My tech blog.

Technologies I'm passionate about.

Evaluating Text Extraction Algorithms

Posted on June 9, 2011 by tomaz

***UPDATE 11/6/2011:** Added the summary and the results table*

Lately I've been working on evaluating and comparing algorithms, capable of extractinguseful content from arbitrary html documents. Before continuing I encourage you to pass trough some of my previous posts, just to get a better feel of what we're dealing with; I've written a short overview, compiled a list of resources if you want to dig deeper and made a feature wise comparison of related software and APIs.

Summary

- The evaluation environment presented in this post consists of 2 datasets with approximately 650 html documents.
- The gold standard of both datasets was produced by human annotators.
- 14 different algorithms were evaluated in terms of precision, recall and F1 score.
- The results have show that the best opensource solution is the boilerpipe library.
- Commercial APIs included in the evaluation environment produced consistent results on both datasets. Diffbot and Repustate API performed best, while others follow very closely.
- Readability's performance is surprisingly poor and lacking consistency between both ports that were put to use in the evaluation setup.
- Further work will include: adding more APIs and libraries to the setup, working on a new extraction technique and assembling a new dataset.

Evaluation Data

My evaluation setup consists of 2 fairly well known datasets:

- The final **cleaneval** evaluation dataset, with 681 documents (created by the ACL web-as-corpus community).
- **Google news** dataset, with 621 documents. (harvested by the authors of the boilerpipe library)

The former was harvested from all sorts of web pages, including: articles, blog posts, mailing list archives, forum conversations, dedicated form submission pages etc. The latter was gathered by scraping the google news stream for a longer period of time (cca 6 months). The random sample that became the final dataset of 621 documents from 408 different news type web sites, came from a larger set of 250k documents assembled during the scrapping process.

From their description and related documentation we can conclude that they represent 2
slightly distinctive domains: the google news dataset represents a specific domain of news articles and the

cleaneval dataset represents a cross domain collection of documents.

The gold standard counterpart of both datasets was created by human annotators who assessed each document individually. Annotators who assembled the cleaneval gold standard, first used lynx text based browser to filter out all the non-text building blocks. Each annotator than cleaned out the redundant non-content text using his own visual interpretation of the website. On the other hand, annotators of the google news dataset inserted additional span tags (using class names as labels) into the original html document. The class names of span tags indicate the following content labels: headline, main text, supplemental text, user comments, related content. In my experimental setup I use only the content labeled under *headline* and *main text* as the gold standard.

As you might have noticed, these datasets are not as large as we would like them to be and there is a fairly good reason behind that: assessing and labeling html documents for content is a tedious task. So was the preprocessing of the cleaneval dataset on my part. The methodology of storing raw html documents for the cleaneval shared task included inserting a special <text> tag around the whole html markup to hold some meta data about the document itself. I had to remove these tags from all documents and insert the basic <html> and <body> tags to those who were obviously stripped of some starting and trailing boilerplate html markup. The reasoning of inserting such tags back into the raw html document is based on the nature of particular extraction software implementations which assume the existence of such tags.

Metrics

I'm using *precision*, *recall* and *f1 score* to evaluate the performance. I covered these in one of my previous posts, but here is a short recap:

Definitions:

$$precision = \frac{|S_{rel} \cap S_{ret}|}{|S_{ret}|} \; recall = \frac{|S_{rel} \cap S_{ret}|}{|S_{rel}|} \; F_1 score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Given an HTML document we predict/extract/**retrieve** a sequence of words and denote them as S_{ret}. Consequentially the sequence of words representing **relevant** text is denoted as S_{rel}.

Both sequences are constructed by the following procedure:

1. Remove any sort of remaining inline html tags
2. Remove all punctuation characters
3. Remove all control characters
4. Remove all non ascii characters (due to unreliable information of the document encoding)
5. Normalize to lowercase
6. Split on whitespace

The intersection of retrieved and the relevant sequence of words is calculated using the Ratcliff-Obershelff algorithm (python difflib).

As we're dealing with raw data, we have to account for 4 special cases of these metrics:

Precision	**Recall**	**F1-score**	**Case**
0	0	inf	Missmatch – both retrieved and relevant sequences are **not** empty, but they don't intersect.
inf	0	nan	Set of retrieved words is empty – the extractor predicts that the observed document does not contain any content.
0	inf	nan	Set of relevant words is empty – the document itself does not contain anything useful.
inf	inf	nan	Both retrieved and relevant set is empty – the document does not contain nothing useful and the extractor comes back empty handed.

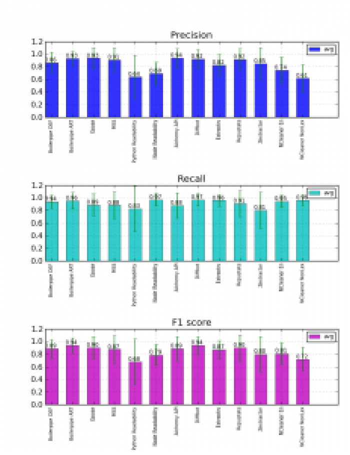
Results

Currently my evaluation setup includes the following content extractors that were either already reviewed or mentioned in my preceding blog posts, so I won't get into the details of every single one in the context of this writeup:

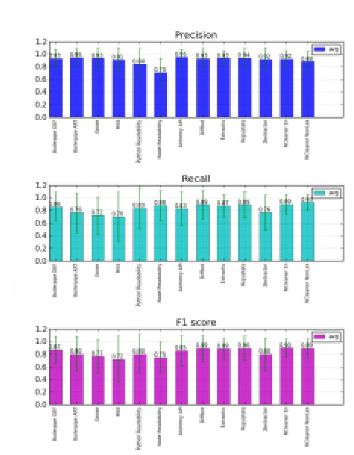
- Boilerpipe – using two similar variants; the **default** extractor and the one especially tuned for **article** type of content.
- NCleaner – again using two of its variants; the **non lexical** n-gram character language independat model and the model trained on **english** corpora.
- Readability – using its python port and readability ported to node.js on top of jsdom.
- Pasternack & Roth algorithm (MSS) – authors provided me with access to the implementation presented in the www2009 paper.
- Goose – using my fork to expose the core content extraction functionality.
- Zextractor – internally used service developed by my friends/mentors at Zemanta.
- Alchemy API – using their *text extraction API.*
- Extractiv – using their semantic *on demand RESTul API.*
- Repustate API – using the *clean html API call.*
- Diffbot – using the *article API.*

Admittedly they are not all specialized for solving problems in a specific domain, say news article content extraction, but they all share one common property: they're capable of distinguishing useful textual content apart from boilerplate and useless content when observing an arbitrary html document.

Results were obtained by calculating precision and recall for each document in the dataset for every content extractor. Then we calculate the arithmetic mean of all three metrics for every extractor. Bar charts are employed to visualize the results.



Precision, recall & F1 score mean - Google news dataset

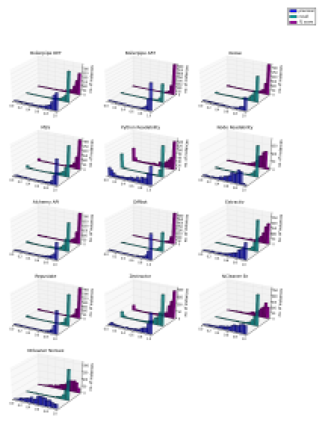


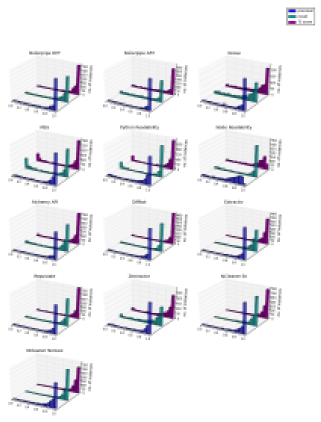
Precision, recall & F1 score mean - Cleaneval dataset

Cleaneval→	*precision*	*recall*	*f1 score*
Boilerpipe DEF	0.931	0.856	0.872
Boilerpipe ART	0.949	0.764	0.804
Goose	0.934	0.719	0.770
MSS	0.911	0.699	0.718
Python Readability	0.841	0.833	0.803
Node Readability	0.704	0.878	0.753
Alchemy API	0.950	0.828	0.854
Diffbot	0.932	0.890	0.891
Extractiv	0.935	0.871	0.887
Repustate	0.940	0.889	0.896
Zextractor	0.916	0.763	0.800
NCleaner En	0.923	0.895	0.897
NCleaner NonLex	0.882	0.927	0.892

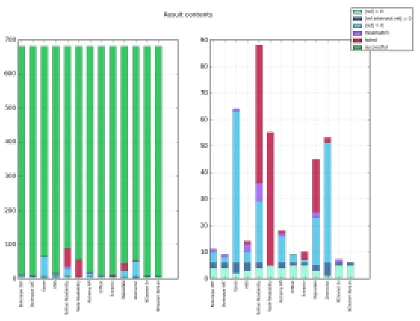
Google news→	*precision*	*recall*	*f1 score*
Boilerpipe DEF	0.863	0.938	0.887
Boilerpipe ART	0.931	0.955	0.939
Goose	0.934	0.887	0.901
MSS	0.907	0.882	0.875
Python Readability	0.638	0.825	0.681
Node Readability	0.688	0.968	0.789
Alchemy API	0.936	0.876	0.888
Diffbot	0.924	0.968	0.935
Extractiv	0.824	0.956	0.870
Repustate	0.917	0.907	0.898
Zextractor	0.850	0.806	0.803
NCleaner En	0.742	0.947	0.812
NCleaner NonLex	0.613	0.959	0.723

Next we explore the distribution of per document measurements. Instead of just calculating the mean of precision, recall and F1 score we can explore the frequencies of per document measurements. In the following chart we visualize frequencies for each metric for all extractors. Notice that we're splitting the [0,1] interval of each metric into 20 bins with equidistant intervals.

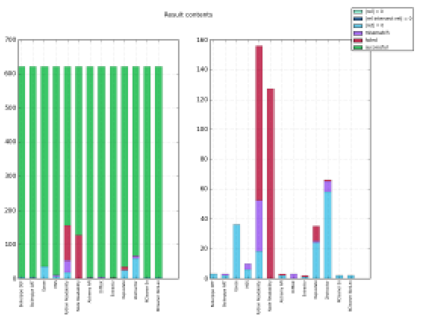

Metric distributions - Google news dataset


Metric distributions - Cleaneval dataset

To account for previously mentioned 4 special cases, we employ a stacked bar chart to inspect the margins of useful/successful per document measurements for every dataset.



Special cases of per document measurements - Cleaneval dataset



Special cases of per document measurements - Google news dataset

Every per document measurement can fall into one of the "special cases" category, "successful" category, or an extra "failed" category. The latter stands for measurement instances that failed due to: parsing errors, implementation specific errors, unsupported language errors etc. The right hand side of both figures depicts the same categories without the "successful" part.

This chart is important, because we only make use of measurements who fall under the "successful" category to obtain the distribution and mean for each metric, respectively.

Observations

The foremost observation is the varying performance of NCleaner outside the cleaneval domain, since both variants seem to perform poorly on the google news collection. The cause of such behavior might be the likelihood that NCleaner was trained on the cleaneval corpus.

Readability's poor performance came as a surprise, moreover, its varying results between both ports. Relatively low precision and high recall indicate that readability tends to include large portions of useless text in its output. I'm not quite satisfied of how readability is represented in this experiment as I'm not making use of the original implementation. The node.js port seems not to differ from the original as much as the python port, but I still worry I'll have to use the original implementation on top of a headless webkit browser engine like PhantomJS, to get a better representation.

Notice the consistent performance of commercial APIs (and zemanta's internal service), Alchemy API and Diffbot in particular. According to diffbot's co-founder Michael Tung, their article API is only a small portion of their technology stack. They're using common visual features of websites to train models in order to gain the understanding of various types of web pages (forum threads, photo galleries, contact info pages and about 50 other types). The article API used in this setup is just an additionally exposed model, trained on visual features of article type pages. On the other hand, zextractor (zemanta's internal service) leverages on libsvm to classify atomic text blocks of the newly observed document.

What I find especially interesting are the bimodal distributions of precision and/or recall for MSS and readability's python port. I suspect that they're failing to produce relevant content on really big documents or they're capable of extracting only a limited amount of content. It'll be interesting to explore this phenomena with some additional result visualizations.

Conclusion and Further Work

According to my evaluation setup and personal experience, the best open source solution currently available on the market is the boilerpipe library. If we treat precision with an equal amount of importance as recall (reflected in the F1 score) and take into account the performance consistency across both domains, then boilerpipe performs best. Performance aside, its codebase is seems to be quite stable and it works really fast.

If you're looking for an API and go by the same criteria (equal importance of precision and recall, performance consistency across both domains), diffbot seems to perform best, although alchemy, repustate and extractiv are following closely. If speed plays a greater role in your decision making process; Alchemy API seems to be a fairly good choice, since its response time could be measured in tenths of a second, while others rarely finish under a second .

My further work will be wrapped around adding new software to my evaluation environment (I was recently introduced to justext tool), compiling a new (larger) dataset and working on my own project, that'll hopefully yield some competative results.

Stay tuned for I'll be releasing all the data and code used in this evaluation setup in the near future.

Posted in text extraction | Tagged evaluation, information retrieval, text extraction | 24 Comments

Feature-wise Comparison of HTML Article Text Extractors

Posted on April 19, 2011 by tomaz

In one of my previous posts I compiled quite a decent list of software (and other resources) all capable of extracting article content from an arbitrary HTML document. While I was gathering all the relevant papers and software I kept updating a handwritten spreadsheet that compares the listed software from a feature-wise viewpoint. So I updated it and decided to dump it on my blog. Hopefully this comparison table will mitigate the decision making process of developers whose products are dependent on such software.

Firstly; let's review some shared functionality features explored for each piece of software in the table:

- structure retainment – Articles are usually formatted using various html tags – paragraphs, lists hyperlinks and ohers. Some text extractors tend to remove such structure and yield only the plain text of the article.
- inner content cleaning – The article content is sometimes broken into non-consecutive text blocks by ads and other boilerplate structures. We're interested in capabilities to remove such inline boilerplate.
- implementation
- language dependency – Some are limited to only one language.
- source parameter – Can we fetch the document by ourselves or does the extractor fetch it internally?
- additional features (and remarks)

	structure retainment	**inner content cleaning**	**implementation**	**source parameter**	**language dependancy**	**additional features and remarks**
Boilerpipe	plain text only	uses a classifier to determine whether or not the atomic text block holds useful content	open source java library	you can fetch documents by yourself or use built-in utilities to fetch them for you	should be language independent since the text block classifier observes language independent text features	implements many extractors with different classification rules trained on different datasets
Alchemy API	text only (has an option to include relevant hyperlinks)	n/a	commercial web api	include the whole document in the post request or provide an url	*observation:* returns an error for non-english content e.g. the document contains "unsupported text language"	extra API call to extract the title
Diffbot	plain text or html	an option to remove inline ads	web api (private beta)	does fetching for you via provided url	n/a	extracts: relevant media, titile, tags, xpath descriptor for wrappers, comments and comment count, article summary
Readability	retains original structure	uses hardcoded heuristics to extract content divided by ads	open source javascript bookmarklet	via browser	language independent but it relies on language dependent regular expressions to match id and class labels	
Goose	plain text	n/a	open source java library	url only (my fork enables you to fetch the document by yourself)	language independent but it relies on language dependent regular expressions to match id and class labels	uses hardcoded heuristics to search for related images and embedded media
Extractiv	depends on the chosen output format – e.g. xml format breaks the content into paragraphs	n/a	commercial web api	include the whole document in post request or provide an url	n/a	capable of enriching the extracted text with semantic entities and relationships
Repustate API	plain text	n/a	commercial web api	url only	n/a	
Webstemmer	plain text	n/a	open source python library	first runs a crawler to obtain seed pages, then it learns layout patterns	language independent	the only piece of software on this list that requires a cluster

				that are later put to work to extract article content		of similar documents obtained by crawling
NCleaner (paper)	plain text	uses character level n-grams to detect content text blocks	open source perl library	arbitrary html document	depends on the training language	reliant on lynx browser for converting html to structured plain text

The reason why some cells in the table are marked as "n/a" was that of this table was built by inspecting the respective software documentation or research papers where the information of an observed feature was absent.

Posted in text extraction | Tagged comparison, information retrieval, semantic web, software, text extraction, web api | 0 Comments

Evaluation Metrics for Text Extraction Algorithms

Posted on March 31, 2011 by tomaz

In my two previous posts *(both were issued on hacker news, ReadWriteWeb and O'Reilly Radar)* I've covered quite a decent array of various text extraction methods and related software. So before reading this one I encourage you to read them to get a better feel of the topic. I bet those of you who read both posts, are eager to ascertain which method performs best. Before getting into performance and accuracy evaluation, we first have to set the groundwork by defining evaluation metrics.

Precision, Recall & F1-score

Both precision and recall are widely used metrics in various fields of study; classification, pattern recognition and information retrieval being most noticeable. When evaluating text extraction algorithms, we feel most comfortable using their respective information retrieval context.

Let's pretend for a while we are evaluating a search engine that retrieves a set of documents given a specific search term. Using this generic example we will review the formal definitions of precision and recall in their information retrieval context.

Formally precision and recall are defined as:

$$precision = \frac{|S_{rel} \cap S_{ret}|}{|S_{ret}|} \; recall = \frac{|S_{rel} \cap S_{ret}|}{|S_{rel}|}$$

Where S_{ret} denotes the set of retrieved documents and S_{rel} the set of documents relevant to the provided search term.

This interpretation of both measures can easily be applied to text extraction. Instead of retrieved and relevant documents for the given search term, we now have the following situation: given an HTML document we predict/extract/retrieve a set of words and denote them as S_{ret}. Consequentially the set of words of the relevant text is denoted as S_{rel}.

Formal definitions aside we must now explore the construction of these sets of words. There are many representations for text documents and but not all are suited for determining such sets. The most common model for representing text documents widely known in NLP, the bag-of-words model, has a tendency to produce biased results when used to construct sets of relevant and retrieved words. Since BOW discards all grammatical information and even word order of the original text, we have no way of determining if multiple instances of the retrieved word are part of the article content or part of the website's boilerplate structure (navigation, menus, related articles …).

A quick observation leads us to an improved model (for representing retrieved and relevant text) which retains word order - the array-of-words. This model is constructed by the following procedure:

1. remove punctuation
2. split the text into an array of words using whitespace as a delimiter
3. lowercase all words
4. (optional) stem all numbers and dates to neutral tokens

Calculating the denominator of both precision and recall using this model is trivial. But what about the numerator – the intersection between retrieved and relevant segment of text? The array-of-words model allows us to make use of delta calculating procedures. The idea is to compute a list of largest continuous subsequences of matching words inside two sequences of words.

Luckily for us, python programmers, there is a module inside the standard library that provides helpers for comparing sequences. Here is a quick example:

```python
import difflib

# initialize
matcher = difflib.SequenceMatcher()
ret = ['this', 'is', 'sparta']
rel = ['this', 'here', 'is','not','acutally', 'sparta']

# calculate matching blocks
matcher.set_seqs(ret, rel)
matches = matcher.get_matching_blocks()[:-1]

# get_matching_blocks returns a list of triples (named tuples)
# each triple (i, j, n) denoting that ret[i:i+n] == rel[j:j+n]
# the last 'dummy' triple is discarded since it's equal to (len(rel), len(ret), 0)
rel_intersect_ret = sum(i.size for i in matches)

print rel_intersect_ret
```

diff.py hosted with ❤ by GitHub view raw

With precision and recall in the bag, we now review a measure that combines both. F1-score is the harmonic mean of precision and recall and is computed using the following equation:

$$F_1 score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

F1-score is derived from a generalized F-beta-measure equation:

$$F_\beta score = (1+\beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

Wikipedia holds a quote, that perfectly articulates the relation of beta, precision and recall in the F-beta-measure equation:

> *β measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision*

Levenshtein Edit Distance

Apart from precision and recall, Levenshtein distance comes from the world of information theory. It's used to determine the number of edit operations needed for transforming one string into the other. Edit operations in the original version of Levenshtein distance are: insertion, deletion and substitution of a single character.

We could employ this measure to compare retrieved and relevant text represented as single strings. The downfall: computationally intensive and biased results due to whitespace character noise. But rather than using the single string representation, we use the array-of-words to represent both texts and than count the edit operations based on words to compute the mutual similarity.

A fairly simmilar technique was put into action during the CLEANEVAL competition, dedicated to competitive evaluation of cleaning arbitrary web pages. Their scoring was based on:

> *Levenshtein edit distance: the smallest number of 'insert word' and'delete word' steps to get from the one text to the other. Prior to calculating edit distance, we normalised both files by lower-casing, deleting punctuation and other nonalphanumeric characters and normalising whitespace.*

My Ideas & Conclusion

Metrics discussed above are based solely on text and that's fine. But what if you're also interested in the retained DOM structure of the content and other related attributes such as links, images and videos? Some implementations of text/content extractors have this capability that is widely ignored in all research based experimental evaluation setups since they're focused only on pure text.

Testing if the retrieved content includes relevant links, videos and images is somehow trivial, but what about the retained DOM structure? Assuming text extractors retain only the basic blocks (paragraphs, lists, anchor tags) of the original structure, comparing them back to the cleaned segment makes the problem even worse.

Challenged by the arisen issue I've been forging a technique to evaluate the correctness of the retained DOM structure. It roughly follows these guidelines:

- Flatten the DOM trees of the retrieved and relevant segments
- Compare the flattened DOM trees with a tree edit distance. Analogous to the Levensthtein distance this measure computes the number of operations needed to transform one tree structure into the other. It was originally developed for comparing ASTs of source code to detect plagiarism, but was later applied to other areas of study. *(see chapter 4.2. in this paper)*
- Employ levenshtein distance to compare contents of each block

Hopefully I'll be able to report about the success (or failure due to expected time complexity) in one of my next posts. I'll also do my best to reveal some early evaluation results of my experimental setup making use of metrics discussed in this post. So feel free to subscribe to my RSS feed or follow me on twitter for more updates.

Posted in text extraction | Tagged evaluation, information retrieval, metrics, text extraction | 2 Comments

List of resources: Article text extraction from HTML documents

Posted on March 11, 2011 by tomaz

UPDATE 21/3/2011: *Added reader contributed links to software and API section*

Following up to my overview of article text extractors, I'll try to compile a list of research papers, articles, web APIs, libraries and other software that I encountered during my research.

Research papers and Articles

Just to summarize the ones mentioned in my previous post:

- Boilerplate Detection using Shallow Text Features by Kohlschütter et al
- Extracting Article Text from the Web with Maximum Subsequence Segmentation by Pasternack & Roth
- Text Extraction from the Web via Text-to-Tag Ratio by Weninger & Hsu
  and the extension of the main algorithm to 2D histogram clustering: Web Content Extraction Through Histogram Clustering (another version)
- VIPS: a Vision-based Page Segmentation Algorithm

Others not mentioned in the overview:

- Fairly old, but cited by nearly every paper with a similar topic: Roadrunner is a wrapper induction technique based on pattern discovery within similar HTML documents.
- Automatic Web News Extraction Using Tree Edit Distance: This algorithm uses a tree comparison metric analogous to Levenshtein distance to detect relevant content in a set of HTML documents.
- Discovering Informative Content Blocks from WebDocuments: A bit outdated due to some assumptions, but interesting because it employs entropy as a threshold metric to predict informative blocks of content.
- Web Page Cleaning with Conditional Random Fields: If you will be reading any of the articles above, sooner or later you'll notice everybody is citing the CleanEval shared task which took place in 2007. This paper presents the best performing algorithm which makes use of *CRF* to label blocks of content as *text* or *noise* based on block level features.

I’ve also stumbled upon these:

- Hierarchical wrapper induction for semistructured information sources
- Template detection for large scale search engines
- Web Page Cleaning for Web Mining through Feature Weighting
- Eliminating noisy information in Web pages for data mining

The only good blog article I came across is the one at *ai-depot.com*: The Easy Way to Extract Useful Text from Arbitrary HTML. The author is using examples written in python to employ a fairly similar technique described in the *text-to-tag ratio* paper listed above.

Software

There is only a small amount of competition when it comes to software capable of [removing boilerplate text / extracting article text / cleaning web pages / predicting informative content blocks] or whatever terms authors are using to describe the capabilities of their product.

The least common denominator for all the listed software below is the following criteria: given an HTML document with an article in it, the system should yield the incorporated article itself (and maybe it’s title and other meta-data):

- Boilerpipe library: an open source Java library. The library itself is the official implementation of the overall algorithm presented in the previously mentioned paper by Kohlschütter et al.
- Readability bookmarklet by arc90labs is open sourced. Originally written in JavaScript it was also ported to other languages:
  - python-readabilty – using BeautifulSoup (slow)
  - fork of python-readability employing lxml for faster parsing
  - ruby-readability
  - PHP port
  - jReadability
  - C# port
- Project Goose by Gravity labs
- Perl module HTML::Feature
- Webstemmer is a web crawler and page layout analyzer with a text extraction utility
- Demo of VIPS packaged in a .dll (it’s use is limited to research purposes only)

Web APIs

After a short inquiry I came across some very decent web APIs:

- Alchemy API Web Page Cleaning – a well known commercial API with a limited free service
- ViewText.org – they’re asking you to be kind to their servers, so this is not your typical commercial service
- DiffBot API – describes itself as: *“Statistical machine learning algorithms are run over all of the visual*

elements on the page to extract out the article text and associated metadata, such as its images, videos, and tags."

- Purifry – is promising high performance and good accuracy. It's also available as a binary.
- Extractiv – text extraction is just a side feature
- Repustate API – includes a *clean html* call

Related

There is a ton of stuff out there that is somehow related to items listed above. Perhaps you'll find them interesting or at least useful:

- Full text RSS feed builder at fulltextrssfeed.com – a very neat example of putting article text extraction into practice
- FiveFilter.org – another full text RSS builder. According to their FAQ, they're using readability ported to PHP
- Boilerpipe has a nice demo running on appengine
- Demo of the previously mentioned extraction algorithm using maximum subsequence optimization

This is it. Hopefully you now have a better perspective of the sparse literature and software on the topic in question. If you're aware of an item that should be added to any of the lists in this post, please drop me a line in the comments. Don't forget to subscribe to my RSS feed for related updates.

Posted in text extraction | Tagged information retrieval, text extraction | 17 Comments

Overview: Extracting article text from HTML documents

Posted on March 2, 2011 by tomaz

In the world of web scraping, text mining and article reading utilities (readability bookmarklet) there is an ever growing demand for utilities that are capable of distinguishing parts of a HTML document which represent an article apart from other common website building blocks like menus, headers, footers, ads etc.

Turns out this problem was baffling researchers since the early 00s. Keep in mind, that back then, folks were still making websites with microsoft frontpage. The majority of methods presented in research papers from that epoch are nowadays useless due to strong assumptions and heuristics that don't apply on today's web development practices.
In the following chapters I'll try to review some article text extraction methods that are applicable to today's websites. They mostly leverage on machine learning, statistics and a wide rage of heuristics.

Boilerpipe library: Boilerplate Removal and Fulltext Extraction from HTML pages

Boilerpipe is probably one of the best open source packages when it comes to full article text extraction that leverages on machine learning. The overall algorithm works by computing both text and structural features on parts of the document and based on these features, decide if the observed part of the document belongs to the

article text or not. The decision rules are inferred by a classifier trained on a set of labeled documents.
To obtain a set of learning examples that are fed into a classifier, boilerplate observes a list of features on different levels of the HTML document:

- text frequency in the whole corpus: to obtain phrases commonly used in useless parts of the document
- presence of particular tags that enclose a block of text: <*h#*> headline, <*p*> paragraph, <*a*> anchor and <*div*> division
- shallow text features: average word length, average sentence length, absolute number of words in the segment
- local context of text: absolute and relative position of the text block
- heuristic features: number of words that start with an uppercase letter, number of words written in all-caps, number of date and time tokens, link density and certain ratios of those previously listed
- density of text blocks: number of words in a wrapped fixed column width text block divided by number of lines of the same block

Each HTML document is segmented into atomic text blocks that are annotated with features listed above and labeled (by a human annotator) with *content* or *boilerpate* class.

In the original paper authors used decision trees and SVMs, trained on previously mentioned set of learning examples, to classify parts of the newly observed document as content or boilerplate text.

Extracting Article Text from the Web with Maximum Subsequence Segmentation

The overall algorithm (MSS) transforms the problem of detecting article text in HTML documents to *maximum subsequence optimization*.

Let's start of by examining maximal subsequence optimization and it's problem statement. Given a sequence of numbers, we've been given a task of finding a continuous subsequence where the sum of it's elements is maximal. This is trivial if the elements of the sequence are all non-negative. In other words:

$$(a, b) = argmax_{(x,y)} \sum_{i=x}^{y} s_i$$

In terms of time complexity this can easily be computed in *O(n),* where *n* stands for number of elements in the original sequence.

Now let's take a look at the representation of documents used in MSS. Each document is tokenized using the following procedure:

1. discard everything between <*script*> and <*style*> tags
2. break up HTML into a list of tags, words and numbers
3. apply porter stemming to all words
4. generalize numeric tokens

To apply maximum subsequence optimization to the particular representation of the document, MSS uses local token-level classifiers to find a score for each token of the document. A negative score indicates that the observed token is not likely to be considered as content and vica-versa for tokens with positive scores. Because MSS uses a global optimization over all scores, there is no need for highly accurate local classifiers.

Experimental setup in the original paper uses Naive Bayes with 2 types of features for every labeled token in the document:

- trigram of token: the token itself and its 2 successors
- parent tag of token in the DOM tree (this can easily be implemented by maintaining a stack of tags when passing through the token array)

Each score produced by the NB classifier is than transformed with *f(p) = p – 0.5* to obtain a sequence of scores ranging from *[-0.5, 0.5]*.

This is basically it; local token-level classifier outputs a sequence of scores to which we apply maximum subsequence optimization. Indexes of the subsequence are used to result the set of tokens that represent the extracted content.

Text Extraction from the Web via Text-to-Tag Ratio

If MSS transformed the problem of article text extraction to maximum subsequence optimization; this approach transforms it to histogram clustering.

Clustering is applied to the text-to-tag ratio array (TTRArray) which is obtained by the following procedure:

1. remove all empty lines and everything between <*script*> tags from the given HTML document
2. initialize the *TRRArray*
3. for each line in the document
   A. let *x* be the number of non-tag ASCII characters
   B. let *y* be the number of tags in that line
   C. if there are no tags in the current line than *TTRArray[current line] = x*
   D. else *TTRArray[current line] = x/y*

The resulting TTRArray holds a text-to-tag ratio for every line in the filtered HTML document.

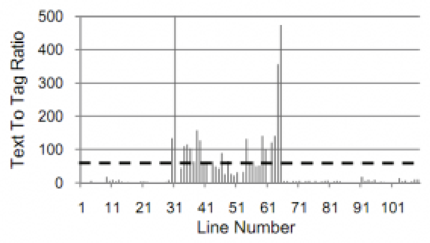
Clustering can be applied to the TTRArray to obtain lines of text that represent the article by considering the following heuristic:

> *For each k in TTRArray, the higher the TTR is for an element k relative to the mean TTR of the entire array the more likely that k represents a line of content-text within the HTML-page.*

Prior to clustering, TTRArray is passed by a smoothing function to prevent the loss of short paragraph lines that might still be part of the content at the edges of the article.

Weninger & Hsu propose the following clustering techniques to obtain article text:

- K-means
- Expectation Maximization
- Farthest First
- Threshold clustering; using standard deviation as a cut-of threshold



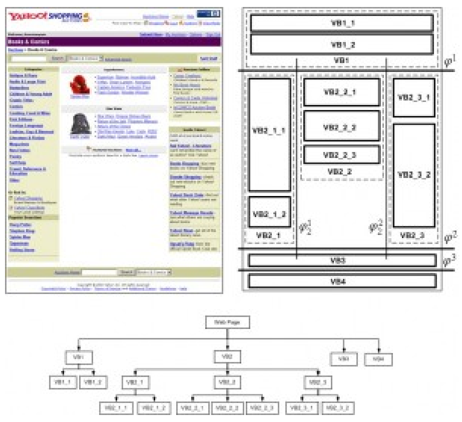
TTRArray (image courtesy of Weninger & Hsu: Text Extraction from the Web via Text-to-Tag Ratio)

VIPS: a Vision based Page Segmentation Algorithm

What's distinctive about this algorithm from others mentioned in this post is making full use of the visual page layout features.

It first extracts blocks from the HTML DOM tree and assigns them a value that indicates the coherency of the block based on visual perception. A hierarchical semantic structure is then employed to represent the structure of the website. The algorithm applies vertical and horizontal separators to the block layout to construct such a semantic structure.

The constructed structure is then used to find parts of the website that represents article text and other common building blocks.


Semantic tree structure (source http://www.zjucadcg.cn/dengcai/VIPS/VIPS.html)

Readabillity

A perfect example of good ol' hand crafted heuristics. Readability relies solely on common HTML coding practices to extract article text, title, corresponding images and even the next button if the article seems to be segmented into several pages.

Readability was originally developed as a JavaScript bookmarklet with a mission to mitigate the troubles of reading trough ad cluttered news websites. Readability is nowadays the most popular article text extraction tool.

Originally written and maintained in JavaScript it was also ported to python and ruby.

Conclusion

I admit I haven't covered all the text extraction algorithms lurking in the wild, but hopefully this post will save some time for those of you who want to dive straight into sparse literature and software packages.

If you've been working on something similar or stumbled across a library that's capable of full article text extraction from HTML documents, please drop me a line in the comments.

In my next post I'll try to bundle up a repository of available open source packages and web services capable of text extraction, so keep in touch by subscribing to my RSS feed.

Posted in text extraction | 6 Comments

My tech blog.
WordPress.