2
Important concepts

In this chapter we will go over some important concepts for the full understanding of this work.

2.1
HyperText Markup Language (HTML)

The HyperText Markup Language (HTML) [15] is the markup language used to describe webpages. Each *HTML element* is represented by a *tag*, such as `<html>`. Tags are usually found in pairs, with content in between them, like this: `<p>example text</p>`. Exceptions which do not contain content, and thus do not require a closing tag, include `<br>`, which represents a line break; `<img>`, which represents an image; and `<hr>`, which represents an horizontal line. Also, tags must be closed in reverse order of opening; that is, more recently opened tags must be closed before their predecessors, creating an hierarchy of elements contained in others. En example of an HTML document is available in Figure 2.1

```
<html>
   <head>
      <title>Useful things</title>
   </head>

   <body>
      <p>Some <em>really</em> useful things:</p>
      <ul>
         <li>coffee, in the morning</li>
         <li>the wheel</li>
         <li><a href="google.com">Google</a></li>
      </ul>
   </body>
</html>
```

Figure 2.1: Example of an HTML document.

In its early versions, some tags did not require an matching closing tag. Instead, they were implicitly terminated when the enclosing tag closed or, in the case of `<p>` tags, when another `<p>` tag was defined. This caused some

confusion and several incompatibilities between browsers, which ultimately led to more strict rules and the creation of the eXtensible HyperText Markup Language (XHTML) [24]. In valid XHTML, every tag must have a matching closing tag or, in case of the self-contained tags, represented by a new syntax: `<br/>`, `<img/>` and `<hr/>`.

The current version of HTML is 4.01 [15]. Despite still being considered a Working Draft by the World Wide Web Consortium (W3C) [45], HTML version 5 [13] has been seeing increasing adoption, specially in mobile phones.

2.2 Document Object Model (DOM)

The Document Object Model (DOM) [26] is a standard created by the World Wide Web Consortium (W3C) [45] to represent HTML, XHTML and XML documents such as webpages. It doesn't define only a conceptual model, but a language-independent convention to represent and manipulate documents.

A document is seen as a rooted tree where each tag or element is represented by an *element node*. Other elements defined inside of it are represented as child nodes, thus recreating the hierarchy present in HTML documents. Text contained in these elements is placed in *text nodes*, which are leaves of this tree.

Other types of nodes exist, such as comment and attribute nodes, but won't have any particular impact on our approach. We will be focusing on the text nodes, as that is where the content lies.

In Figure 2.2 we see an example of how a snippet of an HTML document is represented as a DOM tree.

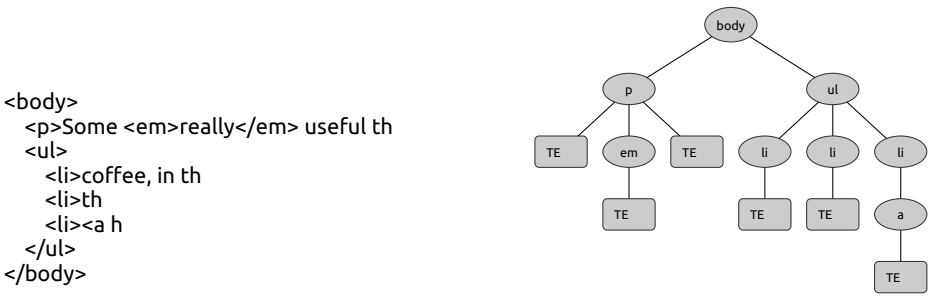

Figure 2.2: Example of a DOM tree representation of HTML documents

2.3 Cascading Style Sheets (CSS)

Cascading Style Sheets [4] is a language used to describe presentation attributes of HTML and XML documents, such as font face, color and size; absolute and relative positioning; etc. It allows the separation of content and presentation in a webpage and the use of several style sheet definitions to be combined in a predictable (or cascade) manner.

It was created in 1996, with the latest revisions dating from 2008. Over the years, it has seen wide adoption by most web sites as the browsers started providing support for it and presentational elements were deprecated from the HTML specification [15, 38]. All current major browsers have a complete implementation of CSS Level 1 and a near-complete implementation of CSS Level 2.1 [8, 27, 37, 44]. CSS Level 3 is currently being worked on [7], and partial implementations of it can be seen in modern browsers, mobile phones and tablets.

A style sheet consists of a list of *rules*. Every rule can be broken down into two parts: one or more *selectors* and a *declaration block*. The selectors, as the name suggests, describe to which nodes of the DOM tree a rule will be applied. A selector can describe nodes in various ways: by tag name, style class, element id, relative and hierarchical positioning, etc. Declaration blocks are much simpler: they consist of a list of *property name* and *value* pairs. There is a long list of properties, which can be used to define font size, font color, bold text, margin, element border, positioning, etc. An example of how a style sheet document looks like is available in Figure 2.3.

```
p {
   font-family: sans-serif;
}

li {
   font-family: cursive;
   list-style-type: lower-roman;
}
```

Figure 2.3: Example of a style sheet document.

Rules are applied by increasing level of *specificity* and, in case of ties, by declaration order. This means that rules describing a node by its element id (which is unique throughout the document) will be applied last, thus overwriting any previously applied rules. Another example is an hierarchical selector, which can be written to mean "apply to every `p` node that is a

descendant of a `div` node"; these are applied after more generic rules such as one that matches all `p` nodes. We can see a document with and without style sheets applied in Figure 2.4.

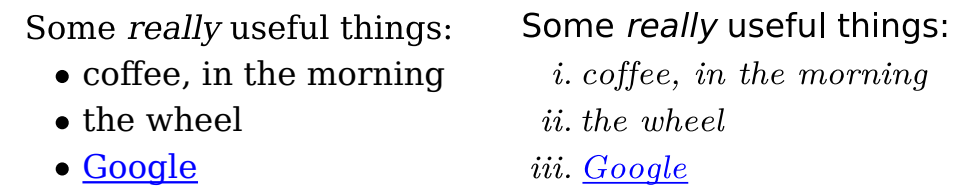

Figure 2.4: Example of a document's presentation before and after applying some CSS rules.

2.4 Machine learning methods used

In this section, we will go over some machine learning methods used in this work. Our approach makes use of an ensemble of decision trees and the work of Wang, et al. (2009) [40, 41], which we implemented for comparison in Chapter 4, uses Support Vector Machine (SVM).

2.4.1 Decision trees

In this work we used classification decision trees as our machine learning technique. Decision trees split the training dataset recursively by the attribute that best differentiates the given examples. For instance, if given a set of weather prediction examples, it might be easier to differentiate rainy and sunny days by separating examples based on the values of a feature describing a cloudy day or clear sky. Naturally, the classifications won't be perfect with only this step as not all cloudy days are also rainy, but other features could then be used on these subsets until we can reach satisfactory results. An example of a decision tree is provided in Figure 2.5.

In more formal terms, decision trees choose, at each node, the feature that provides the higher *information gain*. For each subset of training examples created by the first node, we will create a child node and reapply the algorithm until all examples in that subset belong to the same class. However, it might not be a good idea to perfectly separate all examples since it may cause *overfitting*; that is, rules become too specific and do not generalize well to unseen examples, thus providing incorrect classifications. To avoid overfitting it is common to *prune* a tree. This pruning can be made by various criteria: by the depth of tree, by the size of the current subset of examples, etc.

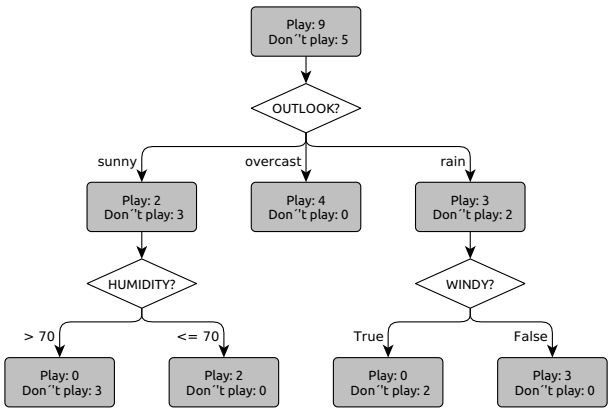

Figure 2.5: Example of a decision tree to decide if the weather is good to play outside.

Several algorithms for the creation of decision trees exist. They differ in how they choose the best separating feature and how they prune the tree. Throughout this work we will be using the REPTree algorithm to construct our decision trees, in an implementation provided by the Weka tool [12], since it presented better results without the need for tweaking parameters to our features and datasets.

2.4.2 Ensemble models

Ensembles consist of a set of machine-learning models, which can be combined to provide a classification that is often better than that of a single model. They can be used in various ways, with various objectives. For instance, a technique called *boosting* successively trains models that favor instances that have been misclassified so far. The end result would be a set of models that is able to identify the easy instances, while having some degree of specialization for the classification of harder instances.

In our approach, we have been using the *bagging* technique, which consists of training a set of models, usually with random subsets of the training instances, and considering the classification given by each of them as a vote. All votes have the same weight, and the most common classification is elected as the final classification of the ensemble model. We chose bagging because of its simplicity and the improvement observed in our results.

2.4.3
Support Vector Machine (SVM)

Support Vector Machine (SVM) [6] is a technique for binary classification of examples. Given a set of points in a plane, we calculate the hyperplane that best separates points of one class from the other. In this work we will not go into details of how to calculate this hyperplane, but an illustration of how it can separate data is available in Figure 2.6.

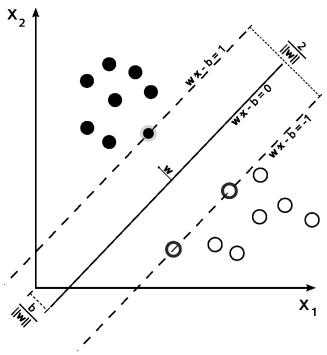

Figure 2.6: Illustration of a hyperplane that linearly separates points from two distinct classes.

Sometimes the example data cannot be linearly separable; that is, there is no hyperplane that can separate points of both classes. To solve this problem, we can apply the *kernel trick* [1, 2], which effectively maps the problem to a higher-dimensional space where points can be linearly separable. An example of the effects of the kernel trick on a set of points is illustrated in Figure 2.7.

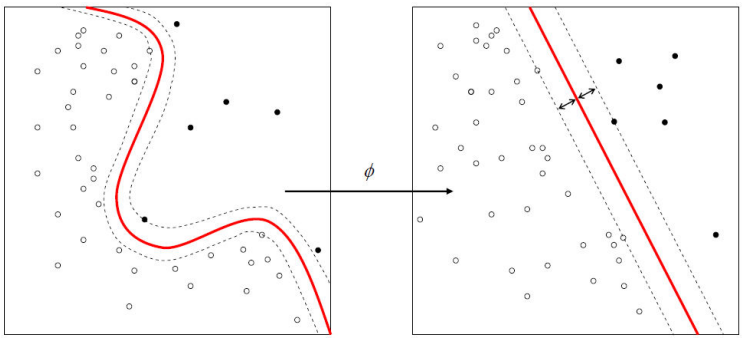

Figure 2.7: Illustration of how the kernel trick can affect the data points.

2.5
Metrics

The output of our algorithms is a set of DOM text nodes, from which we can extract the textual content. To assess our results, we have been using two

different metrics: the first is node-based and the second uses bag of words. A discussion on the advantages and disadvantages of each of them is provided in the following subsections.

The final user application should dictate which metric to use. In this work, unless otherwise noted, all results reported use the bag of words metric.

For evaluating both methods, we use the concepts of *precision*, *recall* and *F-measure*. A high value for precision indicates few wrong classifications, while a high value for recall indicates most of the annotated data is retrieved. They are calculated for each class (title, date, etc.) as follows:

$$\text{precision} = \frac{\#correctly\ classified\ elements}{\#total\ classified\ elements}$$

$$\text{recall} = \frac{\#correctly\ classified\ elements}{\#total\ elements\ in\ class}$$

It can be tricky to improve the results on these metrics: to improve recall, more elements may be classified; however, the more elements classified, the smaller the precision obtained.

As these two measures are often reported together, it is common to use the harmonic mean of both measurements, called F-measure or F-score. Here, we will be working with the *F1-score*, which means precision and recall are given the same weight. To calculate the F1-score, we use the following formula:

$$\text{F1-score} = \frac{2 \times precision \times recall}{precision + recall}$$

Throughout this work, unless otherwise noted, precision and recall is reported as the average from all documents in a corpus, with the F1 calculated from these averages.

2.5.1 Node-based

Since we are dealing with a DOM tree, it is natural and convenient to think of a node-based metric. That is, a DOM node becomes the smallest unit of information. However, this may not always be the best solution.

One issue we found is that all nodes have the same weight when calculating precision and recall, regardless of their content. This means that small nodes with little information are as valuable as large nodes with lots of information. While they could be weighted by character size or word count, small nodes might contain crucial information such as the publication date or the news title. Consider the case of news titles: detecting a node that does not

represent the exact news title would be a miss, but its contents might highly resemble the title, which would make it valuable in a search engine setting. Likewise for dates: detecting a node that only partially contains the date, for instance the day and month of publication, but not the exact time, would have a high impact on the precision and recall metrics, but may do no harm for a given application that disregards such time information.

Another possible issue is how precise we require the matches to be. If the text under a given node and its parent node is exactly the same, which one is to be considered as the date? In theory, since both have the same contents, they should both be a valid answer, but it is possible an application might require more accuracy than others. Because our interest is in the data, rather than the presentation, we prefer a metric more focused in the content, such as bag of words.

2.5.2 Bag of Words

We employ a metric based on bag of words for measuring the results of our extraction, turning words into the smallest unit of information measured. In this approach, text is represented as a collection of unordered tokens (referred to as words) which may appear repeatedly. Words are often defined as a contiguous sequence of either non-whitespace characters or characters from a given set (for instance, letters and numbers, but no punctuation). Neither approach for tokenization requires linguistic information for this task, which does not invalidate the language independence of our method for languages that make use of the Latin alphabet. This is because letters from A to Z are always encoded the same – the problem lies with diacritics and other alphabets such as cyrillic, hangeul, etc., which often have their own encoding – and some non-latin languages do not require the use of whitespaces and thus are harder to tokenize, as is the case with Japanese.

We prefer the bag of words approach because we have found that it more accurately represents what a human would consider as a correct or incorrect extraction of content. The issues present on the node-based approach are not completely absent, but smoothed out in a way that they cause less negative impact for an imperfect match.