CS-Web: A Lightweight Summarizer for HTML

Tony Rose and Ave Wrigley
Canon Research Centre Europe Ltd
tgr@tonyrose.net, ave.wrigley@itn.com

1. LESS IS MORE: A LIGHTWEIGHT SUMMARIZER FOR HTML

Canon, like many other large companies, is a multi-national organisation with several (26) individual web sites. Many of these web sites were set up in the early nineties as a result of the independent initiative of different parts of the company. Consequently, as each web site grew, it tended to carry information that was specific to its host organisation. That may be fine for local users, who only need to access that particular site and know that the information they seek will be on that site (somewhere).

But what of the typical Canon customer, who cares nothing for Canon's internal organisation, but simply wants to find information on the SLR camera range or download a new printer driver? For them, instead of a single, clear way into Canon's web space, there was a bewildering array of individual sites, each carrying separate information. Clearly, something had to be done. And in 1997, it was.

CS-Web: a search engine for Canon's web space

What the company needed was a search engine that could collect information from all of Canon's web sites, index it, and then present it via a single, consistent access point to the user. At that time, there seemed to be two choices of available technology: public web services (such as Alta Vista) that were designed for searching the entire web, or ”off the shelf”, bespoke solutions that were designed for searching a given site, hosted on a single machine. However, there wasn't a product or service for a related set of web sites, like Canon's. So, we had to roll our own (in Perl, of course!)

The result was CS-Web. CS-Web consists of a robot that traverses all of Canon's web sites, collecting information about each page, and a web interface that can be used to search these pages in various ways. In between is a database in which all of the URL information is stored. You can try CS-Web for yourself: it is linked from the main “gateway” page for Canon (http://www.canon.com/). All the main components of CS-Web are written in Perl. The web interface itself is generated using mod_perl. CS-Web presented a variety of challenges, many of which would make a suitable “war story” for TPJ! However, for this this article, we will focus on one particular problem that was crucial to the performance of CS-Web (but is generally applicable elsewhere too): the production of text summaries from HTML documents.

HTML META tags

One of the pieces of information that the CS-Web robot looks for in its traversal of Canon's web sites is a short, textual description of each of the pages. These descriptions are often provided by the page authors as metadata, using HTML META tags. META tags use 'NAME' and 'CONTENT' attributes to specify different types of information. Although there

is currently no official standard for the naming of these META tags, there are existing conventions for things like “keywords” and “description” data (see http://searchenginewatch.internet.com/webmasters/meta.html).

CS-Web does not use full-text search (i.e. scan the entire text of each HTML page for matching words). Instead, it looks for matches in the metadata (which in this case are the page descriptions, titles and keywords). Consequently, these page descriptions have become very important: not only are they one of the main database entries that CS-Web searches in the matching process, but they are also displayed as part of the listing for each of the “hits” that match a given query. One further constraint is that these descriptions have to be of fixed length. This is not only because search engine results need to present brief summaries; in CS-Web, each URL takes up one row in the SQL database table, and the size of each field in each row is fixed in advance. It is possible to have variable width fields in some SQL databases, but only at a performance cost that would have been unacceptable in this application. In fact, looking at other search engines it looks very much as if everyone has made the same decision.

One immediate problem was that (in general) very few page authors consistently provide accurate metadata. Moreover, some page authors often provide “bogus” metadata in an attempt to “outsmart” the public search engines and get a better ranking for certain search queries. As a result, the descriptions they provide can often be bizarre or just plain irrelevant. Consequently, many public search engines now completely ignore META tags for precisely this reason. However, the outlook for CS-Web was a little more promising. Since Canon's webmasters are generally working together, we could expect a certain level of trust, and assume that they were not trying to deceive the CS-Web robot. In turn, the CS-Web robot could acknowledge this trust: if it found a page description within a META tag, it would accept it as being legitimate.

However, there were still a great many pages in Canon's webspace that did not provide this metadata. So, in such cases, where should the description come from? The only answer was to generate a text description directly from the raw HTML. And how could we do that? By using some natural language processing techniques, and Perl. The result was HTML::Summary.

HTML::Summary

HTML::Summary is available from CPAN (http://search.cpan.org/~tgrose/HTML-Summary-0.017/), and is currently at version 0.017. It has a fairly simple interface. First, you create an HTML::Summary object, using the `new` method. `new` can take a number of configuration parameters, that are passed as a hash; e.g.:

```
my $html_summarizer = new HTML::Summary LENGTH => 200;
```

The LENGTH parameter here is the maximum length in bytes for the generated summary. Next, you need an HTML::Element object which corresponds to the HTML page that you want to generate the summary for; e.g. one generated by HTML::TreeBuilder:

```
my $html_tree = new HTML::TreeBuilder;
$html_tree->parse( $html_document );
```

In this case `$html_document` is a scalar containing an HTML string; this could have been read in from a file, or returned as the contents of an HTTP request, for example. Finally, you call the generate method of the HTML::Summary object, with the HTML::Element object as an argument, which returns the summary of the page as a string:

```
$html_summary = $html_summarizer->generate( $html_tree );
```

The Summarization Algorithm

One of the main problems we faced in writing HTML::Summary was finding a strategy for generating a good abstract of a fixed length from arbitrary text. This is known to be an important and difficult problem, and a real quality solution requires sophisticated natural language techniques which can analyse the structure of the original, identify key phrases and concepts, and regenerate these in a more succinct format.

Fortunately for us, there is a range of techniques for doing this kind of thing, some of which are more "quick and dirty" than others. Bearing in mind the application, the quality of the summaries generated needs only be sufficient to give a good gist of the content of the original to the user of CS-Web browsing search results. In addition, for retrieval purposes, the more likely the summary is to contain important keywords related to the page, the better.

One advantage that we had over people trying to generate summaries from plain text, which is the more usual case, is that HTML pages contain markup that can give strong clues to the structure of the content, and also to its relative importance. For example, it is usually clear in HTML pages where paragraphs begin and end. It is also usually clear when important text is italicised, emboldened, or made into a heading.

The HTML::Summary module uses a technique known as the location method of text summarization. Basically, this consists of identifying important sentences, based primarily on their location in the text, and concatenating them together to produce an abstract. A simple example of this would be to take the first sentence of every paragraph in an article and string them together. This can sometimes be surprisingly effective; here is an example:

> "Canon, like many other large companies, is a multi-national organisation with several (26) individual web sites. What the company needed was a search engine that could collect information from all of Canon's web sites, index it, and then present it via a single, consistent access point to the user. The result was CS-Web. CS-Web presented a variety of challenges, many of which would make a suitable "war story" for TPJ!"

The text summarization method used in HTML::Summary is an adaptation of the location method. Basically, the way it works is as follows:

1. **split into sentences**

First of all the text is split into sentences. More about this later.

2. **score the sentences**

The sentences are scored according to the element that they appear in, and whether or not they are the first sentence in that element. The algorithm here is pretty simple: each element has a score; the first sentence in that element gets this score; the rest of the sentences get nothing.

3. **sort the sentences by score**

The sentences are stored in an array of hashes. Each hash corresponds to a sentence, and contains information about the text in the sentence, its length, the HTML element it appeared in, the score given to it, and its original order in the text.

```
$summary[ scalar( @summary ) ] = {
'text' => $text,
'length' => length( $text ),
'tag' => $tag,
'score' => $score,
'order' => scalar( @summary ),
};
```

The scores, as described above, are based on the HTML element that the sentences appear in. These score are stored in a global hash:

```
my %ELEMENT_SCORES = (
'p' => 100,
'h1' => 90,
'h2' => 80,
'h3' => 70,
);
```

These scores were arrived at by empirical investigation; there is no real theoretical justification for them!

4. **truncate the list of sentences**

Calculate how many sentences just take you over the requested summary length.

5. **Sort the sentences by original order again**

Having remembered the original sentence order in the text in the hash for that sentence, you can now re-sort the sentences in that order.

6. **concatenate the sentences to create the summary**

Spacing between sentences needs to be added here, because this is stripped in the sentence splitting process (see Sentence Splitting).

7. **truncate the summary at the requested length**

This last step assumes that if you say you want a summary of 200 characters, that is what you want, even if this means chopping the summary off mid-sentence. This is what we wanted in CS-Web. Maybe in other applications a less severe approach would be

appropriate - it would be easy to add more options to HTML::Summary, so let us know what you think.

Sentence Splitting

OK, now for the nitty gritty. The remainder of the article will focus on one aspect of the HTML::Summary code: splitting the element contents into sentences. Japanese character encodings, which were a particular problem for CS-Web, are dealt with in a sidebar (Truncating Japanese Text).

The task of splitting text into sentences seemed like a more general problem than its application to text summarization, so this is contained in a separate module: Text::Sentence. For the moment, this is distributed as part of the HTML::Summary package, but we would be interested to hear if there is any interest in using this module independently.

Text::Sentence is basically just a regex. It is has a non-object oriented interface that exports one function, `split_sentences`, that takes the text to be split into sentences as an argument, and returns a list of the sentences.

```
sub split_sentences
{
my $text = shift;
return () unless $text;
```

The function first checks if there really is any text to split into sentences; if not, it just returns the empty string.

```
# capital letter is a character set; to account for locale, this
# includes all characters for which lc is different from that character

my $capital_letter =
'[' .
join( '',
grep { lc( $_ ) ne ( $_ ) }
map { chr( $_ ) } ord( "A" ) .. ord( "\xff" )
) .
']'
;
```

Although it would be more efficient to compute this regex component once at the package level, doing it in `split_sentences` allows the user to change locales between calls.

The next few lines start to build up the components of the regex that will be used to split the text into sentences. The first of these components is the capital letter that is found at the start of a sentence. Instead of using the character class [A-Z] as you might normally do, Text::Sentence tries to account for locale specific capital letters. For example, in French, a capital A acute (Á) will not be matched by [A-Z]. The method used in Text::Sentence makes use of the fact that the lc builtin is sensitive to locale settings, and will return a lowercase version of all capitalized characters. For more information on how Perl handles locales, see the perllocale documentation.

```
@PUNCTUATION = ( '\.', '\!', '\?' );
```

The @PUNCTUATION array is a global package variable in Text::Sentence which contains any punctuation that can be used to indicate the end of a sentence. The fact that it is a global means that if you want to change the set of punctuation characters you can. You might, for example, want to add locale specific punctuation for Spanish ``¡'':

```
my $html_summarizer = new HTML::Summary LENGTH => 200;
push( @HTML::Summary::PUNCTUATION, chr( 161 ) );
```

This functionality is not currently documented for the module, but may be in future versions.

```
# this needs to be alternation, not character class, because of
# multibyte characters
my $punctuation = '(?:' . join( '|', @PUNCTUATION ) . ')';
```

As mentioned above, one of the concerns with CS-Web is dealing with multibyte character encodings (see the sidebar Truncating Japanese Text). Japanese punctuation characters may be more than one character long - for example, an exclamation mark in EUC would be "\xA1\xAA".

```
# return $text if there is no punctuation ...
return $text unless $text =~ /$punctuation/;
```

If there isn't any sentence final punctuation in the text, then you might as well return the text now.

```
my $opt_start_quote = q/['"]?/;
my $opt_close_quote = q/['"]?/;

# these are distinguished because (eventually!) I would like to do
# locale stuff on quote characters

my $opt_start_bracket = q/[[({]?/; # }{
my $opt_close_bracket = q/[\])}]?/;
```

Sentences sometimes have quotation marks or parentheses which come before the capital letter at the beginning, or after the full stop, etc. at the end. For example, the following sentence:

```
Larry said "let there by light!" (And there was.)
```

is strictly speaking two sentences (the first sentence ends after the second double quote). However:

```
Larry said "let there by light!" (and there was).
```

would be one sentence. Finally, the regex itself, built up from these components:

```
my @sentences = $text =~ /
(
# sentences start with ...
$opt_start_quote # an optional start quote
$opt_start_bracket # an optional start bracket
```

```
$capital_letter # a capital letter ...
.+? # at least some (non-greedy) anything ...
$punctuation # ... followed by any one of !?.
$opt_close_quote # an optional close quote
$opt_close_bracket # and an optional close bracket
)
(?= # with lookahead that it is followed by ...
(?: # either ...
\s+ # some whitespace ...
$opt_start_quote # an optional start quote
$opt_start_bracket # an optional start bracket
$capital_letter # an uppercase word character (for locale
# sensitive matching)
| # or ...
\n\n # a couple (or more) of CRs (i.e. a new para)
| # or ...
\s*$ # optional whitespace, followed by end of string
)
)
/gxs
;
return @sentences if @sentences;
return ( $text );
}
```

This regex makes use of the lookahead feature in regular expresssions introduced in Perl5. In this case, it allows us to specify that a sentence must not only start with a capital letter, and end in a full stop (or question mark, etc.), but that there must be another capital letter which follows the full stop, exclamation mark, etc. The only exception to this is the case where the sentence is either at the end of a paragraph, or the last non-whitespace text.

Incidentally, the fact that the whitespace between sentences is accounted for in the lookahead means that it is not part of the matched patterns that end up in the `@sentences` array. Because of this, concatenating the sentences does not necessarily give you back the original text.

The main problem with trying to split text into sentences is that there are several uses for full stops. The most common and problematic is in abbreviations. The following example:

```
Dr. Livingstone, I presume.
```

would unfortunately count as two sentences according to Text::Sentence - the first sentence ending after the first 3 characters. The performance of Text::Sentence could be improved by taking into account some common special cases; honorifics (Mr., Mrs., Dr.), common abbreviations (e.g., etc., i.e.), and so on. However, as with many natural language problems, this obeys the law of diminishing returns; a little bit of effort will do a decent 90% job, but that last 10% starts to get real hard! Luckily, again, for our purposes, as with the text summarization, 90% is good enough.

Conclusion

We chose to use Perl for CS-Web mainly because of its more obvious benefits for this kind of project: the LWP modules for web programming, DBD/DBI, mod_perl, and so on. However, we found that Perl is also a very useful tool for doing natural language work. Its

text processing ability, rapid development cycle, and ability to generate complex data structures on the fly make it particularly appropriate in this field.

A lot of interesting work in natural language research involves analysis of corpus data; collecting statistics about language use over large databases of typical usage. The web is an obvious rich source of this type of data, and in view of this, it is a little surprising how few tools and modules appeared to be available in Perl for this field. Certainly, when we were working on the Text::Sentence regex, we posted something to a language processing mailing list, and there seemed to be quite a lot of interest in what we were doing, as well as extensive Perl expertise in that community. Hopefully natural language processing will become yet another nut for Perl to crack in the near future!

2. SIDEBARS

Truncating Japanese Text

Canon is a Japanese company, so dealing with Japanese text in its web pages was an important issue. Japanese text is usually encoded in one of several possible multibyte encoding schemes. At least some of these schemes use variable numbers of bytes to represent single Japanese characters, or allow Japanese and ascii characters to be intermingled. This presented us with a serious problem.

The summaries generated by Text::Summary are truncated at a fixed length, and this length is specified in bytes, rather than characters. If Japanese text is truncated at an arbitrary byte length, this may mean that it is truncated in the middle of a multi-byte character.

This would be bad enough, if it just affected the truncated text itself. In CS-Web page abstracts can appear in result listings for keyword searches. If a “broken” page summary is inserted into running text, the byte immediately following the summary may be interpreted as the next byte of the uncompleted Japanese character at the end of the broken summary. This could seriously impair the rendering of the remaining text.

The Text::Sentence includes another supporting module, Lingua::JA::Jtruncate which addresses this problem. Lingua::JA::Jtruncate contains just one function; `jtruncate`, which is used as follows:

```
use Lingua::JA::Jtruncate qw( jtruncate );
$truncated_jtext = jtruncate( $jtext, $length );
```

where `$jtext` is some Japanese text that you want to truncate, $length is that maximum length that the text needs to be truncated to, and `$truncated_text` is the result of truncating the text. Here's how it works.

First of all, some regexes are defined that match characters in each of the three main Japanese coding schemes; EUC, Shift-JIS, and JIS.

```
%euc_code_set = (
ASCII_JIS_ROMAN => '[\x00-\x7f]',
JIS_X_0208_1997 => '[\xa1-\xfe][\xa1-\xfe]',
HALF_WIDTH_KATAKANA => '\x8e[\xa0-\xdf]',
```

```
JIS_X_0212_1990 => '\x8f[\xa1-\xfe][\xa1-\xfe]',
);
%sjis_code_set = (
ASCII_JIS_ROMAN => '[\x21-\x7e]',
HALF_WIDTH_KATAKANA => '[\xa1-\xdf]',
TWO_BYTE_CHAR => '[\x81-\x9f\xe0-\xef][\x40-\x7e\x80-\xfc]',
);
%jis_code_set = (
TWO_BYTE_ESC =>
'(?:' .
join( '|',
'\x1b\x24\x40',
'\x1b\x24\x42',
'\x1b\x26\x40\x1b\x24\x42',
'\x1b\x24\x28\x44',
) .
')'
,
TWO_BYTE_CHAR => '(?:[\x21-\x7e][\x21-\x7e])',
ONE_BYTE_ESC => '(?:\x1b\x28[\x4a\x48\x42\x49])',
ONE_BYTE_CHAR =>
'(?:' .
join( '|',
'[\x21-\x5f]', # JIS7 Half width katakana
'\x0f[\xa1-\xdf]*\x0e', # JIS8 Half width katakana
'[\x21-\x7e]', # ASCII / JIS-Roman
) .
')'
);
%char_re = (
'euc' => '(?:' . join( '|', values %euc_code_set ) . ')',
'sjis' => '(?:' . join( '|', values %sjis_code_set ) . ')',
'jis' => '(?:' . join( '|', values %jis_code_set ) . ')',
);
```

Each of the regexes in the hash `%char_re` should match one character encoded in the scheme corresponding to the keys of the hash.

Now for the definition of the `jtruncate` function; first, some fairly obvious sanity checks:

```
sub jtruncate
{
my $text = shift;
my $length = shift;
# sanity checks
return '' if $length == 0;
return undef if not defined $length;
return undef if $length < 0;
return $text if length( $text ) <= $length;
```

Now save the original text; this is used later, if the truncation process fails for some reason.

```
my $orig_text = $text;
```

Now use Linga::JA::Jcode::getcode to detect which code the text is encoded in. Linga::JA::Jcode::getcode is a simple wrapper around the jcode.pl Perl library for

Japanese character code conversion by Kazumasa Utashiro <utashiro@iij.ad.jp>, which he kindly agreed to let us distribute with HTML::Summary.

```
my $encoding = Lingua::JA::Jcode::getcode( \$text );
```

If getcode returns undef, or a value other than euc, sjis, or jis, then it has either failed to detect the encoding, or detected that it is not one of those that we are interested in, so we take the brute force approach, and use `substr`.

```
if ( not defined $encoding or $encoding !~ /^(?:euc|s?jis)$/ )
{
return substr( $text, 0, $length );
}
```

The actual truncation of the string is done in `chop_jchars` - more about this later.

```
$text = chop_jchars( $text, $length, $encoding );
```

`chop_jchars` returns `undef` on failure. If we have failed to truncate the Japanese text properly we need to resort to `substr` again; here the decision is over whether it is more important to come in under the `$length` constrain, or risk returning a Japanese string with broken character encoding, and we plump for the former.

```
return substr( $orig_text, 0, $length ) unless defined $text;
```

Next a special case: JIS encoding uses escape sequences to shift in and out of single-byte / multi-byte modes. If the truncation process leaves the text ending in multi-byte mode, we need to add the single-byte escape sequence. Therefore, we truncate (at least) 3 more bytes from JIS encoded string, so we have room to add the single-byte escape sequence without going over the $length limit.

```
if ( $encoding eq 'jis' and $text =~ /$jis_code_set{
TWO_BYTE_CHAR }$/ )
{
$text = chop_jchars( $text, $length - 3, $encoding );
return substr( $orig_text, 0, $length ) unless defined $text;
$text .= "\x1b\x28\x42";
}
```

And we're done!

```
return $text;
}
```

Now for `chop_jchars`; this highly sophisticated technique simply lops off Japanese characters from the end of the string until it is shorter than the requested length. OK, it is pretty ugly, and slow for large strings truncated to small values, but it does the job!

```
sub chop_jchars
{
my $text = shift;
my $length = shift;
my $encoding = shift;
while( length( $text ) > $length )
```

```
{
return undef unless $text =~ s!$char_re{ $encoding }$!!o;
}
return $text;
}
```

Basic Summarization Methods

The basic approach of most simple summarisation systems is to examine each sentence in the original document, assess its importance (using one or more known heuristics) and then output an summary of the desired length, by omitting the less important sentences. Clearly, the success of this approach relies on the accuracy of the methods by which importance is measured. Usually, one (or more) of the following six simple methods is applied (Paice, 1990):

The location method

Sentences are scored according to their position or location within the document. For example, sentences occurring at the beginning or end of the first paragraph, or within a heading may be given a higher score than sentences in the middle of a paragraph.

The cue method

This method is based on the notion that certain words in the document indicate the presence of more (or less) important material. For example, strongly positive words like “best”, “significant” or “greatest” would tend to increase the sentence score. By contrast, negative words like “impossible” or “hardly” would tend to decrease the sentence score.

The title-keyword method

The title of the document is assumed to be a reliable indication of the focus of its contents, so sentences that refer to the same concepts as those found in the title are given a higher score. This process, like any other that uses such lexical data, may be assisted by the application of various lexical pre-processes. For example a stemmer may be used to conflate inflected terms to a single root, e.g. “runs” and “running” become “run”. Similarly, a stop list may be used to filter out stop words (i.e. function words, such as “the”, “of”, “and”, etc.)

The frequency-keyword approach

The idea behind this approach is that the important concepts in a document will be represented by certain keywords that occur with a higher than expected frequency. Therefore, sentences that contain these words will be given a higher score. The keywords are usually identified by taking a sorted word-frequency list and removing stop words. A slightly more sophisticated variant on this involves the use of 'distinctiveness' rather than raw frequency, i.e. to normalise the frequency counts by a-priori frequency counts taken from an independent large text corpus.

The indicator phrase method

This method is similar to 2. The cue method except that in this case one looks for certain phrases rather than words. For example, “The aim of this paper is ...” and “This document attempts to review ...” both tend to indicate that some important concept is about to be introduced. Consequently, sentences containing such constructions should receive higher scores. Evidently, there are many different types of indicator phrase, but research has

shown that these may be derived from a smaller number of underlying templates (Paice, 1990).

The syntactic method

Experiments from as far back as the 1970's (e.g. Earl, 1970) have attempted to correlate sentence importance with syntactic structure, but so far without conclusive results. Evidently, the above six methods differ in their complexity and performance. Edmundson (1969) performed a comparative evaluation, and found methods 1 - 4 to be superior, with the order of decreasing performance corresponding to the order shown above. In addition, he evaluated their performance in combination, and found a linear combination of the first three to be optimal (with an appropriate weighting given to the scores obtained from each method).

3. REFERENCES

1. Earl (1970) Earl, L.L. “Experiments in automatic extracting and indexing”. Information Storage and Retrieval, 6, pp 313-334, 1970.

2. Edmundson (1969) Edmundson, H. P. “New methods in automatic extracting”. Journal of the ACM, 16(2):264-285, 1969.

3. Paice (1990) Paice, C. “Constructing literature abstracts by computer”, Information Processing and Management, Vol. 26(1), 1990.