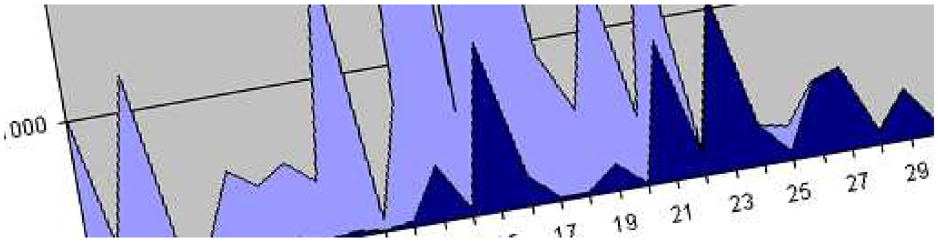
- News
- Articles
- Knowledge
- Resources
- Site

The Easy Way to Extract Useful Text from Arbitrary HTML

By alexjc | April 5, 2007



You've finally got your hands on the diverse collection of HTML documents you needed. But the content you're interested in is hidden amidst adverts, layout tables or formatting markup, and other various links. Even worse, there's visible text in the menus, headers and footers that you want to filter out. If you don't want to write a complex scraping program for each type of HTML file, there is a solution.

This article shows you how to write a relatively simple script to extract text paragraphs from large chunks of HTML code, without knowing its structure or the tags used. It works on news articles and blogs pages with worthwhile text content, among others...

Do you want to find out how statistics and machine learning can save you time and effort mining text?

The concept is rather simple: use information about the density of text vs. HTML code to work out if a line of text is worth outputting. (This isn't a novel idea, but it works!) The basic process works as follows:

1. Parse the HTML code and keep track of the number of bytes processed.
2. Store the text output on a per-line, or per-paragraph basis.
3. Associate with each text line the number of bytes of HTML required to describe it.
4. Compute the text density of each line by calculating the ratio of text to bytes.

5. Then decide if the line is part of the content by using a neural network.

You can get pretty good results just by checking if the line's density is above a fixed threshold (or the average), but the system makes fewer mistakes if you use machine learning — not to mention that it's easier to implement!

Let's take it from the top…

Converting the HTML to Text

What you need is the core of a text-mode browser, which is already setup to read files with HTML markup and display raw text. By reusing existing code, you won't have to spend too much time handling invalid XML documents, which are very common — as you'll realise quickly.

As a quick example, we'll be using Python along with a few built-in modules: `htmllib` for the parsing and `formatter` for outputting formatted text. This is what the top-level function looks like:

```
def extract_text(html):
    # Derive from formatter.AbstractWriter to store paragraphs.
    writer = LineWriter()
    # Default formatter sends commands to our writer.
    formatter = AbstractFormatter(writer)
    # Derive from htmllib.HTMLParser to track parsed bytes.
    parser = TrackingParser(writer, formatter)
    # Give the parser the raw HTML data.
    parser.feed(html)
    parser.close()
    # Filter the paragraphs stored and output them.
    return writer.output()
```

The TrackingParser itself overrides the callback functions for parsing start and end tags, as they are given the current parse index in the buffer. You don't have access to that normally, unless you start diving into frames in the call stack — which isn't the best approach! Here's what the class looks like:

```
class TrackingParser(htmllib.HTMLParser):
    """Try to keep accurate pointer of parsing location."""
    def __init__(self, writer, *args):
        htmllib.HTMLParser.__init__(self, *args)
        self.writer = writer
    def parse_starttag(self, i):
        index = htmllib.HTMLParser.parse_starttag(self, i)
        self.writer.index = index
        return index
    def parse_endtag(self, i):
        self.writer.index = i
        return htmllib.HTMLParser.parse_endtag(self, i)
```

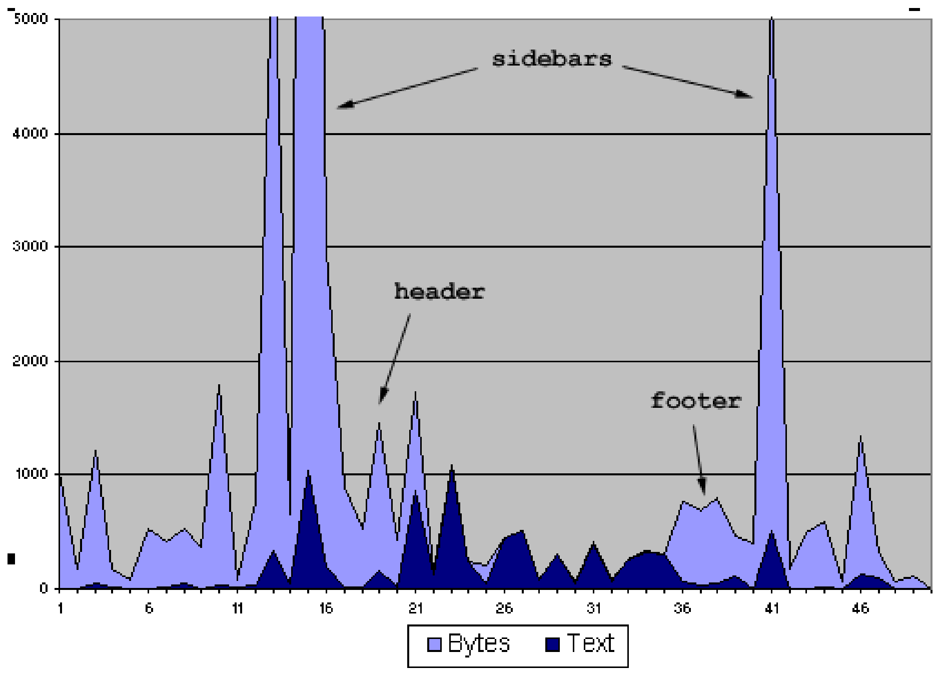
The `LineWriter` class does the bulk of the work when called by the default formatter. If you have any improvements or changes to make, most likely they'll go here. This is where we'll put our machine learning code in later. But you can keep the implementation rather simple and still get good results. Here's the simplest possible code:

```
class Paragraph:
    def __init__(self):
        self.text = ''
        self.bytes = 0
        self.density = 0.0

class LineWriter(formatter.AbstractWriter):
    def __init__(self, *args):
        self.last_index = 0
        self.lines = [Paragraph()]
        formatter.AbstractWriter.__init__(self)

    def send_flowing_data(self, data):
        # Work out the length of this text chunk.
        t = len(data)
        # We've parsed more text, so increment index.
        self.index += t
        # Calculate the number of bytes since last time.
        b = self.index - self.last_index
        self.last_index = self.index
        # Accumulate this information in current line.
        l = self.lines[-1]
        l.text += data
        l.bytes += b

    def send_paragraph(self, blankline):
        """Create a new paragraph if necessary."""
        if self.lines[-1].text == '':
            return
        self.lines[-1].text += 'n' * (blankline+1)
        self.lines[-1].bytes += 2 * (blankline+1)
        self.lines.append(Writer.Paragraph())

    def send_literal_data(self, data):
        self.send_flowing_data(data)

    def send_line_break(self):
        self.send_paragraph(0)
```

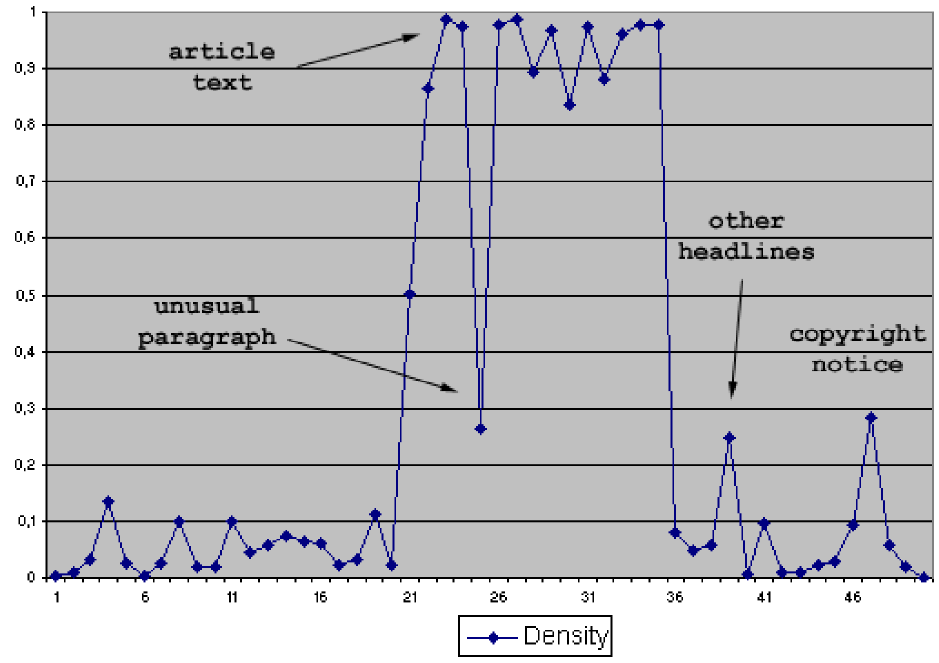
This code doesn’t do any outputting yet, it just gathers the data. We now have a bunch of paragraphs in an array, we know their length, and we know roughly how many bytes of HTML were necessary to create them. Let’s see what emerge from our statistics.

Examining the Data

Luckily, there are some patterns in the data. In the raw output below, you’ll notice there are definite spikes in the number of HTML bytes required to encode lines of text, notably around the title, both sidebars, headers and footers.



While the number of HTML bytes spikes in places, it remains below average for quite a few lines. On these lines, the text output is rather high. Calculating the **density** of text to HTML bytes gives us a better understanding of this relationship.



The patterns are more obvious in this density value, so it gives us something concrete to work with.

Filtering the Lines

The simplest way we can filter lines now is by comparing the density to a fixed threshold, such as 50% or the *average* density. Finishing the `LineWriter` class:

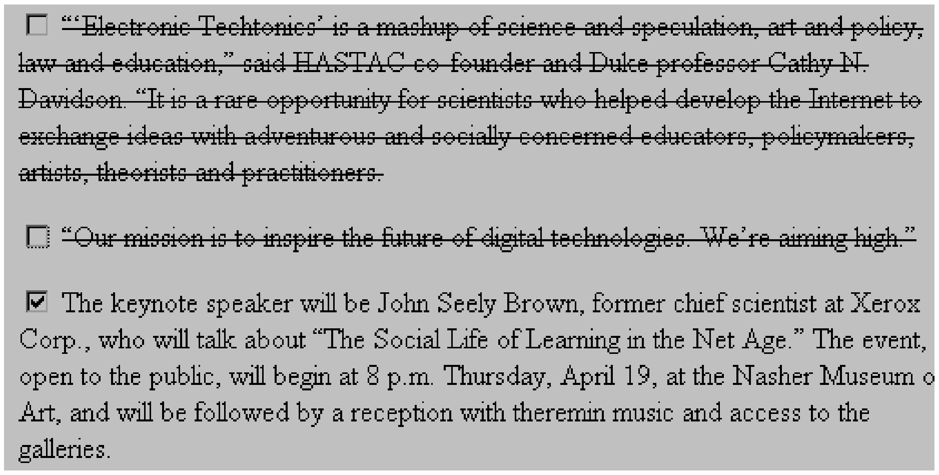
```
    def compute_density(self):
        """Calculate the density for each line, and the average."""
        total = 0.0
        for l in self.lines:
            l.density = len(l.text) / float(l.bytes)
            total += l.density
        # Store for optional use by the neural network.
        self.average = total / float(len(self.lines))

    def output(self):
        """Return a string with the useless lines filtered out."""
        self.compute_density()
        output = StringIO.StringIO()
        for l in self.lines:
            # Check density against threshold.
            # Custom filter extensions go here.
            if l.density &gt; 0.5:
                output.write(l.text)
        return output.getvalue()
```

This rough filter typically gets most of the lines right. All the headers, footers and sidebars text is usually stripped as long as it’s not too long. However, if there are long copyright notices, comments, or descriptions of other stories, then those are output too. Also, if there are short lines around inline graphics or adverts within the text, these are not output.

To fix this, we need a more complex filtering heuristic. But instead of spending days working out the logic manually, we’ll just grab loads of information about each line and use machine learning to find patterns for us.

Supervised Machine Learning

Here’s an example of an interface for tagging lines of text as content or not:



☐ ~~“‘Electronic Techtonics’ is a mashup of science and speculation, art and policy, law and education,” said HASTAC co-founder and Duke professor Cathy N. Davidson. “It is a rare opportunity for scientists who helped develop the Internet to exchange ideas with adventurous and socially concerned educators, policymakers, artists, theorists and practitioners.~~

☐ ~~“Our mission is to inspire the future of digital technologies. We’re aiming high.”~~

☑ The keynote speaker will be John Seely Brown, former chief scientist at Xerox Corp., who will talk about “The Social Life of Learning in the Net Age.” The event, open to the public, will begin at 8 p.m. Thursday, April 19, at the Nasher Museum o Art, and will be followed by a reception with theremin music and access to the galleries.

The idea of supervised learning is to provide examples for an algorithm to learn from. In our case, we give it a set documents that were tagged by humans, so we know which line must be output and which line must be filtered out. For this we’ll use a simple neural network known as the perceptron. It takes floating point inputs and filters the information through weighted connections between “neurons” and outputs another floating point number. Roughly speaking, the number of neurons and layers affects the ability to approximate functions precisely; we’ll use both single-layer perceptrons (SLP) and multi-layer perceptrons (MLP) for prototyping.

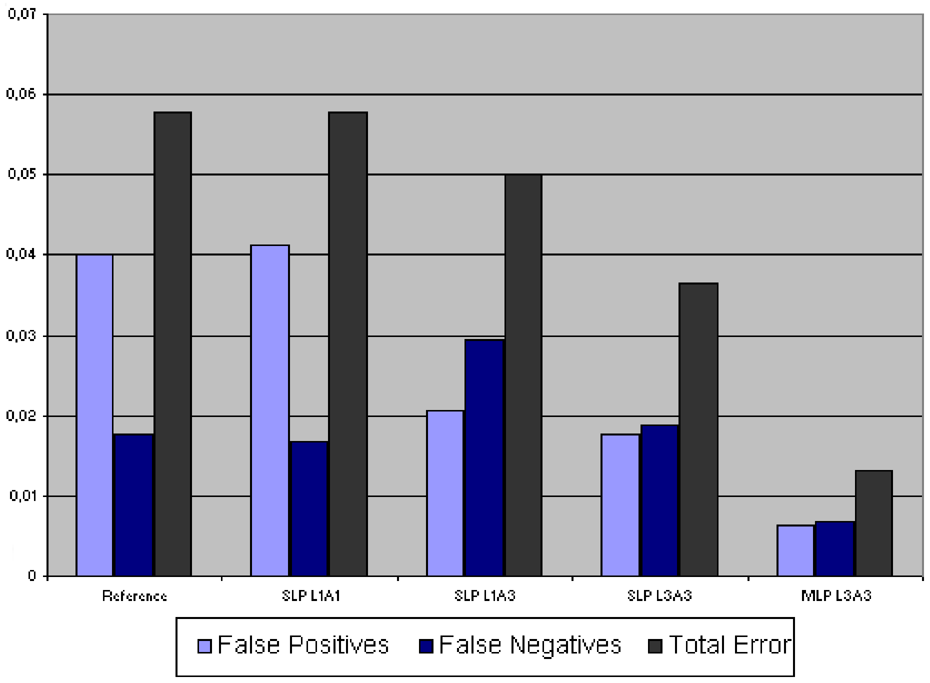
To get the neural network to learn, we need to gather some data. This is where the earlier `LineWriter.output()` function comes in handy; it gives us a central point to process all the lines at once, and make a global decision which lines to output. Starting with intuition and experimenting a bit, we discover that the following data is useful to decide how to filter a line:

- Density of the **current** line.
- Number of HTML bytes of the line.
- Length of output text for this line.
- These three values for the **previous** line,
- … and the same for the **next** line.

For the implementation, we’ll be using Python to interface with *FANN*, the Fast Artificial Neural Network Library. The essence of the learning code goes like this:

```
from pyfann import fann, libfann

# This creates a new single-layer perceptron with 1 output and 3 inputs.
obj = libfann.fann_create_standard_array(2, (3, 1))
ann = fann.fann_class(obj)

# Load the data we described above.
patterns = fann.read_train_from_file('training.txt')
ann.train_on_data(patterns, 1000, 1, 0.0)

# Then test it with different data.
for datin, datout in validation_data:
    result = ann.run(datin)
    print 'Got:', result, ' Expected:', datout
```

Trying out different data and different network structures is a rather mechanical process. Don’t have too many neurons or you may train too well for the set of documents you have (overfitting), and conversely try to have enough to solve the problem well. Here are the results, varying the number of lines used (1L-3L) and the number of attributes per line (1A-3A):



The interesting thing to note is that 0.5 is already a pretty good guess at a fixed threshold (see first set of columns). The learning algorithm cannot find much better solution for comparing the density alone (1 Attribute in the second column). With 3 Attributes, the next SLP does better overall, though it gets more false negatives. Using multiple lines also increases the performance of the single layer perceptron (fourth set of columns). And finally, using a more complex neural network structure works best overall — making 80% less errors in filtering the lines.

Note that you can tweak how the error is calculated if you want to punish false positives more than false negatives.

Conclusion

Extracting text from arbitrary HTML files doesn't necessarily require scraping the file with custom code. You can use statistics to get pretty amazing results, and machine learning to get even better. By tweaking the threshold, you can avoid the worst false positive that pollute your text output. But it's not so bad in practice; where the neural network makes mistakes, even humans have trouble classifying those lines as "content" or not.

Now all you have to figure out is what to do with that clean text content!

Tags: machine learning, neural network, python, scraping, statistics, text mining
Category: tutorial |

10 Responses to "The Easy Way to Extract Useful Text from Arbitrary HTML"

1. *Cenny Wenenr* Says:
   April 10th, 2007 at 7:40 am

   For the simple greedy immidietly local threshold, we could introduce a few assumptions to find a perhaps better estimate than the average. The average of some set is frequently used because it is the point with the least square distance over all values in the set. In this case however, there are two separate sets: text items and markup items.
   If we assume that each follows a normal distribution (which is common) and there is some probability to pick from one of them, we may estimate these parameters and find the values Thete for which the probability of either classification is 0.5.
   Solving this we receive the threshold:
   theta* = (m_y t_y - m_z t_z +- (m_y - m_z) sqrt(t_y t_z)) / (t_y - t_z)
   where m_y is the average of the text items, m_z the average of the markup items and t_z and t_y are given by:
   t_y = s_y ln p
   t_z = s_z ln(1-p)
   where s_y is the estimated standard deviation, which is
   sum of all i: (y_i - m_y)^2
   and similar for z.
   p is the ratio of text items, i.e. |y| / (|y| + |z|).

   A LaTeX formula can be found here:
   http://ai-freenode.wikidot.com/the-easy-way-to-extract-useful-text-from-arbitrary-html

   I can add additional details if anyone wishes. Reservations for errors, did this hastily and it is not double checked in any way.

   It should yield a bit better estimate than just the average but only relies on the immidietely local features and not for instance the relation to neighbors and global features. These should have a great impact. Hence, ANNs should work better if feasible.

2. *Anand Muthu* Says:
   April 19th, 2007 at 7:02 am

   Great thought ! Superb :)
   Thinking to implement this in Java too.

3. *Phil* Says:
   June 23rd, 2007 at 3:41 pm

   This approach to extract text for html page is interesting. Good idea.

4. *Lawrence* Says:
   October 15th, 2007 at 11:23 am

That is quite interesting. I want to find a similar approach only with C++. In my opinion, this is probably the only way to do an AI capable of learning from the internet or books.

5. *Betty* Says:
   December 11th, 2007 at 9:27 pm

   good idea!similar to the 4th floor,I want to find a approach with C++.

6. *veelion* Says:
   December 25th, 2007 at 9:42 pm

   good idea! but what's the content of the file 'training.txt' while using FANN? Thanks. If possible, please send an example to veelion@gmail. It is appreciated.

7. *Einar Vollset's Blog. » Blog Archive » Extracting human text from web pages using Ruby* Says:
   February 12th, 2008 at 10:40 am

   […] number of available options out there, including using NLTK (a pretty good Python NLP library), and this approach using machine learning (also in Python - it seems to be the language of choice for AI these […]

8. *natch* Says:
   August 6th, 2008 at 1:57 pm

   If the goal is "Easy" as you say, just do this:

   lynx –dump URL_HERE > output.txt

9. *AlanT* Says:
   August 6th, 2008 at 7:48 pm

   Interesting article! From my own experience, I will also recommend using Feedity (http://feedity.com), a simple data extraction service which can convert any webpage to structured information and generate a RSS feed for it. I use it daily for newsreading, mashups, and even for data integration at work. Its really awesome!

10. *Srinivasan R* Says:
    August 7th, 2008 at 3:13 am

    Great work. We actually use a HTML parser (BeautifulSoup) and then find then in each div (or any container) tag decide whether to include the text in the output or not based on the link to text ratio. We mostly need to remove the sidebar menus and top menus which mostly have anchor tags inside the container tags.
    We also rule out some basic script/object/embed tags and also use common css classes to identify comments, header, etc.
    Works pretty good for us, but would also like to try this out.

Comments

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

Search

Subscribe Now

Subscribe to artificial intelligence articles in a reader.

Or get notified by email:

Subscribe

Recent Posts

- Artificial Intelligence in Games
- More AI Content & Format Preference Poll
- What's Your Biggest Question about Artificial Intelligence?
- The Easy Way to Extract Useful Text from Arbitrary HTML
- AI Knowledge At Your Fingertips
- Inside DARwIn the Humanoid Robot
- IRC Bot Announces Headlines
- AI-Depot.com Renovated
- The Random Test
- Solving Integer Programming Problems Using Genetic Algorithms

Archives

- September 2007
- April 2007
- March 2007
- July 2003
- May 2003
- April 2003
- March 2003
- February 2003
- January 2003
- December 2002
- November 2002
- October 2002
- September 2002
- August 2002
- July 2002
- June 2002
- April 2002

Categories

- analysis
- announcement
- essay
- interview
- review
- tutorial