3
Our approach

In this work, we design an efficient page-level approach that lies in between the strictly structural vs. full rendering spectrum. It uses structural properties of news webpages and visual presentation information from cascading style sheets, such as font size and color, which may be calculated without a full rendering of the webpage, providing us with good execution times. With this information available, we train a machine learning model to classify DOM nodes and employ some post-processing afterwards. Also, we strive to keep it language independent. Since we don't rely on textual cues, we can apply our approach to virtually any document and observe little variation in the results obtained.

To accomplish this, we first detect the relevant content of the news page and discard all the rest. We then proceed by calculating our lightweight features only in this relevant subset of the page, and follow with machine-learning models to identify the title, date and body. An outline of our execution pipeline is displayed in Figure 3.1.

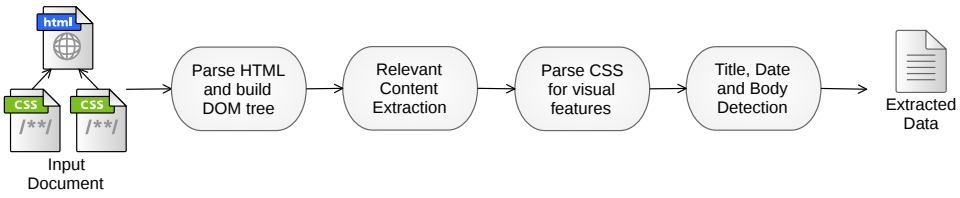

Figure 3.1: Outline of our execution pipeline

3.1
Simplified CSS parser

We have developed a simplified CSS parser that exempts us from using a rendering engine such as Mozilla Firefox's Gecko [11], Microsoft Internet Explorer's MSHTML [25] or Apple Safari and Google Chrome's WebKit [42]. These engines are capable of a much more accurate parsing, but end up computing much more information than we require, raising execution time. With our parser we have access to all information we need much faster, making

our approach ideal for use in time-sensitive tasks such as crawling the web. A time comparison between rendering engines and our approach is made in chapter 5.

Our parser was originally developed to compute font sizes and font colors. This is relatively simple: any implementation of the CSS specifications should provide this. However, we did run into a lot of issues, mostly caused by pages which did not follow the specifications. Modern browsers are capable of rendering such pages through what they call *quirks mode*, a more flexible rendering mode, backwards compatible with older versions of current standards and which often does several assumptions and error corrections. These are not documented, and we had to create our own heuristics based on trial and error. Common examples of problems we found are lack of units for property values and wrong declaration order for CSS selectors and property values.

Computing the positioning and space that each element occupies on screen once rendered is not impossible, but considerably more complicated and computationally intensive. The speed comes then from two different fronts: first, since we do not require geometric attributes, we do not need to render the whole webpage at once (it is often the case that one element impacts on the rendering of another through properties such as `float`) and can compute visual properties on-demand. Since our algorithm does not go through every node in the DOM tree, we save time by only computing properties for the nodes we are interested in. Second, because we have a predefined subset of properties that are of interest, we can discard several CSS rules that do not impact on those properties, thus analyzing less rules, reducing the size of our data structures and increasing performance. Our numbers show that about 80% of the properties and 70% of the rules can be discarded, which translates into an execution time of about 40% of the total time that would be spent with CSS.

3.2 Attributes

For each text node, we use 9 attributes computed from the parsed webpage as features for our machine learning models, chosen by human observation. These are:

1. the text length, measured in characters;

2. amount of digit characters present in the text;

3. percentage of digit characters in the text;

4. font size, normalized to pixels;
5. font size relative to the biggest font size seen, measured in percentage;
6. whether the node is presented in bold letters;
7. the amount of similarly styled nodes in the document;
8. a measure of distance between the current node and the document's node that contains the title;
9. edit distance between the node text and the document's `<title>` tag contents.

The first six attributes are very natural. The last three, however, deserve some further explanation.

Attribute 7 attempts to capture the uniqueness of a presentation style. This is motivated by the observation that the title is often easily recognized by humans because it stands out from the rest of the text in the page, usually because of a combination of style attributes. Two presentation styles are considered similar if the font size, font color, font family (serif, sans-serif font, etc.) and bold text attributes are the same.

Attribute 8 is discarded for title detection. It is only used for date detection, measuring the distance to the detected title node. We define distance in this case as the signed difference between the current node's and the detected title node's position in the ordered list of all text nodes of the document. More details as to how the title node is selected is given in Section 3.4.2.

Attribute 9 measures the edit distance between the current node and the document's `<title>` tag text. This is used only for title detection, and is calculated as

$$EditDistance(node\ text,\ title\ tag)\ /\ length(title\ tag)$$

where $EditDistance(\cdot, \cdot)$ is the classic Levenshtein metric [19] with weights adjusted to penalize deletions from the node text much more than insertions[1]. This is done to smooth out the impact of the web site identification that usually comes in the document title while penalizing the loss of the node's information. For example, an example news story from CNN has a document title of "Apple's Lion makes a move towards mobile – CNN.com", and the title node contains only "Apple's Lion makes a move towards mobile". When calculating the edit distance between these two strings, the "– CNN.com" part

[1]We empirically set the *insertion cost* to 1, *change cost* to 2 and *deletion cost* to 4.

is to be inserted to the node's title when comparing them, which is why the weights are adjusted to penalize such operation much less. One might argue that it would be enough to check whether one is a substring of another, but not only this is not always the case, but some encoding issues might get in the way. Using edit distance, this is less of an issue.

3.3 Training process

We perform some preprocessing to remove some noise before training a model for date detection. Consider the example in Figure 3.2, which will be revisited in more details in Section 5.1, Notice that, unless we annotate DOM text nodes individually, the `div` node is to be annotated as date. While the `b` nodes can also be annotated to avoid that it inherits the date label, the text node with the text "John" still inherits this label.

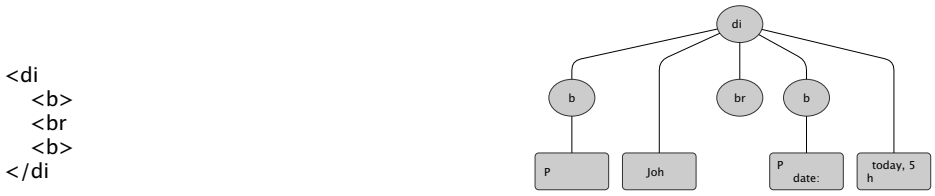

Figure 3.2: Example of a DOM subtree that will receive preprocessing.

To resolve this issue, we programatically remove the date label from all text nodes that do not contain any numeric characters. The idea behind this is based on the human observation that, like the example shows, dates tend to use numbers in one way or another to report the publication date: either in the time (hours and minutes), the date (day and year, mostly) or age of publication (such as the example – "5 hours ago"). Some exceptions exist, such as "yesterday", but these are not normally used in such broad terms in big news portals.

After this preprocessing, we proceed by training two binary classification models: one for title detection and one for date detection.

3.4 Classification process

3.4.1 Extraction of relevant content

The first step in our approach is the extraction of relevant content from the news page. We define as *relevant* the body of the news story, its title

and associated metadata, such as author and publication date. That is, the relevant content is what is left after removing all templates associated with the webpage.

To detect the relevant content, we employ the NCE algorithm described in [18]. This algorithm is quite fast and provides a very good starting point for the news body, which will later be improved by the next steps of our approach. It is based on the assumption that a *separator node* exists in the webpage; defined as a node for which the textual content of the subtree rooted on it contains most or all of the relevant content of the page. We proceed by refining the selection of relevant content with the identification and removal of subtrees that present a high link density or repeated textual patterns. The former is indicative of navigational links (menus), advertisements, related stories, etc.; and the latter often indicates a comments section at the end of the news body. As a definition of link density we use the number of characters in `<a>` tags in a subtree over the number of characters in the textual content of that subtree. The remaining subtrees combined should provide high values of precision and recall for the relevant content.

More formally, consider two families of parameters, $\{\alpha_i\}$ and $\{\beta_j\}$, and let T_u denote the subtree rooted at node u, $text(T_u)$ denote all text in T_u and $linkdensity(T_u)$ denote the link density of T_u. We will be denoting by $F_{u,\gamma}$ the forest composed by subtrees, with link density at most γ, that are rooted at children of u; that is, $F_{u,\gamma} = \{T_v | v \text{ is a child of } u \text{ and } linkdensity(T_v) \leq \gamma\}$. The separator node is a node u that satisfies the following conditions:

1. contains a leaf node L with $|text(L)| \geq \beta_3$;
2. is not a paragraph tag (`<p>`);
3. has $|text(T_u)| \geq \beta_2$;
4. has at most one sibling u' for which $|text(T_{u'})| > \beta_3$ and $linkdensity(T_{u'}) < \beta_1$; and
5. u is the closest ancestor of L that simultaneously satisfies the above conditions.

If a separator node cannot be located, the algorithm return the entire DOM tree. Otherwise, it returns the forest F_{u,β_4}.

A postprocessing is then performed on the returned forest to detect and remove comment sections. Consider a depth-first search in this forest, where a node u is added to a list L if it satisfies $\alpha_1 \leq |text(T_u)| \leq \alpha_2$. In this case, the search continues from the first node that is not a descendant of u. We

then build a graph $G = (L, E)$ where there is an edge (u, v) if and only if u and v differ by at most β_5 position in L and the size of the longest common subsequence between $text(T_u) and text(T_v)$ is at least β_6 times the size of the smallest of these two texts. A connected component in G then corresponds to a set of subtrees that are closely located and have similar content. The motivation is that the content in these subtrees is similar due to the user identification and date timestamps of these comments and closely located. As soon as a connected component with at least 3 nodes is detected, we identify its least common ancestor z and remove it from the solution and also every node that comes after z in a depth-first search.

We suggest the reader to refer to [18] for more details on the implementation of NCE and the different steps of the algorithm.

From this point forward, all steps are performed on a subset of nodes composed by the relevant nodes detected and all nodes that come before the first detected relevant node when doing a depth-first search. We extend the subset of evaluated nodes this way to ensure the title and date are contained in the subset we will analyze, but with further experimentation this could be tuned with a parameter.

3.4.2 Title and date detection

We apply a binary classification model at each step, separating the title from all the rest, then the date from all the rest. For this task, we use an ensemble of decision tree models [36], which was introduced in Section 2.4. We used the Weka tool [12] for training and model settings were kept with default values, with no specific optimizations.

First, we classify the document title. Second, we classify the publication date. Our premise is that dates are generally close to the document title [41], so we use the classified title as a feature for the date classification. For each node, we calculate its distance to the classified title and use that to guide our model. This constitutes attribute 8, mentioned in Section 3.2.

It's worth considering what happens with the date classification model when no title node is detected. We make some considerations on this regard in Chapter 5.

3.4.3 Post-processing

Each of the previously mentioned steps receive some post-processing.

For titles, we only allow one node per document to be detected. In case two or more are detected (excluding the `<title>` tag), we always pick the one that comes first on the document. However, if no title nodes are detected, we take the `<title>` tag as the title.

For dates, we cluster all detected nodes by their proximity in the document and select the cluster closest to the detected title. Two nodes belong to the same cluster if there is at most two other text nodes in the DOM tree between them. If either of them is already in a cluster, the clusters are combined. In case no title was detected (and thus the `<title>` tag is used), no clustering takes place and all nodes are considered as date. Next, nodes which would add large amounts of text are discarded. This is usually a sign of noise being added as our observations show that publication dates are well separated from the document's content.

Finally, we redefine as body every non-title and non-date relevant node visited in a depth-first search that starts at the detected title and ends at the last node from the extracted relevant content.