture the common presentation style of web pages and entropy of its elements is computed to determine which element should be removed. In [2], a two-stage web page cleaning method is proposed. First, web pages are segmented into blocks and blocks are clustered according to their style features. Second, the blocks with similar layout style and content are identified and deleted. As far as the authors know, none of the published method was based on sequence labeling or a similar learning algorithm.

# Chapter 2

## User documentation

The system we developed is called *Victor* and is distributed along with this paper. Following are instructions on installing, optionally configuring and using *Victor*.

### 2.1 Installation

Being still somewhat experimental, the package does not have any install script. The suggested usage for now is running the scripts from the unpacked source directory. Following Perl modules are required (available from CPAN):

- `Curses` (for the annotation tool only)
- `HTML::Entities`
- `HTML::Parser`
- `HTML::Tidy` (optional)

And the following programs:

- CRF++, available from <http://crfpp.sourceforge.net/>
- HTML Tidy, available from <http://tidy.sourceforge.net/>

The included `check-deps.pl` script can be used to check if all the required packages are installed.

## 2.2 Configuration

There are two sources of configuration in *Victor*. The so called config file, placed in the `configs` sub-directory, and the CRF++ template file, placed in the `templates` sub-directory. All *Victor* scripts accept the `--config <name>` command line option to switch between configs. There is a preset config for use in the CLEANVAL shared task in `configs/cleaneval.conf` (the command line option then reads `--config cleaneval`). The config file has a simple format:

```
variable: value
```

Where `value` can be either a single token or an array of these, depending on the type of the variable. In case of an array, the definition can be split into multiple lines:

```
variable: value1
variable: value2
```

is equivalent to

```
variable: value1 value2
```

The meaning of the individual variables is discussed throughout the text.

## 2.3 Usage

To use *Victor* for cleaning web pages, a set of annotated HTML files for training is needed. *Victor* comes with files from the CLEANVAL development set, converted to the *Victor* annotation format. These files are found in `cleaneval/*.anno`. The `cleaneval-extra/` directory contains additional training data prepared by us. To annotate own files, run the following command in the unpacked source directory:

```
./annotate.pl --config cleaneval file.html file.anno
```

This will launch a full-screen terminal application that allows you to assign an annotation tag to each segment of text by pressing one of the keys displayed on the bottom. Besides the `paragraph`, `header` and `list` tags specified by the CLEANVAL task, there is the `other` tag for marking text



that should be cleaned away, and the `continuation` tag, denoting text segments that connect to the last segment marked as paragraph, header or list. The `continuation` tag is needed, because *Victor* splits the input on HTML tags, but paragraphs, headers or list items can span multiple segments, for example

```
<p> See <a href="http://example.org">our website</a>
for more details.</p>
```

would be annotated as (assuming the text is to be annotated as a single paragraph)

```
[p] See
[c] our website
[c] for more details.
```

Training the CRF++ engine on the annotated data is done by running the `learn.pl` script:

```
./learn.pl --config cleaneval file1 file2 ...
```


This will store the results in a “model” file in the `models` sub-directory. To train on the included CLEANVAL development dataset, run

```
./learn.pl --config cleaneval cleaneval/*.anno
```


Finally cleaning web pages is done using the `clean.pl` script. By default, its output is the same as that of the annotation tool. This can either be converted to the CLEANVAL annotation format by the `victor2cleaneval` script, or by running `clean.pl` with the `--format cleaneval` parameter:

```
./clean.pl --config cleaneval --format cleaneval files ...
```




The advantage of the latter method is that it supports the special `<text id="...">` tag and adds the URL: `... line` to the output file, as required by CLEANVAL.



# WAC3 - 2007



Web as Corpus 2007, UCLouvain, Louvain-la-Neuve, September 15-16 2007 (Belgium)

WAC3
<ul style="list-style-type: none"> <li>o Call for papers</li> <li>o Submit a paper</li> <li>o Registration</li> <li>o Program</li> <li>o Scientific committee</li> <li>o Travel info. &amp; venue</li> <li>o Local organisation team</li> <li>o Associated events</li> <li>o SIGWAC</li> <li>o Previous Workshops</li> <li>o Pictures</li> </ul>
Cleeveval
<ul style="list-style-type: none"> <li>o Information</li> <li>o Scientific committee</li> </ul>
Organisation
 <p style="margin: 5px 0;"><b>UCL</b> Université catholique de Louvain</p>  <p style="margin: 5px 0;"><b>UNIVERSITEIT GENT</b></p>  <p style="margin: 5px 0; font-size: small;">CENTAL Centre de traitement automatique du langage</p>

### SIGWAC

SIGWAC is the Special Interest Group of the Association for Computational Linguistics (ACL) on Web as Corpus

Its objectives are

- ◆ to promote interest in the use of the web as a source of linguistic data, and as an object of study in its own right;
- ◆ to provide members of the ACL with a special interest in the web-as-corpus with a means of exchanging news of recent research developments and other matters of interest;
- ◆ to sponsor meetings and workshops on the web as corpus that appear to be timely and worthwhile.

More info on [SIGWAC](#)

---

Last update : July, 2007

Figure 2.1: SIGWAC website as rendered by a web browser. Text that should be extracted is marked red.

```
<h> SIGWAC
<p> SIGWAC is the Special Interest Group of the
Association for Computational Linguistics (ACL)
on Web as Corpus

<p> Its objectives are

<l> to promote interest in the use of the web as a
source of linguistic data, and as an object of study
in its own right;
<l> to provide members of the ACL with a special
interest in the web-as-corpus with a means of
exchanging news of recent research developments
and other matters of interest;
<l> to sponsor meetings and workshops on the web as
corpus that appear to be timely and worthwhile.
<p> More info on
```

Figure 2.2: The same page as cleaned by *Victor*, with line breaks added. Except for the stray “More info on” paragraph, the result looks promising.

# Chapter 3

## System Overview

The system is based on sequence labeling with CRF++<sup>1</sup>, an implementation of Conditional Random Fields [4]. It is aimed at cleaning arbitrary web pages as specified in the CLEANVAL shared task description<sup>2</sup>. Processing the HTML input consists of several steps:

### 3.1 Standardizing HTML code

The raw HTML input is passed through Tidy<sup>3</sup> (either via pipes to the tidy binary, or using the `HTML::Tidy` Perl module), in order to get a valid and parsable HTML tree. During development, we found only one significant problem with Tidy, namely that it does not interpret code inside the `<script>` element, which means that it will be confused by JavaScript strings containing `<script>` or `</script>`. We employed a simple workaround for it in the `preclean` module. Except for this particular problem which occurred only once in our training data, Tidy has proved to be a good choice.

---

<sup>1</sup><http://crfpp.sourceforge.net/>

<sup>2</sup><http://cleaneval.sigwac.org.uk/>

<sup>3</sup><http://tidy.sourceforge.net/>

## 3.2 Precleaning

Afterwards, the HTML text “precleaned”: It is parsed<sup>4</sup> and some elements (like scripts, style definitions, embedded objects, etc) are already deleted. The exact list of elements to delete is specified by the `tag-delete` configuration variable. Remaining text is split by HTML tags into a sequence of so called text blocks. For example, the snippet

```
<p>Hello <b><i>world</i></b>!</p>
```

would be split into three blocks, “Hello”, “world”, and “!”. The rest of the system then views the document as a series of blocks. This approach has two potential problems:

- A given block is classified as whole, there is no way to classify a part of a block differently than the rest. In practice, this turned out to be a significant problem only for text inside the `<pre>` element. Therefore, inside the `<pre>` element, each line of text is separated with a `<br>` tag, splitting the content into multiple blocks. This allows each line to be classified separately.
- Some text rendered on web pages is contained in tag attributes, which are ignored by the parser. This is especially true for the `alt` attribute of the `<img>` tag and the `value` attribute of the `<input>` tag. For `<img>`, this is solved by copying the alternative text into an artificial text child of the `<img>` element in the preclean stage. We believe that most `<input>` tags on the Web do not carry any useful information in their `value` attributes, therefore we ignore them (although the work-around would be the same).

## 3.3 Parsing Precleaned HTML

In this step, the precleaned HTML text is again parsed with `HTML::Parser` and the blocks identified in the previous stage are loaded into memory. Each block is internally stored as a Perl hash with these fields:

`id`

The number of the block (first block in the document has `id 0`).

---

<sup>4</sup>Using the `HTML::Parser` Perl module, which provides an easy-to-use SAX-like API.

**text**

Text of the block, with HTML entities decoded.

**containers**

An array of parent elements, from outermost to innermost.

**distance**

An array of tags seen since last block, "**x**" stands for opening tag `<x>` and `"/x"` stands for closing tag `</x>`.

**td\_group, div\_group**

Reference to an array of numbers of blocks that belong to the same `td_group` or `div_group`. The meaning of these fields is explained in 4.2.

**fv**

The feature vector of this block. This field created in the next stage.

**class**

For annotated HTML, this field stores the assigned result tag.

These fields are then used by the feature extraction modules to compute the feature vector of each block. Currently, these fields are always generated, even when running the annotation tool (which does not need any feature vectors)<sup>5</sup>.

## 3.4 Feature extraction

In this step, a feature vector is generated for each block and stored in the `fv` field. The list of features and their detailed description is presented in the next chapter. The features must have a finite set of values<sup>6</sup>. The mapping of integers and real numbers into finite sets was chosen empirically and is specified in the configuration. Possible values for feature  $X$  are specified in the variable `feature-X-values`, the mapping is “largest less-or-equal”.

---

<sup>5</sup>On a 2.40GHz Pentium 4 machine, not generating these fields would save only about 0.06 seconds when parsing a 150 Kb HTML file, so optimizing here is not worth it.

<sup>6</sup>This is a limitation of the CRF tool used.

Features that are based on a percentage (e.g. relative position in the document) are scaled in a similar way: The number of possible values for feature  $Y$  is specified in the variable `feature- $Y$ -scale`.

Most features are generated separately by independent modules. This allows for adding other features and switching between them for different tasks easily.

Not all features that are generated in this stage are actually used for learning and cleaning, the idea is that the feature modules should be as simple as possible and fill the feature vector with all features they can generate and that the next phase then picks those features that are wanted. This is a memory waste, but so far it does not seem to be a problem.

### 3.5 Learning And Cleaning

The CRF++ tool expects the document to have the following tabular format:

```
feature1 feature2 ... featureN tag
feature1 feature2 ... featureN tag
```

that is one row for each “word” (a text block in our case), one column for each feature and the result tag in the last column (the input of the cleaning tool does not have the last column of course). Furthermore, the learning tool needs a so called “template”, that defines actual features used by CRF++. A feature in CRF++ is one expanded line in the template, with each line being a reference into the document table, relative to the current line<sup>7</sup>. *Victor* does not use the CRF++ template directly, it generates the templates from its own format. See the `templates/cleaneval.tpl` file for an example.

The `crf-features` configuration variable defines, which features are output into the document table. Its purpose is also to provide a stable mapping of feature names to column numbers<sup>8</sup>.

To be able to map rows in the document table back to the text blocks, the first column is not a real feature, but the id of the text block. The

---

<sup>7</sup>In fact, this is more complicated. CRF++ defines a binary feature for each unique string the line can expand to, and the line can be composed of multiple references into the document table. See CRF++ documentation for details.

<sup>8</sup>Feature vectors are stored as Perl hashes, which do not have any defined sorting.

template file never references the first row, so the block ids do not influence the learning and cleaning.

The output format of the CRF++ cleaning tool `crf_test` is the same as the input format of the learning tool. When cleaning, *Victor* dumps the document table without the last columns, runs `crf_test`, and set the `class` field of each block according to the last column. The array of blocks is then passed to one of the output handlers `Victor::Output::*`.



# Chapter 4

## Features

Features recognized by *Victor* can be divided by their scope into three subsets: features of individual text blocks, features of groups of blocks and features of the whole document. Furthermore, features can be divided by their source into features based on the HTML markup and features based on the text content of the blocks.

### 4.1 Markup-based Features

**container.p, container.a, container.u, container.img, container.class-header, container.class-bold, container.class-italic, container.class-list, container.class-form**

For each parent element of a block, a corresponding *container.\** feature will be set to 1, e.g. a hyperlink inside a paragraph will have the features *container.p* and *container.a* set to 1. This feature is especially useful for classifying blocks: For instance a block contained in one of the `<h $x$ >` elements is likely to be a header, a *container.li* feature suggests a list item, etc. The *container.class-\** features refer to classes of similar elements rather than to elements themselves. This grouping is done in order to reduce the length of the feature vector<sup>1</sup>.

**split.p, split.br, split.hr, split.class-inline, split.class-block**

For each opening or closing tag encountered since the last block, we generate a corresponding *split.\** feature. This is needed to decide,

---

<sup>1</sup>and therefore be able to train on a smaller set of documents.

whether a given block connects to the text of the previous block (classified as *continuation*) or not. Also, the number of encountered tags of the same kind is recorded in the feature. This is mainly because of the `<br>` tag; a single line break does not usually split a paragraph, while two or more `<br>` tags usually do. The *split.class-\** features again refer to classes of similar elements.

## 4.2 Content-based Features

### **char.alpha-rel, char.num-rel, char.punct-rel, char.white-rel, char.other-rel**

These features represent the absolute and relative counts of characters of different classes (letters, digits, punctuation, whitespace and other) in the block.

### **token.alpha-rel, token.num-rel, token.mix-rel, token.other-rel, token.alpha-abs, token.num-abs, token.mix-abs, token.other-abs**

These features reflect distribution of individual classes of tokens<sup>2</sup>. The classes are words, numbers, mixture of letters and digits, and other.

### **sentence.count**

Number of sentences in a block. For English, we use a naive algorithm basically counting periods, exclamation marks and question marks, without trying to detect abbreviations. Given that the actual count is mapped into a small set of values anyway, this does not seem to be a problem. For Czech language, we use a module, developed by one of our colleagues, that also detects abbreviations. But we did not think it is worth the effort creating a new module for English.

### **sentence.avg-length**

Average length of a sentence, in words.

### **sentence-begin, sentence-end**

---

<sup>2</sup>In this context, tokens are sequences of arbitrary characters separated by whitespace.

These features identify text blocks that start or end a sentence. This helps recognizing headers and list items (as these usually do not end with a period) as well as continuation blocks (*sentence-end=0* in the previous blocks and *sentence-start=0* in the current block suggest a continuation).

### **first-duplicate, duplicate-count**

The *duplicate-count* feature counts the number of blocks with the same content (ignoring white space and non-letters). The first block of a group of duplicates is then marked with *first-duplicate*. This feature serves two purposes: On pages where valid text interleaves with noise (blogs, news frontpages, etc), the noise often consists of some phrases like “read more...”, “comments”, “permalink”, etc, that repeat multiple times on the page. The second purpose of this feature is to identify quotes in discussions, which are deleted in the CLEANVAL development set. The *first-duplicate* feature is then used to recognize the original post. Especially for the second use case, there is room for improvement in being able to express different levels of text similarity among the blocks, to be able to also detect quotes of parts of the original text.

### **regexp.url, regexp.date, regexp.time**

While we try to develop a tool that works independently of the human language of the text, some language-specific features are needed nevertheless. The configuration defines each *regexp.\** feature as an array of regular expressions. The value of the feature is the number of the first matching expression (or zero for no match). We use three sets of regular expressions: to identify times and dates (lines with creation date/time are marked as *header* in CLEANVAL data) and URLs (these are usually cleaned away in the CLEANVAL data).

### **bullet**

This is a specialized version of the *regexp.\** features, matching different patterns that suggest a list item (number or a single letter, closing parenthesis, dash, etc). Each combination of matches results in a different value of the feature, so that lists can be recognized as a sequence of blocks with the same *bullet* value.

### **div-group.word-ratio, td-group.word-ratio**

The layout of many web pages follows a similar pattern: The main content is enclosed in one big `<div>` or `<td>` element, as are the menu bars, advertisements etc. To recognize this feature and express it as a number, the parser groups blocks that are direct descendants of the same `<div>` element (`<td>` element respectively). A direct descendant in this context means that there is no other `<div>` element (`<td>` element respectively) in the tree hierarchy between the parent and the descendant. For example in this markup

```
<div> a <div> b c </div> d <div> e f </div> g </div>
```

the *div-groups* would be (a, d, g), (b,c) and (e, f). The *div-group.word-ratio* and *td-group.word-ratio* features express the relative size of the group in number of words. To better distinguish between groups with noise (e.g. menus) and groups with text, only words not enclosed in `<a>` tags are considered.

## **4.3 Document-related features**

### **position**

This feature reflects a relative position of the block in the document (counted in blocks, not bytes). The rationale behind this feature is that parts close to the beginning and the end of documents usually contain noise.

### **document.word-count, document.sentence-count, document.block-count**

This feature represents the number of words, sentences and text blocks in the document.

### **document.max-div-group, document.max-td-group**

The maximum over all *div-group.word-ratio* and a maximum over all *td-group.word-ratio* features. This allows us to express “fragmentation” of the document – documents with a low value of one of these features are composed of small chunks of text (e.g. web bulletin boards).

# Chapter 5

## Data and Evaluation

For our development purposes we had 104 manually cleaned HTML documents available: 51 of them were provided by CLEANVAL and the rest was randomly selected, downloaded, and cleaned following the same guidelines by a volunteer. In this development data set, 22 501 blocks were identified and assigned appropriate labels. Their distribution is shown in the following table.

label	count
header	1 996
list	1 149
paragraph	3 419
continuation	3 380
total ( <i>content</i> )	9 944
other ( <i>noise</i> )	12 557

Table 5.1: Labels distribution in the development data set.

The data set was randomly split into six subsets and for better estimation of performance measures used in a six-fold cross-validation. Evaluation measures were of two types:

1. **labeling accuracy** – the ratio of correctly assigned block labels (1) from the full label set and (2) distinguishing only between *content* and *noisy* blocks;
2. **cleaning performance** – the official CLEANVAL scoring measures based on (3) edit distance and the extent to which the markup tags

indicate blocks of text starting and ending in the same place and (4) alignment of text alone, ignoring the markup tags plus (5) a combination of the latter two referred as the total score.

# Chapter 6

## Experiments and Results

First, we have performed a series of experiments where we disabled each of the 46 features one by one with the following observations:

The variance among the results on the six cross-validation subsets was relatively high in all experiments; the total score ranged from 66% to 80%. This is partially caused by the relatively small amount of training and test data in each run but it also proves the heterogeneous character of the task.

In contrast, the difference among the results of all experiments was relatively low. The total scores ranged from 71.9% to 74.5%. A possible explanation is a certain redundancy in the feature set.

Disabling some of the features slightly improved the results. The total score with all features enabled was 73.9%, while disabling the *document.-word-count* feature resulted in a score of 74.6%. Although this difference is probably not statistically significant we used this criterion and disabled some features for cleaning the evaluation data.

We also performed two other experiments: one with only the content-based features enabled and one with only the markup-based features enabled. Surprisingly, the results only dropped to 65.3% for markup-only and 66.5% for content-only features. This gives us an idea how much information is taken from these two types of features and how much we gain from their combination.

For the CLEANVAL evaluation, we have chosen three experimental settings: Exp-1 with all features enabled, Exp-2 with the features *word-ratio*, *numeric-count*, *mixed-count* and *nonword-count* disabled, and Exp-3 with two more additionally disabled features *regexp.url* and *document.word-count*. The cross-validated results on the development data set for these

experiments are displayed in Table 6.1.

	labeling accuracy (%)		cleaning performance (%)		
	full set	content-noise	markup-only	text-only	total
Exp-1	74.45	82.60	66.71	81.11	73.92
Exp-2	75.09	83.09	67.31	81.68	74.50
Exp-3	75.01	82.88	68.46	81.83	75.15

Table 6.1: Labeling accuracy and cleaning performance on the development data.



# Chapter 7

## Conclusions

We proposed a method for web page cleaning based on sequence labeling with Conditional Random Fields and presented a few initial experiments evaluated on the development data for CLEANVAL 2007. With very limited costs and manual work we were able to achieve encouraging results<sup>1</sup>. Using supervised training methods seems as a reasonable approach and we believe that a better set of features and a larger collection of training data can bring additional performance improvement. Nevertheless, we are aware of some weaknesses of our system:

- There is no natural language detection. Therefore, it is also very easy to “trick” *Victor* into accepting something that is no valid text at all – any longer sequence of random characters with enough spaces and full stops enclosed in the `<p>` element is very likely to be labeled as a paragraph. While we agree that a rudimentary natural language detection is necessary in order not to pollute the corpus with texts in foreign languages, it could be easily implemented outside of the system. Recognizing intentionally mangled content would be harder. Theoretically, some spam detection software could be used for that.
- Currently, *Victor* does not perform very well on pages where valid text is split up into small pieces, such as bulletin boards. While the `document.max-div-group` and `document.max-td-group` features might help recognizing such pages, the actual cleaning performance on such

---

<sup>1</sup>Unfortunately, we have not received results from the CLEANVAL shared task as of writing this paper. Therefore, our conclusions are backed up only by the results of the cross-validation experiments.

pages is still far from being optimal. Depending on the task however, it could be acceptable discard pages that are known to cause problems, preferring quality of the cleaned text over quantity.

## 7.1 Future Ideas

Below are some ideas for additional features and other improvements.

### **Extract features from CSS style rules**

Currently, the only meta data *Victor* uses is the HTML tree. A challenging task would be to implement parsing CSS style rules and extract features such as font sizes or even positioning. We believe that this would improve performance on many web pages that use `<span>` or similar tags with style rules instead of “semantic” tags.

### **Recognize common Content Management Systems**

It should not be difficult to recognize popular Content Management Systems (CMS), such as Drupal or MediaWiki. This would mainly allow to hint the CRF algorithm by marking blocks that are part of the main content and are therefore more likely to contain valid text.

# Bibliography

- [1] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, 2002.
- [2] Liang Chen, Shaozhi Ye, and Xing Li. Template detection for large scale search engines. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, Dijon, France, 2006.
- [3] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1), 1999.
- [4] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, USA, 2001.
- [5] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Intelliclean: a knowledge-based intelligent data cleaner. In *Knowledge Discovery and Data Mining*, 2000.
- [6] Shian-Hua Lin and Jan-Ming Ho. Discovering informative content blocks from web documents. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, Edmonton, Alberta, Canada, 2002.
- [7] Lan Yi and Bing Liu. Web page cleaning for web mining through feature weighting. In *Proceedings of Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, 2003.

- [8] Lan Yi, Bing Liu, and Xiaoli Li. Eliminating noisy information in web pages for data mining. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, Washington, DC, USA, 2003.