BER$_y$L: A Unified Approach to Web Block Classification

Andrey Kravchenko

Magdalen College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2015

Abstract

Web blocks such as navigation menus, advertisements, and headers and footers are key components of web pages which define not only the appearance of a web page but also the way in which humans interact with different parts of the page. For machines, however, classifying and interacting with these blocks is a surprisingly hard task. Yet, web block classification has varied applications in the fields of wrapper induction, assistance for visually impaired people, mobile web browsing, web page topic clustering and web search. Our system for web block classification, BER$_y$L, performs the automated classification of web blocks through a combination of machine learning and declarative, model-driven feature extraction based on Datalog rules. BER$_y$L uses refined feature sets for the classification of individual blocks to achieve accurate classification for all of the block types that we have so far observed. The high accuracy is achieved through these carefully selected features. Some are even tuned to the specific block type. At the same time, BER$_y$L avoids the high cost of feature engineering through a model-driven rather than programmatic approach to feature extraction. Not only does this reduce the time for feature engineering, the model-driven, declarative approach also allows for semi-automatic optimisation of the feature extraction system. BER$_y$L also employs a holistic approach to web block classification where individual blocks are considered within the context of a web page powered by the knowledge representation rules specific to that domain. We validate these claims for a broad range of web blocks in an extensive evaluation.

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Georg Gottlob for his suggestions, ideas, and feedback throughout my years of research. I would also like to express my sincere gratitude to my college supervisor and ex-tutor Professor Oege de Moor for teaching me from my first year as an undergraduate, for encouraging me to undertake research in Computer Science, for showing to me the limitlessness of the subject, and for his constant support and advice throughout my time at Oxford. I would like to thank my colleagues on the DIADEM and VADA projects: Dr Giovanni Grasso, Dr Giorgio Orsi, Dr Christian Schallhart, Dr Emanuel Salinger, and Dr Ruslan Fayzrakhmanov and my fellow DPhil students Cheng Wang, Omer Gunes, Xiaonan Guo, and Andrew Sellers for their collaboration and friendship. I am also thankful to Professor Thomas Lukasiewicz and Dr Alessandro Provetti for kindly agreeing to be my DPhil examiners and their constructive comments on the thesis. Finally, I want to thank my parents, my girlfriend, and my friends in Oxford, London, and Moscow for supporting and caring for me all these years.

Contents

List of Figures

Chapter 1

Introduction

When a human looks at a web page, he or she sees a meaningful and well-structured document, in which the semantics of different functional blocks and elements is mainly defined by their layout and the textual content. However, such interpretation is not accessible for the computer, and it only "sees" the technical layers of the web page [36] represented merely by the source code (e.g., HTML, CSS, and JavaScript files) and the rendered models (e.g., the DOM tree, CSSOM with computed attributes, and executed JavaScript). Whilst it is probably infeasible for a machine to replicate the human's perception and derive his or hers mental model, it would be highly useful for it to understand the logical structure and functional role of various elements of the web page for a wide range of different applications through the analysis of the layout, as well as visual and textual features. Web search is an especially important potential application, since semantic understanding of a web page allows the restriction of link analysis to clusters of semantically coherent blocks. Hence, we aim to build a system which provides a functional and semantic understanding of web pages.

We are primarily concerned with the task of web block classification. Informally speaking, a web block is a logically consistent segment of a web page layout, an area which can be identified as being visually separated from other parts of the web page. A web block carries a certain semantic meaning, such as title, main content, advertisement, login area, footer, etc. It is through the semantic meaning of individual web blocks that a human understands the overall meaning of a web page. There are many blocks with a common semantic meaning (i.e., a layer of web specific objects [36]) among different websites and domains (e.g., headers, navigation menus, logos, pagination elements, and maps) that share common web patterns. This diversity of blocks makes the task of their accurate and fast detection challenging from a research and implementation perspective. In general, the difficulty of the block classification problem lies not only in the complexity of individual classifiers, but

also in the complexity of the entire system which needs to balance the individual accuracies of its constituent classifiers and its overall performance.

There are several important applications of web block classification. One is automatic and semi-automatic wrapper induction [5, 23, 55, 64], as the ability to identify blocks such as navigation menus and pagination bars will automatically provide the generated wrappers with a reliable way of crawling through the pages of a website. Another application is to assist visually impaired people with navigating the website's internal content [28]. For example, a system which is able to accurately identify navigation menus will be able to read aloud to a visually impaired user all of the options that they may select to get from the current page to other web pages of the same website. Web block classification can also help in the task of mobile web browsing [4, 28, 43, 47, 57, 58]. For example, if we want to project only the main content of a web page onto a user's mobile screen, an accurate domain-independent block classifier will be able to solve that task. Currently, most extraction systems attempt to solve the problem of eliminating irrelevant advertisements by keeping manually written block lists of URLs. However, this approach has a significant weakness since designers can simply change the domain names from which the advertisements originate, thereby avoiding the block lists. On the contrary, a classifier which is able to distinguish advertisements from non-advertisements based on structural and visual features would be much more robust than a naïve URL stop-list approach. We may also use a block classification system for web page topic clustering [42], e.g., in a system which can distinguish real estate input forms (an example of a search request in such a form would be "All houses in Oxford, at most 5 miles away from the city centre, priced from £300,000 to £1,000,000") from all other types of forms. We can then use the presence of a real estate input form on a certain web page as a strong indicator that this web page belongs to the real estate domain. In a similar way, a web block classification framework can be used to enhance the performance of focused crawlers. Finally, web block classification can be useful for the ubiquitous task of a web search, as demonstrated by [11, 12, 56, 62]. Rather than performing the search and the inter-link analysis on individual pages, the enhanced web-search system would be able to search for specific blocks and would improve the quality of the extracted information by running a page-to-block analysis and a block-to-page analysis rather than a page-to-page link analysis.

Our web block classification system is known as BER$_y$L (**B**lock classification with **E**xtraction **R**ules and machine **L**earning). BER$_y$L is a module of DIADEM (**D**omain-centric **I**ntelligent **A**utomated **D**ata **E**xtraction **M**ethodology)[1], a large-scale domain-dependent system for data extraction [21] which interacts with its other modules OPAL (**O**ntology

[1] http://www.cs.ox.ac.uk/projects/DIADEM/

based web **P**attern **A**nalysis with **L**ogic) and AMBER (**A**daptable **M**odel-**B**ased **E**xtraction of **R**esult pages) which are employed for understanding web forms and analysing result pages.

In the context of this work, the efficiency of web block classification is measured by five main parameters: **(1)** *precision*, the percentage of true positives among the nodes identified with a certain block type, **(2)** *recall*, the percentage of nodes identified with a certain block type among all nodes that are actually of this block type (and thus lower recall means more false negatives), **(3)** F_1, the harmonic mean of precision and recall, **(4)** *accuracy*, the percentage of nodes correctly identified as true positives or true negatives among all nodes classified, and **(5)** *performance* which measures the time in milliseconds it takes for BER$_y$L to perform the task of web block classification with fixed computing power.[2]

A considerable amount of research has been undertaken in the field of web block classification [9, 13, 16, 25, 31, 32, 34, 37, 39, 41, 43, 50, 53, 55, 59, 60, 64, 65]. However, most of the approaches taken in this research have attempted to classify a relatively small set of domain-independent blocks with a limited number of features, whereas we aim to develop a unified approach to classify both domain-dependent and domain-independent blocks. Furthermore, none of the papers with which we are familiar discussed the extensibility of their approaches to new block types and features.

Finally, although most of these machine learning-based approaches require significantly more training than BER$_y$L, they usually reach a precision level below 70% [25, 31, 34, 37, 43]. The highest value of F_1 that any of these machine learning-based approaches reach is around 85% [43]. An intensive evaluation of our BER$_y$L system (Chapter 6) has shown that it can achieve much higher levels of precision and recall. This can partly be explained by the fact that other approaches have attempted to classify different blocks with the same set of features, whereas our approach employs individual feature sets for different types of blocks.

Although most of the current block classification approaches are based on machine learning methods, to our knowledge there is no universal approach to domain-dependent and domain-independent block classification.

Challenges of the web block classification problem The challenges of the current web block classification approaches can be divided into three main categories:

1. they cannot address a large variety of blocks;

[2]Precision = TP/(TP+FP), Recall = TP/(TP+FN), Accuracy = (TP+TN)/(TP+FP+TN+FN), $F_1 = 2 \times \text{Precision} \times \text{Recall} / (\text{Precision} + \text{Recall})$, where TP means true positives, FP means false positives, FN means false negatives, and TN means true negatives.

2. domain-independent methods [32, 41, 43] run in performance limitations;

3. domain-dependent methods [9, 37, 55] have proven too costly in terms of supervision.

A detailed analysis of three seminal state-of-the-art classification methods [32, 37, 43] showed that none of these approaches employ domain-specific knowledge, that they all use a single set of features, and that they cannot perform fine-grained block classification. It is partially due to these reasons that state-of-the art approaches cannot achieve the necessary accuracy of classification on wide ranges of blocks.

System requirements Therefore, BER$_y$L must meet three main requirements:

1. it must be able to cover a diverse range of domain-independent and domain-dependent blocks;

2. it must achieve acceptable precision and recall results for each individual block in the classification system, and maximise the overall performance of the classification system;

3. it must be adaptive to new block types and domains.

As discussed above, the failure of state-of-the-art classification systems in terms of covering diverse sets of block types can be partially explained by the fact that they all attempt to classify these blocks with one single set of features.

We thus aim to build a system which can handle block-specific features. We would also like to introduce features for new block types whilst sacrificing as little automation as possible. This leads us to a separation of BER$_y$L into two general components:

1. the classification of individual web blocks;

2. the large-scale classification of domain-independent web blocks.

For individual blocks, we can define the features and build the training corpus manually. However, outliers are very common for many block types (such as pagination bars), and we must define highly block-specific features in order to take them into account.

Example 1. The pagination bar in Figure 1.1 has two "next" links, one numeric and one non-numeric, identified as "4" and "›" respectively. In order to identify the whole block as a pagination bar, we need to know what the "next" links are. If we want to identify the non-numeric "next" link, we do so by looking at the links that contain non-numeric "next"

Figure 1.1: An example of a pagination bar and its sub-component 'next' links

link annotations such as "›" or "»". The "last page" link denoted by "»" is also present in this pagination bar. Since the "next" links can also be denoted by the "»" symbol, the only way to distinguish between the "next" and "last" links in this case is to look at whether the candidate link contains a link with "next" annotation to its visual left. This is not a straightforward feature, however, there is no way in which an accurate classification of the "next" links in the outlier pagination bars can be performed without it. The general problem of an entirely manual approach to web block classification is that, for many block types, we must generate very specific and complicated features. Some of these blocks are highly diverse in their visual and structural layout, which requires a great deal of data to build the training corpus, which in turn involves increased human effort. We aim to automate this process as much as possible. Hence, this is one of the problems of building a large-scale classification system for domain-independent and domain-dependent blocks.

In the case of large-scale classification for domain-independent blocks (such as navigation menus or advertisements), the significant number of blocks (many of which have diverse structural and visual representations) makes the set of potential features so large that the feature extraction process can become a bottleneck in the system. This will lead us either to contract the feature space or to optimise the performance of the feature extraction process. We also aim to employ techniques for minimising the training data (such as semi-supervised learning) as the number of labelled instances required for the classifier to work can be very large, even with a diverse and representative set of features.

Example 2. The following is an example of an application of the large-scale classification of domain-independent web blocks. If we want to extract the office locations for a very large number of different websites (e.g., the current DIADEM system supports around 100,000 websites) in different domains (such as real estate, used cars, retail stores, restaurants, etc.), most locations are given at the footer of each website. Therefore, we can assume with a strong degree of certainty that, if we have a robust classifier for footers, we can extract the correct office locations from the large majority of websites, irrespective of to which domains these websites belong. In addition, most of the features of the footer classifier can be borrowed from the global feature repository (such as the width and height of the respective CSS box) or instantiated from the template relations (e.g., we can check

whether the candidate for the footer block contains copyright information in it by using the aggregate annotation span relation with the respective instantiation).

For both domain-dependent and domain-independent blocks, we aim to combine human-generated rules (e.g., "floor plans can only appear on real estate result pages") and the training data. If the training data strongly contradicts the rule, we can then remove that rule from BER$_y$L.

The task of web block classification is technically challenging due to the diversity of blocks in terms of their internal structure (their representation in the DOM tree and the visual layout) and the split between domain-dependent and domain-independent blocks. Hence, from a technical perspective, it is important to have a global feature repository which can provide a framework for defining new features and block type classifiers through the instantiation of template relations. A BER$_y$L user will be able to extend it with new features and classifiers with ease by generating them from existing template relations, rather than writing them from scratch. This will make the whole hierarchy of features and classifiers leaner and the process of defining new block types and their respective classifiers more straightforward and less time-consuming. Ideally, we aim to generate new block classifiers in a fully automated way such that, given a set of structurally and visually distinct web blocks of the same type, the block classification system would be able automatically to identify the list of optimal features for describing that block, taking some of these features from the existing repository and generating new ones which did not exist in the repository beforehand. However, this approach would almost be infeasible in the case of BER$_y$L since the diversity of block types that we want to classify is likely to cause the space of potential features to be extraordinarily large, if not infinite. Hence, we need to limit the approach of the generation of new features and block type classifiers to a semi-automated approach.

Contributions of BER$_y$L The contributions of our approach in BER$_y$L have two main aims: improving the quality of classification and minimising the effort of generating new classifiers (as explained in the above paragraph). Contributions 1-3 and 4-6 refer to the quality of classification and generation aspects respectively.

1. We provide a holistic view of the page which gives a coherent perspective of the way in which it is split into web blocks and the way in which these blocks interact.

2. We employ domain-specific knowledge to enhance the accuracy and performance of both domain-dependent (e.g., price, location, and floor plan images in the real estate domain) and domain-independent (e.g., navigation menus) classifiers.

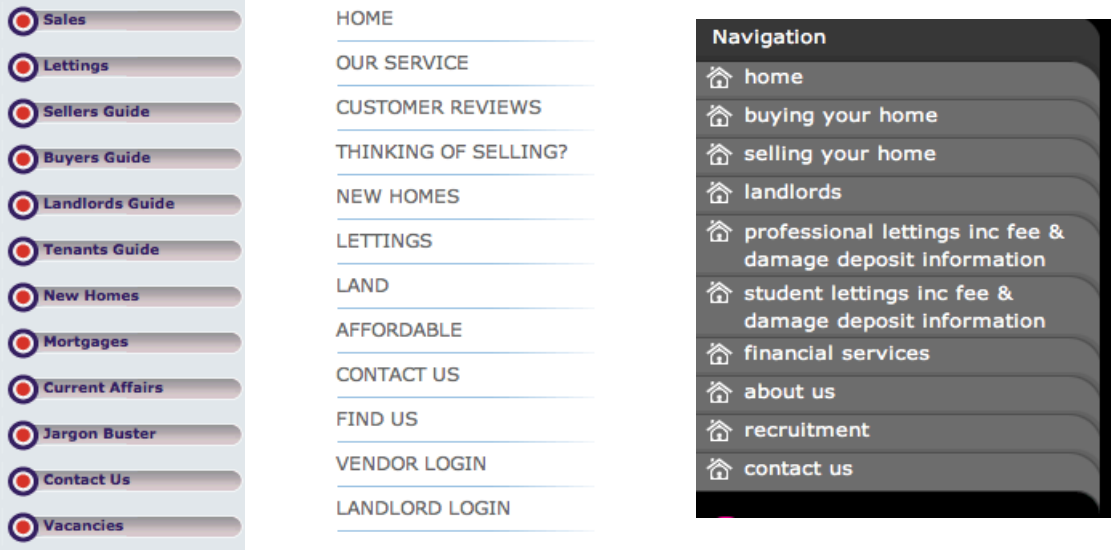


Figure 1.2: Navigation menus contained within sidebars

3. We provide a global feature repository which allows users of BERyL to easily add new features and block classifiers.

4. The holistic view of the page is implemented through a system of mutual constraints.

5. We encode domain-specific knowledge through a set of logical rules; for example, that the navigation menu is always at the top or bottom of (and rarely to the side of) the main content area of the page for the real estate and used cars domains. Alternatively, e.g., for the real estate domain, the floor plans can only be found in the main content area of the page.

6. The global feature repository is implemented through baseline global features and template relations used to derive local block-specific features. We also use textual annotations from the global annotation repository as features for web block classification in order to further enhance the accuracy of individual classifiers.

As explained above, we distinguish between domain-independent and domain-dependent classifiers. Domain-independent classifiers typically determine blocks which have a certain function; however, in some cases they can also determine blocks with clearly semantic meanings. On the other hand, it is almost always the case that domain-dependent blocks are semantic rather than functional. A domain-independent block can be a part of another one, connected through a *part-of* relation.

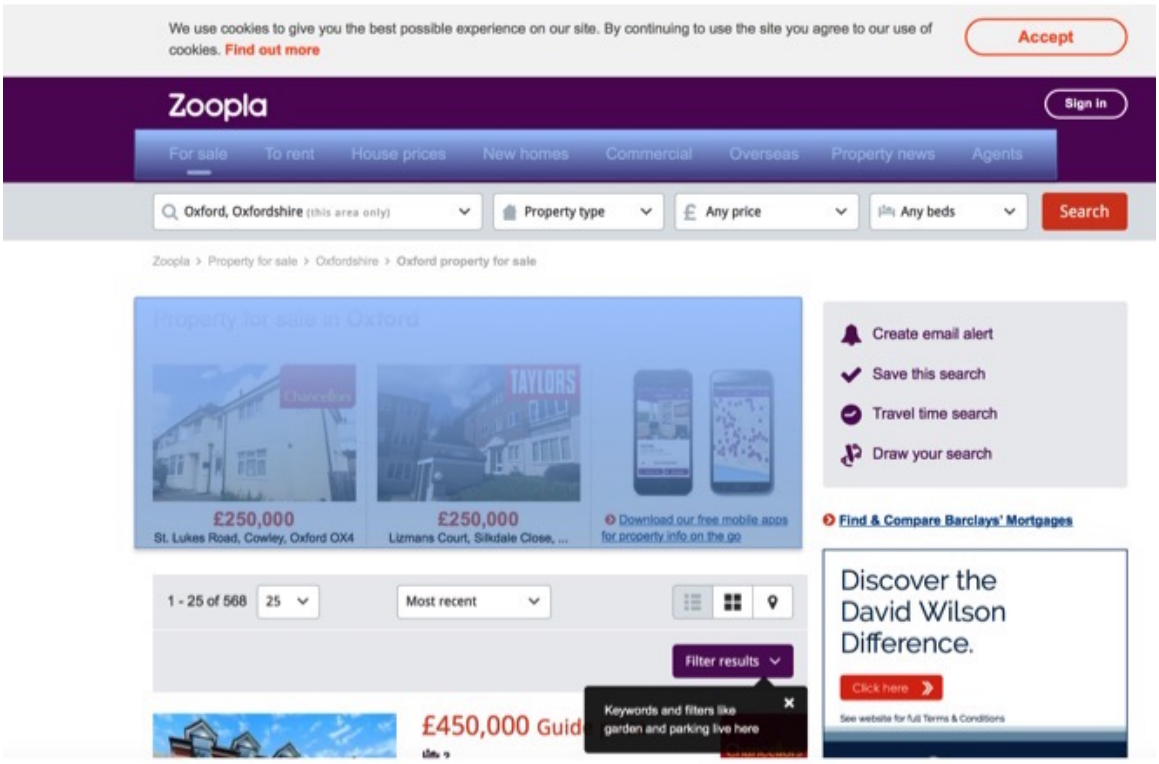


Figure 1.3: Mutual interaction between the block classifiers

Example 3. For example, each of the sidebars on the real estate websites in Figure 1.2 contains a navigation menu. Both of these blocks are domain-independent. However, the sidebar is a functional block as it does not determine the contents of what is contained inside it (e.g., it can also contain an advertisement), whilst the navigation menu is a block with well-defined semantic meaning.

We provide an example of how a holistic view of a page can enhance the accuracy of classification for individual blocks as well as the overall performance of BER_yL in Figure 1.3.

Example 4. A system consists of two classifiers – headers and navigation menus – and the navigation menu classifier has much higher accuracy than the header classifier. The individual classifiers determine the block highlighted in blue at the top of Figure 1.3 correctly as a navigation menu and the block highlighted in the middle as a header. However, our knowledge tells us that, in all domains, a navigation menu is located either in the header, just below the header, or in the sidebar. This constraint is violated by the given classifications and, therefore, because we trust the navigation menu classifier more, we can remove the classification of the header from the output as a FP.

With respect to the employment of domain-specific knowledge for enhancing the performance of classifiers (contributions 2 and 5), our new approach to feature extraction allows us to easily integrate domain-specific pre- and post-classification filters which ensure that the classifiers in question meet all of the additional restrictions imposed by the domain in which they are applied. This approach may be used for both domain-dependent and domain-independent classifiers, but is of particular evaluative interest for domain-independent classifiers, such as pagination links and navigation menus. We discuss our implementation of the filtration rules in more detail in Chapters 3 and 4 of this report.

We have also implemented a global feature repository (contributions 3 and 6) which is crucial to the automation and large-scale evaluation of the BER$_y$L system. Our system supports baseline global features which can be shared between different classifiers. The system also includes global features which can be instantiated with parameters and connected to textual annotations from the global annotation repository. This approach allows us to create feature template relations and their specific instantiations by labelling some of the parametrised global features as template relations and their evaluation with concrete parameters as instantiations of those relations. We discuss the new framework for the global feature repository of BER$_y$L in Chapters 3 and 4 of this report.

The creation of the global feature repository has allowed us to implement stages 2 and 3 of the BER$_y$L system (the large-scale classification of domain-independent web blocks and the large-scale classification of domain-dependent web blocks). In particular, it has allowed us to easily extend the system with new classifiers without reducing the accuracy of the classifiers that are already present or significantly reducing the performance of the system.

Our verification framework allows for the fast creation of comprehensive and robust evaluation corpora in order to measure the accuracy of individual classifiers, their individual performance rates, and the performance rate of the whole system. This helps us to test our hypotheses for stage 2 of the system.

Declarative approach to feature extraction In BER$_y$L, we use a declarative approach to feature extraction wherever possible, since that **(1)** allows us to combine it with relations and global features which provide a succinct representation of the current feature set. This, in turn, allows us to simplify the definition of new features through the employment of existing global features or relation instantiations and to learn new features by automatically finding the right combination of parameters for relation instantiations. **(2)** It is also much easier to learn Datalog [1, 18] predicates automatically than to learn procedural language programs, which is likely to come in useful in the large-scale domain-dependent block

classification stage of BER$_y$L when we will have to infer new features automatically. In other cases which require the use of efficient libraries and data structures or intense numerical computation (e.g., features acquired from image processing), we employ a procedural approach for feature extraction implemented through Java.

Web block classification It does not seem feasible to solve the block classification problem through a set of logic-based rules, as it is often the case that there are many potential features which can be used to characterise a specific block type. However, only a few play a major role in uniquely distinguishing this block type from all others. Some of these features are continuous (e.g., the block's width and height) and it can be difficult for a human to manually specify accurate threshold boundaries. Hence, for BER$_y$L, we decided to use a machine learning (ML) approach to web block classification.

Scientific novelty of our approach The main scientific novelties of our approach in BER$_y$L based on combining machine learning with declarative-driven feature extraction are the following:

1. the enhancement of ML-generated web block classifiers with domain-specific knowledge improves the accuracy of classification;

2. there is a substantial improvement in the manual effort of defining new features acquired through the introduction of relation instantiations and the global feature repository;

3. the mutually interconnected system of classifiers which is built on a holistic view of the page improves the accuracy of classification and the overall performance of the web block classification system;

4. there is a substantial improvement in the performance of feature extraction acquired by relation instantiations and the global feature repository.

Chapter 2

Problem Definition and Architecture

In this chapter, we define the problem of web block classification and outline the architecture of the approach taken in BER$_y$L to solve that problem.

2.1 Problem definition

The first question to address in terms of building a block classification system is how to define "web block". For example, on the web page presented in Figure 2.1, it is obvious that the large bar at the top of the page is a navigation menu, but what about the block on the left-hand side? Do we consider it a navigation menu, a sidebar, or both? In addition, do we classify the bar at the bottom as a navigation menu or a footer or a navigation menu inside a footer? The answers to these questions are not obvious. Hence, we aim to employ human annotators to give their definitions of the block types present on a page and then build an ontology of block types based on a synchronised view of their definitions.

We give an informal definition of a *web block* as a part of a web page which has a semantic meaning which is different to other parts of that web page and/or which has a certain role within the overall structure of the page.

Example 5. A "next" link is a block of the first type as it conveys a certain semantic meaning. A sidebar is a block of the second type since it serves as a structural element of the page and can either be an advertisement or a navigation menu.

To define the underlying web block classification problem formally, we need to capture both how a "part of a web page" and a "semantic meaning" are represented. The possible semantic and functional roles of blocks are characterised by *labels* such as `numeric_next_link` or `nav_menu`. We distinguish between *positive classification labels* $L^+ \in \mathbb{L}^+$ and *negative classification labels* $L^- \in \mathbb{L}^-$. A block labelled with a positive label holds a certain semantic or functional meaning, such as `sidebar` or `header`. A block

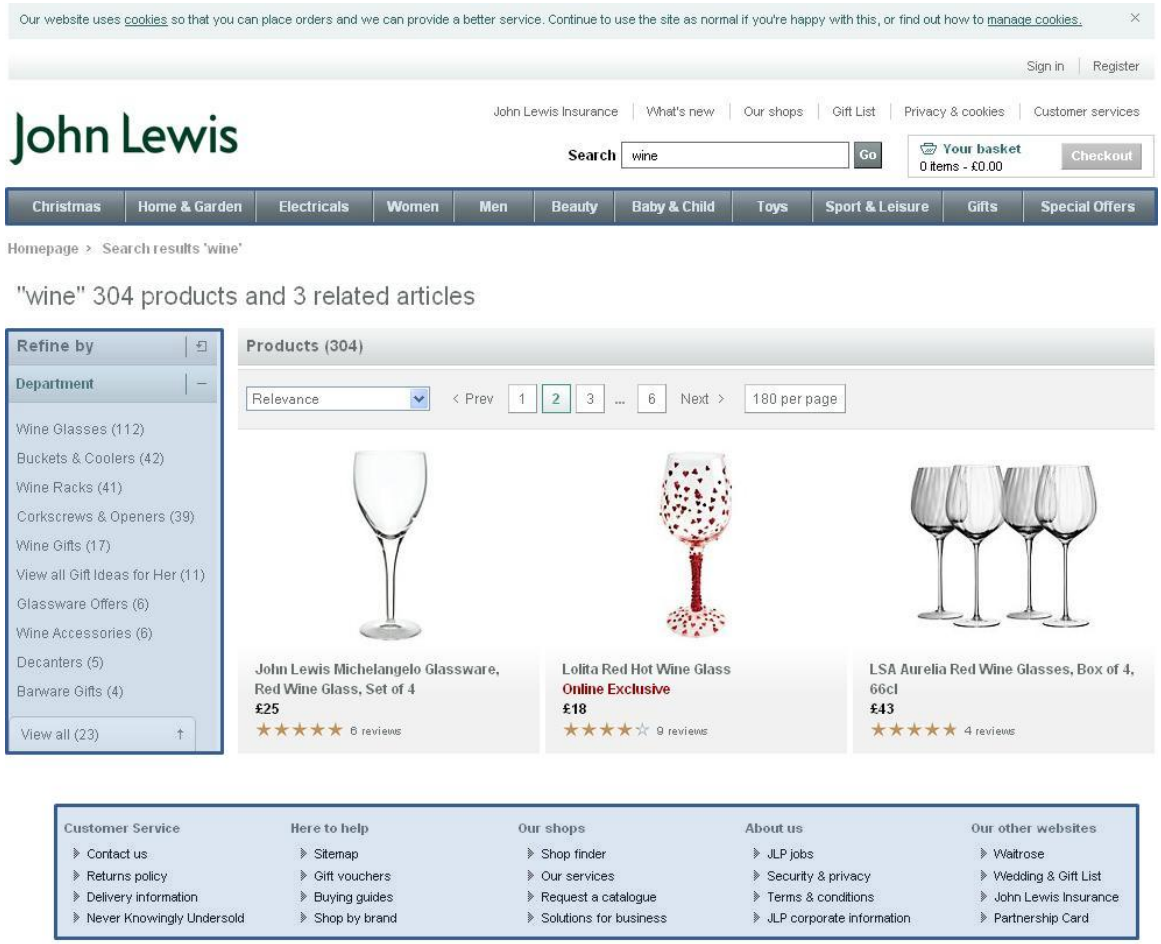


Figure 2.1: An example of ambiguity that can arise when defining block types

with a negative label, such as `non_sidebar`, `non_header`, or `non_next_link`, *does not* hold a certain semantic or functional meaning. Note that $\mathbb{L}^+ \cap \mathbb{L}^- = \emptyset$. The union of $\mathbb{L}^+ \cup \mathbb{L}^- = \mathbb{L}$ forms a set of all possible labels $\mathbb{L}$, which we refer to as a *global labelling schema*.

We also distinguish between different block types. Let $\mathbb{BT}$ be the set of all block types. Then, for each block type $BT \in \mathbb{BT}$, there is a non-empty set of positive labels L^+_{BT} and a non-empty set of negative labels L^-_{BT}. $L_{BT} = L^+_{BT} \cup L^-_{BT}$ is the set of all labels for BT. These sets of labels are disjointed among the BT, i.e., for all $BT, BT' \in \mathbb{BT}$, $L^{\theta}_{BT} \cap L^{\theta'}_{BT'} = \emptyset$ with $\theta, \theta' \in \{+, -\}$.

Example 6. For the `PAGINATION_LINK` block type, the set of positive labels is $L^+_{\texttt{PAGINATION\_LINK}}$=`{numeric_next_link, non_numeric_next_link}` and the set of negative labels is $L^-_{\texttt{PAGINATION\_LINK}}$=`{non_next_link}`.

Parts of web pages are represented in the following as DOM tree fragments.

Definition 1. Let P be a web page. Then, the *DOM tree* of that page represents the hierarchy and order among the HTML elements, attributes, and text fragments on that page. Specifically, a DOM tree T_P is a tuple $(\mathcal{N}, \text{CHILD}, \text{NEXT}, \text{ATTR}, \text{CSS}, \tau, \sigma)$ such that

- $\mathcal{N}$ is the set of all DOM nodes in T_P;
- CHILD, NEXT, ATTR, CSS are four binary relations among DOM nodes which capture the structure of the tree;
- τ is a (partial) mapping from DOM nodes to "tags" such as `table` or `div`; and
- σ is a (partial) mapping from DOM nodes to string values associated with that node, e.g., the content of a text fragment or the value of an attribute or CSS property.

There are a number of structural constraints on the tree. Let e be a DOM node.

- All four relations are irreflexive and anti-symmetric.
- The CHILD relation forms a tree.
- A node n with $(e,n) \in \text{CHILD}$ and $\sigma(n) = v$ is referred to as a *text node* with value v. A text node may not occur in the domain of CHILD, CSS, or ATTR.
- A node n with $(e,n) \in \text{CSS}$, $\tau(n) = t$ and $\sigma(n) = v$ is called a *CSS node* with property name t and value v. A CSS node may not occur in NEXT, ATTR or CHILD, nor in the domain of CSS.
- A node n with $(e,n) \in \text{ATTR}$, $\tau(n) = t$ and $\sigma(n) = v$ is referred to as an *attribute node* with tag t and value v. It may not occur in CHILD, or CSS.
- A node n with $\tau(n) = t$ and $(e,n) \in \text{CHILD}$ or $(n,e) \in \text{CHILD}$ is referred to as an *element node*. Element nodes may not occur in the range of ATTR or CSS and have no associated text value.
- The NEXT relation forms an acyclic sequence over all element and text nodes.

In the following section, we abstract from text content values and move on to annotations, which represent occurrences of specific entities in the text content.

Definition 2. An *annotated DOM* is a DOM tree T_P decorated with annotation types from an *annotation schema* $\Lambda = (\mathscr{E}, \mu)$ where $\mathscr{E}$ is a set of entity types and $\mu : \mathscr{E} \times \mathcal{N} \times \mathscr{U}$ a relation on $\mathscr{E}$, the set of DOM nodes $\mathcal{N}$ and the union of all domains of entity types $\mathscr{U}$. $\mu(A,n,v)$ holds if n is a text node containing an instance of entity type A and the normalised representation of that instance is v.

Typically, an annotated DOM is obtained from a plain DOM through the use of a battery of (named) entity recognisers. This is discussed in more detail in Chapter 4.

Finally, given the annotated DOM, we can formally define the problem of web block classification.

Definition 3. Let $\mathbb{BT}$ be a set of block types and $\mathbb{L}$ the corresponding set of labels. Given a web page P with associated annotated DOM T_P, the problem of *web block classification* is the problem of finding a mapping M from the set of sub-trees $T'_1, \ldots, T'_n$ of T_P to sets of labels from $\mathbb{L}$, such that each sub-tree is labelled with, at most, one label per block type, e.g., for each $BT \in \mathbb{BT}$ and sub-tree T_i, it must hold that $|M(T_i) \cap \mathbb{L}_{BT}| \leq 1$.

The above definition establishes the general classification problem. To apply standard classification algorithms, we need to map this general problem to a specific problem with a suitable feature space. To this end, we need to define the set of the features of the said feature space and explain how to compute them for a given DOM sub-tree and, of course, the actual classifiers.

We can now define the key concept used in the problem of web block classification, namely the concept of a feature that serves as the "alphabet" for defining unique characteristics of web blocks.

Definition 4. Let $BT \in \mathbb{BT}$ be a web block type and T_P an annotated DOM tree. Then a *feature* F_{BT} for BT is a mapping from nodes in T_P to values. The features can be Boolean, numeric, or nominal with regards to a given set of categories. We denote with $\textsc{type}(F)$ this type of F. The set of all features for BT is denoted as $\mathbb{F}_{BT}$. A *feature vector* $\vec{v}$ is an associative array over $\mathbb{F}_{BT}$ and $\textsc{featurise}_{BT}(T_P)$ maps the sub-trees of a given page to the corresponding feature vector such that $\textsc{featurise}_{BT}(T_P)[F_{BT}] = F_{BT}(T_P)$.

It is important to note that it is primarily the feature set rather than the chosen classification method that determines the accuracy of a classifier. Therefore, the selection of the most discriminating features for a given block type is crucial.

Definition 5. Let $BT \in \mathbb{BT}$ be a web block type, $\mathbb{F}_{BT}$ the corresponding set of features, and T_P a sub-tree of an annotated DOM. Then, a *classifier* for BT is a function $CL_{BT} : \mathbb{F}_{BT} \to L_{BT}$ such that, given a feature vector $\vec{v} = \textsc{featurise}_{BT}(T_P)$ over $\mathbb{F}_{BT}$, $CL_{BT}(\vec{v}) = L$ where L is a label from L_{BT}. We say that T_P is classified as BT if $L \in L^+_{BT}$ and that its classification for BT is L. A classification $\mathscr{C}$ of T_P over $\mathbb{BT}$ is then a mapping from DOM nodes in T_P to sets of labels from $\mathbb{L}$ such that, for each node n in T_P, $\mathscr{C}(n) = \{L \in \mathbb{L} : \exists B \in \mathbb{BT} : CL_{BT}(\textsc{featurise}_{BT}(n)) = L\}$. We say that the assignment size of a classification is the sum of the sizes of all the label sets.

Example 7. For a sub-tree T_P with feature vector $\vec{v'}$ and $CL_{\texttt{next\_link}}(\vec{v'}) = L$, we say that T_P is classified as a *numeric next-link* if $L =$`numeric_next_link`. However, if L is `non_next_link`, T_P is not classified as a next link at all.

Unfortunately, individual classifiers only go so far. A key challenge in web block classification is that the classifications of different block types affect each other; for example, if we determine that footers and sidebars are mutually exclusive and that both may be useful in determining the list of the main content paragraphs. Therefore, BER$_y$L uses a holistic approach to web block classification (as explained in Chapter 1). The core mechanism of the holistic approach is a set of constraints to which the set of all web block classifications of DOM nodes on a page must conform. BER$_y$L ensures that, if any of these constraints are violated, the existing classification is refined until all the web block classifications conform to the constraints.

It is now necessary to define a system of holistic constraints which indicates if a block classification is likely to be reasonable.

Definition 6. Let T_p be an annotated DOM and $\mathscr{C}$ a classification for T_P over a set of given block types $\mathbb{BT}$. Then, a *constraint* $\xi(T_P, \mathscr{C})$ over $\mathscr{C}$ and T_P is a first-order formula over three types of atoms:

- equality atoms between first-order variables and/or constants;
- atoms over annotated DOM relations as defined in Definition 1, such as CHILD or CSS;
- atoms over the classification relation CLASS which relates a DOM node to each of the class labels from $\mathscr{C}(n)$.

A non-empty set of such constraints over T_P and $\mathscr{C}$ is also referred to as a *system of constraints* $\Xi(T_P, \mathscr{C})$. We extend the notion of satisfaction from first-order formulae to constraints and systems of constraints in the obvious way.

We give examples of the actual constraints used by the BER$_y$L system in Section 3.9 of Chapter 3.

Definition 7. Let T_p be an annotated DOM, $\mathscr{C}$ a classification for T_P over a set of given block types $\mathbb{BT}$, and $\Xi(T_P, \mathscr{C})$ a corresponding system of constraints. Then $\mathscr{C}$ is a *valid classification* for Ξ, if Ξ is satisfied by T_P and $\mathscr{C}$, i.e., if all constraints in Ξ are satisfied.

It is often the case that a system of constraints is not satisfied by a given set of classifications. We then need to find a subset of the given classification which satisfies all constraints in Ξ with that subset fulfilling some *optimisation criteria* and for which the system of constraints is satisfied. We use a combined optimisation criterion: first, we optimise the number of labels assigned to DOM nodes (i.e., the assignment size of the classification) and, in the case of ties, the relative coverage of the classified nodes on the page. The relative *coverage* of a set of nodes N in an annotated DOM T_P is the average of (1) the

percentage of nodes occurring in at least one of the sub-trees rooted in nodes in N among all element nodes in T_P and (2) the percentage of the area covered by at least one node in N among the whole area of the page. We generalise the relative coverage from a set of nodes to a classification in the obvious way.

Definition 8. Let T_P be an annotated DOM, $\mathscr{C}$ a classification for T_P over a set of given block types $\mathbb{BT}$, and $\Xi(T_P, \mathscr{C})$ a corresponding system of constraints. The *maximal valid classification* problem is the optimisation problem of finding the classification $\mathscr{C}'$ for T_P over $\mathbb{BT}$ such that:

1. if n is a DOM node and $l \in \mathscr{C}'(n)$, then $l \in \mathscr{C}(n)$;
2. $\mathscr{C}'$ has maximal assignment size among all classifications for which (1) holds;
3. $\mathscr{C}'$ has the maximal relative coverage of T_P among all classifications for which (1) and (2) hold.

The underlying decision-related problem for the maximal valid classification is NP-complete. We present concrete examples of how the holistic approach works in our BER$_y$L system in Section 3.9 of Chapter 3 and evaluate how it improves the overall accuracy of the system in Chapter 6.

2.2 Architecture of our approach

BER$_y$L performs the actual block classification of a newly selected page in the following way: **(1)** converting the given page to basic predicates (described in Sections 3.2 and 3.6 of Chapter 3), **(2)** performing the feature extraction on these basic predicates (which results in a new feature file), **(3)** converting this feature file to an unlabelled ARFF file, and **(4)** applying the pre-computed web block classifier to this ARFF file.

With respect to **(1)**, we have a standard mechanism for generating the facts that represent all content and structural knowledge we acquire from the DOM tree of the given page, as well as its CSS encoding, provided by the DIADEM system [22].

We present a general scheme of the architecture of BER$_y$L and most of supervised classification approaches, which covers the phases **(2)**-**(4)** mentioned above in Figure 2.2. We extract the features, build the training corpus, and then perform the classification on the evaluation corpus. If the accuracy of classification on the evaluation corpus does not meet our requirements, we try to devise a new set of features and/or a new training corpus which is more representative of the diversity of a particular block type. We need to generate a

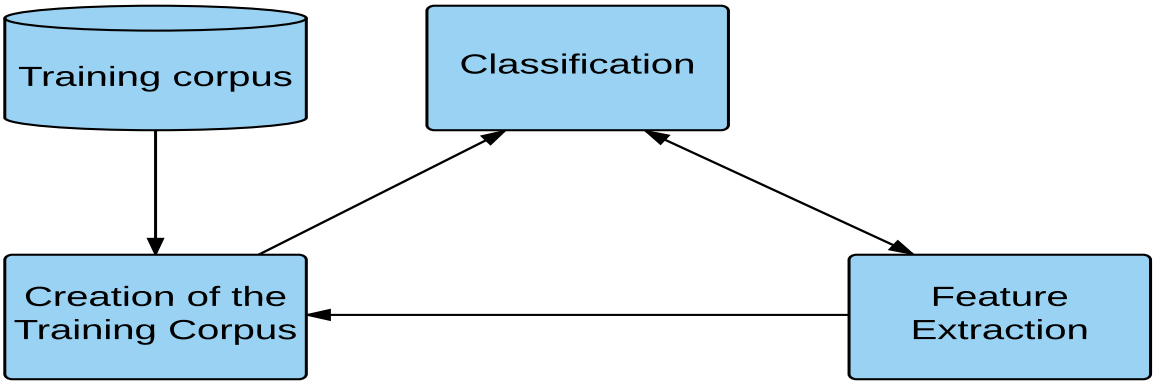


Figure 2.2: A general scheme of the architecture of supervised classification approaches

feature set every time we perform web block classification on a newly acquired web page, hence the backward connection between these two stages in Figure 2.2.

It is important to note this approach can also be applied to most other classification frameworks in general and web block classification frameworks in particular. However, as discussed in Chapter 1, there are some subtle differences between the approaches taken by state-of-the-art web block classification frameworks and the approach taken in BER$_y$L.

2.2.1 Architectures of state-of-the-art approaches and BER$_y$L

We illustrate the most common architecture of the state-of-the-art approaches to classification in Figure 2.3. A single set of features is extracted for block types from one domain or different domains and all blocks are then classified with that set of features.

The main difference between BER$_y$L 's architecture (presented in Figure 2.4) and the architectures used in the state-of-the-art approaches is that each block has its own tailored set of features which have been strongly engineered through the use of domain knowledge. These sets are easily adaptable to extension due to their use of modular component-based approach to feature extraction based on the use of template relations and global features. This also allows the BER$_y$L system to be adaptable to extensions to new classifiers and domains. BER$_y$L also employs a holistic approach based on viewing all blocks as interrelated with regards to the page on which they are located. This approach is implemented through a system of constraints after the execution of individual block classifiers to further enhance precision and recall of the output classifications. We also use domain-specific knowledge to write correct and powerful systems of constraints.

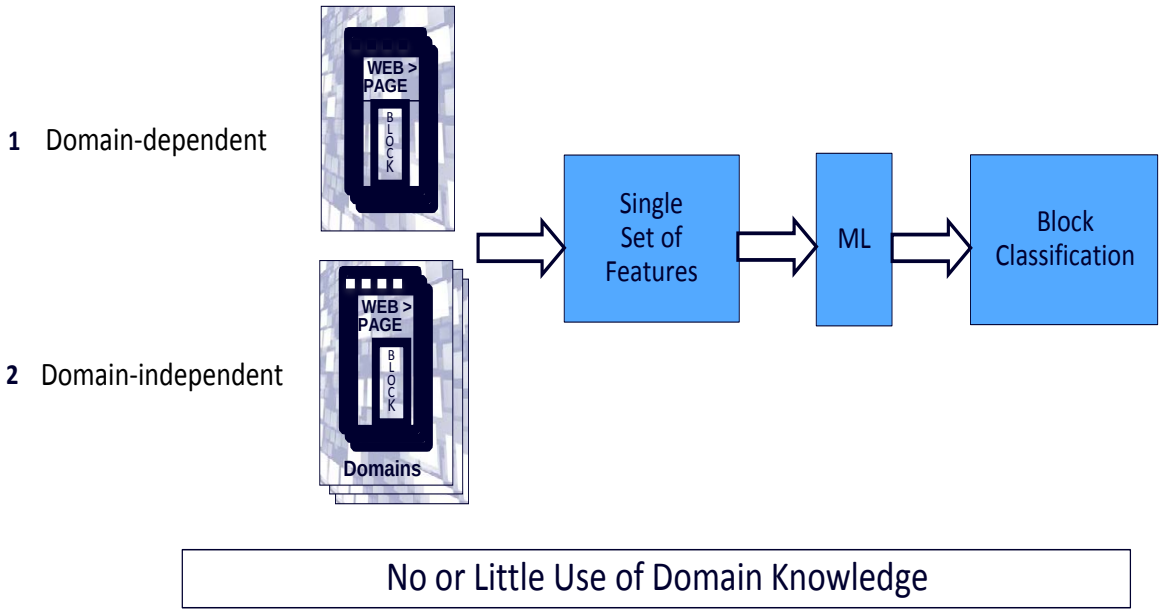


Figure 2.3: Most common architecture of the state-of-the-art approaches

2.2.2 Architecture of feature extraction in BER$_y$L

Feature extraction in BER$_y$L is of particular interest as we employ a novel approach which allows us to easily define complex, representative, and robust feature sets for new block types. The more blocks that have already been covered by the system, the easier it is to define classifiers for new types. This is because it is highly likely that such classifiers reuse many features and patterns which have already been used by existing classifiers. This is achieved by the mechanism of a **Component Model (CM)** which consists of a set of **components** and defines the process of feature extraction through the recurring use of **relations**. We use relations to represent global features, such as the width and height of the DOM nodes' respective CSS boxes and patterns which commonly occur for many block types, such as the patterns of proximity, containment, and annotation shown in Figure 2.5. The general motivation for using the CM approach is two-fold: **(1)** it minimises the human effort required for the definition of new block types and their respective features and **(2)** it improves the performance of the feature extraction process.. It is now necessary to consider the process of defining and extracting a new feature for a specific block type.

The first step is to analyse the current set of features for the block type in question $F_{BT} = F_1, \ldots, F_n$ and understand what is the strongest characteristic of this block type which discriminates it from other blocks and which is not present among any of the features that have already been included in F_{BT}. These characteristics are typically related to the content and structural presentation of the block (such as the number of characters or the

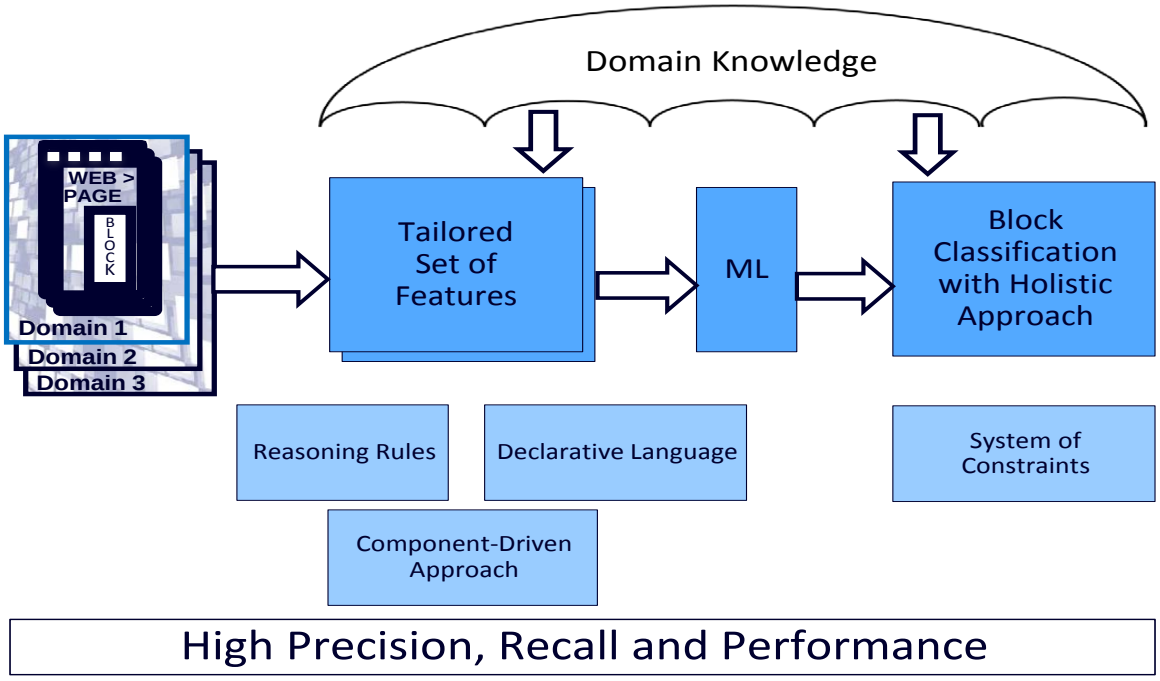


Figure 2.4: Detailed scheme of BER$_y$L's architecture

font used) encoded in the DOM tree or visual information (such as the width and height of the block or its relative position on the web page) encoded in CSS. In order to observe the discriminatory characteristics, we need to examine the occurrences of the given block type on a set of pages which are as diverse as possible. It is also beneficial to examine the presentation of a given block type on pages from different domains, even if the block type itself is domain-independent.

Example 8. In the case of a domain-independent block type of navigation menus, we have examined how it is presented in five different domains (real estate, used cars, online retail, forums and blogs, and restaurants).

We can now define the extraction process of the newly added feature F_{n+1}. In BER$_y$L we do so either in a declarative way (e.g., through Datalog rules) or a procedural way (e.g., through Java classes).

It is very often that different features use the same predicates or follow the same pattern in their respective definitions.

Example 9. The `direct left visual numeric sibling is a non-link` and `the number of visually neighbouring numeric next links` features of the "next" link classifier use the common `numeric` predicate, whilst both the `first page` feature for the header

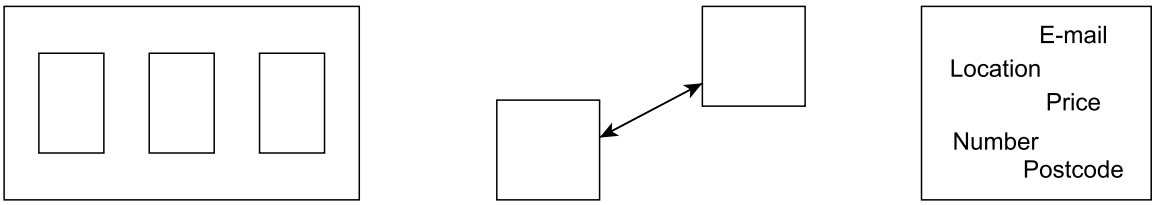


Figure 2.5: The patterns of containment, proximity and annotation

classifier and the `last page` feature for the footer classifier follow the same pattern of containment (the visual representation of this pattern and examples of other patterns such as annotation and proximity are presented in Figure 2.5).

In BER$_y$L, we take care of these common predicates and patterns through **relations** which, in the case of declarative implementation, are normal predicates which can be instantiated with specific parameter values (e.g., the width and height for the `in the visual proximity` relation that defines the boundaries for determining whether two DOM nodes are in visual proximity of each other and are set at the time of instantiation of this relation; our definition of visual proximity differs between classifiers). Alternatively, they could be lower-level predicates or functions (e.g., the `font family` function at the time of instantiation of the `property` function in the `two nodes which differ with respect to a specific property` relation). In the case of procedural implementation, relations are abstract classes which are overridden at the time of instantiation. We refer to relations which represent common patterns which can occur between different classifiers (such as the `two nodes which differ with respect to a specific property` relation) as **template relations**. Template relations present a powerful mechanism for the modular and scalable introduction of new features and classifiers. We cover relations in more detail in Section 3.7 of Chapter 3. In the vast majority of cases, relations are defined in a declarative way; however, in some cases (those discussed in Chapter 1 in the case of feature extraction) which require the use of efficient libraries and data structures or intense numeric computation (e.g., relations acquired from image processing), we employ a procedural approach to the extraction of relations implemented through Java. We provide a more detailed description of relations and our approach to their extraction in Chapters 3 and 4.

It is important to note that many features (such as the width and height of the block) tend to be used between different classifiers. We refer to these as **global features** and extract them before we extract features which are specific to one classifier. This is done so that they can be reused in all classifiers that need them without any computational overhead.

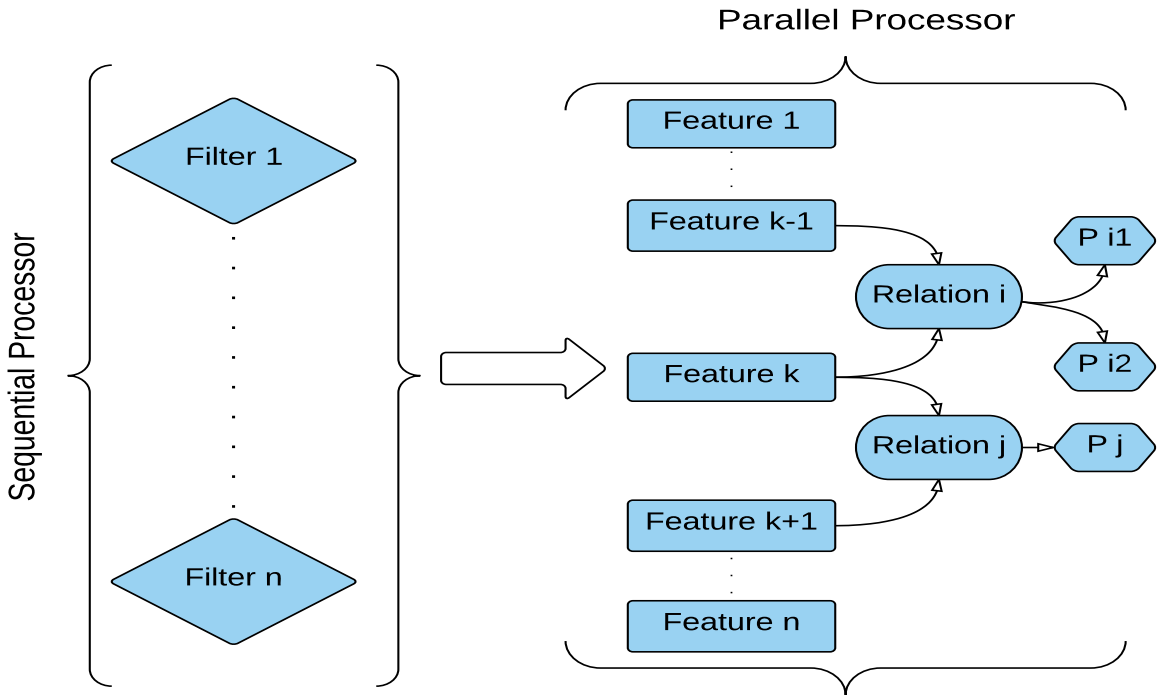


Figure 2.6: A general scheme of BER$_y$L's approach to feature extraction

We also pre-compute all relation templates for further improvements to the performance of the BER$_y$L system.

2.2.3 General architecture of the component-driven approach

In Figure 2.6, we provide a diagram which outlines the CM-driven approach to feature extraction. We first filter out the facts from the original set of DOM and CSS facts which present interest to our classification system with the help of **filter components**. These filters can be of a different nature.

Example 10. We can first employ the node of interest filter which filters out only certain nodes and the facts that relate to them from the full list of nodes (e.g., `<a,href>` and `<div,onclick>` nodes for the "next" link classifier, since we are only interested in links in this case) and then the attributes of interest filter which only filters facts for the attributes that we are specifically interested in (e.g., `class`, `title`, `name`, `value`, `id`, `alt`, and `src` attributes for the "next" link classifier). The filters are organised in a **sequential processor** $C_{11},\ldots,C_{1m}$ which uses the output produced by component C_i as the input for component C_{i+1}.

We then perform the actual feature extraction. Each feature is extracted either through a **rule component** or a **procedural component**. Both rule and procedural components can

use relation components, and two or more features can use the same relation component.

Example 11. In the case of the "next" link classifier, both the `non-link direct left visual numeric sibling` and `the non-link direct left structural sibling` features use the same `numeric` relation.

Both features and relations can make use of parameters. For example the `numeric` relation employs the `maximum annotation span` parameter which sets the maximal number of characters in a numeric node for the "next" link classifier to 3 (in practice, we do not observe search outputs split over more than 999 pages). Individual feature components are united in a **parallel processor** $C_{21},\ldots,C_{2n}$ *which allows us to execute each of the constituent components in a separate thread* and evaluates those features irrespective of the order in which they are given.

We currently use the J48 Decision Tree as a default classifier due to its ability to render good accuracy in terms of linearly separable feature spaces and the transparency of how the classification decision is being made. However, if we encounter feature spaces which are not linearly separable, we aim to switch to Support Vector Machine (SVM) classifiers. We can also use an ensemble classification method (if it will increase the accuracy of our system) for certain block types. In general, BER$_y$L is flexible with respect to which classifier to use and users can easily switch between classification methods according to the block types that they intend to classify and general system requirements.

Example 12. One example of when we prefer high precision over high recall would be the classification of "next" links. In the case of the "next" link classifier, we distinguish between numeric and non-numeric "next" links. It is usually the case that both numeric and non-numeric "next" links are present on the same page (e.g., see the pagination bar in Figure 1.1 in Chapter 1). For the extraction system to be able to directly navigate from one result page to the next, it would be sufficient to simply identify one "next" link. However, classifying a non-"next" link as a "next" link will lead the system to navigate to an incorrect page. An example of when we prefer high recall over high precision would be the classification of input forms. If the block classification module of the system wrongly classifies a non-form as a form, the form processing module would be able to identify that block is not a form. However, a form which is not identified as such will simply be ignored by the system and the form-filling component will have no input on which to work.

The trade-off between accuracy (precision and recall) and performance comes with an increasing number of features used by the classifier. Therefore, we aim to find an optimal set of features which would render both acceptable accuracy and performance results.

Chapter 3

Framework of our Approach

As we have discussed in Chapter 2, there are three main components of the BER$_y$L system: **(1)** feature extraction; **(2)** creation of the training corpus; and **(3)** web block classification. We now give a description of the framework for the approach taken in BER$_y$L, along with our solutions to the problems outlined in Chapter 2.

3.1 Feature selection

It is crucial that we select a correct set of features for the classifier in question. If we select an unrepresentative set of features, then the classifier will work fine on some of the input instances, but will not generalise to be representative on the whole space of possible inputs.

The features that we select can be both domain-independent (e.g., width and height of the CSS boxes for the candidate DOM nodes) and domain-dependent (e.g., whether the node in question has a `NEXT` annotation or not). Domain-independent features are likely to be stored in a global feature repository or instantiated through relation templates that are discussed in more detail further in this section, whilst domain-dependent features are usually specific to a given classifier and get defined ad hoc.

The novelty of our approach is that we try to tailor feature sets for different web blocks by application of declarative programming to feature extraction. We propose a component-driven approach combined with the use of Datalog-based BER$_y$L language and powered by the use of domain-specific knowledge. This, together with the repository of global features and relation templates, helps to create tailored features for different blocks and domains.

3.2 Component-based approach to feature extraction

We give a description of how our approach to feature extraction works in practice by introducing the concept of a **Component Model (CM)-driven approach to feature extraction**.

We now define the mechanics of the CM-based approach.

Definition 9 (Component type)**.** A *component type* $\mathfrak{C}$ is a triple $\langle I, O, U \rangle$ of two relation schemas I and O, the respective schema of the input and output relations for a component, and a set of formal parameters U.

Definition 10. A *formal component parameter* is a mapping of a unique name p, the parameter name, to a parameter type $\tau_p \in \mathscr{P}$. $\mathscr{P}$ is the set of permissible types and defined recursively as follows: Let $\mathscr{P}_A$ be the set of *atomic* types such as *Integer* or *String*. Then a permissible type is either an atomic type from $\mathscr{P}_A$, a sequence type $[\tau_i]$ where τ_i is itself another type from $\mathscr{P}$, or a component type.

$$\tau_p = \begin{cases} \alpha & \text{with } \alpha \in \mathscr{P}_A \\ [\tau_i] & \text{with } \tau_i \in \mathscr{P} \\ \mathfrak{C} & \text{with } \mathfrak{C} \text{ a component type} \end{cases}$$

For each parameter type τ, we denote with $\mathsf{domain}(\tau)$ the set of permissible values for τ.

We call a parameter of atomic type an atomic parameter, of sequence type a sequence parameter, and of component type a component parameter.

For an atomic type τ, $\mathsf{domain}(\tau)$ is the corresponding set of values. For a sequence type, it is the set of sequences over the values of the constituent type. For a component type, it is the set of (references to) ground components of type $\mathfrak{C}$. We call a component *generic* if it has at least one formal parameter, and *higher-order* if there is at least one component type parameter in U.

Let $\mathscr{N}$ be the set of component names. Then $\mathfrak{C}[p \mapsto v]$ is an *instantiation* for component type $\mathfrak{C}$ that binds the formal parameter with name p in $\mathfrak{C}$ to the value v, where $v \in \mathsf{domain}(p)$. We call an instantiation ground if all parameters in $\mathfrak{C}$ are bound to an actual value. For component parameters, we allow *references* as values, i.e., expressions of the shape $@n$ where n is the unique name of the referenced component.

Definition 11. Let $\mathscr{K}$ be the set of primitive components. A *(component) composition specification* describes how to combine components into more complex ones. A composition specification for a component type $\mathfrak{C}$ is then a triple $C = (n, \mathfrak{C}', \mathsf{E})$ where n is a unique name, $\mathfrak{C}'$ a ground instantiation for component type $\mathfrak{C} = (I, O, U)$, and E an expression that describes how to compose this component from other, more primitive ones. Specifically, a

composition expression is an expression of the form:

$$\mathsf{E} = \begin{cases} k & \text{where } k \in \mathscr{K} \\ @n & \text{where } n \text{ is a reference to an already defined component} \\ \mathsf{E}_1 \lhd \ldots \lhd \mathsf{E}_\mathrm{r} & \text{where } \mathsf{E}_\mathrm{i} \text{ are composition expressions} \\ \mathsf{E}_1 \parallel \ldots \parallel \mathsf{E}_\mathrm{r} & \text{where } \mathsf{E}_\mathrm{i} \text{ are composition expressions} \\ \mathsf{E}_1 \oplus \ldots \oplus \mathsf{E}_\mathrm{r} & \text{where } \mathsf{E}_\mathrm{i} \text{ are composition expressions} \end{cases}$$

We call $\lhd$ isolated *sequential* composition, $\parallel$ *parallel*, but isolated composition, and $\oplus$ sequential, but *possibly dependent* composition.

We write $\text{TYPE}(C) = \mathfrak{C}$, $\text{INPUT}(C) = I$, and $\text{OUTPUT}(C) = O$. Component expressions form trees (with possibly shared branches) and thus no cycles between components are possible. We refer to the name n assigned to a composition expression e as $\text{NAME}(e)$.

We refer to the components that are referenced in a composition expression as sub-components $C_1, \ldots, C_r$ and r the arity of C. We call the set of *defined sub-components* in E_C the set $\text{SUBS}(C)$ of all components defined in E_C or one of its sub-expressions, i.e., $\text{SUBS}(C) = \bigcup_{1 \leq i \leq r} \text{SUBS}(C_i) \cup \{(n_i : C_i) : 1 \leq i \leq r\}$.

A composition expression E_C is *valid*, if

1. the schemata of the sub-components are compatible: $\text{INPUT}(C) = \text{INPUT}(n_1)$, $\text{OUTPUT}(n_{i-1}) = \text{INPUT}(n_i)$ for all $1 < i \leq r$, and $\text{OUTPUT}(n_r) = \text{OUTPUT}(C)$.
2. for each composition parameter $(p, \tau) \in \text{TYPE}(C)$ that is instantiated to component reference $@v$, there is a sub-component $v : C_v \in \text{SUBS}(C)$ such that $\text{TYPE}(C) = \tau$.

Given a component (and corresponding composition expression) the semantic of that component is then quite straightforward: in our framework, components are implemented either as declarative rules (rule-based components, written and evaluated as Datalog rules) or Java classes (procedural components). In both cases, implementations may query parameters of the surrounding component. For atomic parameters, the query returns the corresponding value, for sequence parameters it returns a suitable representation of a sequence in Datalog or Java, and for component parameters it returns an interface that allows the implementation to access the results of the other component.

Definition 12. Let $C = (n, \mathfrak{C}')$ be a component with corresponding composition expression $n : \mathsf{E}_C$ and U the set of parameters for $\text{TYPE}(C)$. Then, we denote with $[\![\mathsf{E}_C]\!](I)$ the output for C under E, if executed on input I. The output is a set of pairs $n : O$ where n is a component name and O an instance of a component output schema. As such, we write $[\![\mathsf{E}_C]\!](I)[n]$ to

access the output instance for the component with name n.

$$[\![n : \mathsf{E}_C]\!](I) = \{n : [\![\mathsf{E}_C]\!](I \cup U)[\textsc{name}(\mathsf{E}_C)]\}$$

$$[\![\mathsf{E}_C(I)]\!] = \begin{cases} \{n_1 : k(I)\} & \text{if } \mathsf{E} = k \in \mathscr{K} \\ \bigcup_{i \le r} [\![C_i]\!](I) & \text{if } \mathsf{E} = C_1 \parallel \ldots \parallel C_r \\ \bigcup_{i \le r} [\![C_r]\!]([\![C_{r-1}]\!](\ldots [\![C_1]\!](I) \ldots)[\textsc{name}(C_{r-1})]) & \text{if } \mathsf{E} = C_1 \lhd \ldots \lhd C_r \\ \bigcup_{i \le r} [\![C_r]\!]([\![C_{r-1}]\!](\ldots [\![C_1]\!](I) \ldots) \cup I) & \text{if } \mathsf{E} = C_1 \oplus \ldots \oplus C_r \end{cases}$$

As already mentioned in Chapter 1, the rule-based components employ a declarative approach, since: **(1)** it allows us to combine it with template relations and global features that provide a succinct representation of the current feature set, which in turn allows us to simplify definition of new features through employing existing global features or template instantiations and learn new features by automatically finding the right combination of parameters for template instantiations; and **(2)** it is much easier to learn logical predicates automatically than to learn procedural language programs, which is likely to come in useful if we have to infer new features automatically.

We use Datalog as the declarative language of choice, since it is fast and widely used [18, 26]. Also, with a Datalog-based approach to feature extraction we can easily extend our extraction rules to run on databases, which can come in useful if we have to process large sets of training or evaluation data. We use DLV [38] to implement our feature extraction rules, as it is one of the few off-the-shelf Datalog engines that are available.

We can now give a definition of a *component model*, which is a crucial part of our BER_yL system, since each of its constituent classifiers corresponds to a unique component model.

Definition 13 (Component model)**.** A component model is a set $\mathbb{M}$ of executable components such that for each component $C \in \mathbb{M}$ with a higher-order parameter p that references a component C', then $C' \in \mathbb{M}$ ("no dangling references").

Example 13. We give an example of a fragment of the real component model we use for the Next Link classifier (Figure 3.1). Let us consider one of the most important features of the Next Link classifier to be that it checks whether the direct left numeric visual sibling of a given numeric node is a link or not (if it is not a link that is a strong indicator that the numeric link in question is a numeric next link). We define this feature from the global template relation *two nodes with unary relation properties connected by a binary relation* in that it denotes two nodes connected to each other by a binary relation that also defines a unary property of the second node. Each of the nodes also holds one unary relation property of their own. That is a very generic template that can be used in many cases for different

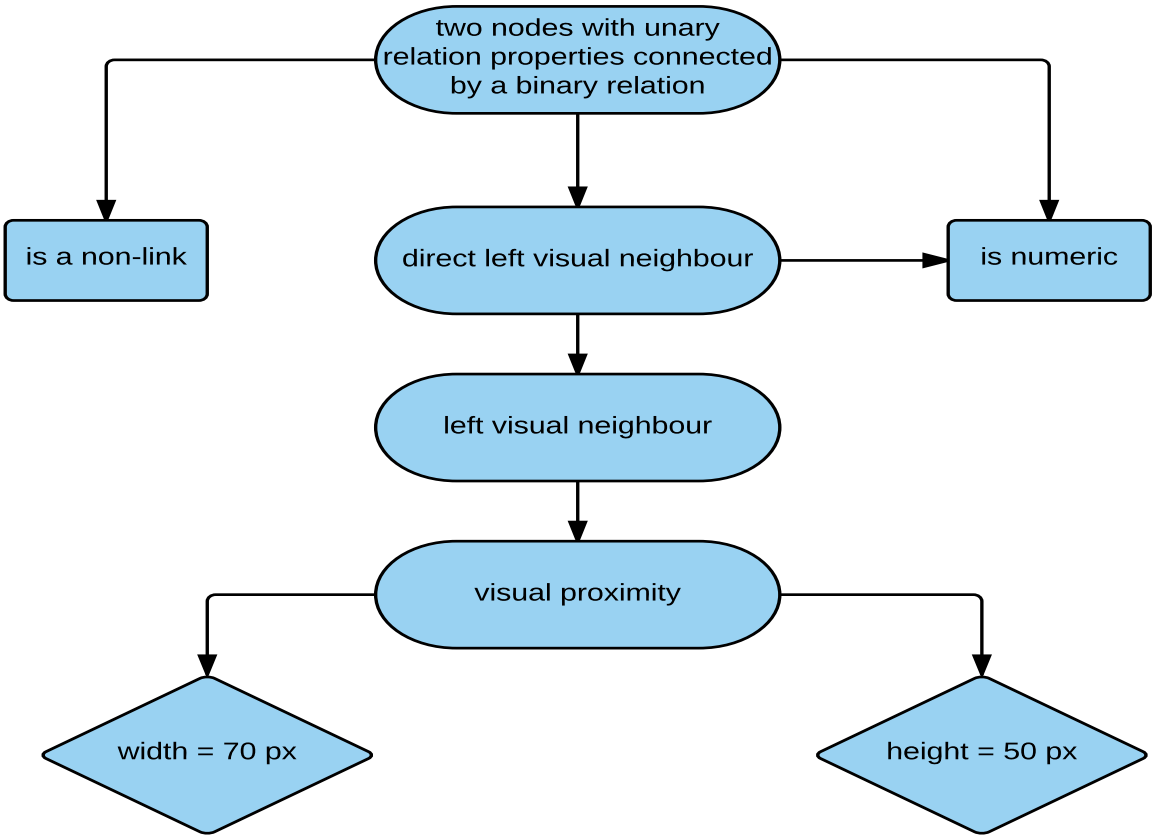


Figure 3.1: An example of a component model for a single feature

classifiers. In our case, the binary relation is the *direct left visual neighbour* relation with the property of the second node (the one to the visual left of the node in question), which is defined as *is numeric*. Note that the *direct left visual neighbour* binary relation, in turn, utilises the *visual proximity* binary relation that defines any visual proximity within the given boundaries regardless of direction. We set the width and height parameters for the visual proximity boundaries to 70 and 50 pixels respectively. The respective unary relation properties of the two nodes in question are *is numeric* for the first node and *is a non-link* for the second node. Note that both the *two nodes with unary relation properties connected by a binary relation* and *direct left visual neighbour* relations make separate calls to the *is numeric* relation, but these calls direct to the same component representing the *is numeric* relation, which allows us to avoid unnecessary overhead.

3.3 Introduction to Datalog

In this section we briefly introduce the declarative language of *Datalog*. Some of the information given here closely follows the presentation given in [1]. We focus on the aspects

of Datalog that are crucial in our work (i.e., the syntax and semantics of the language) and Datalog with safe and stratified negation, aggregation, arithmetic, and comparison operators.

There are strong limitations to the use of relational algebra and the languages based thereon (such as traditional versions of SQL prior to SQL:1999) to query databases. One of the most crucial limitations is the lack of support for *recursion*. Datalog supports recursion and enhances the querying power of relational algebra.

Example 14. One of the most important queries that a user can run on a DOM tree is to find all ancestors and descendants of a given node. However, as relational algebra does not support recursion, we cannot define a query in the traditional versions of SQL (prior to SQL:1999) for finding **all** ancestors or descendants of the given node, but only its parent and children. However, this can easily be achieved by using Datalog.

Syntax of Datalog A *Datalog rule* is an expression which is written in the form of $R_1(u_1) \Leftarrow R_2(u_2),\ldots,R_n(u_n)$, where $n \geq 1$, $R_1,\ldots,R_n$ are the names of the relations and $u_1,\ldots,u_n$ are tuples of arbitrary arities. Each variable that occurs in u_1 must occur in at least one of the $u_2,\ldots,u_n$ tuples. A *Datalog program* is a finite set of Datalog rules. $R_1(u_1)$ is known as the *head* of the rule and $R_2(u_2),\ldots,R_n(u_n)$ is known as the *body* of the rule. We also distinguish between *extensional (edb)* relations which only occur in the body of the rules and consist of data that is stored in a database and *intensional (idb)* relations which occur in the head of a rule from a Datalog program and consist of computed tuples.

Example 15. It is important to consider the ancestral relation on a DOM tree, such as the following set of edb predicates (predicate `node(Id)` represents a node with a certain unique identifier `Id` and predicate `parent(Id1,Id2)` represents that a node with identifier `Id1` is a parent of node with identifier `Id2`).

```
{node(a), node(b), node(c), node(d),
 parent(b,a), parent(c,a), parent(d,c)}
```

The Datalog program P_1 for finding the ancestors of all nodes is the following:

```
ancestor(Id1,Id2) ⇐ parent(Id1,Id2).
ancestor(Id1,Id2) ⇐ parent(Id1,Id3), ancestor(Id3,Id2).
```

Semantics of Datalog We give a brief description of two semantic interpretations of Datalog: **(1)** model-theoretic semantics and **(2)** fixpoint semantics. The model-theoretic semantics, in particular the Herbrand interpretation of a Datalog program, regards the program as a set of first-order sentences which describes the desired answer. Therefore, the aim is

to find an instance of the database which satisfies the given sentences, also known as the *model* of the sentences. It is important that, of all possible models, we choose the *minimal model*. The fixpoint semantics interpret a Datalog program as the least set of relations (in the pointwise inclusion order on vectors of sets of tuples) that satisfies the recursion equations in the program.

Example 16. Suppose we have the set of edb and idb predicates described in Example 15. In both the minimal Herbrand interpretation and fixpoint semantics, the resultant model is the following:

```
{node(a), node(b), node(c), node(d), parent(b,a), parent(c,a), parent(d,c),
 ancestor(b,a), ancestor(c,a), ancestor(d,c), ancestor(d,a)}
```

Datalog with safe and stratified negation In order to compute the intensional relations in a Datalog program, we require those relations to be finite – otherwise the fixpoint iteration is not guaranteed to terminate. One way of ensuring finiteness is to insist that every variable occurring in the head of a rule also occurs in a non-negated atom in its body: such a Datalog program is said to be *safe*.

Example 17. We give examples of two unsafe Datalog programs P_2 and P_3. P_2 contains an unbound variable:

```
s(X) ⇐ r(Y).
```

In P_3, the binding occurs only in terms of a negation, which makes this program unsafe:

```
s(X) ⇐ r(Y),  ¬ r(X).
```

It is not necessarily the case that a least Herbrand model (or a least fixpoint) of a Datalog program exists.

Example 18. The following unsafe Datalog program P_4 does not have a least Herbrand model or a least fixpoint:

```
S(x) ⇐ ¬ R(x).
R(x) ⇐ ¬ S(x).
```

To ensure the existence of a least model, we restrict the use of negation to *stratified negation*. We apply the rules of the program through *strata*, i.e., a set of rules (programs) where, if a rule from a stratum contains negation of a certain idb predicate, that idb predicate should have appeared in one of the preceding strata. A Datalog program is said to be *stratifiable* if it can be classified into strata. Datalog program P_4 from Example 18 is not stratifiable. We now give an example of distinct stratifications of a Datalog program.

Example 19. In the following Datalog program P_5:

```
r1  S(x) ⇐R'1(x), ¬R(x).
r2  T(x) ⇐R'2(x), ¬R(x).
r3  U(x) ⇐R'3(x), ¬T(x).
r4  V(x) ⇐R'4(x), ¬S(x), ¬U(x).
```

This program has five distinct stratifications, any of which can be applied at the time of evaluation:

```
{r1}, {r2}, {r3}, {r4}
{r2}, {r1}, {r3}, {r4}
{r2}, {r3}, {r1}, {r4}
{r1, r2}, {r3}, {r4}
{r2}, {r1, r3}, {r4}
```

For the applications in this thesis, the restriction to stratified Datalog is acceptable, but for other applications recursion through negations is essential. The above definition of a least model can be appropriately adapted to cater for such generality, as in [54].

Datalog with aggregation, arithmetic operations, and comparisons We are applying Datalog in practical environments and, therefore, we need it to support aggregation as well as arithmetic operations and comparisons. Our version of Datalog supports #count, #sum, #avg, #max, and #min aggregation operators. Stratification applies to aggregation in the same way that it applies to negation. For similar reasons, we need arithmetic and comparison operators. Our version of Datalog supports the $=$, $<$, $>$, $\leq$, and $\geq$ comparison operators as well as $+$, $-$, and $\times$ arithmetic operators. Unrestrained use in recursion could lead to an infinite universe of values; however, in practice and in terms of this thesis, this is not an issue.

3.4 The BER$_y$L language

We provide a tool to the users of our system that allows them to define powerful BER$_y$L component models in an easy and intuitively understandable, declarative way. This tool is a version of Datalog with some extensions and is one of the key modules of our framework.

The BER$_y$L language is a dialect of Datalog that includes safe and stratified negation, aggregation, arithmetic, and comparison operators, denoted as Datalog$^{\neg,\text{Agg},\text{ALU}}$. The usual restrictions apply, i.e., all variables in a negation, arithmetic operation, aggregation, or comparison must also appear in some other (positive) context. Further, there may be no dependency cycles over such operators. In the context of the BER$_y$L system, this language is used in a certain way:

1. there is a number of input facts for representing the annotated DOM (see Table 3.1), as well as the output of previous components, such as the classification facts in Table 3.1;

2. each program carries a distinguished set of output predicates and only entailed facts for those predicates are ever returned.

The BER$_y$L language also uses a number of syntactic extensions to ease the development of complex rule programs. The most important ones are *namespaces*, *parameters*, and the explicit distinction of *output predicates*. All other intensional predicates are temporary ("helper") predicates only.

Definition 14. A BER$_y$L *namespace* is a set of predicate names, itself identified by a unique name among all namespaces. It is denoted as a prefix followed by :: in front of the contained predicate names.

Namespaces serve to ease the use of predicates from different sources or program components, and to isolate these components effectively. However, formally, they do not increase the expressiveness of the used Datalog fragment as they can be rewritten in a linear pre-processing step.

BER$_y$L language parameters in the `param` namespace are predicates that map to formal component parameters of atomic, sequence, or component type, as defined in Section 3.2.

Definition 15. Let C be a component of component type $\mathfrak{C}$ with $C = \mathfrak{C}[p_1 \mapsto v_1, \ldots, p_n \mapsto v_n]$. Let id_i be unique identifiers for the bindings $p_i \mapsto v_i$. For each such binding where p_i has type τ_i, there is a set of input facts BP in the input of C such that:

1. $BP = \{\texttt{param}::p_i(id_i, v_i)\}$ if τ_i is atomic;

2. $BP = \bigcup_j \psi_j \cup \{\texttt{param}::p_i(id_i, sid_1), \ldots, \texttt{param}::p_i(id_i, sid_n)\}$ where $v_i = [q_1, \ldots, q_n]$, sid_j a unique identifier for q_j, and ψ_j is the application of this definition to parameter q_j, if τ_i is a sequence type;

3. $BP = \bigcup_{O \in \mathscr{O}} \{\texttt{param}::p_i(id_i, X_1) \rightarrow O(cid, X_1)\} \cup \{\texttt{param}::p_i(id_i, X_1, X_2) \rightarrow O(cid, X_1, X_2)\} \cup \ldots \cup \{\texttt{param}::p_i(id_i, X_1, \ldots, X_{arity(O)}) \rightarrow O(cid, X_1, \ldots, X_{arity(O)})\}$, where τ_i is a component type, $v_i = C'$, cid is a unique identifier for C', $\mathscr{O}$ is the set of distinguished output predicates of C', and $arity(O)$ is the arity of predicate O.

For every component its input also contains one additional parameter `param::instantiation_id`(id_i) that provides runtime access to the binding Id.

Structural:	
`dom::element(N,T,PStart,Start,End)`	DOM element node `N` $\in \mathscr{N}$ has tag `T`=τ`(N)`, and parent node `P`, such that `(P, N)` $\in$ CHILD, with start label `PStart`, and start and end labels `Start` and `End`.[1]
`dom::attribute(A,N,T,V)`	DOM attribute node `A` of node `N` with `(N,A)`$\in$ ATTR has tag `T`=τ`(A)` and value `V`=σ`(A)`.
`dom::clickable(N)`	`N` is a clickable target (a link or has an onclick handler, i.e., `(`τ`(N)=a)` $\vee$ `(`$\exists$`(N,A)`$\in$ ATTR $\wedge$ τ`(A)=onclick)`.
`dom::content(N,v,O,L)`	Node `N` corresponds to textual content `v` with `(N,`N_i`)`$\in$CHILD$^+\wedge\neg\exists$`(`N_i`,`N_j`)`$\in$CHILD$\wedge$`(`σ`(`N_1`)++...++`σ`(`N_n`)=v)`[2], nodes N_i are sorted in the ascending order of their `Start` labels, node `N` starts at document offset `O` and has length `L`.
Visual:	
`css::box(N,Left,Top,Right,Bottom)`	bounding box of a DOM node `N`, such that $\exists$`(N,`C_1`),(N,`C_2`),(N,`C_3`),(N,`C_4`)`$\in$ `CSS:` τ`(`C_1`)=box_left_coord`$\wedge\tau$`(`C_2`)=box_top_coord`$\wedge$ τ`(`C_3`)=box_right_coord`$\wedge\tau$`(`C_4`)=box_bottom_coord`$\wedge$ σ`(`C_1`)=Left`$\wedge\sigma$`(`C_2`)=Top`$\wedge\sigma$`(`C_3`)=Right`$\wedge\sigma$`(`C_4`)=Bottom`.
`css::page(Left,Top,Right,Bottom)`	bounding box of a DOM node `N` with τ`(N)=html`.
`css::resolution(1800,800)`	the average screen resolution.[3]
`css::font_family(N,Family)`	`N` is rendered with a font from the given family `(`$\exists$`(N,C)` $\in$ CSS $\wedge$ τ`(C)=font_family`$\wedge\sigma$`(C)=Family)`.
`css::font_size(N,Size)`	`N` is rendered with a font of the given size `(`$\exists$`(N,C)` $\in$ CSS $\wedge$ τ`(C)=font_size`$\wedge\sigma$`(C)=Size)`.
Content:	
`gate::annotation(N,A,v)`	holds if μ`(A,N,v)` holds (Definition 2).
Classification:	
`cls::classification(N,`*BT*`,Label)`	a classification `(N,Label)`$\in$ `CLASS` where *BT* $\in \mathbb{BT}$ is a block type with `Label`$\in$ *BT*. For instance the pagination classifier corresponds to the pagination namespace, e.g., `cls::classification(e_200,pagination,numeric_next_link)` and `cls::classification(e_300,pagination,non_num_next_link)`. Different classifiers can produce output on the same node, e.g., `cls::classification(e_100,header,header)` and `cls::classification(e_100,navigation,nav_menu)`.

Table 3.1: BER$_y$L's input facts

Example 20. For an example of an atomic parameter, consider component *C* with parameter *p* that is used in two different places in a larger component: in one place where *p* is bound to 70, and in another where *p* is bound to 30. Then, the first binding is represented by a fact `param::maxWidth(1,70)`, the second by `param::maxWidth(2,30)`, if 1 and 2 are corresponding unique identifiers.

Example 21. For an example of a sequence parameter, consider component C with a sequence parameter *clickables*, which is used to determine the DOM nodes that are clickable. It is mapped to a list of lists of strings ($\tau_{clickables}$=[[String]]), where each of the inner lists represents tuples of an element tag, an optional attribute tag, and an optional attribute value. A DOM node is then clickable, if it matches such a list, i.e., has the same tag, an attribute with the given attribute tag and attribute value (if any).

Let p be the Id of the `clickables` parameter and its value be a list of sequence parameters with IDs $s_1,\ldots,s_n$. Then, there is one parameter fact `param::clickables(`p`,`s_i`)` for each s_i. The values of the nested lists are lists of strings. In this case, only inner lists of length up to 3 are allowed. For the j-th element e_j of the i-th inner list there is a fact `param::clickable(`s_i`,`e_j`)` and a second fact `param::string(`e_j`,`S`)` where S is the value of e_j:

```
clickable(X) ⇐ dom::element(X,Tag,_,_,_), param::clickables(PId,SId1),
 param::clickable(SId1,VId1), param::string(VId1,Tag).
clickable(X) ⇐ dom::element(X,Tag,_,_,_), dom::atribute(_,X,Attr,_)
 param::clickables(PId,SId2), param::clickable(SId2,VId1), param::string(VId1,Tag),
 param::clickable(SId2,VId2), param::string(VId2,Attr).
clickable(X) ⇐ dom::element(X,Tag,_,_,_), dom::atribute(_,X,Attr,Value),
 param::clickables(PId,SId3), param::clickable(SId3,VId1), param::string(VId1,Tag),
 param::clickable(SId3,VId2), param::string(VId2,Attr), param::clickable(SId3,VId3),
 param::string(VId3,Value).
```

In the example, we have three cases of inner lists: lists of lengths 1, 2, and 3. Lists of length 1 are treated in the first rule (lines 1–2) and the single containing string is interpreted as a tag. Lists of length 2 are treated in the second rule (lines 3–5) and the containing strings are interpreted as a tag and its attribute. Finally lists of length 3 are treated in the third rule (lines 6–9) and the containing strings are interpreted as a tag, its attribute, and the value of that attribute.

Assume the `clickables` parameter is instantiated to `[["button"], ["a","href"], ["a","onclick"], ["input","onclick"], ["input","image"], ["input","button"], ["div","onclick"], ["input","type","button"], ["input","type","image"], ["input","type","submit"]]`. The corresponding set of facts for this instantiation is then:

```
param::clickables(1,2). param::clickable(2,3). param::string(3,"button").

param::clickables(1,4). param::clickable(4,5). param::clickable(4,6).
param::string(5,"a"). param::string(6,"href").
```

[1] Start and end labels of a node correspond to a pre-order traversal of the DOM tree with a single incremental counter that assigns the start label the first time the node has been explored, and the end label when all of the node's descendants have been explored.

[2] ++ is a concatenation operator.

[3] According to a study by http://www.w3schools.com/browsers/browsers_display.asp.

```
param::clickables(1,7). param::clickable(7,8). param::clickable(7,9).
param::string(8,"a"). param::string(9,"onclick").
param::clickables(1,10). param::clickable(10,11). param::clickable(10,12).
param::string(11,"input"). param::string(12,"onclick").
param::clickables(1,13). param::clickable(13,14). param::clickable(13,15).
param::string(14,"input"). param::string(15,"image").
param::clickables(1,16). param::clickable(16,17). param::clickable(16,18).
param::string(17,"input"). param::string(18,"button").
param::clickables(1,19). param::clickable(19,20). param::clickable(10,21).
param::string(20,"div"). param::string(21,"onclick").

param::clickables(1,22).
param::clickable(22,23). param::clickable(22,24). param::clickable(22,25).
param::string(23,"input"). param::string(24,"type"). param::string(25,"button").
param::clickables(1,26).
param::clickable(26,27). param::clickable(26,28). param::clickable(26,29).
param::string(27,"input"). param::string(28,"type"). param::string(29,"image").
param::clickables(1,30).
param::clickable(30,31). param::clickable(30,32). param::clickable(30,33).
param::string(31,"input"). param::string(32,"type"). param::string(33,"submit").
```

Based on the above definition of BER$_y$L language parameters, we can give a generic definition of a program in the BER$_y$L language.

Definition 16. A BER$_y$L program is a set of components $C_1, \ldots, C_n$, each with a type $\mathfrak{C}_i \in \mathbb{C}_{BL}$, where $\mathbb{C}_{BL}$ are the component types supported by the BER$_y$L language.

Specifically, there are three classes of components and component types, which are used heavily in BER$_y$L language programs: components representing (1) relations, (2) features, and (3) constraints. All of these operate on the universe $\mathscr{U}$ of DOM nodes and atomic values appearing in the DOM of a page.

Definition 17. Let C be a component of type $\mathfrak{C}$ and $\mathfrak{C}$ be a relation, feature, or constraint type. The three classes of types are distinguished by their input and specific, distinguished output signatures. These distinguished predicates differ in name and arity, but all have an Id for the specific instantiation of C as first parameter. Specifically, the single distinguished output predicate is (with I the unique identifier for the concrete instantiation of C and N the unique name of $\mathfrak{C}$):

1. `relation`$::N(I, X_1, \ldots, X_k)$ if $\mathfrak{C}$ is a relation component and $X_1, \ldots, X_k \in \mathscr{U}$. The input schema of relation components is the union of `dom`, `css`, `gate`, and `relation` (containing all known relations for possible use as sub-relations). If $\mathfrak{C}$ has parameters, the input schema also contains its `param` predicates (analogous for the other cases).

2. `feature::N(I,X,V)` if $\mathfrak{C}$ is a feature component, $X, V \in \mathscr{U}$, typically with X being a DOM node and V an atomic value. The input schema of feature components is the union of `dom`, `css`, `gate`, and `relation`.

3. `constraint::N(I,X,Y,V)` if $\mathfrak{C}$ is a constraint component and $X, Y, V \in \mathscr{U}$, typically with X and Y being DOM nodes and V a Boolean value. The input schema of constraint components is the union of `dom`, `css`, `gate`, `relation`, and `cls`.

We call components of relation types relation components, components of feature types feature components, and components of constraint types constraint components. As the input/output schemata for these three classes of component types are fixed, in future discussion we omit stating them explicitly.

We typically use relations to either (a) distinguish sets of nodes, (b) relate sets of nodes to each other, or (c) attach additional information to nodes. Therefore, we call relations of the first type where, in addition to the unique identifier there is a single other parameter, namely the node to be characterised, **unary relations**, relations of the second type, where there are two other parameters, namely the nodes in a relation, **binary relations**, and relations of the third type, where there are n other parameters that attach additional information to the given node, **function relations**.

Example 22. Examples for these three types of relations:

1. `relation::is_numeric(Id,X)` is a unary relation;

2. `relation::is_visual_neighbour(Id,X,Y)` is a binary relation;

3. `relation::num_characters(Id,X,Num)` is a function relation.

We cover constraints in detail in Section 3.9. We give concrete examples of the use of relations, features, and constraints by the BER$_y$L language in Sections 3.5 and 3.9

The standard library of the BER$_y$L language

The BER$_y$L language also provides a set of predefined components that a specific block type may import, which are defined through the input facts from Table 3.1 and are written in the `relation` namespace. We call these predefined components **standard relations**. Figure 3.2 shows some of the standard relations in BER$_y$L: these range from structural relations between nodes (similar to XPath relations) over visual relations (such as proximity) to information about the rendering context (such as the dimensions of the first and last screen).

```
relation::preceding(Id,X,Y) ⇐ dom::element(X,_,_,_,End),
  dom::element(Y,_,_,Start,_), End < Start, param::instantiation_id(Id).
relation::descendant(Id,X,Y) ⇐ dom::element(X,_,_,StartX,EndX),
  dom::element(Y,_,_,StartY,EndY), StartY < StartX,
  EndX < EndY, param::instantiation_id(Id).
relation::leaf_descendant(Id,X,Y) ⇐ dom::element(X,_,_,_,_),
  relation::descendant(CId1,X,Y), ¬relation::descendant(CId2,Z,X),
  param::instantiation_id(Id).
relation::visual_proximity(Id,X,Y) ⇐
  css::box(X,LeftX,TopX,_,_),
  css::box(Y,LeftY,TopY,_,_),
  TopY − DVert ≤ TopX ≤ TopY + DVert,
  LeftY − DHor ≤ LeftX ≤ LeftY + DHor,
  param::dH(PId1,DHor), param::dV(PId2,DVert),
  param::instantiation_id(Id).
relation::left_visual_neighbour(Id,X,Y) ⇐
  relation::visual_proximity(CId,X,Y),
  css::box(X,_,_,RightX,_), css::box(Y,LeftY,_,_,_),
  RightX ≤ LeftY, param::instantiation_id(Id).
relation::first_screen(Id,Left,Top,Right,Bottom) ⇐
  css::page(Left,Top,_,_), css::resolution(H,V),
  Right = Left + H, Bottom = Top + V, param::instantiation_id(Id).
relation::last_screen(Id,Left,Top,Right,Bottom) ⇐
  css::page(_,_,Right,Bottom), css::resolution(H,V),
  Left = Right − H, Top = Bottom − V, param::instantiation_id(Id).
```

Figure 3.2: The standard library of the BER$_y$L language

3.5 Examples of the use of the BER$_y$L language

In this section, we give examples of two of the three components that the BER$_y$L language uses the most, namely the relation and feature components. We cover constraints in more detail in Section 3.9. In the following examples we identify component types by means of their parameters, as the input and output are fixed by Definition 17.

Example 23. To illustrate the use of *relations* in the BER$_y$L language, let us consider the following example. We want to define the pairs of nodes X and Y such that X is displayed to the left of Y ("visual left neighbour"). We also assume that their top and bottom boundaries are aligned within a certain threshold `dV`.

This relation component thus has one parameter `dV`, the maximum vertical distance between the respective top and bottom boundaries of X and Y, with $\tau_{dV} =$`Integer`. The rules for this relation are:

```
relation::visual_left_neighbour(Id,X,Y) ⇐
  css::box(X,LeftX,TopX,RightX,BottomX),
  css::box(Y,LeftY,TopY,RightY,BottomY),
```

```
  TopY − DVert ≤ TopX ≤ TopY + DVert,
  BottomY − DVert ≤ BottomX ≤ BottomY + DVert, RightX ≤ LeftY,
  param::dV(PId,DVer), param::instantiation_id(Id).
```

BER$_y$L provides a high-order relation `node_in_between`, which has a parameter `base` that is bound to a (partial) order such as "visual left neighbour". It holds for all pairs of nodes X and Y for which `base` holds and for which there is at least one other node Z such that `base` holds for (X,Z) and (Z,Y). The corresponding rule for `node_in_between` is thus (with `bid` the Id of the `base` parameter):

```
relation::node_in_between(Id,X,Y) ⇐
  param::base(bid,X,Y),
  param::base(bid,X,Z),
  param::base(bid,Z,Y),
  param::instantiation_id(Id).
```

A typical use of this relation is to instantiate base to `visual_left_neighbour`:

```
relation::node_in_between[base↦relation::visual_left_neighbour[dV↦50]]
```

The segment of the BER$_y$L program obtained after the instantiation of `node_in_between` is given below (assuming that `bid` is the unique identifier of `base` and `cid` the identifier for the specific instantiation of `visual_left_neighbour`):

```
relation::node_in_between(Id,X,Y) ⇐
  param::base(bid,X,Y),
  param::base(bid,X,Z),
  param::base(bid,Z,Y),
  param::instantiation_id(Id).

param::base(bid,X,Y) ⇐
  relation::visual_left_neighbour(cid,X,Y),
  param::instantiation_id(Id).
```

For the purpose of brevity in the notation in this and further examples we omit giving the `param::`$p_i(id_i,X_1,\ldots,X_k) \rightarrow$`relation::`$r_j(cid_j,X_1,\ldots,X_k)$ rules in the instantiated segments of BER$_y$L programs and instead write `relation::`$r_j(cid_j,X_1,\ldots,X_k)$ in the bodies of the rules that use respective parameters.

Example 24. Suppose we want to define the *feature* that computes the font size of the given DOM node. This numeric feature has no formal parameters and can be written in the following way (note that in the case where the node does not have a corresponding font size input fact we assume that the font size is 0, as feature mappings between DOM nodes and values need to be one-to-one):

```
feature::font_size(Id,X,FontSize) ⇐
 dom::element(X,_,_,_,_), css::font_size(X,FontSize),
 param::instantiation_id(Id).

feature::font_size(Id,X,0) ⇐
 dom::element(X,_,_,_,_),  ¬ css::font_size(X,_),
 param::instantiation_id(Id).
```

3.6 Component composition with the BER$_y$L language

Let us now define the computation of the feature introduced as an example in Section 3.2 using the component model syntax and semantics, as well as the BER$_y$L language that we have introduced in the sections above.

Example 25. The graph representation of the component model for this feature is given in Figure 3.1. For the purpose of brevity in our notation we abbreviate the main component and its constituent sub-components in the following way: (a) is numeric as C_1, (b) is a non-link (or non-clickable) as C_2, (c) visual proximity as C_3, (d) left visual neighbour as C_4, (e) direct left visual neighbour as C_5, and (f) two nodes with unary relation properties connected by a binary relation as C_6. C_1-C_2 are unary relation components, whilst C_3-C_6 are binary relation components. We give the rules for these components in Figure 3.3.

C_1 and C_2 are primitive components with no parameters attached to them and therefore have trivial empty instantiations and identity-binding expressions:

```
𝔈1[]; C1
𝔈2[]; C2
```

The rules of C_3 and C_4 correspond to standard binary relations `relation::visual_proximity` and `relation::left_visual_neighbour` as defined in Figure 3.2. C_3 has two parameters dH and dV that represent the rectangle coordinates of possible positions of the top-left node, the CSS boxes of which are considered to be in the visual proximity of the CSS box of the given node. The instantiations and composition expressions of these two components are the following:

```
𝔈3[dH↦70,dV↦50]; C3
𝔈4[]; C3◁C4
```

We now give the instantiation and the binding expression of binary relation component C_5 that specifies whether one node is a direct left visual neighbour of another node and has a single unary relation component parameter:

```
C1: relation::numeric(Id1,X) ⇐gate::annotation(X,"NUMBER",_),
    param::instantiation_id(Id1).
C2: relation::non_clickable(Id2,X) ⇐dom::element(X,_,_,_,_),
    ¬(dom::clickable(X)), param::instantiation_id(Id2).
C3: relation::visual_ proximity(Id3,X,Y).
C4: relation::left_visual_neighbour(Id4,X,Y).
C5: relation::direct_left_visual_neighbour(Id5,X,Y) ⇐
    relation::left_visual_neighbour(CId1,X,Y),
    ¬relation::indirect_left_visual_neighbour(CId2,X,Y),
    param::sibling_pred(CId3,Y),
    param::instantiation_id(Id5).
   relation::indirect_left_visual_neighbour(CId2,X,Y) ⇐
    relation::left_visual_neighbour(CId4,X,Y),
    relation::left_visual_neighbour(CId5,X,Z),
    css::box(Y,LeftY,_,RightY,_), css::box(Z,LeftZ,_,RightZ,_),
    RightY ≤ LeftZ, RightX ≤ LeftY,
    param::instantiation_id(CId2).
C6: relation::binary_unary(Id6,X,Y) ⇐
    param::binary_pred(CId1,X,Y),
    param::node_pred(CId2,X),
    param::sibling_pred(CId3,Y),
    param::instantiation_id(Id6).
```

Figure 3.3: Rules corresponding to components C_1-C_6

$$\mathfrak{C}_5[\texttt{sibling\_pred} \mapsto C_1];\ ((C_3 \lhd C_4) \parallel C_1) \lhd C_5$$

Finally, we give the instantiation and binding expression of binary relation component C_6 according to our example:

$$\mathfrak{C}_6[\texttt{binary\_pred} \mapsto C_5, \texttt{node\_pred} \mapsto C_1, \texttt{sibling\_pred} \mapsto C_2];$$
$$((((C_2 \parallel C_3) \oplus C_4) \oplus C_5) \parallel C_2) \lhd C_6$$

Note that it seems more obvious to define the composition expression for C_5 as $((C_1 \parallel C_3) \lhd C_4) \lhd C_5$, but in that case we would have to recompute C_1 at the time of evaluation of the final component C_6, and the composition expression for C_6 would have been $((((C_1 \parallel C_3) \lhd C_4) \lhd C_5) \parallel (C_1 \parallel C_2)) \lhd C_6$, as the semantics of the $\lhd$ operator would have restricted visibility of C_2 at the time of the computation of C_6 and we would have encountered an additional computational overhead.

For the purpose of brevity, later in this chapter and in the following chapters we omit the definition of component types, links to the rules, and their attachment to the sub-components through the $\lhd$ operator and combine the ground instantiations and composition expressions in a more concise notation.

Example 26. In the concise notation component C_5 can be defined as

$\mathfrak{C}_6$[binary_pred$\mapsto C_5$,node_pred$\mapsto C_1$,sibling_pred$\mapsto C_2$]; $(((C_1 \parallel C_3) \oplus C_4) \oplus C_5) \parallel C_2$

3.7 Case study: template relations and global features

Different block types often have overlapping features that can be covered through the global feature repository, but it is even more common for them to differ only slightly, e.g., with respect to what DOM nodes to consider. Therefore, it would be highly beneficial both for the flexibility of the system to the introduction of new classifiers and for overall performance reduction to introduce template relations that follow certain patterns (we list examples of common patterns in Chapter 1) and that differ only in narrow aspects of their definition (e.g., the width and height of visual proximity in pixels or a specific annotation that we want the candidate nodes to have, such as numeric annotation for the numeric "next" links or the price annotation for the price tags for the Real Estate, Used Cars and Online Retail domains).

Example 27. Let us reconsider the example of a component model from Section 3.2, the definition and computation of which has been given in Section 3.6. We have defined two namespaces: `param` for instantiable parameters and `relation` for instantiable relations. The `param` namespace defines the name of the parameter and the value within the rule component to which it links, whilst the `relation` namespace fixes a certain predicate or function (in case it has an associated value) as instantiable. We distinguish between binary relations that hold between two nodes and unary relations that hold for one node.

We add a general `relation::binary_unary(Id`$_1$`,X,Y)` relation template through which we instantiate the feature, as well as `relation::direct_left_visual_neighbour(Id`$_3$`,X,Y)`, `relation::numeric(Id`$_6$`,X)` and `relation::non_clickable(Id`$_7$`,X)` relations that represent that node `X` is a direct left visual neighbour of node `Y`, nodes `X` and `Y` are numeric, and node `Y` is non-clickable (we omit giving the rules for `relation::left_visual_neighbour(Id`$_4$`,X,Y)` and `relation::visual_proximity(Id`$_5$`,X,Y)` as these are standard relations, and their definitions are presented in Figure 3.2):

```
relation::binary_unary(Id1,X,Y) ⇐
    param::binary_pred(CId1,X,Y), param::node_pred(CId2,X),
    param::sibling_pred(CId3,Y), param::instantiation_id(Id1).
relation::indirect_left_visual_neighbour(Id2,X,Y) ⇐
    relation::left_visual_neighbour(CId4,X,Y),
    relation::left_visual_neigbour(CId5,X,Z),
    css::box(Y,LeftY,_,RightY,_), css::box(Z,LeftZ,_,RightZ,_),
    RightY ≤ LeftZ, RightX ≤ LeftY,
```

```
    param::instantiation_id(Id2).
relation::direct_left_visual_neighbour(Id3,X,Y) ⇐
    relation::left_visual_neighbour(CId6,X,Y),
    ¬(relation::indirect_left_visual_neighbour(CId7,X,Y)),
    param::sibling_pred(CId8,Y),
    param::instantiation_id(Id3).
relation::numeric(Id6,X) ⇐ gate::annotation(X,"NEXT",_),
    param::instantiation_id(Id6).
relation::non_clickable(Id7,X) ⇐ dom::element(X,_,_,_,_), ¬(dom::clickable(X)),
    param::instantiation_id(Id7).
```

All parameters get assigned with concrete values or predicates at the time of instantiation, i.e., the semantics of the instantiation operator on a given relation component C that contains atomic, sequential, and component parameters is that all component parameters $\{\texttt{param::p}_1,\ldots,\texttt{param::p}_n\}$ included in this component need to be mapped to lower-order relations (or grounded relations; if that predicate in turn has constituent relations, these sub-relations need to be resolved as well), and all atomic and sequential parameters of this component $\texttt{param::p}_{n+1},\ldots,\texttt{param::p}_{n+m}$ of types $\tau_{n+1},\ldots,\tau_{n+m}$ need to be mapped to concrete values:

$$\mathfrak{C}[\texttt{p}_1\mapsto\texttt{relation::rel}_1,\ldots,\texttt{p}_n\mapsto\texttt{relation::rel}_n,\texttt{p}_{n+1}\mapsto\texttt{val}_1\in\text{domain}(\tau_{n+1}),\ldots,\texttt{p}_{n+m}\mapsto\texttt{val}_m\in\text{domain}(\tau_{n+m})]$$

Each time a feature or a relation $\texttt{relation::rel}_1$ gets defined through another relation $\texttt{relation::rel}_2$, $\texttt{relation::rel}_2$ gets attached to $\texttt{relation::rel}_1$ through either an isolated sequential composition operator ($\lhd$) or a sequential but possibly dependent composition operator ($\oplus$).

We can now extract the feature by connecting and instantiating these relations through the component model operators described in Section 3.2 (for the purpose of notational concision we map $\texttt{relation::binary\_unary(Id}_1\texttt{,X,Y)}$ to binary relation component $\texttt{C}_1$, $\texttt{relation::direct\_left\_visual\_neighbour(Id}_3\texttt{,X,Y)}$ to binary relation component $\texttt{C}_2$, $\texttt{relation::left\_visual\_neighbour(Id}_4\texttt{,X,Y)}$ to binary relation component $\texttt{C}_3$, $\texttt{relation::visual\_proximity(Id}_5\texttt{,X,Y)}$ to binary relation component $\texttt{C}_4$, $\texttt{relation::numeric(Id}_6\texttt{,X)}$ to unary relation component $\texttt{C}_5$, and $\texttt{relation::non\_clickable(Id}_7\texttt{,X)}$ to unary relation component $\texttt{C}_6$) :

$$\mathfrak{C}_1[\texttt{binary\_pred}\mapsto C_2,\texttt{node\_pred}\mapsto C_5,\texttt{sibling\_pred}\mapsto C_6],\ \mathfrak{C}_2[\texttt{sibling\_pred}\mapsto C_5],\ C_4[\texttt{dH}\mapsto\texttt{70},\texttt{dV}\mapsto\texttt{50}];\ ((((C_4 \parallel C_5)\oplus C_3)\oplus C_2)\parallel C_6)\lhd C_1$$

Note that the $\texttt{param::sibling\_pred(Id}_3\texttt{,Y)}$ parameter gets instantiated with two different relations ($\texttt{relation::numeric(\_,Y)}$ and $\texttt{relation::non\_clickable(\_,Y)}$ for

```
relation::annotated_by(Id,X,Atype) ⇐ param::node_of_interest(CId,X),
 gate::annotation(X,AType,_), param::annotation_type(PId,AType),
 param::instantiation_id(Id).
relation::annotation_coverage(Id,X,Atype,Coverage) ⇐
 param::node_of_interest(CId,X),
 AnnotationSpan = #sum{N1: relation::leaf_descendant(Y,X), dom::content(Y,_,_,N1),
 gate::annotation(Y,AType,_)},
 Span = #sum{N2: relation::leaf_descendant(Y,X),  dom::content(Y,_,_,N2)},
 Coverage = AnnotationSpan/Span, param::annotation_type(PId,AType),
 param::instantiation_id(Id).
relation::in_visual_proximity_property(Id,X) ⇐ param::node_of_interest(CId1,X),
 relation::visual_proximity(CId2,Close,X), param::property(CId3,Close),
 param::instantiation_id(Id).
relation::num_in_visual_proximity_property(Id,X,Num) ⇐
 param::node_of_interest(CId1,X), relation::visual_proximity(CId2,Close,X),
 Num = #count{Close: param::property(CId3,Close)},
 param::instantiation_id(Id).
relation::relative_position_within(Id,X,(PosH,PosV)) ⇐
 param::node_of_interest(CId1,X), css::box(X,LeftX,TopX,_,_),
 PosH = LeftX/Width, PosV = TopX/Height, param::boundaries(CId2,Height,Width),
 param::instantiation_id(Id).
relation::contained_in(Id,X) ⇐
 param::node_of_interest(CId1,X), css::box(X,LeftX,TopX,RightX,BottomX),
 Left < LeftX < RightX < Right, Top < TopX < BottomX < Bottom,
 param::container(CId2,Left,Top,Right,Bottom),
 param::instantiation_id(Id).
```

Figure 3.4: A sample of BER$_y$L's template relations

`relation::binary_unary(Id1,X,Y)` and `relation::direct_left_visual_proximity(Id3,X,Y)` relations. The `relation::binary_unary(Id1,X,Y)` relation template represents a very powerful pattern that can be used in multiple classifiers, whilst the `relation::numeric(Id6,X)` relation is less powerful, but can also be used in multiple classifiers and among different features within the same classifier.

Example 28. Figure 3.4 shows a sample of other template relations in BER$_y$L. The first two templates define Boolean features for any annotation type `AType`, indicating whether a certain node of interest is annotated with `AType` and the coverage of that annotation over the full text of the given node. We instantiate them for numbers for the Pagination Link classifier in the following way:

```
relation::annotated_by[annotation_type↦["NUMBER"]]
```

The third and fourth templates define a Boolean feature that holds for nodes of interest if there is another node in their visual proximity for which `Property(Close)` is true (this

template is similar to the `binary_unary(Id,X,Y)` template, but less powerful) and the number of such nodes in the visual proximity of the given node. We can instantiate it to nodes that are annotated with PAGINATION for the Pagination Link classifier in the following way:

```
((relation::annotated_by[annotation_type↦["PAGINATION"]])◁
(relation::visual_proximity[dh↦150,dV↦50]))◁
relation::in_visual_proximity_property[property↦relation::annotated_by]
```

The fifth template defines a numeric feature that determines the relative horizontal and vertical positions of the CSS box of a given node within a predetermined rectangle. Note that we need an additional Java arithmetic component to compute the relative positions, as Datalog does not support division. To check the relative position of the node within the page we can perform the following instantiation:

```
relation::relative_position_within[boundaries↦relation::page_boundaries]
```

Finally, the sixth template defines a Boolean feature that determines whether the CSS box of a given node is fully located within a predetermined rectangle. If we want to check whether a node is located within the first screen of the page, we can instantiate this template in the following way:

```
relation::contained_in[container↦relation::first_screen_container]
```

In BER$_y$L, we store all template relations and global features that get used in more than one classifier (such as `relation::annotated_by(Id,X,AType)` template relation and `feature::box_width(Id,X)` global feature) under a separate namespace, called Global Component or `GM`. This allows us to optimise performance, as we can pre-compute these template relations and global features at the beginning of the feature extraction process, rather than re-compute them individually for each classifier that employs them.

3.8 Web block classification

In this section we briefly describe BER$_y$L's classification component, discuss the challenges of feature extraction, and the ways in which we can optimise the manual effort for building the training corpus.

Classification component

We start by briefly discussing the procedural classification component. The classification component takes features as its input and produces classifications as output, hence its input/output schema is INPUT=`feature` and OUTPUT=`cls`. This component can run in two

Figure 3.5: An example of a pagination bar with two elements that hold 'next' annotations

modes: the training mode and the classification mode. In the training mode, we run the classification component on a trained corpus of labelled feature vectors (therefore the input of the classification component in the training phase is in fact INPUT =`feature`∪`cls`), and the side effect is the generation of a serialisable classifier for a given block type. We run the classification component in the training mode only once for each given block type, and can re-use the generated classifier multiple times during the run-time when we execute the component in the classification mode. Therefore, it is acceptable if the classifier takes a long time to create from the training corpus, as long as its running time in the classification mode is acceptable. We discuss the implementation details of the classification component as well as the auxiliary arithmetic component that allows us to use arithmetic operations not supported by BER$_y$L language, such as division, and real numbers in Chapter 4.

Feature extraction in BER$_y$L

Suppose we want to write a BER$_y$L-language rule-based component to extract a basic feature for a given block type. How do we approach this problem? We first have to convert a HTML representation of a page into a set of input facts, which can then be processed by a BER$_y$L-language program. The task of building an input facts file corresponds to the shredding of a DOM tree into relations and using an XML labelling schema to encode the IDs of the DOM nodes. The `Start` and `End` IDs of a node correspond to a pre-order traversal of the DOM tree with a single incremental counter that assigns the start label the first time the node has been explored, and the end label when all of the node's descendants have been explored.

Example 29. Let us assume we have the following sample of an annotated DOM:

```
dom::element(e_2167_a,a,2150,2167,2169)
dom::clickable(e_2167_a)
gate::annotation(e_2167_a,"NUMBER","15")
dom::element(e_2189_a,a,2150,2189,2198)
dom::clickable(e_2189_a)
gate::annotation(e_2189_a,"NEXT","Next Page")
```

In this sample the `dom::element(N,T,PStart,Start,End)` input fact represents a DOM element identified by its Id `N`. `Start` and `End` identify the start and end labels of this node, whilst `PStart` identifies the start label of its parent node. `T` identifies the tag of a given DOM

element, such as `div` or `img`. The annotation input fact `gate::annotation(N,A,v)` identifies that DOM node `N` is annotated with an annotation of a certain type `A` with normalised value `v`. Here NEXT and NUMBER are annotation types that come from a GATE [24] gazetteer list that includes values such as 'next' and '›'.

Example 30. Now, let us consider a sample rule-based feature component and define its respective extraction rule in BER_yL. We take one of the features used in the pagination link classifier. Suppose we want to check whether the closest link to the visual left of a given link has a NEXT link annotation (suppose we are extracting this feature for the pagination bar presented in Figure 3.5).

Both the "Next ›" and "Last ››" DOM element nodes contain NEXT annotations, which are {'next', '›'} and {'››'}, respectively. Hence, the feature renders **`False`** for the "Next ›" node and **`True`** for the "Last ››" node. We can now define the rules that represent this particular feature for the pagination link block type (we use negation to ensure that there is no node `Z`, which holds a NEXT annotation located between nodes `X` and `Y`):

```
feature::pagination::left_visual_neighbour_is_next_link(Id,X) ⇐ dom::clickable(X),
  relation::left_visual_proximity(X,Y), gate::annotation(Y,"NEXT",_),
  ¬(relation::left_visual_proximity(X,Z), relation::left_visual_proximity(Z,Y),
  gate::annotation(Z,"NEXT",_)), param::instantiation_id(Id).
```

Note that we could have defined the `pagination::left_visual_neighbour_is_next_link` feature in a much easier way, by using template relations; that is described in detail in Section 3.7.

Feature extraction is the bottleneck of our classification approach, as the actual execution of the classifier takes significantly less time. Hence, we would like to optimise the rules written in the BER_yL language as much as possible or trim the input facts (we describe this method in more detail further in this section), as both factors affect the performance of the feature extraction approach. An alternative approach to performance optimisation would be the use of global features and feature templates to instantiate features that are the same, or similar, for different block types synchronously.

A completely manual design of new classifiers is a very labour-intensive process that takes a lot of human effort, not only to find a correct set of features for individual blocks, but also to build a training corpus that represents the diversity of these blocks. Hence, when we attempt to build a classification system for dozens of domain-independent blocks, we need to solve two tasks: **(1)** find a minimal set of features that will be able to render required accuracy results at an acceptable performance rate; and **(2)** minimise the human effort required to build a training corpus, whilst preserving the diversity of the corpus in

regard to the variety of its constituent blocks' structural and visual representations on the web.

Creation of an optimal set of features for web block classification

There are several ways in which we can find a minimal set of features that meets the precision, recall, and performance constraints imposed by our system for individual web blocks. To find an optimal set of features that meets the accuracy requirements we can start with a broad set of global features, new template-instantiated features, and *ad hoc* defined features, where the total number of features is some number n, and then perform an introspective evaluation on its subsets of cardinality $n-1$. We can then select a subset that yields a maximal F_1 measure **and** surpasses both the precision and recall thresholds, and perform an introspective evaluation on its subsets of cardinality $n-2$. We carry on with this greedy algorithm approach until we reach the point at which no subset of cardinality $k-1$ yields precision and recall values that surpass the respective thresholds and select its superset of cardinality k that has yielded a maximal F_1 measure.

Another way of improving the performance of feature extraction is the optimisation of the BER_yL language extraction rules, which has to be done manually, and contraction of the input facts file, which can be done automatically. We propose performing the contraction in the following way: **(1)** extract all the input facts needed for the initial set of features; **(2)** after multiple steps of the greedy algorithm described in the paragraph above, which yields an optimal subset of features, build a classifier based on this set; **(3)** after obtaining such a classifier we can determine which input facts are precisely needed for the process of block classification; **(4)** we can then trim the initial input facts according to these criteria and use only the trimmed set of facts for the feature extraction for this specific block type on the runtime pages. We now present an example of this approach.

Example 31. Suppose we start with the following subset of the input facts required for 'next' link identification:

```
dom::element(e_2189_a,a,2150,2189,2198)
dom::clickable(e_2189_a)
gate::annotation(e_2189_a,"NEXT","Next Page")
dom::element(e_2303_a,a,2302,2303,2325)
dom::clickable(e_2303_a)
gate::annotation(e_2303_a,"NEXT","Look at more images >>")
dom::element(e_2435_a,a,2405,2435,2453)
dom::clickable(e_2435_a)
gate::annotation(e_2435_a,"NUMBER","+44 (0)1865 273838")
```

We assume the resultant classifier implies that only DOM nodes with 'numeric' and 'next' annotations can be identified as 'next' links, and furthermore that nodes with the 'next' annotation can have at most 15 characters, and that nodes with the 'numeric' annotation can have at most five characters. In that case, the initial subset of the DOM facts gets trimmed to the following:

```
dom::element(e_2189_a,a,2150,2189,2198)
dom::clickable(e_2189_a)
gate::annotation(e_2189_a,"NEXT","Next Page")
```

Optimisation of the manual effort for building the training corpora

The training corpora we have to build for domain-independent web block classifiers can substantially vary in size, depending on the diversity of the structural and visual representation of particular web blocks. However, as we have mentioned above, due to the large-scale nature of the domain-independent approach to web block classification, we need to minimise the size of training corpora, as it takes a lot of human effort to find representative web pages and manually label nodes corresponding to different block types. A possible solution to this problem would be to take a **semi-supervised learning approach** to training the classifiers.

We start with a relatively small kernel of web pages, selected in such a way that it is as diverse as possible with regard to the structural and visual layout of the block in question. We then run the classifier generated from this kernel on a wide set of pages selected from the unified corpus of web pages and re-train it based on the newly labelled data. If the new classifier does not render acceptable precision and recall results on a subset of the evaluation corpus, the user can investigate the instances of the evaluation corpus that have been mislabelled (the implementation of our system allows the highlighting of the CSS boxes of classified blocks in the browser's visual rendering, as exemplified in Figure 3.6, so it should not be a time-consuming task), find the k-closest neighbours of each of these instances in the extended training corpus, check whether they have been correctly labelled, alter the incorrectly classified instances, retrain the classifier, and then run it on another subset of the evaluation corpus.

3.9 Holistic approach to web block classification in BER$_y$L

Let us revisit the problem of web block classification based on a holistic view of the page that we have introduced in Chapter 1. The motivation for this is to increase the accuracy of

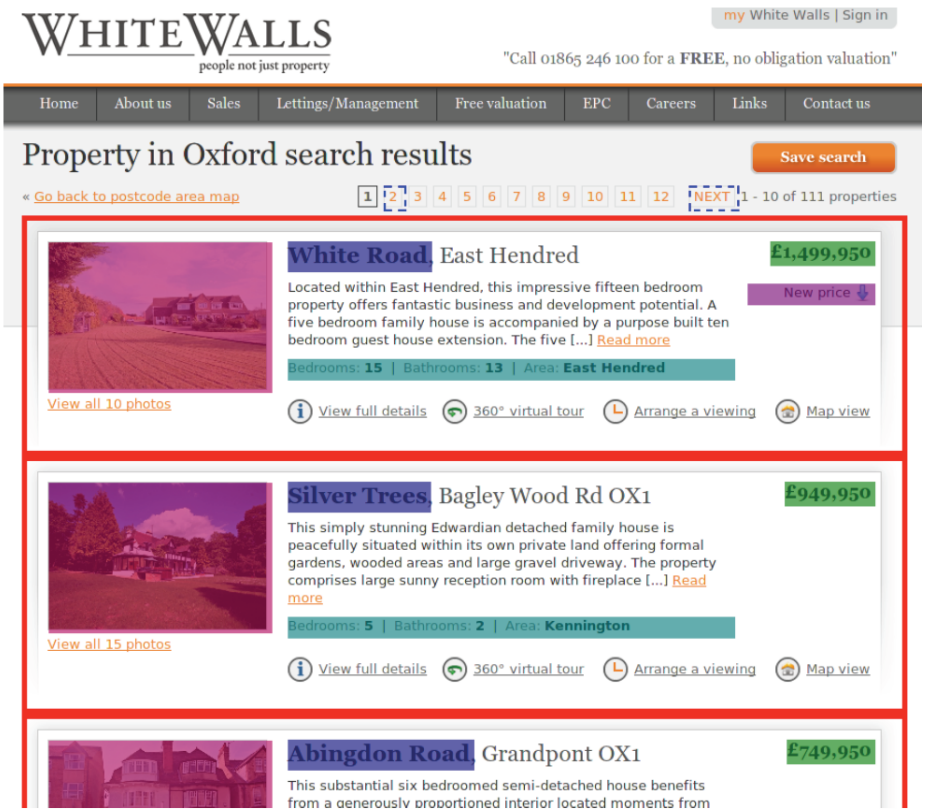


Figure 3.6: An example of web block highlighting in DIADEM

classifiers for blocks of different types located on the same page. We give an example of how it works in practice in Chapter 1 (Figure 1.3).

Given the set of classifiers, how do we build a set of constraints that increases the accuracy for some classifiers whilst not dropping the accuracy for other classifiers?

BER$_y$L tries to derive a holistic interpretation of a web page from the individual classifiers. As such, it uses constraints to find possible conflicts between classifiers and resolve them. Formally, we recall the notion of a classifier from Definition 5. As noted there, classifiers are applied to features of DOM nodes (identifying a sub-tree in the input DOM) and produce (positive or negative) labels.

Constraints are built from predicates, which represent that a given node $n \in T_P$ is classified with a certain classification label $L \in L^+_{BT}$. They may also use other predicates presented in Table 3.1, e.g., `dom::clickable(N)` and relations, but not features.

Example 32. An example of a constraint is that a header and a footer cannot be contained within each other or overlap. This constraint translates into the following BER$_y$L fragment (here and in the following we use $\Longrightarrow$ as syntactic sugar):

```
constraint::containment(Id,N,N′,true) ⇐
  param::instantiation_id(Id), relation::contained_in(RId,_,_),
  (cls::classification(N,header,header), cls::classification(N′,footer,footer)) ⟹
   ¬relation::contained_in(RId,N,N′), ¬relation::contained_in(RId,N′,N)).
```

This maps all pairs of nodes for which the above constraint holds to **true**. We call these pairs *witness pairs* of the constraint. An analogous rule maps all *violator pairs* for which the constraint is not satisfied to **false**. If there is at least one violator pair for a given constraint ξ , there is a conflict for this constraint on the given set of classifications, and BER$_y$L drops one of the violating classifications (see Algorithm 1), and then checks whether ξ is now satisfiable and that there are no violator pairs produced by it.

We verify the constraints over the set of initial classifications in the `cls` namespace produced by all constituent classifiers of the BER$_y$L system. As mentioned above, if a constraint ξ between classification types BT_1 and BT_2 is not satisfied on some pair (N,N'), a heuristic must be applied to remove one or more of the violating classifications on the involved nodes. In the current version of BER$_y$L (in the greedy constraints resoluion Algorithm 2), we employ a heuristic that first maximises the number of newly satisfied constraints. If there is a tie, it breaks that tie by removing a violating classification on a node with smaller relative size. The size is measured as the average of **(a)** the area of the CSS box of the node, and **(b)** the total number of DOM nodes in sub-tree(s) rooted at this node.

3.9.1 Heuristic constraint resolution for BER$_y$L

We now describe the heuristics we use to determine the subset of the original set of classifications according to our optimisation criteria with regard to the given system of constraints, first for the naïve algorithm and then for the greedy algorithm.

The input is the system of constraints that we check for being satisfied and the set of classifications produced for all block types present in this system of constraints.

In the naïve constraints resolution algorithm, we traverse the set of constraints, for each constraint ξ_i checking which classification labels are present in ξ_i (by calling function $GET\text{-}LABELS(\xi_i)$) and evaluating the constraint on the subset $\mathscr{C}_i \in \mathscr{C}_{in}$ of input classifications (acquired through $\mathscr{C}_i \leftarrow cls :: classification(X,BT,Label) \wedge Label \in \mathscr{L}_{BT} \subset \mathscr{L}_i$). If there is a set of classifications for which the given constraint is not satisfied (denoted as $\mathscr{C}_{conflict}$), we randomly remove one of these classifications from the conflicting set of classifications (by calling function $REMOVE\text{-}RANDOM\text{-}CLASSIFICATION(\mathscr{C}_{conflict})$) and re-run the algorithm on the reduced set of classifications. We repeat this process of trimming the set of classifications until all constraints get satisfied on the trimmed set. The

Algorithm 1: The naïve constraints satisfaction algorithm

Data: DOM tree T_p with classifications as in Table 3.1, a system of constraints $\Xi(T_P,\emptyset) = \{\xi_i | i \leq |\Xi(T_P,\emptyset)|\}$, and a classification $\mathscr{C}_{in}$ over a set of given block types $\mathbb{BT}$

Result: A classification $\mathscr{C}_{out}$ over a set of given block types $\mathbb{BT}$ on the nodes of the DOM tree T_p

Procedure $SATISFYING\text{-}CLASSIFICATIONS(\Xi(T_P,\emptyset),\mathscr{C}_{in})$
  for $(i \leftarrow 0;\ i \leq |\Xi(T_P,\emptyset)|;\ i{+}{+})$ **do**
    $\mathscr{L}_i \leftarrow GET\text{-}LABELS(\xi_i)$;
    $\mathscr{C}_i \leftarrow cls :: classification(\_, BT, Label) \wedge Label \in \mathscr{L}_{BT} \subset \mathscr{L}_i$;
    $\mathscr{C}_{conflict} \leftarrow EVALUATE\text{-}CONSTRAINT(\xi_i, \mathscr{C}_i, T_P)$;
    if $\mathscr{C}_{conflict} \neq \emptyset$ **then**
      $l \in \mathscr{C}(n) \leftarrow REMOVE\text{-}RANDOM\text{-}CLASSIFICATION(\mathscr{C}_{conflict})$;
      $\mathscr{C}' \leftarrow \mathscr{C}_{in} \setminus l$;
      $SATISFYING\text{-}CLASSIFICATIONS(\Xi(T_P,\emptyset),\mathscr{C}'))$;
  $\mathscr{C}_{out} \leftarrow \mathscr{C}_{in}$;
  return $\mathscr{C}_{out}$;

running time complexity of this algorithm is $O(Eval(\xi,\mathscr{C}_{in}) \times |\Xi(T_P,\emptyset)| \times |\mathscr{C}_{in}|)$, where $Eval(\xi,\mathscr{C}_{in})$ is the worst-case data complexity of evaluating a constraint $\xi \in \Xi(T_P,\emptyset)$ (i.e., the polynomial data complexity of evaluating a fixed Datalog program) on classification $\mathscr{C}_{in}$ (acquired through function $EVALUATE\text{-}CONSTRAINT(\xi_i,\mathscr{C}_{in},T_P)$).

Note that deleting one classification and making the conflicting constraint satisfiable can turn another constraint that was previously satisfiable into unsatisfiable as demonstrated in Example 33.

Example 33. Suppose there are two constraints in our system for two different classification labels A and B. The first constraint ξ_1 states that if there is a DOM node with classification label B, the CSS box of which is located under the CSS box of a node with classification label A, and these two boxes have an overlap in their horizontal coordinates, then the width of the CSS box with classification label B should be smaller than the width of the CSS box with classification label A. The second constraint ξ_2 states that if there are multiple nodes with classification label A the vertical coordinates of which have an overlap, the number of such nodes must be even. Now, let us assume we have four nodes with classification label A, four nodes with classification label B and the visual configuration of their respective CSS boxes is the one shown in Figure 3.7. Suppose our algorithm first checks for the satisfaction of constraint ξ_2 and then for the satisfaction of constraint ξ_1. Constraint ξ_2 is satisfied by all six nodes, and we move on to constraint ξ_1, which is not satisfied by node 3 with label A and node 7 with label B. We then randomly remove node 3 from the set of classifications. However, then constraint ξ_1 is no longer satisfied for the top

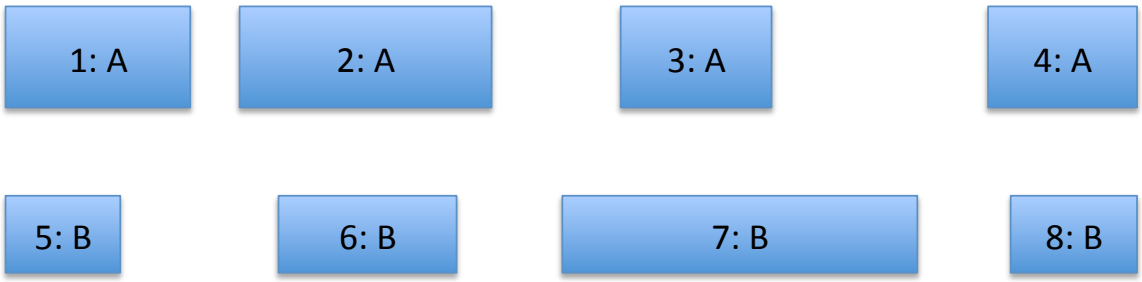


Figure 3.7: An example of a constraint that was previously satisfied, no longer being satisfied after the set of classifications gets reduced at the application of another constraint

row, as the number of nodes (and corresponding CSS boxes) in that row is now odd, and we have to randomly remove node 4, so that the system of two constraints gets satisfied on all remaining nodes.

It is obvious that deleting classifications at random in the case of a conflict can eliminate a lot of True Positive classifications. We therefore need to define a more refined strategy for conflict resolution. We now give our greedy algorithm for constraints satisfaction based on the two heuristics introduced in Definition 8 of Chapter 2.

In the greedy algorithm (Algorithm 2), in case of a conflict within a constraint we apply two optimisation heuristics for deleting a conflicting classification: (1) we first check for a classification the removal of which makes the maximal number of constraints that were not satisfied with it present in the classification set into valid ones (by calling function $GET\text{-}OPTIMAL\text{-}SUBSET(\mathscr{C}_{conflict}, constraints\text{-}resolved)$), and (2) in case there are multiple classifications the removal of which achieves the maximal number of constraints that get satisfied, we remove the one with the lowest coverage ratio described in Definition 8 (by calling function $MINIMAL\text{-}COVERAGE\text{-}RATIO(\mathscr{C}')$). The running time complexity of the greedy algorithm is $O(Eval^2(\xi, \mathscr{C}_{in}) \times |\Xi(T_P, \emptyset)|^2 \times |\mathscr{C}_{in}|^2)$, where $O(Eval^2(\xi, \mathscr{C}_{in})$ corresponds to the polynomial data complexity of Datalog. The semantics of both the naïve and greedy algorithms can be implemented in the form of a declarative program in the BER$_y$L language, with constraint resolution rules based on the sets of violator pairs produced by individual constraints.

3.9.2 Case study: BER$_y$L's system of constraints

In the current version of BER$_y$L there are six constraints: **(1)** the topological containment constraint; **(2)** the width constraint; **(3)** the vertical distance constraint (the navigation menu must be visually contained in either a header or a sidebar, the footer must have the same width as the header, the width of the sidebar can be at most half the width of the header,

Algorithm 2: The greedy constraints satisfaction algorithm

Data: DOM tree T_p with classifications as in Table 3.1, a system of constraints $\Xi(T_P, \emptyset) = \{\xi_i | i \leq |, \Xi(T_P, \emptyset)|\}$ and a classification $\mathscr{C}_{in}$ over a set of given block types $\mathbb{BT}$

Result: A classification $\mathscr{C}_{out}$ over a set of given block types $\mathbb{BT}$ on the nodes of the DOM tree T_p

Procedure $SATISFYING\text{-}CLASSIFICATIONS\text{-}GREEDY(\Xi(T_P, \emptyset), \mathscr{C}_{in})$
  for $(i \leftarrow 0;\ i \leq |\Xi(T_P, \emptyset)|;\ i{+}{+})$ **do**
    $\mathscr{L}_i \leftarrow GET\text{-}LABELS(\xi_i)$;
    $\mathscr{C}_i \leftarrow cls :: classification(\_, BT, Label) \wedge Label \in \mathscr{L}_{BT} \subset \mathscr{L}_i$;
    $\mathscr{C}_{conflict} \leftarrow EVALUATE\text{-}CONSTRAINT(\xi_i, \mathscr{C}_i, T_P)$;
    if $\mathscr{C}_{conflict} \neq \emptyset$ **then**
      for $(j \leftarrow 0;\ j \leq |\mathscr{C}_{conflict}|;\ j{+}{+})$ **do**
        $\mathscr{C}_{deleted} = \{\mathscr{C}_{conflict}\}_j$;
        $constraints\text{-}resolved_j \leftarrow 0$;
        for $(k \leftarrow 0;\ k \leq |\Xi(T_P, \emptyset)|;\ k{+}{+})$ **do**
          $\mathscr{L}_k \leftarrow GET\text{-}LABELS(\xi_k)$;
          $\mathscr{C}_k \leftarrow cls :: classification(\_, BT, Label) \wedge Label \in \mathscr{L}_{BT} \subset \mathscr{L}_k$;
          $\mathscr{C}'_{conflict} \leftarrow EVALUATE\text{-}CONSTRAINT(\xi_k, \mathscr{C}_k, T_P)$;
          $\mathscr{C}''_{conflict} \leftarrow EVALUATE\text{-}CONSTRAINT(\xi_k, \mathscr{C}_k \setminus \mathscr{C}_{deleted}, T_P)$;
          if $(\mathscr{C}'_{conflict} \neq \emptyset \wedge \mathscr{C}''_{conflict} = \emptyset)$ **then**
            $constraints\text{-}resolved_j \leftarrow constraints\text{-}resolved_j + 1$;
          else if $(\mathscr{C}'_{conflict} = \emptyset \wedge \mathscr{C}''_{conflict} \neq \emptyset)$ **then**
            $constraints\text{-}resolved_j \leftarrow constraints\text{-}resolved_j - 1$;
      $\mathscr{C}' \leftarrow GET\text{-}OPTIMAL\text{-}SUBSET(\mathscr{C}_{conflict}, constraints\text{-}resolved)$;
      if $|\mathscr{C}'| = 1$ **then**
        $\mathscr{C}'' \leftarrow \mathscr{C}_{in} \setminus \{\mathscr{C}'\}$;
      else
        $l \in \mathscr{C}(n) \leftarrow MINIMAL\text{-}COVERAGE\text{-}RATIO(\mathscr{C}')$;
        $\mathscr{C}'' \leftarrow \mathscr{C}_{in} \setminus l$;
      $SATISFYING\text{-}CLASSIFICATIONS\text{-}GREEDY(\Xi(T_P, \emptyset), \mathscr{C}'')$;
  $\mathscr{C}_{out} \leftarrow \mathscr{C}_{in}$;
  return $\mathscr{C}_{out}$;

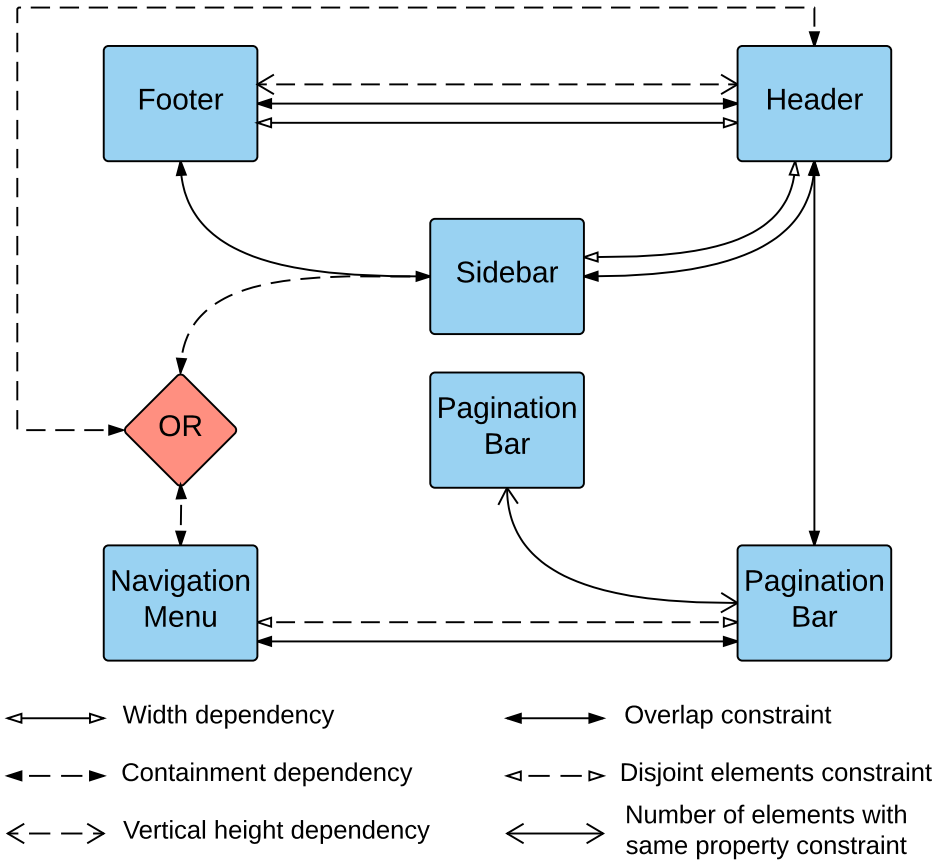


Figure 3.8: Holistic approach to web block classification in BER$_y$L

and the vertical distance between a header and footer must be at least the height of a screen if the overall height of a page is bigger than one screen, and half the screen's height if not); **(4)** there is no topological overlap between the CSS boxes and DOM sub-trees corresponding to the two blocks in question; **(5)** the contents of the structural elements of two blocks are disjointed (e.g., there cannot be a link leading to the same page that is present in both blocks); and **(6)** same or similar number of elements sharing the same characteristic property (e.g., the number of pagination links in two pagination bars cannot differ by more than one in cases where their width is the same or almost the same).

We now present the system of constraints used for providing a holistic view of the page in the current version of BER$_y$L. Note that we use the template approach described in Section 3.7 to define constraints in a generic way that is agnostic to concrete classifiers and classification types. We can then instantiate those constraints with concrete classifiers and classification types (such as `pagination_link` and `non_numeric_next_link`) specific to the current implementation of the BER$_y$L system. We elaborate more on the Datalog implementation of the system of constraints, the resolution heuristic into BER$_y$L, and its use of instantiable relations and parameters in Chapter 4. We also use the `constraint` names-

pace along with unique predicate names to define constraints. The visual representation of these constraints is given in Figure 3.8, and the BER_yL language rules are given below (for the purpose of brevity we omit the exact definitions of relations used in the definitions of constraints and introduce division into the BER_yL language, although the actual division is done in the arithmetic procedural component introduced in Chapter 4, and the same applies to real non-integer values, as well as $\Longrightarrow$ implication and $|Value|$ modulus operators):

```
constraint::width(Id,N1,N2,true) ⇐
  param::instantiation_id(Id),
  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
   param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
   param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2) ⟹
   param::box_width(PId5,N1,Width1), param::box_width(PId6,N2,Width2),
   Width1 ≤ Width2/WidthFactorMin, Width2 ≤  Width1*WidthFactorMax,
   param::width_factor_min(PId7,WidthFactorMin),
   param::width_factor_max(PId8,WidthFactorMax)).

constraint::containment(Id,N1,N2,true) ⇐
  param::instantiation_id(Id),
  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
   param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
   param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2)) ⟹
   relation::visually_contained_in(Id1,N1,N2),
   relation::structurally_contained_in(Id1,N1,N2)).

constraint::vertical_height(Id,N1,N2,true) ⇐
  param::instantiation_id(Id),
  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
   param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
   param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2) ⟹
   relation::top_coordinate(Id1,N1,TC1), relation::top_coordinate(Id2,N2,TC2),
   param::screen_height_adjustment(PId5,ScreenHeightAdjustmentFactor),
   css::resolution(H,V), |TC1 − TC2| ≥ ScreenHeightAdjustmentFactor*V).

constraint::no_overlap(Id,N1,N2,true) ⇐
  param::instantiation_id(Id), relation::visual_overlap(Id1,_,_),
 relation::structural_overlap(Id2,_,_),
  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
   param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
   param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2) ⟹
   ¬(relation::visual_overlap(Id1,N1,N2),
   ¬(relation::structural_overlap(Id2,N1,N2)).

constraint::disjoint_elements(Id,N1,N2,true) ⇐
  param::instantiation_id(Id),
  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
   param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
```

```
    param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2) ⟹
42   relation::disjoint_elements(Id,N1,N2)).

44 constraint::num_elements_with_same_property(Id,N1,N2,true) ⇐
    param::instantiation_id(Id),
46  (cls::classification(N1,BT1,Label1), cls::classification(N2,BT2,Label2),
    param::firstBlockType(PId1,BT1), param::firstLabel(PId1,Label1),
48  param::secondBlockType(PId3,BT2), param::secondLabel(PId4,Label2) ⟹
    relation::count_elements_with_property(Id1,N1,Count1),
50  relation::count_elements_with_property(Id1,N2,Count2),
    param::max_difference_in_counters(PId5,MaxDifferenceInCounters),
52  |Count1- Count2| ≤ MaxDifferenceInCounters, param::instantiation_id(Id)).
```

We can then instantiate these generic constraints to fit with the requirements of the BER$_y$L system:

```
constraint::width[firstBlockType↦header, firstLabel↦header,
2 secondBlockType↦footer, secondLabel↦footer,
 width_factor_min↦1, width_factor_max↦1]
4
constraint::width[firstBlockType↦header, firstLabel↦header,
6 secondBlockType↦sidebar, secondLabel↦sidebar,
 width_factor_min↦2, width_factor_max↦#maxint]
8
constraint::containment[firstBlockType↦nav_menu, firstLabel↦nav_menu,
10  secondBlockType↦[header,sidebar], secondLabel↦[header,sidebar]]

12 constraint::width[firstBlockType↦header, firstLabel↦header,
 secondBlockType↦footer, secondLabel↦footer, screen_height_adjustment↦0.8]
14
constraint::no_overlap[firstBlockType↦header, firstLabel↦header,
16  secondBlockType↦footer, secondLabel↦footer]

18 constraint::no_overlap[firstBlockType↦header, firstLabel↦header,
 secondBlockType↦sidebar, secondLabel↦sidebar]
20
constraint::no_overlap[firstBlockType↦footer, firstLabel↦footer,
22 secondBlockType↦sidebar, secondLabel↦sidebar]

24 constraint::disjoint_elements[firstBlockType↦pagination_bar,
 firstLabel↦pagination_bar, secondBlockType↦nav_menu,
26 secondLabel↦nav_menu, element_value↦relation::link_url]

28 constraint::num_elements_with_same_property[firstBlockType↦pagination_bar,
 firstLabel↦pagination_bar, secondBlockType↦pagination_bar,
30 secondLabel↦pagination_bar, property↦relation::pagination_link,
 max_difference_in_counters↦1]
```

Chapter 4

System Description

In this chapter we discuss the implementation side of the main concepts of our system. We start by outlining the overall architecture of our implementation of Component Models. We then discuss the DLV implementation of filter processors that narrow the space of DOM, CSS and Annotation facts only to those needed by the specific classifier or a system of classifiers. We then give a detailed description of the implementation of the feature extraction part of the BER$_y$L system, through both declarative and procedural approaches (realised in DLV[1] and Java respectively), and the way it employs relations and parameters defined in component models. We then talk about the implementation of the classification processors through the WEKA framework.

We conclude the chapter by giving a UML diagram that covers the most important parts of our BER$_y$L system and their interaction between each other, along with a list of examples to illustrate their concrete implementations in the context of different classifiers.

4.1 Main aspects of BER$_y$L's implementation

A typical component model that represents a single classifier consists of three processors: **(1)** a filter processor, **(2)** a feature extraction processor, and **(3)** a classification processor. Processors **(1)**-**(3)** get executed in a sequential order. Filter and classification processors are sequential, whilst feature extraction processors are parallel. The filter processor consists of an ordered list of filters, which are applied sequentially on a set of DOM, CSS, and Annotation facts acquired from a web page, with the output set of facts for each filter in the list serving as the set of input facts for the next filter in this list. The feature extraction processor computes all features required by the given component model. The computation of features is done in a parallel way, with no given order of their evaluation, for the purpose

[1] http://www.dlvsystem.com/

of optimising the performance of computing their constituent relations that often get shared between different features in the same classifier. The classification processor consists of an ordered list of classifiers that operate on the set of features acquired from the feature extraction processor, with each of the constituent classifiers operating on a subset of this feature set. The classifiers get executed in a sequential order set by the user. Note that there are two modes in which we can run the classification processor: **(1)** the **training mode**, in which BER$_y$L creates a training corpus from the given set of DOM, CSS and Annotation facts files (with each file corresponding to an individual web page) and a mapping between the tuples of `(Classifier, URL, XPath Expression)` and `Label` where the node with a `Label` from the labelling scheme of a given `Classifier` gets uniquely determined by its respective `XPath` expression on the page with a given `URL`, and **(2)** the **evaluation mode**, in which the system runs a set of precomputed classifiers on the features extracted for one given page, and enforces the system of constraints stored in the inter-classifier components on the final outputs of these classifiers.

In Figure 4.1, we present the UML diagram that provides the main interfaces of these three processors. As we can see, the interfaces that represent all three processors (`FFFilterProcessor` for the filter processor, `FFParallelProcessor` for the feature extraction processor, and `FFClassificationProcessor` for the classification processor) are implementations of the more general `FFProcessor` interface, with `FFFilterProcessor` and `FFParallelProcessor` being the implementations of the `FFComponentProcessor` interface. All components of the given classifier are dependent on the configuration file of this classifier (we also keep a global configuration file that handles global features and template relations, along with the dependencies and constraints used in the inter-classifier stage of our BER$_y$L system that is described in Section 3.9 of Chapter 3) represented by the `FFConfiguration` class, from which we can access any component, processor, parameter, or relation stored in that configuration file.

Filter processors play a crucial role in the optimisation of the system's overall performance, as they allow its constituent classifiers to work solely on the DOM, CSS, and Annotation facts that they need for extraction of their respective features. Each consecutive filter in the processor takes as its input the output of the previous filter, and hence at each stage of the filtering process the set of facts being processed decreases in size. Filter processors are implemented in DLV, and hence can be easily created and modified to handle our representation of the DOM, CSS, and Annotation facts that are also implemented in DLV. In Figure 4.2, we present a UML diagram that shows the main interfaces of a filter processor, along with relationships that hold between them.

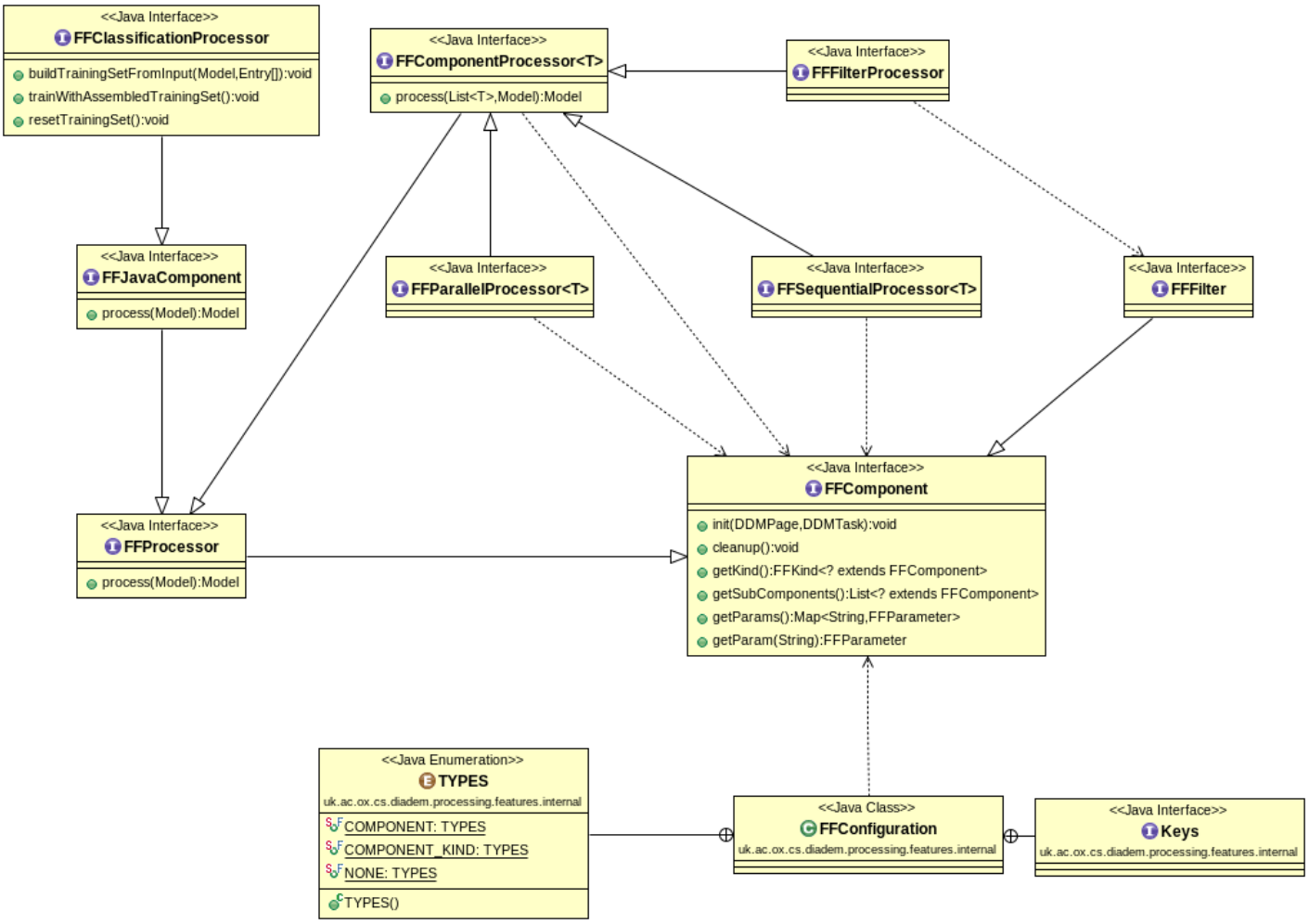


Figure 4.1: A UML diagram of the framework for the processor components of the system

Feature extraction processors play the most important role in BER$_y$L, as a modular approach to feature engineering is one of the main contributions of our system. Feature extraction can be done through **(1)** rule-based components, **(2)** procedural components, or **(3)** mixed components that combine rule-based and procedural approaches for feature extraction. In Figure 4.3, we give the UML diagram that represents the architecture of a feature extraction processor. In BER$_y$L we implement rule-based components through DLV (`FFRuleComponent`) and procedural components through Java (`FFJavaComponent`). Parameters (`FFParameter`) and relations (`FFRelation`) are implemented through rule-based components. A mixed component (`FFMixedComponent`) can take the values of unary function relations computed through the use of rule-based components as its input and apply procedural Java code to those values.

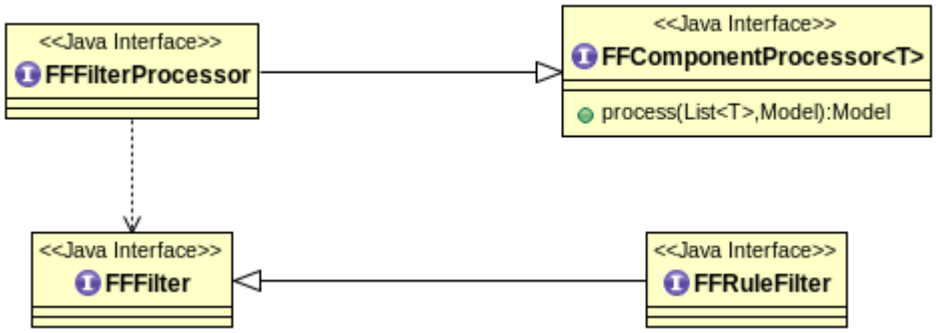


Figure 4.2: A UML diagram of the framework for a filter processor

Example 34. We can take the coordinates of the CSS box representing a particular element on the page and apply Java procedural code involving division to check whether that box is in the first screen of that page.

We use the `Evalex`[2] library implemented in the `FFEvalexArithmeticFeatureComponent` class for computing expressions from the mixed components that take the values of unary function relations as their inputs.

Parameters and relations can be of the following types: **(1)** atomic variable parameters (`FFAtomicParameter`), **(2)** list parameters (`FFListParameter`), **(3)** hierarchical parameters (`FFHierarchicalParameter`), **(4)** hierarchical list parameters (`FFListParameter`), **(5)** unary and unary function relation predicate parameters (`FFRelation`), and **(6)** binary relation predicate parameters (`FFRelation`).

Example 35. We provide examples for each type of parameter:

1. **atomic variable parameter** – a `max_number_of_characters` parameter passed to an annotation template that specifies the maximum number of characters an annotation specific to a given classifier can hold (e.g., for the numeric links in the Pagination Link classifier, we are only interested in numeric annotations that hold no more than three characters);

2. **list parameter** – an `attributes_of_interest` parameter that specifies which attributes we need for a specific classifier at a filtering stage (e.g., for the Pagination Link classifier, we only need the `class`, `title`, `name`, `value`, `id`, `alt`, `src`, `href`, `onclick`, `image`, `button`, and `type` attributes);

3. **hierarchical parameter** – an `annotation_kind_type` parameter that specifies the kind and type of annotation we want to instantiate an annotation template with (e.g., `label>next` for the Pagination Link classifier);

[2]https://github.com/uklimaschewski/EvalEx

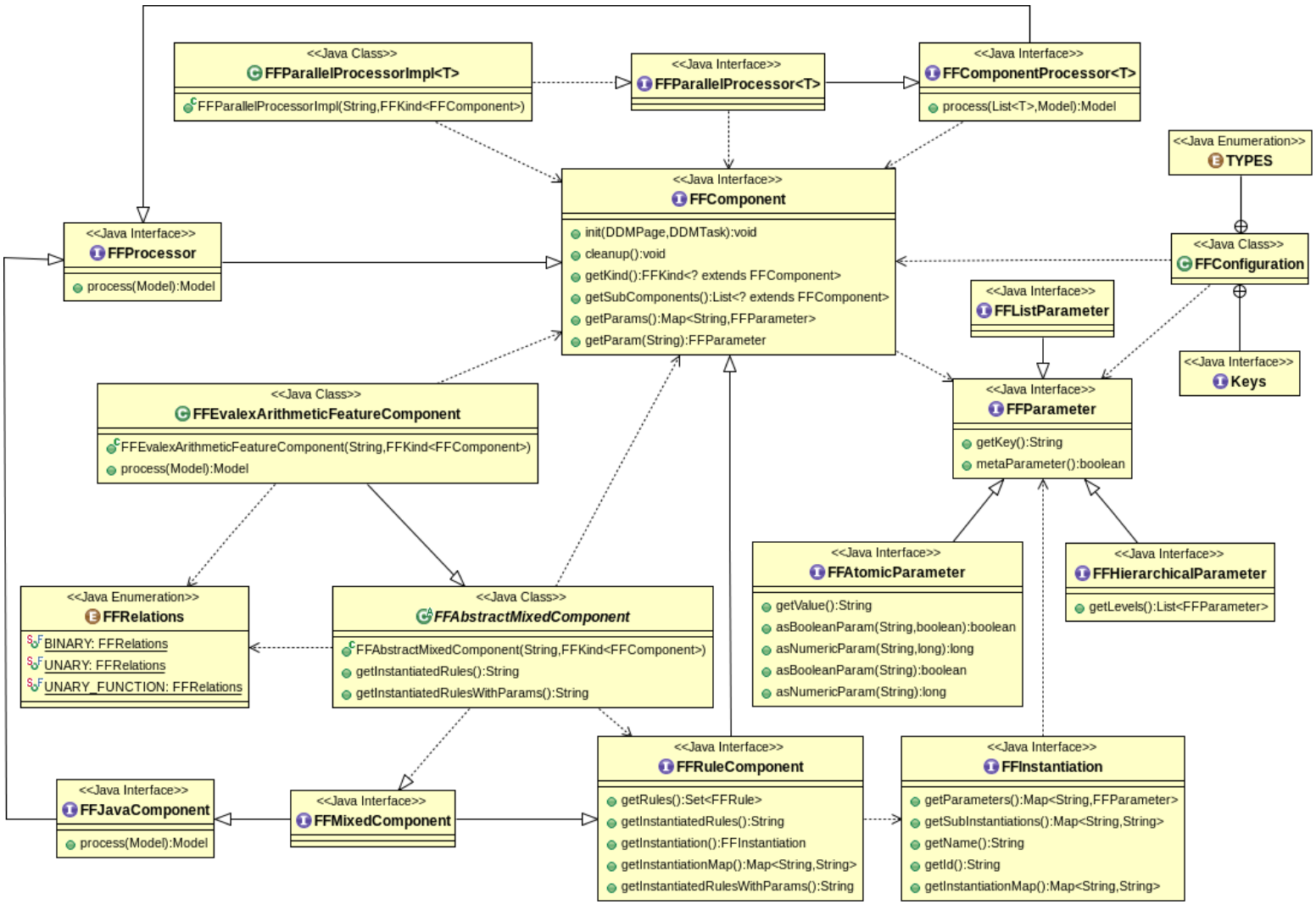


Figure 4.3: A UML diagram of the framework for a feature extraction processor

4. **hierarchical list parameter** – a `filter_by_tag_attribute` parameter that we use for some of instantiations of the `node_of_interest` template that specifies the nodes of interest through the tag-attributes tuples, where a node of interest must have a tag from the given list and at least one of the attributes linked to that tag, and in certain cases that attribute must have a specific value, e.g., in the case of the Pagination Link classifier, the hierarchical list for the `filter_by_tag_attribute` parameter is the following: `button`, `a>onclick`, `input>onclick`, `input>image`, `input>button`, `div>onclick`, `input>type>"button"`, `input>type>"image"`, `input>type>"submit"`; note that we include all elements with the `button` tag into the node of interest domain for that classifier, irrespective of the attributes they hold);

5. **unary relation predicate parameter** – an annotation predicate passed as a parameter for a more complicated template (e.g., the `holds_numeric_annotation` predicate passed to the `closest_node_in_proximity_of_a_given_type_holds_a_property` template for the case of the Pagination Link classifier to specify that in this instantiation we are interested only in pagination nodes in the visual neighbourhood of the given node; note that we also pass an `is_link` predicate as a unary relation parameter for that template instantiation to specify we are looking for non-link neighbours of the node in question);

6. **unary function relation predicate parameter** – a `number_of_characters` relation that specifies the number of characters in the text of a given node;

7. **binary relation predicate parameter** – a proximity predicate passed as a parameter for a more complicated template (e.g., the `left_proximity` predicate passed to the `closest_node_in_proximity_of_a_given_type_holds_a_property` template to show that we are only interested in nodes `Y` that are to the visual left of the given node `X`; note that we can also pass the width and height parameters to the `left_proximity` relation predicate to specify the exact width and height of the neighbourhood rectangle to the visual left of the node in question, in which the candidate nodes can be located).

Hierarchical lists and relation predicates are powerful parameter types and allow us to define highly configurable templates that can cover a very diverse range of instantiations. Relations and parameters are instantiated through the `FFInstantiation` interface, and the concrete mappings for those instantiations are provided in the configuration file implemented through the `FFConfiguration` class.

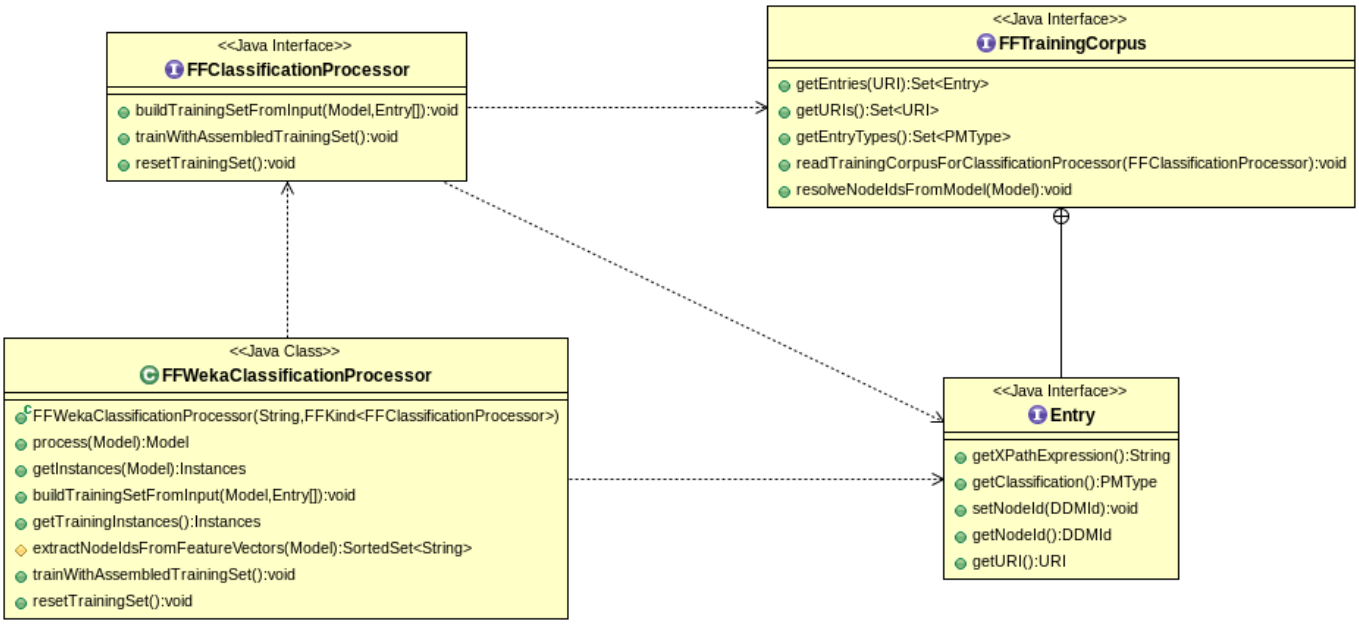


Figure 4.4: A UML diagram of the framework for a classification processor

In Figure 4.4, we give the UML diagram that represents the architecture of a classification processor. A classification processor takes as its input the feature set produced by the feature extraction processor. It then has two options, **(1)** to run in the evaluation mode or **(2)** to run in the training mode. The training mode creates a classifier from the set of feature vectors and a training corpus represented through the node (or sets of nodes) to classification labels mapping where the nodes (or sets of nodes) are represented by their unique XPath expressions[3], and class labels come from the labelling schema of the classifier we are building. Once the classifier has been created and serialised it can then be applied to the feature vectors extracted from a single page by the feature extraction processor to create mappings between each feature vector and a label from the labelling schema of the serialised classifier (note that we store only mappings for positive classification labels, such as `numeric next link` and `non-numeric next link` for the Pagination Link classifier, in the output of a classification processor and do not store mappings for negative classification labels, such as `not a next link` for the Pagination Link classifier). We implement the training and evaluation modes of the classification processor in Java through the `WEKA` library.[4] At the moment, we use the J48 decision tree model for our classifiers, but if another classification model, such as SVMs, will increase the accuracy of our BER_yL system, we can easily switch due to the flexibility of our system and the `WEKA` platform.

[3]https://www.w3.org/TR/xpath20/
[4]http://www.cs.waikato.ac.nz/ml/weka/

We conclude this section by giving the UML diagram for the overall architecture of BER$_y$L (Figure 4.5) that covers the main aspects of the system that we have discussed above: **(1)** the relational organisation between the three main processors of the system (filter processors, feature extraction processors, and classification processors), **(2)** filter processors, **(3)** feature extraction processors, and **(4)** classification processors.

4.2 Case studies for BER$_y$L's main components

We first give an example of a filter processor for the Pagination Link classifier.

Example 36. Consider a processor that consists of three consecutive filters: **(1)** the element nodes filter, **(2)** the attribute filter, and **(3)** the annotation filter. They are in the given order as the attribute filter only searches through the nodes filtered out by the element nodes filter and the annotation filter only gets applied to the element and attribute nodes filtered out by the two previous filters.

In the snippets below we give the rules that correspond to those three filters, along with the assignment of their parameters in the concrete instantiation of the filter processor of the Pagination Link classifier (we assign names to filters, such as `element_filter`, `attribute_filter` and `annotation_filter`, and mark the outputs of different filters with the `filter` namespace). Filter components form a sub-class of relation components (with the obvious adjustments to Definition 17 where relations are defined to only produce a predicate in the `relation::` namespace).

For the Pagination Link classifier we are only interested in the button nodes and nodes that have certain attributes, and in some case attribute and value pairs (as in the example for a list of hierarchical parameters from Section 4.1 of this Chapter [`button`, `a>onclick`, `input>onclick`, `input>image`, `input>button`, `div>onclick`, `input>type>"button"`, `input>type>"image"`, `input>type>"submit"`]):

```
nodeWithAttribute(N,Tag,AttrTag,AttrVal) ⇐ dom::element(N,Tag,PStart,Start,End),
 dom::attribute(Attr,N,Tag,AttrVal).

filter::element_filter(Id,N) ⇐ nodeWithAttribute(N,Tag,_,_), param::tag(Tag),
 param::instantiation_id(Id).
filter::element_filter(Id,N) ⇐ nodeWithAttribute(N,Tag,AttrTag,_),
 param::tag(Tag,AttrTag), param::instantiation_id(Id).
filter::element_filter(Id,N) ⇐ nodeWithAttribute(N,Tag,AttrTag,AttrVal),
 param::tag(Tag,AttrTag,AttrVal), param::instantiation_id(Id).
```

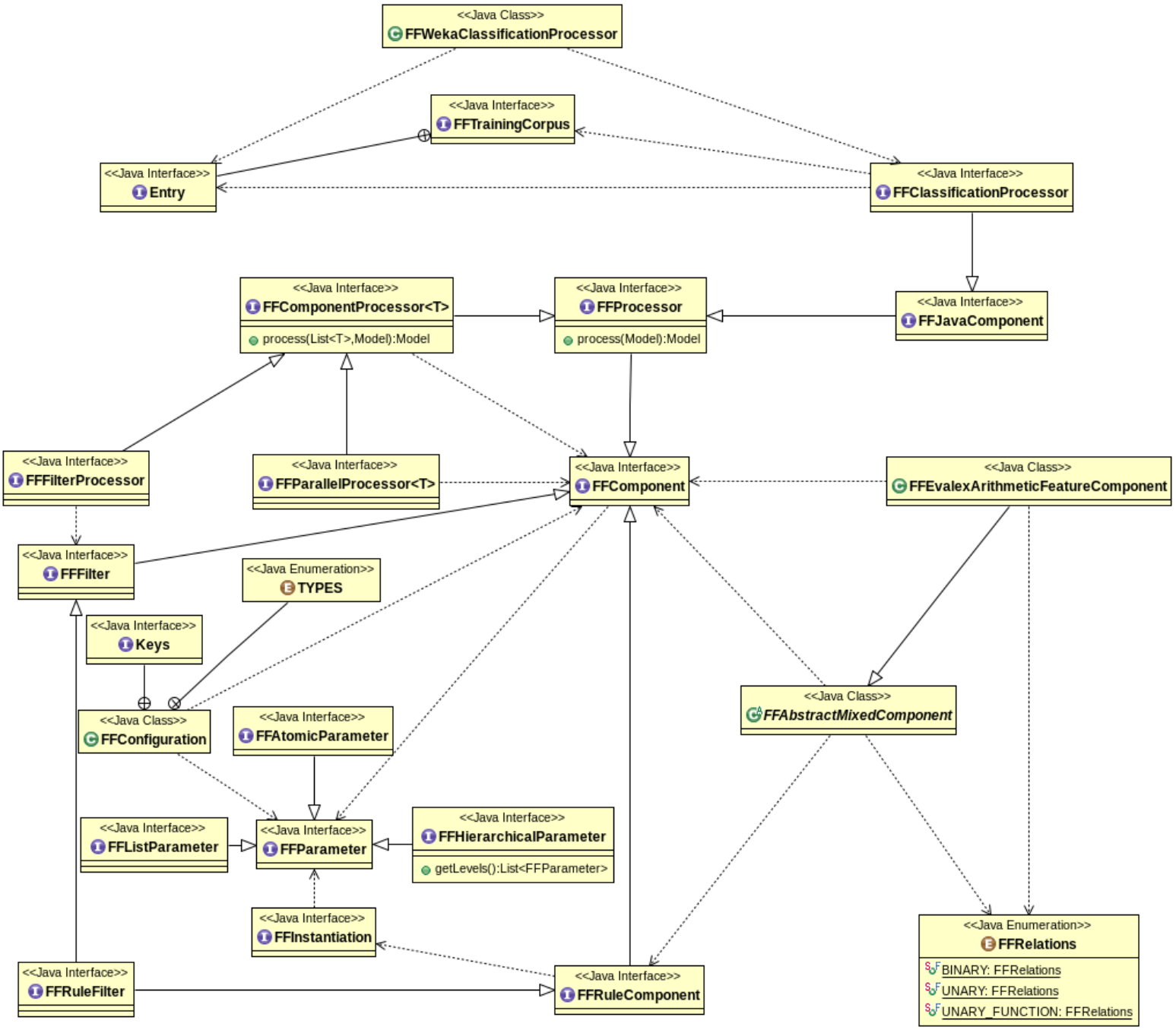


Figure 4.5: A UML diagram of the framework for the BER$_y$L system

The instantiation of the element filter is:

```
element_filter[tag↦["button","a">"onclick","input">"onclick","input>image",
 "input">"button","div">"onclick","input">"type">"button",
 "input">"type">"image","input">"type">"submit"]]
```

In the Pagination Link classifier we use a selected range of attributes (as in the example for a list parameter from Section 4.1 of this Chapter [`class`, `title`, `name`, `value`, `id`, `alt`, `src`, `href`, `onclick`, `image`, `button`, `type`]):

```
filter::attribute_filter(Id,N) ⇐ dom::attribute(_,N,Tag,_), param::tag(Tag),
  param::instantiation_id(Id).
```

This filter is instantiated in the following way:

```
attribute_filter[tag↦["class","title","name","value","id","alt","src","href",
 "onclick","image","button","type"]]
```

The Pagination Link classifier employs numeric and next link annotations represented by a hierarchical list parameter [`instance>number`, `label>next`, `label>next_attr`], where `next` and `next_attr` are the annotations from the gazetteers that cover the next link annotations that can be found in element nodes and their attributes). The corresponding annotation filter is the following:

```
filter::annotation_filter(Id,N) ⇐ gate::annotation(N,Type,Kind),
 param::annotatedBy(Kind,Type), param::instantiation_id(Id).
```

The annotation filter gets instantiated as:

```
annotation_filter[annotatedBy↦["instance">"number","label">"next",
 "label">"next_attr"]]
```

We now give an example of a feature extraction processor in action.

Example 37. Consider a subset of the features used in the Pagination Link classifier: **(1)** the numeric feature `feature::num_chars` that represents the number of characters in the text of the element node, **(2)** the Boolean feature `feature::direct_left_visual_numeric_neighbour_is_not_a_link` that checks whether the direct left visual neighbour node of the given node is a link or not (we cover this example in Section 3.7 of Chapter 3), **(3)** the Boolean feature `feature::direct_left_structural_numeric_neighbour_is_not_a_link` that checks whether the direct left structural neighbour node of the given node is a link or not, and **(4)** the numeric feature `feature::relative_vertical_position_in_the_first_screen` that if the CSS box of the given node is located within the first screen of the page, returns the top coordinate of that box divided by the height of the screen and returns -1 if that is not the case. We start by defining the `feature::num_chars` feature:

```
feature::num_chars(Id,X,Num) ⇐ function::char_num(X,Num), Num≤4,
 param::instantiation_id(Id).
```

Note that for the Pagination Link classifier we impose a limitation on the maximal number of characters in the text of the node; since we have not observed any scenarios in which there are more than 10,000 pages, the output is split over and the respective numeric pagination links contain more than four digits. It is practical to store this number as a `param::maxSpan` parameter, rather than set it manually:

```
feature::num_chars(Id,X,Num) ⇐ function::char_num(X, Num), Num≤MaxSpan,
 param::maxSpan(MaxSpan), param::instantiation_id(Id).
```

We can then instantiate this feature for the Pagination Link classifier as:

```
feature::num_chars[maxSpan↦4]
```

Let us revisit `feature::direct_left_visual_numeric_neighbour_is_not_a_link` from Section 3.7 of Chapter 3. The set of relations we use for the definition of this feature is the following:

```
relation::binary_unary(Id,X,Y) ⇐
  param::binary_pred(CId1,X,Y), param::node_pred(CId2,X),
  param::sibling_pred(CId3,Y), param::instantiation_id(Id).
relation::direct_left_visual_neighbour(IdX,Y) ⇐
  param::left_visual_neighbour(Id3,X,Y),
   ¬ relation::indirect_left_visual_neighbour(Id7,X,Y),
  param::siblingPred(CId4,Y), param::instantiation_id(Id).
relation::left_visual_neighbour(Id,X,Y) ⇐ relation::visual_proximity(Id4,X,Y),
  css::box(X,_,_,RightX,_), css::box(Y,LeftY,_,_,_), RightX ≤ LeftY,
  param::instantiation_id(Id).
relation::indirect_left_visual_neighbour(Id,X,Y) ⇐
  param::left_visual_neighbour(_,X,Y), param::left_visual_neighbour(_,X,Z),
  css::box(Y,LeftY,_,RightY,_), css::box(Z,LeftZ,_,RightZ,_),
  RightY ≤ LeftZ, RightX ≤ LeftY, param::instantiation_id(Id).
relation::vis_prox(Id,X,Y) ⇐
  css::box(X,LeftX,TopX,_,_), css::box(Y,LeftY,TopY,_,_),
  TopY − DVert ≤ TopX ≤ TopY + DVert, LeftY − DHor ≤ LeftX ≤ LeftY + DHor,
  param::dH(DHor), param::dV(DVer), param::instantiation_id(Id).
relation::numeric(Id,X) ⇐ gate::annotation(X,"NEXT",_),
  param::instantiation_id(Id).
relation::non_clickable(Id,X) ⇐ dom::element(X,_,_,_,_), ¬(dom::clickable(X)),
  param::instantiation_id(Id).
```

We extract the feature instantiating these relations in the following way (for the purpose of making the notation more concise we abbreviate `relation::binary_unary(Id1,X,Y)` to C_1, `relation::direct_left_visual_neighbour(Id2,X,Y)`

to C_{12}, relation::left_visual_neighbour(Id_3,X,Y) to C_{13}, relation::vis_prox(Id_4,X,Y) to C_{14}, relation::is_numeric(Id_5,X) to C_5, and relation::non_clickable(Id_6,X) to C_6) :

```
((((C14[dH↦70,dV↦50]◁C13)‖C5)⊕C12[sibling_pred↦C5])‖C6)◁
C1[binary_pred↦C12,node_pred↦C5,sibling_pred↦C6]
```

Now let us consider feature::direct_left_structural_numeric_neighbour_is_not_a_link that is very similar to the feature above, with the exception that we are now interested in structural neighbours rather than visual neighbours. Two nodes N1 and N2 are structural neighbours if **(1)** they have a common ancestor, **(2)** are located at the same depth level from this common ancestor, **(3)** the distance between them, computed as the sum of the depth levels of the two nodes from their common ancestor (e.g., the depth level of a grandchild node of the common ancestor is 2), is smaller than some predefined threshold param::maxDistance(MaxDistance), and **(4)** the horizontal distance between those two nodes, computed as Start2−End1 (dom::element(N1, _, _, Start1, End1), dom::element(N2, _, _, Start2, End2), and Start2>End1), is smaller than another predefined threshold param::maxHorizontalDistance(MaxDistance). For conciseness we omit the exact rules for the structural neighbour relations. The instantiation of the feature::direct_left_structural_numeric_neighbour_is_not_a_link feature for the Pagination Link classifier is the following: (relation::binary_unary(Id_1,X,Y) is abbreviated to C_1, relation::direct_left_structural_neighbour(Id_2,X,Y) to C_{22}, relation::left_structural_neighbour(Id_3,X,Y) to C_{23}, relation::struct_prox(Id_4,X,Y) to C_{24}, relation::is_numeric(Id_5,X) to C_5, and relation::non_clickable(Id_6,X) to C_6):

```
((((C24[maxDistance↦7,maxHorizontalDistance↦50]◁C23)‖C5)⊕
C22[sibling_pred↦C5])‖C6)◁
C1[binary_pred↦C22,node_pred↦C5,sibling_pred↦C6]
```

Note that we reuse the relation::binary_unary template that we have already used for the instantiation of the feature::direct_left_visual_numeric_neighbour_is_not_a_link feature to instantiate the feature::direct_left_structural_numeric_neighbour_is_not_a_link feature. We also reuse the relation::is_numeric and relation::non_clickable relations. These relations get computed only once, since all features are extracted at the same time through a parallel processor.

Finally let us consider the feature::relative_vertical_position_in_the_first_screen feature that combines both logic-based and procedural approaches to compute the final output. We use mixed components defined through the FFMixedComponent interface and implemented through the FFAbstractMixedComponent and FFEvalexArithmeticFeatureComponent classes (Figure 4.3) to perform this kind of operation. We first compute two unary function relations, relation::top_coordinate(Id_1,N,Top) and

relation::screen_height(Id$_2$,ScreenHeight), through logic-based components and then use a procedural component to compute the relative vertical position of the node on the first screen (if it is in the first screen) defined by Top/ScreenHeight. The rules for the two relations that are required to compute this feature are the following:

```
relation::top_coordinate(Id1,N,Top) ⇐ css::box(N,Left,Top,Right,Bottom),
 param::instantiation_id(Id1).
relation::screen_height(Id2,ScreenHeight) ⇐
 param::averageScreenHeight(ScreenHeight),
 param::instantiation_id(Id2).
```

We instantiate the average screen height to 800 pixels[5]:

```
relation::screen_height[averageScreenHeight↦800]
```

We then perform the division of the output of those two relations through a procedural Java part of the mixed component. The exact expressions we evaluate for mixed components are set in the Configuration of the mixed component and generated through the EvalEx library.

We conclude this section by describing the implementation of our classification approach through the WEKA library. We start by building a WEKA training corpus from the feature vectors of a given classifier computed for multiple pages. This is done through the FFWEKAClassificationProcessor class (Figure 4.4). We then build a classifier based on this training corpus (at the moment we use a J48 decision tree classifier, but our system provides a mechanism for an easy switch to any other classifier, such as an SVM or a neural network). The final step of the training phase is the serialisation of the classifier. We can now apply the classification processor in the evaluation mode on new pages, by extracting their respective feature sets, converting them to the WEKA format, deserialising the classifier, and applying it on those feature vectors.

[5]According to a study by http://www.w3schools.com/browsers/browsers_display.asp.

Chapter 5

Examples of Individual Classifiers

In this chapter we discuss individual classifiers included in the BER$_y$L system. We first give an example of a feature set for a given classifier (the Pagination Link classifier), discuss how the features are split into different types, and talk about the training phase [22]. We also show how templates and global features are used in defining the feature set of this classifier. We then give descriptions of other domain-independent classifiers included in the BER$_y$L system.

5.1 Pagination links

To motivate the need for pagination link detection and give an impression of some of the issues involved, Table 5.1 presents a selection of popular websites from the evaluation corpus (on all of which we happen to identify pagination links with 100% precision and recall). n is the number of links on the result page, n_1 (n_2) is the number of immediate numeric (non-numeric) pagination links on the page, and P, R are precision and recall for our approach. For each website we also present a screenshot of either its pagination links or a potential false positive. Even in this small sample of web pages, we can observe the diversity of pagination links: only six of the twelve websites have a typical pagination link layout (non-numeric link containing a NEXT keyword and a list of numeric links with the current page represented as a non-link). Some of the challenges evident from this table are:

1. for FindAProperty and IKEA the index of the current page is a link and thus we need to consider e.g., its style to distinguish it from the other links;

2. for Zoopla the "50" for the results per page can be easily mistaken for an immediate numeric pagination link;

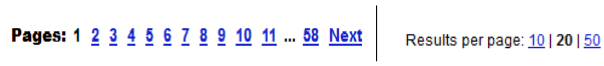
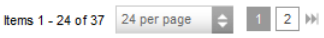
	Website	n	n_1	n_2	P	R	Screenshot
Real Estate	FindAProperty	370	1	1	1	1	
	Zoopla	332	1	1	1	1	
	Savills	234	2	2	1	1	
Used Cars	Autotrader	262	2	2	1	1	
	Motors	472	2	2	1	1	
	Autoweb	103	2	2	1	1	
Retail	Amazon	448	1	1	1	1	
	Ikea	290	2	0	1	1	
	Lands' End	527	2	2	1	1	
Forums	TechCrunch	279	0	1	1	1	
	TMZ	200	2	2	1	1	
	Ars Technica	341	2	2	1	1	

Table 5.1: Sample pages

3. for Savills numeric links come as intervals. However, our NUMBER annotations also cover numeric ranges (as well as "2k" or "two");

4. for Amazon the result page contains a confusing scrollbar for navigation through the related products (right part of the screenshot);

5. for Lands' End the non-numeric pagination link is an image. However, our approach classifies it correctly, based on the context and attribute values;

6. TechCrunch contains a single isolated non-numeric pagination link that we are able to identify due to the keyword present in its text and the proximity to "Page 1";

7. TMZ has a pagination link that carries both a NEXT and a NUMBER annotation. From the context, we nevertheless identify it correctly as non-numeric.

To identify pagination links with high accuracy, we create a small component model in BER$_y$L that consists of content, structural, visual proximity, and page position features. In Section 6.2, we show that BER$_y$L not only achieves almost perfect accuracy with this feature model for a wide range of domains and pages, but that these four feature types also contribute notably to the overall performance. A **pagination sequence** is a sequence of

web pages from the same domain that are the result of paginating some information such as an article or a result set of a search. Given a DOM tree P of a page, the (immediate) **pagination link identification problem** is the problem of identifying those nodes in P that must be clicked to get to the following page in any pagination sequence the page is part of. The pagination links should be distinguished into numeric and non-numeric (such as "Next").

With BER$_y$L we reduce this problem to a block classification task over the set of clickable nodes (`dom::clickable`). To do so, we need to: **(1)** define appropriate annotation types if necessary, **(2)** specify an appropriate component model, as discussed in Chapter 3, and **(3)** train a classifier on a small training set.

Annotation types In addition to the standard annotation type NUMBER, we introduce two annotation types specific to pagination link identification: NEXT and PAGINATION. NEXT collects typical keywords used to indicate immediate non-numeric pagination links, e.g., "next", "›", or "»". PAGINATION includes all those, but also keywords related to previous pagination links (e.g., "previous") and to the number of results (e.g., "page", "results").

Feature model Table 5.2 shows the features used in our approach for pagination link identification. They are split into four types: content, page position, visual proximity and structural. For conciseness we use the <> brackets notation to show the parameters or relations we want to instantiate the given templates with. The corresponding extraction rules are given in Figure 5.1.

To define these features, we use a small number of auxiliary relation predicates and functions, which are shown in Table 5.3. The first is required in any BER$_y$L component model (apart from the Global Model `GM`) and specifies the domain of discourse, here all `dom::clickable` nodes (links and other click targets). The second is required in feature models that use proximity predicates and specifies the area we consider to be "in the proximity" of a node:

The *content features* $1-4$ from Table 5.2 specify whether a node is annotated with one of the three annotation types (NEXT, PAGINATION, and NUMBER) and how many characters it contains. They are defined in Figure 5.1 by an instantiation and an import of the global feature `char_num`. The instantiation creates three instances of the `annotated_by` template, one for each of the three annotation types.

The *page position features* $5-8$ are the relative position on the first page and on the first screen, as well as whether a node is on the first or last screen. They are de-

		Description	Type	Predicate
Content	1	Annotated as NEXT	bool	`gm::relation::annotated_by<"NEXT">`
	2	Annotated as PAGINATION	bool	`gm::relation::annotated_by<"PAGINATION">`
	3	Annotated as NUMBER	bool	`gm::relation::annotated_by<"NUMBER">`
	4	Number of characters	int	`gm::feature::char_num`
Page position	5	Relative position on page	int^2	`gm::relation::relative_position<css::page>`
	6	Relative position in first screen	int^2	`gm::relation::relative_position<relation::first_screen>`
	7	In first screen	bool	`gm::relation::contained_in<relation::first_screen>`
	8	In last screen	bool	`gm::relation::contained_in<relation::last_screen>`
Visual proximity	9	Pagination annotation close to node	bool	`gm::relation::in_visual_proximity_property <gm::relation::annotated_by<"PAGINATION">>`
	10	Number of close numeric nodes	int	`gm::relation::num_in_visual_proximity_property <plm::relation::numeric>`
	11	Closest numeric node is a link	bool	`gm::relation::binary_unary<gm::relation::left_visual_proximity>_with <plm::relation::numeric>_is<plm::relation::non_link>`
	12	Closest numeric node has different style	bool	`<plm::relation::numeric>_is<plm::relation::different_style>`
	13	Closest link annotated with NEXT	bool	`<dom::clickable>_is<gm::relation::annotated_by<"NEXT">>`
	14	Ascending w. closest numeric left, right	bool	`plm::ascending_numerics`
Structural	15	Preceding numeric node is a link	bool	`gm::relation::binary_unary<relation::preceding>_with <plm::relation::numeric>_is<plm::relation::non_link>`
	16	Preceding numeric node has different style	bool	`<plm::relation::numeric>_is<plm::relation::different_style>`
	17	Preceding link annotated with NEXT	bool	`<dom::clickable>_is<gm::relation::annotated_by<"NEXT">>`

Table 5.2: PLM: Pagination Link Model

fined by two instantiations in Figure 5.1, one for relative positions (using `css::page` and `relation::first_screen`, respectively) and one for the presence in the first or last screen.

The *visual proximity features* are the most involved. They include a feature on whether there is a node in visual proximity that is annotated with PAGINATION (9), a feature on the number of numeric nodes in the visual proximity (10), and a feature that specifies whether the node and the closest numeric nodes in the visual proximity to the left and to the right form an ascending sequence (14). $11-13$ ask if the closest node with a certain property passes a given test, e.g., whether the closest numeric node to the visual left is a link. Accordingly, $11-13$ are instantiations of `closest`. 9 and 10 are instantiations of `in_proximity` and `num_in_proximity`. 14 is the only feature in this model that is defined entirely from scratch.

```
plm::relation::node_of_interest(Id,X) ⇐ dom::clickable(X), param::instantiation_id(Id).
plm::relation::proximity_dimension(Id,Width,10) ⇐ css::page(_,_,Width,_),
 param::instantiation_id(Id).
plm::relation::numeric(Id,X,Value) ⇐ gate::annotation(X,"NUMBER",Value),
 param::instantiation_id(Id).
plm::relation::numeric(Id,X) ⇐ plm::relation::numeric(_,X,_), param::instantiation_id(Id).
plm::relation::different_style(Id,X,Y) ⇐ css::font_family(X,FX), css::font_family(Y,FY),
 FX≠FY, param::instantiation_id(Id).
```

Table 5.3: Auxiliary relation predicates used in the PLM model

The *structural features* are similar to $11-13$, but use XPath's preceding relation instead of visual proximity. For example, 15 tests whether the numeric node immediately preceding the given node is a link. These are omitted from Figure 5.1 as they are similar to $11-13$.

```
1 – 3 :gm::relation::annotated_by[annotation_type↦["NEXT","PAGINATION","NUMBER"]]
  4 :  gm::feature::char_num
5 – 6 :gm::relation::relative_position_within[boundaries↦
       [relation::first_screen,relation::last_screen]]
7 – 8 :gm::relation::contained_in[container↦[relation::first_screen,relation::last_screen]
  9 :  gm::relation::annotated_by[annotation_type↦["PAGINATION"]]◁
       gm::relation::in_visual_proximity_property[property↦gm::relation::annotated_by]
 10 :  gm::relation::num_in_visual_proximity_property[relation::property↦
         plm::relation::numeric]
11 – 12 :gm::relation::binary_unary[binary_predicate↦relation::left_visual_neighbour;
         node_predicate↦plm::relation::numeric; sibling_predicate↦
         [plm::relation::numeric,plm::relation::non_link]]
 13 :  gm::relation::annotated_by[annotation_type↦["NEXT"]]◁
       gm::relation::binary_unary[binary_predicate↦relation::left_visual_neighbour;
         node_predicate↦dom::clickable; sibling_predicate↦gm::relation::annotated_by]
 14 :  plm::ascending_numerics(Id,X) ⇐ plm::relation::node_of_interest(CId1,X),
         plm::relation::numeric(CId2,X,ValueX),
         relation::left_neighbour(Left,X), relation::left_neighbour(X,Right),
         plm::relation::numeric(CId3,Left,ValueLeft),
         plm::relation::numeric(CId4,Right,ValueRight),
         ValueLeft < ValueX < ValueRight,
         ¬(relation::left_visual_neighbour(CId5,Left,LeftN),
           relation::left_visual_neighbour(CId6,LeftN,X),
           plm::relation::numeric(CId7,LeftN)),
         ¬(relation::left_visual_neighbour(CId8,RightN,Right),
           relation::left_visual_neighbour(CId9,X,RightN),
           plm::relation::numeric(CId10,RightN)),
         param::instantiation_id(Id).
```

Figure 5.1: Extraction rules for pagination link identification

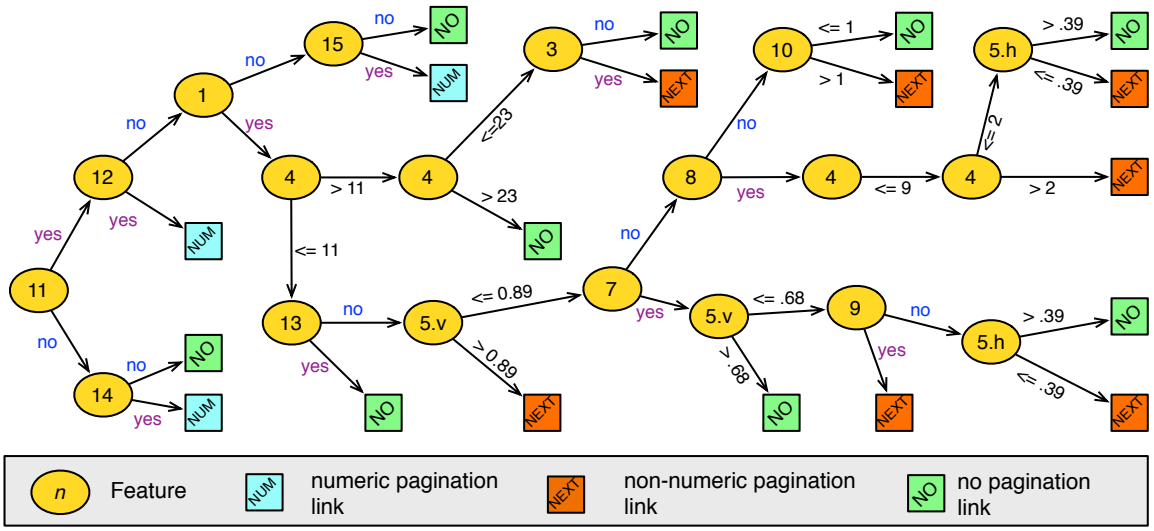


Figure 5.2: Classification tree for the pagination link model

Training the classifier With this feature model, BER$_y$L derives a classifier based on a small training set. For pagination link classification, a training corpus of only two dozen pages suffices to achieve the high accuracy demonstrated in Section 6.2. We expect a trade-off between accuracy, corpus size and the complexity of the feature model, but their relation in this triangle remains an open issue.

Figure 5.2 shows the classification tree derived on a training corpus for websites from the Real Estate and Used Cars domains in the UK. The evaluation results for this classifier are presented in Section 6.2. The tree employs almost all features, though the only structural feature considered is 15. For 5, we use both horizontal and vertical positions, but at different points in the tree (5.h and 5.v). Visual proximity (9 – 14) and 15 are clearly very distinctive features of pagination links. Feature 1 has a key role in distinguishing numeric and non-numeric pagination links, but 11, 12, and 14 are also required to give an almost perfect distinction, as is evident in Section 6.2.

Our classifier employs almost all of the features we have pre-selected. It places a strong emphasis on the visual features, such as relative vertical and horizontal positions, and whether the link to be classified is in the first or last screen. Manually finding rules that correspond to this classifier would be a very error-prone and time-consuming task, particularly where thresholds and complex features are involved. This justifies the use of machine learning to obtain the precise classifiers.

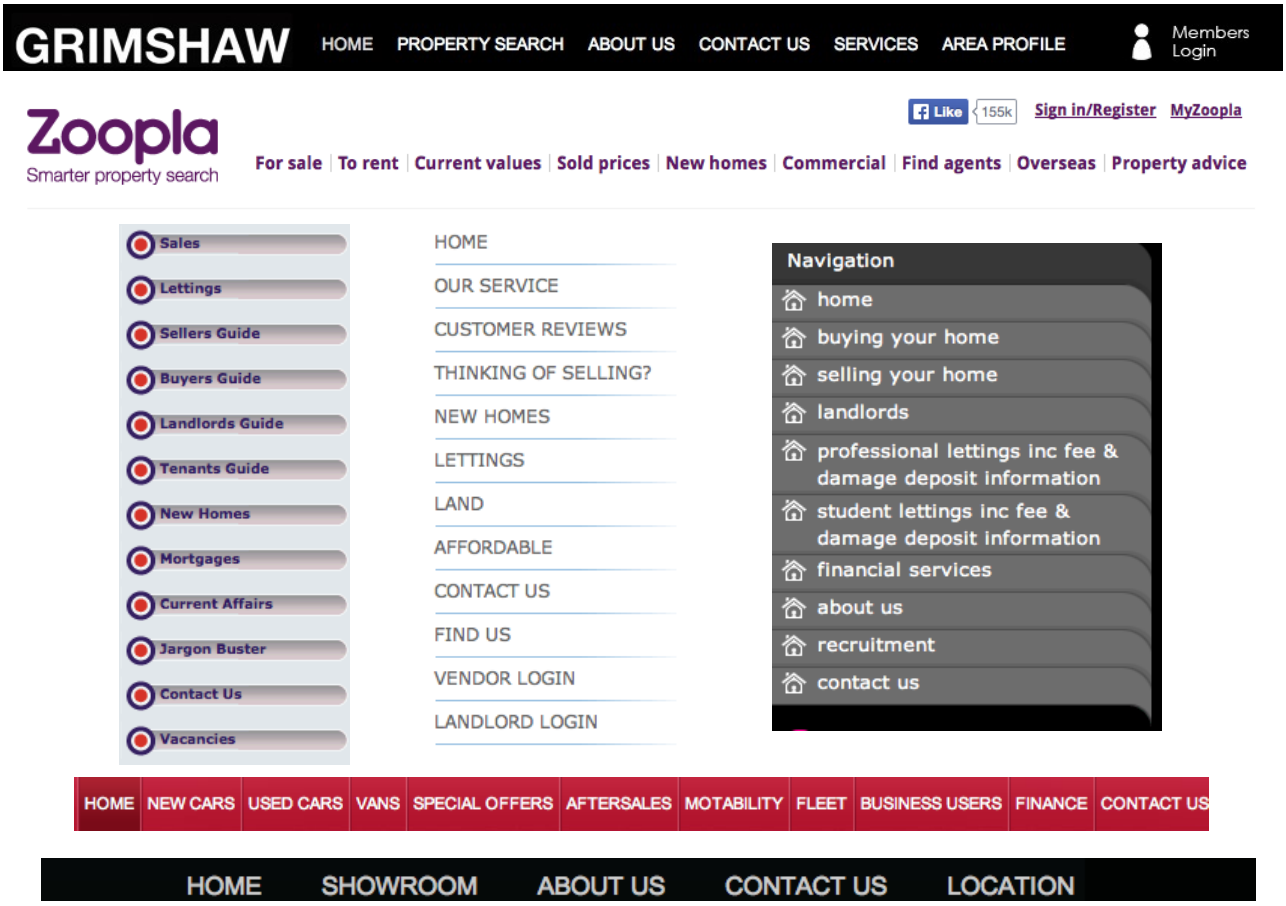


Figure 5.3: The diversity of navigation menu types

5.2 Navigation menus

Navigation menus play a huge role in the structure of web pages. They appear on both home pages and result pages and in almost all of the imaginable web domains. We give a sample of different navigation menus in Figure 5.3. The menus at the top belong to the Real Estate domain, whilst those at the bottom come from the Used Cars domain. The side menus in the centre have a very different structure to the menus with the normal horizontal layout but can also hold semantic roles of main menus. Navigation menus can vary considerably in size and the number of items they contain. Note that menu items can include domain-specific annotations (e.g., "Rent" for the Real Estate domain and "Vehicle" for the Used Cars domain), as well as domain-independent annotations (e.g., "Home" or "Contact Us").

Before going into description of features used in the navigation menu model (`nmm`), we need to decide which nodes to consider as potential navigation menus. After performing a scan of random web pages that contain navigation menus, we have decided to define the nodes with `div`, `ul`, `tr`, `tbody`, `table`, `nav`, and `ol` tags as navigation menu candidates. We

1	Annotation coverage	int^2	`gm::relation::annotation_coverage<"NAVIGATION-MENU">`
2	In first screen	bool	`gm::relation::contained_in<relation::first_screen>`
3	Relative position in first screen	int^2	`gm::relation::relative_position<relation::first_screen>`
4	Relative position on page	int^2	`gm::relation::relative_position<css::page>`
5	Width of the corresponding CSS box	int	`gm::feature::css_width`
6	Height of the corresponding CSS box	int	`gm::feature::css_height`

Table 5.4: NMM: Navigation Menu Model

also augment a set of BER_yL's standard function relations with the DOM nodes' CSS boxes' widths and heights:

```
gm::relation::css_width(Id1,X,Width) ⇐ css::box(X,LeftX,_,RightX,_),
  Width = RightX-LeftX, param::instantiation_id(Id1).
gm::relation::css_height(Id2,X,Height) ⇐ css::box(X,_,TopX,_,BottomX),
  Height = BottomX-TopX, param::instantiation_id(Id2).
```

We list the features used for navigation menu identification in Table 5.4. The navigation menu annotation coverage feature is of particular importance. Note that we specify the annotation procedure in such a way that for a page from a certain domain the navigation menu annotation gazetteer forms a union of a global domain-independent annotation gazetteer and the gazetteer specific for that domain (e.g., the gazetteer for the Real Estate domain contains the "Mortgage" annotation, and the one for the Used Cars domain contains the "Engine" annotation, whilst they both share the "About Us" annotation). The decision tree generated for the Navigation Menu classifier is quite large, so for the sake of brevity we do not present it here.

5.3 Headers, footers, and sidebars

Headers, footers, and sidebars are ubiquitous on the web. They appear in all possible domains and carry more of a structural role than a semantic meaning. For example a header can contain a navigation menu, whilst a sidebar can contain a form, advertisement, navigation menu, etc. Footers can contain very diverse information as well. We present a sample of different headers, footers and sidebars in Figure 5.4.

We define the namespace for the header model as `hm`, for the footer model as `fm`, and for the sidebar model as `sm`. The node of interest tags for these three classifiers are `<header>` and `<div>` for headers, `<footer>` and `<div>` for footers, and `<div>`, `<aside>`, `<nav>`, `<form>`, `<section>`, and `<span>` for sidebars.

These three blocks share most of their features from the global feature repository, which is another reason we list them in one section, apart from the two features for the sidebars

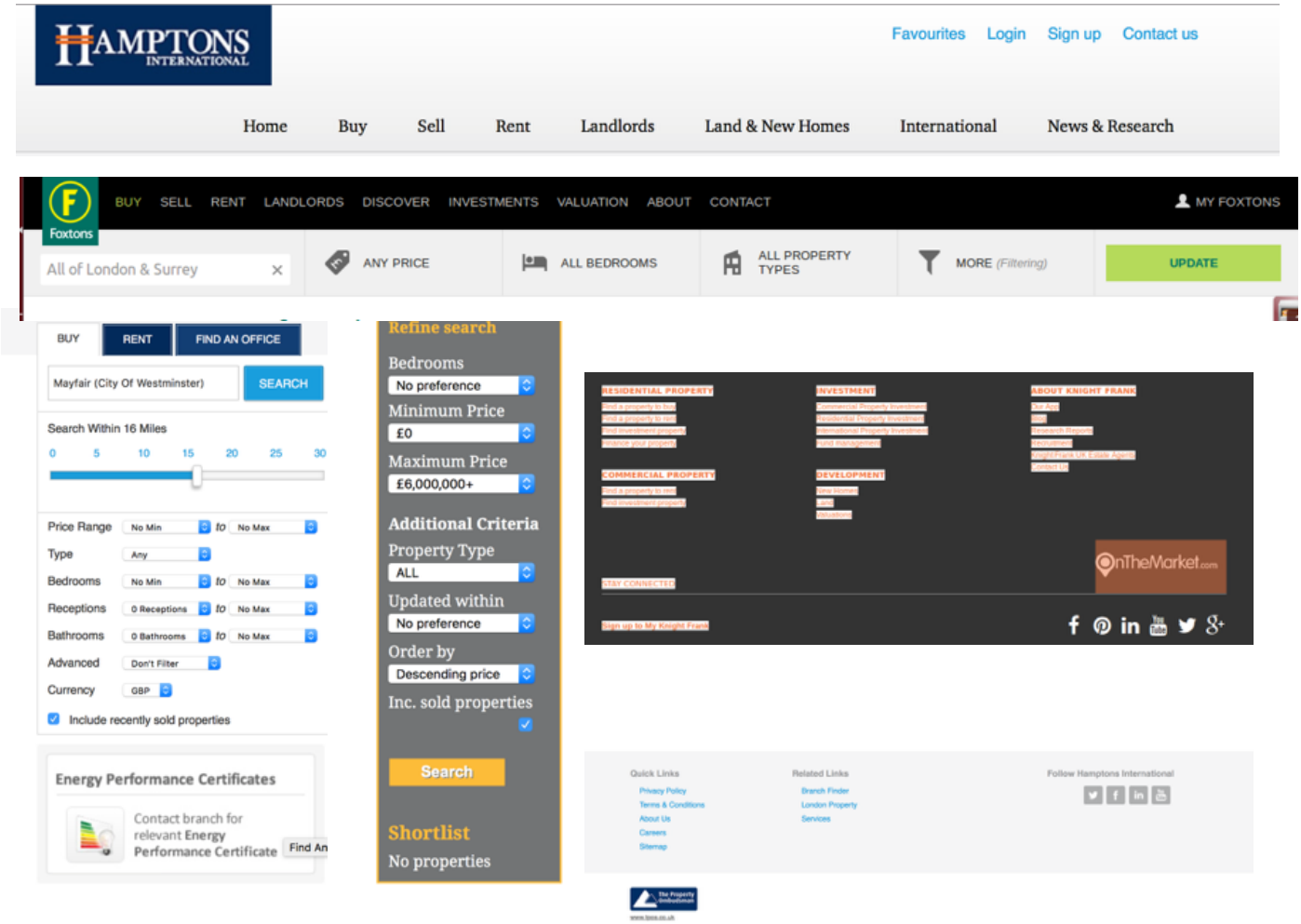


Figure 5.4: The diversity of headers, footers, and sidebars

that determine whether the respective CSS boxes of the candidate nodes are located on the left or right margin of the page, which can serve as a strong clue that the node in question is a sidebar. The two features which check whether the CSS boxes are on the left or right margins, as used in the sidebar classification, are just a trivial parametrised extension of the `gm::relation::relative_position<css::page>` relation.

1	In first screen	bool	`gm::relation::contained_in<relation::first_screen>`
2	In last screen	bool	`gm::relation::contained_in<relation::last_screen>`
3	Relative position on page	int^2	`gm::relation::relative_position<css::page>`
4	Relative position in first screen	int^2	`gm::relation::relative_position<relation::first_screen>`
5	Width of the corresponding CSS box	int	`gm::feature::css_width`
6	Height of the corresponding CSS box	int	`gm::feature::css_height`

Table 5.5: HM, FM and SM: Header, Footer, and Sidebar Models

Chapter 6

Evaluation Results

In this chapter we present the evaluation results of our BER$_y$L system. We start by presenting the verification framework that we use for computing various evaluation metrics. We then show detailed evaluation results for the Pagination Link classifier, by demonstrating the precision and recall results for that classifier applied on several domains, and discuss the impact of different feature types on its overall precision and recall. We then report on the precision, recall and performance results of other classifiers included in the BER$_y$L system. We then analyse the performance of the complete BER$_y$L system with and without the use of global features and template relations. We then perform the evaluation of our solution to the α-accuracy problem on all classifiers in the BER$_y$L system. We conclude this chapter with the evaluation of our holistic approach to web block classification.

6.1 Verification framework

In BER$_y$L we have implemented a verification framework that allows us to easily build new training and evaluation corpora and extend existing ones. In combination with the global feature and template repositories it also allows us to check the importance of individual features and groups of features with respect to improvements of precision and recall, and the reduction in performance rates they bring into respective classifiers. We can also use this framework for evaluating accuracy improvements and losses in performance that automatically generated features bring into BER$_y$L.

In that framework we identify nodes that correspond to web blocks of interest through the URLs of the pages on which they are located and unique XPath expressions that distinguish them on those pages. We give a sample of classified instances for the Pagination Link training corpus in Table 6.1. For the sake of brevity, we replace the full URL with the domain name of that URL.

URL	XPath expression	Label
http://www.beltonduffey.com	`//*[@id='home-right']/table[1]/tbody/tr/td[2]/a[6]`	numeric
http://www.beltonduffey.com	`//*[\@id='home-right']/table[1]/tbody/tr/td[2]/a[6]`	non-numeric
http://www.beltonduffey.com	`//*[\@id='home-right']/table[3]/tbody/tr[1]/td[2]/a[1]`	numeric
http://www.beltonduffey.com	`//*[\@id='home-right']/table[3]/tbody/tr[1]/td[2]/a[6]`	non-numeric
http://www.grampianpropertycentre.co.uk	`//*[\@id='content\_search']/center/table/tbody/tr/td[2]/a`	numeric
http://www.grampianpropertycentre.co.uk	`//*[\@id='content\_search']/center/table/tbody/tr/td[8]/a`	non-numeric
http://www.edwardsyeovil.co.uk	`/html/body/div/div[2]/div[2]/div[3]/div/div[2]/a`	numeric
http://www.edwardsyeovil.co.uk	`/html/body/div/div[2]/div[2]/div[3]/div/div[6]/a`	non-numeric
http://www.edwardsyeovil.co.uk	`/html/body/div/div[2]/div[2]/div[4]/span/div[2]/a`	numeric
http://www.edwardsyeovil.co.uk	`/html/body/div/div[2]/div[2]/div[4]/span/div[6]/a`	non-numeric
http://www.realestates-wsp.co.uk	`//*[@id='listings']/section[1]/p/a[1]`	numeric
http://www.realestates-wsp.co.uk	`//*[@id='listings']/section[2]/p/a[1]`	numeric
http://www.douglasandgordon.com	`//*[@id='pageWrap']/div[3]/div[2]/div[2]/ul/li[2]/a`	numeric
http://www.douglasandgordon.com	`//*[@id='pageWrap']/div[3]/div[2]/div[2]/ul/li[10]/a`	non-numeric
http://www.vebra.com/brannen/	`//*[@id='s-pagenavtop']/ul/li[2]/a`	numeric
http://www.vebra.com/brannen/	`//*[@id='s-pagenavtop']/ul/li[11]/a`	non-numeric
http://www.vebra.com/brannen/	`//*[@id='s-pagenavbottom']/ul/li[2]/a`	numeric
http://www.vebra.com/brannen/	`//*[@id='s-pagenavbottom']/ul/li[11]/a`	non-numeric

Table 6.1: A sample of the verification corpus for the pagination link classifier

6.2 Evaluation results for pagination links

We have carried out a comprehensive evaluation on five block types that we have considered in BER$_y$L so far: the Pagination Links, Navigation Menus, Headers, Footers, and Sidebars. For Pagination Links, we have performed a detailed evaluation on 145 web pages from four different web domains (100 pages from the Real Estate domain, 25 pages from the Used Cars domain, and 10 pages each from the Online Retail and Online Blogs and Forums domains), and we have evaluated the other four classifiers on 50 web pages each. Each block type has been evaluated on four domains: Real Estate, Restaurants, Online Retail, Blogs and Forums.

For each block type and domain, the pages have been selected randomly from a listings page (such as `yell.com`) or from a Google search. The latter favours popular websites, but that should not affect the results presented here. For examples of the pages in the Pagination Links evaluation corpus, consider again Table 5.1 from Section 5.1. For all five block types BER$_y$L exhibits good precision, recall and F_1 results, although recall can be further improved for the search form classifier. Note that we use the F_1 metric rather than $F_{0.5}$ or F_2 since, as we have discussed in Example 12 of Chapter 2, for classification

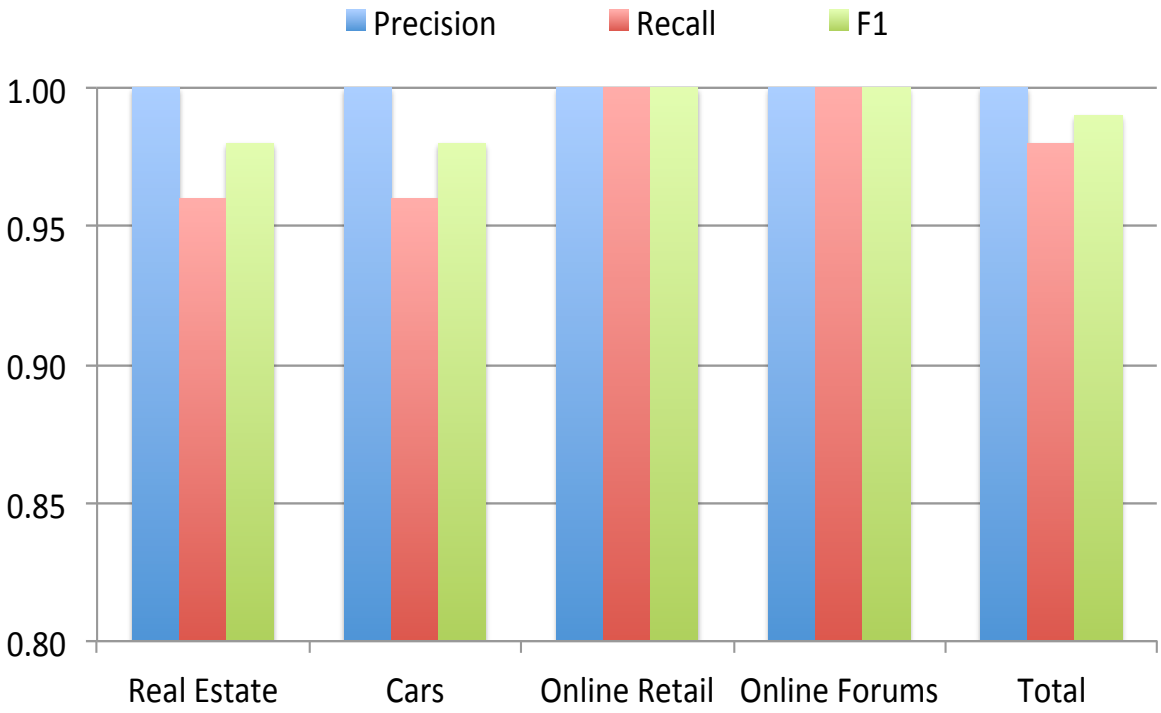


Figure 6.1: Precision and recall of the pagination link model

of some blocks we prefer high precision over high recall, whilst for others we prefer high recall over high precision.

Accuracy of pagination link classification We put special emphasis on the Pagination Link classifier, due to its importance to the DIADEM system, and hence have performed a more detailed evaluation for this block type than for the other three. We present detailed evaluation results for the Pagination Link classifier in Figure 6.1, which illustrates that in all four domains our approach achieves 100% precision, and recall is never below 96%. This high accuracy means that our approach can be used to crawl or otherwise navigate paginated websites with a very low risk of missing information or retrieving unrelated web pages. *Numeric pagination links* are generally harder to classify than non-numeric ones, due to their greater variety and the larger set of candidates. Though precision is 100% for both cases, recall is on average slightly lower for numeric pagination links (98% vs. 99%) and in some domains quite notable (e.g., Real Estate with 96% vs. 99%).

Performance results The speed of feature extraction is crucial for the scalability of our approach to allow the crawling of entire websites. As discussed above, the use of visual features by itself imposes a certain penalty, as a page needs to be rendered for those features to be computed. We present the performance results of $\text{BER}_{\text{y}}\text{L}$ in Figure 6.2, which shows,

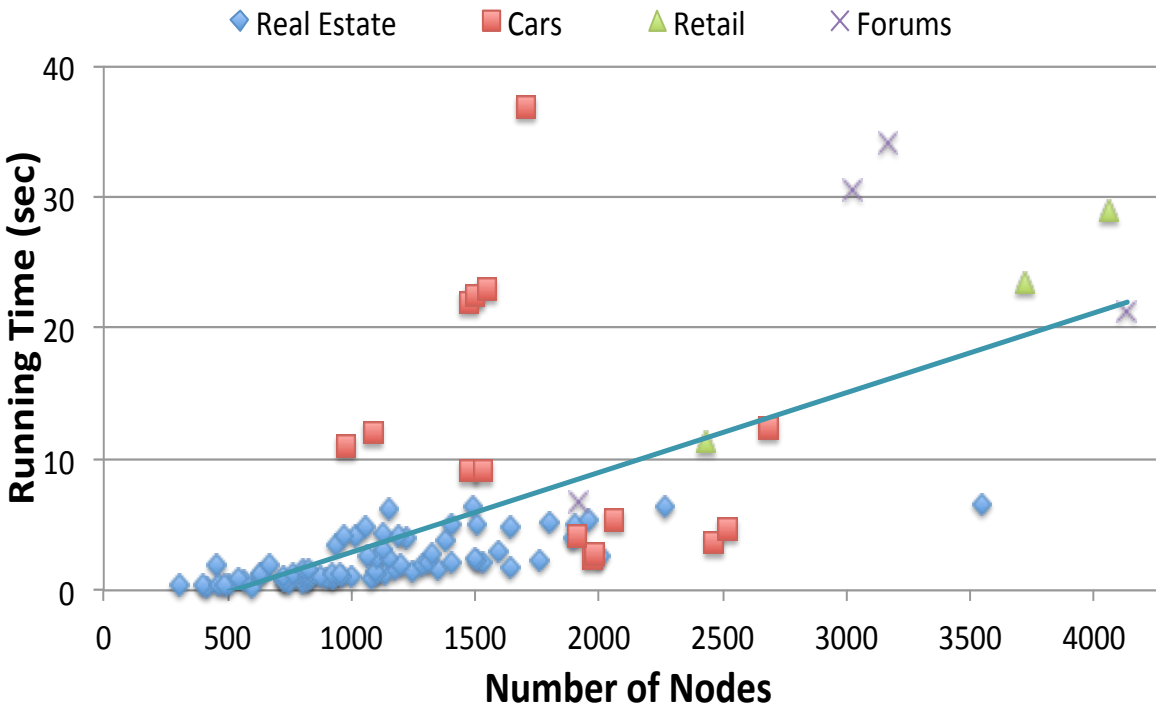


Figure 6.2: Performance of the pagination link model

for the example of the Pagination Link classifier, that the performance is highly correlated to page size, with most reasonably sized pages being processed in well below 10 seconds (including page fetch and rendering). It is interesting to observe that the domains for which we use Google to generate the corpus, and for which the corpus is thus biased towards popular websites, seem to require more time than the Real Estate domain, for which the corpus is randomly picked from `yell.com`.

6.3 Evaluation results for all classifiers of BER$_y$L

We have also performed evaluation on other constituent classifiers of the BER$_y$L system, namely Navigation Menus, Footers, Headers, and Sidebars. For all of these classifiers we have achieved good precision, recall and F_1 scores, although, as shown in the further section that evaluates our page-holistic approach to web block classification, this result can still be significantly improved. We present the precision, recall, and F1 results achieved for each of the individual classifiers in Figure 6.3. All of the classifiers have precision and recall scores above 80% apart from the Sidebar classifier, which has a precision score of just below 80%. This can be explained by the fact that a sidebar usually does not have obvious clues, such as the NEXT annotation for non-numeric next links, and therefore it is harder to distinguish True Positives from False Positives and False Negatives. Note that

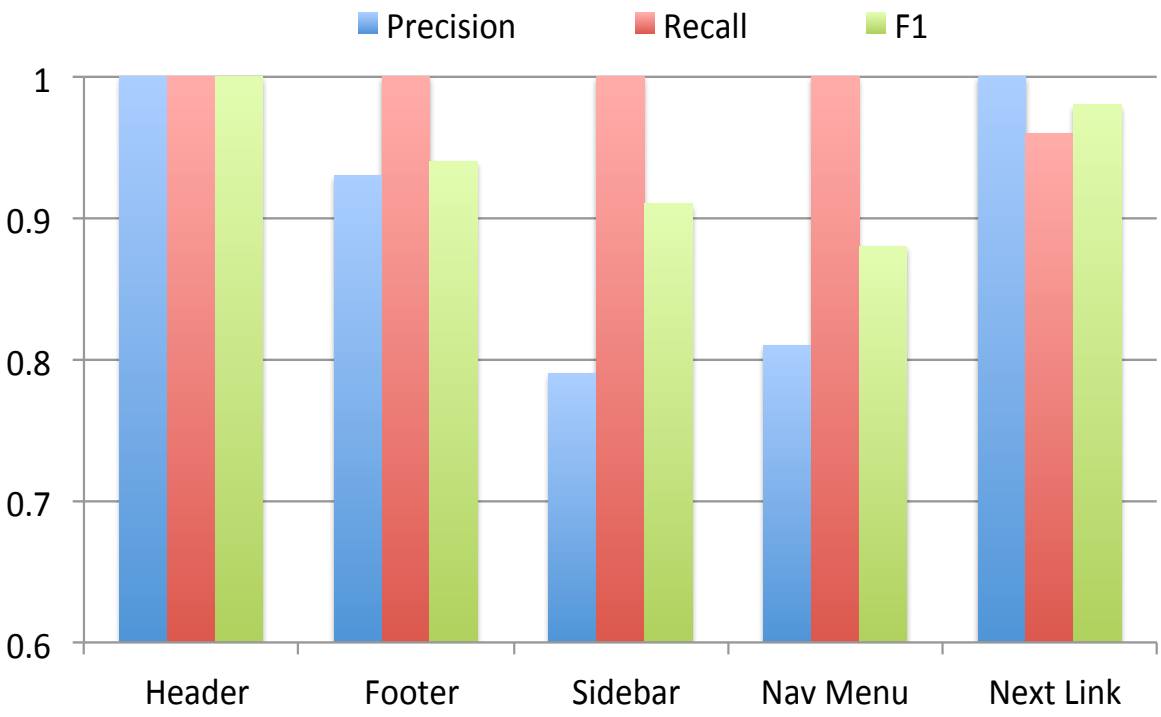


Figure 6.3: BER$_y$L's accuracy results on the five classifiers

the Next Link and Header classifiers achieve a perfect precision of 100%. This can be explained by the fact that we use a highly tailored feature set that includes features specific to the Next Link block, and that headers are very distinct from other blocks and our fairly simple set of six features used for the classification of this block is sufficient to achieve a perfect separation between header and non-header DOM nodes.

Figure 6.4 shows the distribution of processing time over web pages, sorted by processing time, for three different block types. The average running time for a web page is around 7-10 seconds, and it can be seen that the average running time for the domain-dependent block of floor plans is higher than for the domain-independent block types of pagination links and navigation menus.

6.4 Comparison to other approaches

In Table 6.2 we present the accuracy results achieved by the BER$_y$L system compared to other systems that are covered in Chapter 7 on Related Work. Some of the papers analysed there do not present the accuracy results, so we omit them from this comparison, and for others we compare to the approaches, which cover the block types also covered by the BER$_y$L system, and if there is no overlap between the block types covered, we compare on the average accuracy results for all classifiers in the system. As a lot of block types covered by our system are not covered by other approaches, we compare the precision, recall, and F_1 metrics over two block types where there is an overlap (navigation menus and pagination

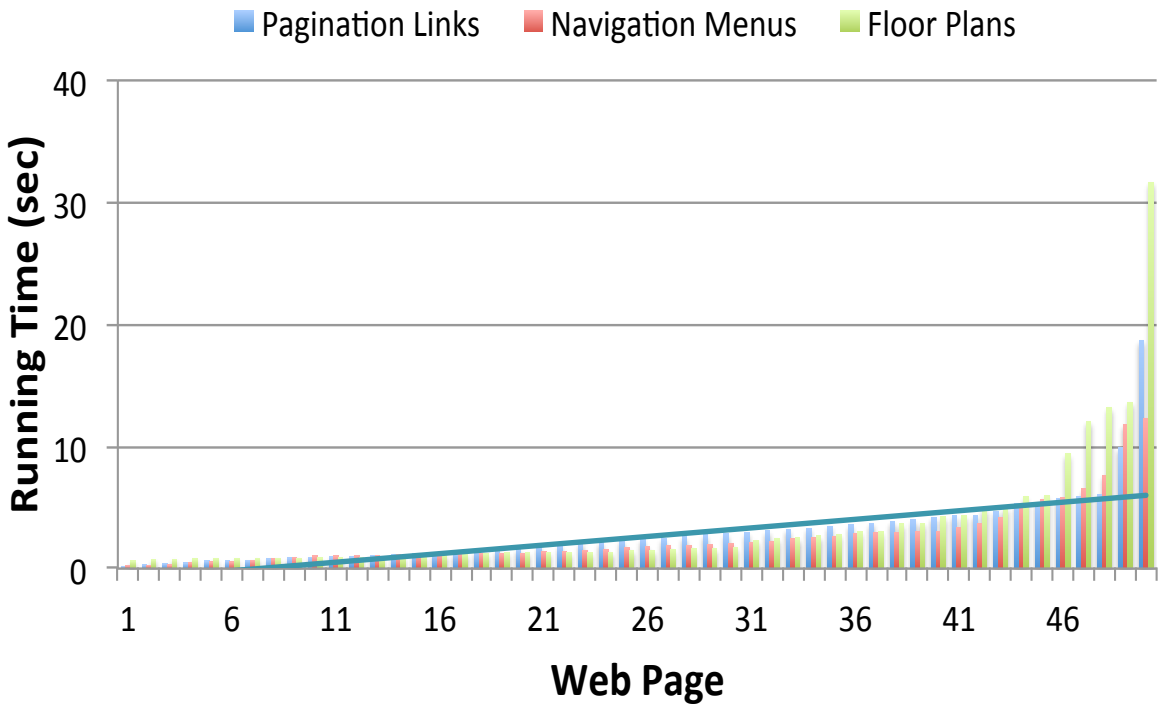


Figure 6.4: Overall performance histogram

bars), and the average precision, recall, and F_1 metrics for entire classification systems. If there is no data for a specific metric or classifier for the method we are benchmarking against, we indicate this by N/A in the relevant cell of Table 6.2.

In all cases BER$_y$L achieves higher precision and recall results for individual classifiers, as well as average precision and recall results for all classifiers. Our intuition is that the better results achieved by our system are due to the fact that we use distinct feature vectors for each of the block types, and each of these feature vectors is highly tailored to the block type it is being extracted on, whilst other approaches attempt to use a single coarse-grained feature vector for all block types their systems classify. In our further evaluation we will aim to reproduce the sets of features proposed by the models we have compared our approach to, reconstruct them in the BER$_y$L system and compare the results achieved with these feature sets on classifiers of interest to the results achieved with feature sets used in the BER$_y$L system, which are combined with the holistic approach to web block classification.

We have not compared the performance of BER$_y$L to other systems, as neither of them offers open-source code, so we have been unable to run BER$_y$L and these systems on the same inputs. However, if we re-engineer the features proposed by the other approaches in BER$_y$L, we will be able to compare the performance of feature extraction for given block types with the feature sets used for these block types by the BER$_y$L system and other sys-

	Pagination Bars			Navigation Menus			System Average		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
BER$_y$L	1.00	1.00	1.00	0.88	1.00	0.94	0.97	1.00	0.98
[31]	0.42	0.96	0.58	0.98	0.37	0.54	0.73	0.65	0.69
[43]	N/A	N/A	N/A	N/A	N/A	0.88	N/A	N/A	0.75
[37]	N/A	N/A	0.82	N/A	N/A	0.82	N/A	N/A	0.52
[25]	N/A	N/A	N/A	N/A	N/A	N/A	0.77	0.71	0.74
[34]	N/A	N/A	N/A	N/A	N/A	N/A	0.75	0.66	0.70

Table 6.2: Precision, recall, and F_1 of the BER$_y$L system compared to other systems

tems. This will allow us to compare the 'computational complexity' of the feature sets used in the BER$_y$L system and other systems within the environment where all features are programmed in the same declarative BER$_y$L language and are extracted on the same machine of a given processing power.

6.5 The α-accuracy evaluation of BER$_y$L

Once we have defined the feature vector for a given block type, we need to build its specific training corpus and then create a classifier based on this corpus.

The training corpus should capture the diversity of a block type, so that we do not encounter the problem of overfitting at the evaluation stage, i.e., that the classifier performs well on the training corpus but shows poor results on the evaluation corpus, due to the training corpus not being fully representative of the block type's domain. We would also like the training corpus to be minimal in size, as finding the labelling data takes a lot of human effort. The task of minimising the training corpus becomes especially important at the second phase of our web block classification system (large-scale classification of domain-independent blocks).

One of the sub-problems we face is the problem of achieving the necessary precision and recall with a set of features as concise as possible, due to performance limitations that are likely to be triggered by adding extra features, as well as minimising the amount of supervision required for classification (one of the main metrics for this is the size of the training corpus, measured by the number of pages included in it). We would therefore generally aim to define a feature vector that maximises accuracy whilst keeping the performance at an acceptable level and minimise the amount of supervision. We define this as the problem of the α-accuracy feature selection.

Definition 18. Given a web block type BT, its respective feature set $\mathbb{F}_{BT}$, and training corpus T_{BT}, the problem of α-accuracy feature selection is to find a subset of given features

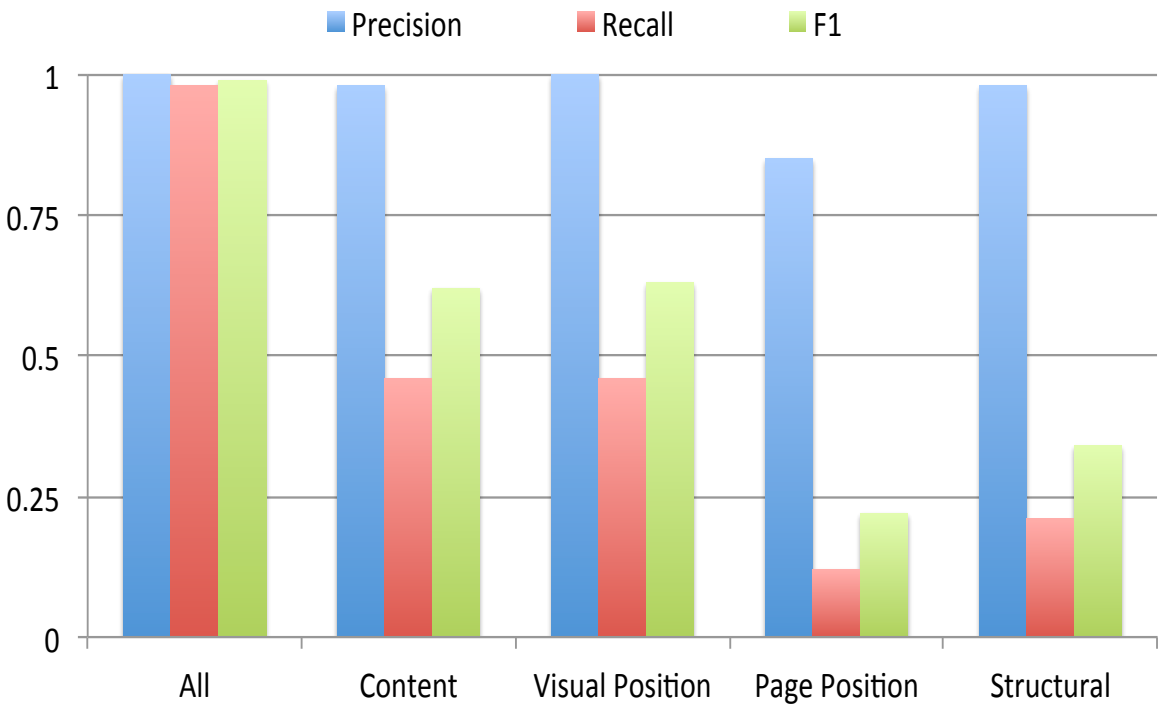


Figure 6.5: α-accuracy evaluation of the pagination link model

$\mathbb{F}'_{BT} \in \mathbb{F}_{BT}$ and a given training corpus $T'_{BT} \in T_{BT}$, such that the accuracy ($acc(CL_{BT})$) of the classifier CL_{BT} obtained by training on T'_{BT} with the feature set of $\mathbb{F}'_{BT}$ is at least α ($acc(CL_{BT}) \geq \alpha$).

Note that it is more important for us to reduce the size of the feature set than the size of the training corpus, since the training corpus is created only once. In Figure 6.5 we show the results of the α-accuracy evaluation of BER$_y$L for the pagination link model (PLM) classifier, acquired through splitting the feature set into four non-overlapping subsets (content features, page position features, visual proximity features, and structural features) and evaluating precision, recall, and F_1 scores on each of these subsets. For space reasons, we do not show the individual impact of each single feature, but note that all features included in the classification tree (Figure 5.2) contribute at least several percentage points to the overall precision and recall. Figure 6.5 also shows that content and visual proximity features are significantly more important for recall than page position and structural features.

6.6 Evaluation of BER$_y$L's holistic approach

We have also performed the evaluation of the holistic approach we take in the BER$_y$L system, which is one of our key contributions. The system of constraints that BER$_y$L utilises

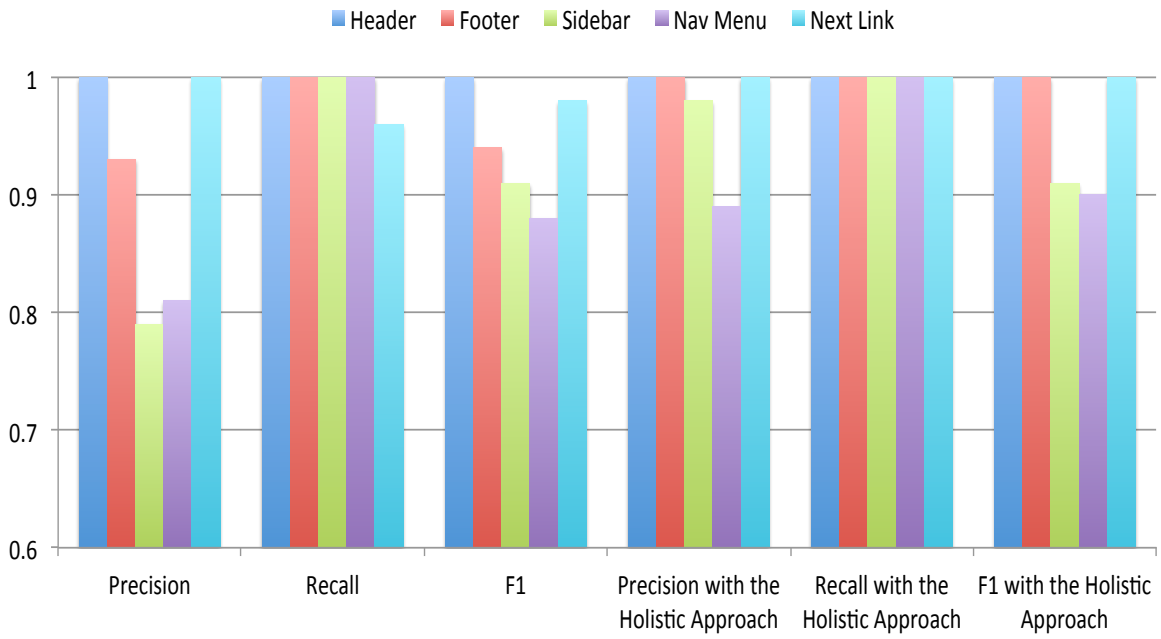


Figure 6.6: Standard per-block classification vs BER$_y$L's holistic approach

is described in Section 3.9 of Chapter 3. As can be seen from Figure 6.6, we get significant improvements in both precision and recall for all classifiers, but especially strong gain in precision for sidebars, navigation menus, and footers and strong gain in recall for next links. This proves that our hypothesis is valid and even with a limited set of inter-classifier constraints (both domain-independent or the ones that involve domain knowledge) can significantly improve the accuracy of classification.

Chapter 7

Related Work

We present an overview of the papers relevant to our main research area, which is web block classification, in Section 7.2. Web block classification aims to assign semantic labels to structural or visual blocks acquired from the DOM tree. Many web block classification approaches are based on machine learning techniques. We also provide a review of papers on the related task of web page segmentation in Section 7.1. Many approaches use web page segmentation as a pre-filter that outputs structural or visual blocks on which the web block classification is then performed. Web page segmentation is concerned with splitting the web page's DOM tree or its visual rendering into semantically coherent blocks, which in most cases do not overlap.

In Section 7.2, we also give a brief overview of the literature on web page classification, a field we want to research in the future, that is concerned with determining what domain the web page in question belongs to.

7.1 Web page segmentation

Web page segmentation is a process of splitting the web page into semantically related content blocks. Each web page segmentation algorithm introduces its own metric for measuring consistency. For example, the popular VIPS algorithm [10] is based on visual consistency of blocks and defines ten different levels of segmentation granularity measured by the 1-10 *degree of coherence (DOC)* metric. We give an example of page segmentation performed by the VIPS algorithm in Figure 7.1. We cover VIPS in more detail further in this section.

There has been quite a lot of research done in the field of web page segmentation. All of the approaches we are familiar with apart from [8, 27, 58] are based on DOM tree traversal. Some of these are based solely on the tree's structure. Others employ the visual layout of the web page. In general, the latter type of approaches produces better results than the

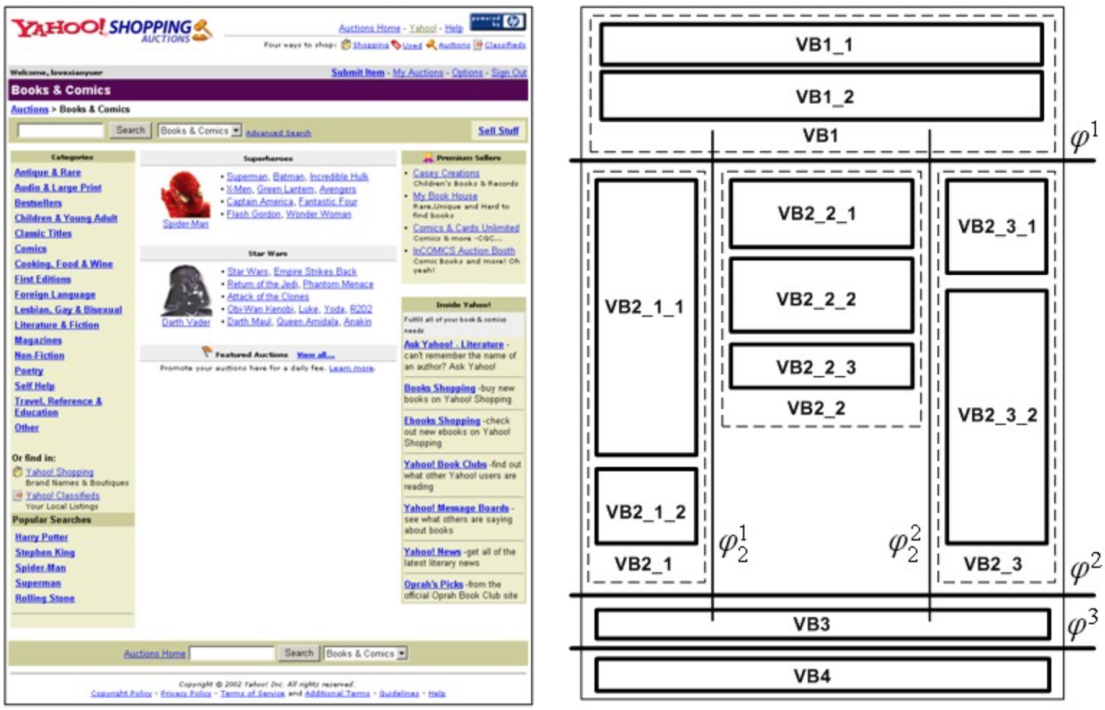


Figure 7.1: Segmentation of a web page performed by the VIPS algorithm [10]

former. We can also distinguish between general and focused segmentation approaches. In focused segmentation, the page is first split into a hierarchy of semantic blocks and then one or more blocks is selected and labelled with a certain semantic meaning, e.g., Main Content or Title. We give an overview of these approaches in the current section, as they present an enhancement of classic web page segmentation with web block classification. There have been a few papers on the applications of web page segmentation. We present them in this section as well.

In Table 7.1 we list the referenced papers along with some of the attributes that the approaches proposed in them possess, namely whether they are based on structural or visual representation, top-down or bottom-up, general or focused, and domain-specific or domain-independent.

We would like to give special consideration to two systems in this review, one visual-based and the other structural-based. VIPS [10] is a visual-based system, first implemented in 2003 and significantly modified in the following years. It is now partially integrated into Microsoft's Bing search engine. VIPS employs a top-down segmentation approach based on visual cue detection. It first segments the DOM tree into blocks by consecutively splitting the element nodes and then assigns weights to separators between blocks that have been constructed. It finally constructs a block tree structure, making sure that each

Referenced Paper	Structural/Visual	Top-down/Bottom-up	General/Focused	Domain dependency
Cai et al. (VIPS) [10]	Visual	Top-down	General	Domain-independent
Kang et al. (REPS) [33]	Structural	Top-down	General	Domain-independent
Vadrevu et al. [52]	Structural	Top-down	General	Domain-independent
Romero and Berger [47]	Structural	Bottom-up	General	Domain-independent
Xiang et al. [58]	Both	Bottom-up	General	Domain-independent
Mehta et al. [44]	Both	Top-down	General	Domain-independent
Sano et al. [48]	Both	Top-down	Focused	Domain-independent
Gupta et al. [28]	Structural	Top-down	Focused	Domain-independent
Wu et al. [56]	Both	Bottom-up	Focused	Domain-dependent

Table 7.1: Taxonomy of segmentation approaches

block meets the granularity requirement, i.e., that its Degree of Coherence (DOC) value is greater than a predefined PDOC value, a natural number ranging from 1 to 10. VIPS is a very popular system and has been cited in every single paper presented in our review of web page segmentation approaches. In Figure 7.2 we present a brief snapshot of the VIPS' architecture.

REPS [33] is a structural-based system developed in 2010. Its main idea is to find repetitive tag patterns in the DOM tree that are then used to segment it into semantic blocks. The authors claim that they mostly outperform VIPS in terms of the correctness of web page segmentation.

In [52], the authors propose using *path entropy* within the DOM tree to segment it into blocks, which for each element node determines the homogeneity of the subtree rooted at that node. [47] suggests a structural bottom-up approach that evaluates a cost function for merging adjacent pairs of segments, which in this case consist of disjointed sets of continuous DOM leaf nodes. [57] builds up an algorithm based on a combination of structural and visual analysis. The structural analysis phase consists of finding repeated continuous patterns in the DOM tree. The visual analysis phase is based on heuristics borrowed from VIPS [10]. Finally, the authors of [44] propose an iterative application of VIPS that runs through different PDOC values ranging from 1 to 10 and is validated by a Naïve Bayes classifier, which checks the semantic consistency of blocks by ensuring that each block in the segmentation tree belongs to exactly one category from the open directory project (ODP) ontology.[1]

There are a few outlier papers that do not quite fit in the presented hierarchy and have not been included in Table 7.1. We give a brief description of each one. The authors of [2] suggest a clustering approach based on three different distance metrics derived from the DOM tree – the actual DOM-based distance, geometric distance and semantic distance.

[1] http://www.dmoztools.net

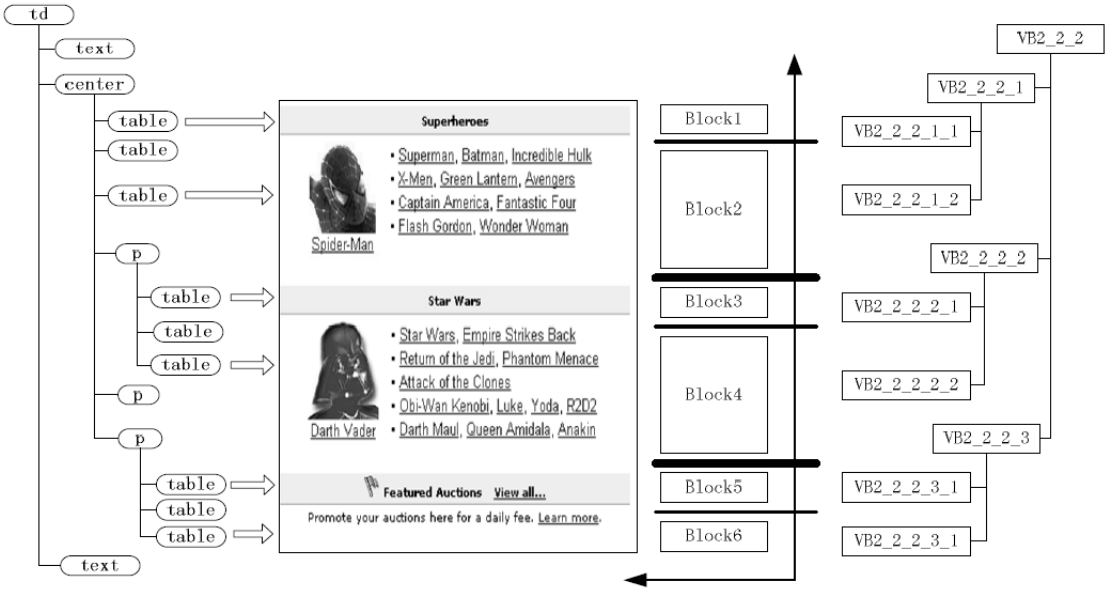


Figure 7.2: An overview of the VIPS' architecture [10]

In [15], a novel graph-based approach is proposed. A web page is viewed as a complete weighted graph in which the DOM nodes correspond to vertices and the edge weights express the cost of placing the end points in the same or different segments. The paper presents two algorithms for analysing the graph, based on correlation clustering and energy minimisation graph cuts, and argues that the latter yields much better results than the former. The authors of [58] propose a novel data structure, which they call a layout tree, as an input for their segmentation method. Their approach is based on the four laws from Gestalt theory, a psychological concept that aims to explain visual perception of humans through proximity, similarity, closure, and simplicity. This algorithm, along with VIPS [10] and REPS [33], can serve as a useful baseline for the evaluation of BER$_y$L's potential segmentation sub-module. [8, 27, 58] are the only papers among those we observed in the literature that base their segmentation algorithms not on the DOM structure directly, but rather on its suitable abstraction. However, in [27], the representation of the CSS containment tree is based on the Mozilla frame tree[2], and [8] uses its own HTML/CSS rendering engine, which essentially limits the range of browser APIs these approaches can be used for to Gecko. The algorithm proposed in [27] proceeds in two steps. It first clusters box elements in a bottom-up way by looking at their vertical and/or horizontal alignments and then performs a top-down segmentation by finding appropriate separators between element DOM nodes, based on their presentation features, such as font type, font size, and font style.

[2]http://wiki.mozilla.org/Gecko:Key_Gecko_Structures_And_Invariants

Focused segmentation

As discussed above, focused segmentation approaches provide classic segmentation followed by the identification of one or more semantic web blocks. We give a brief overview of the papers in this field.

The authors of [48] propose a method for title blocks extraction. Their algorithm first performs a top-down traversal of the DOM tree to determine which one of the eight possible layouts the web page most corresponds to. The page is then segmented into minimal blocks, and title blocks are identified via a J48 machine learning classifier. [56] presents an approach for extracting the main content area (Product Information in the authors' terminology) for the pages in the Online Retail domain. It first performs a bottom-up segmentation, followed by block importance analysis based on the evaluation of a predefined function. Finally, the most important block is extracted as Product Information. [28] introduces a domain-independent approach for extracting the main content from a web page. For that purpose, the authors propose the application of two sets of filters, with different levels of granularity, to a DOM tree. The first, coarser filter simply ignores tags or specific attributes within tags, whereas the second, more refined filter includes the advertisement remover, the link list remover, the empty table remover, and the removed link retainer.

Applications of web page segmentation

There have been a few concrete applications of web page segmentation proposed by the Microsoft Research Asia team that developed VIPS [10].

In [62], the authors suggest the use of VIPS to improve pseudo-relevance feedback by only extracting information from the most relevant segments, determined by a set of ranking methods. The approaches described in [11] and [12] use VIPS as a pre-filter for block-based web search and block-level link analysis, respectively.

There are two major points of relevance that this review presents to our research: **(1)** BER$_y$L's potential segmentation sub-module can contribute from the heuristics proposed in the papers we have reviewed, and we plan to implement some of them in the near future; **(2)** we can use some of the systems presented, such as VIPS [10], REPS [33], and the methods based on Gestalt theory of visual perception [58], as baseline approaches for evaluating our segmentation sub-module.

7.2 Web block and web page classification

Web block classification is the most relevant area of research in the context of our BER$_y$L framework. There have been several papers published on this topic in the past ten years. Broadly speaking, web block classification methods can be split into those based on machine learning (ML) techniques and those based on other approaches (i.e., the rule-based and heuristic approaches). Also, the features used for the analysis of web blocks in ML-driven and other algorithms can be subdivided into structural (based on the structure of a DOM tree or another intermediary structure representing a web page), visual (based on the web page's rendering), and lexical (based on the analysis of the web page's free text). In Table 7.2 we present a taxonomy of papers on web block classification similar to the one we gave in Section 7.1. In general, we distinguish these approaches according to five criteria: types of blocks extracted, types of block features used (St for Structural, V for Visual, and Lex for Lexical), whether they are based on machine learning techniques or not, whether a segmentation algorithm is used as a pre-filter, and domain dependency. Note that we assume the classification approach to be domain-dependent if it is explicitly defined as such in the paper, or only one domain of choice is presented in the evaluation section.

Approaches presented in [13, 32, 39, 53, 60] focus on extracting the main content of a web page. In DIADEM, the content area identification is performed by AMBER, a Datalog-driven rule-based module, which extracts records and their attributes for DIADEM's domains of choice (Real Estate and Used Cars as for the current prototype). We can use some of the methods presented in these papers to verify results produced by AMBER. We can also use AMBER and other main content detection methods as a negative filter when performing classification of other block types such as Navigation Menus, Advertisements, and Footnotes.

[41, 64, 65] propose methods for the identification of records and their respective attributes. The authors of [41] introduce VIDE, a system for record and attribute extraction. It is based on appearance similarity and alignment techniques, and demonstrates very high precision and recall values. [64] suggests a method for record-level wrapper induction that is based on a novel 'broom' data structure, which is used for representing records and generated wrappers. [65] proposes a simultaneous approach for record detection and attribute labelling, which is based on the combination of the two tasks through the use of Hierarchical Conditional Random Fields (HCRFs). However, all three approaches are outperformed by AMBER for DIADEM's domains of choice.

The approach taken in [55] proposes a machine learning method for the extraction of titles and bodies from news web pages. The authors introduce a comprehensive set of

Referenced Paper	Block Labels	Features	ML-based	Segmentation	Domain dependency
Cao et al. [13]	Main Content	St, V, and Lex	Yes	Yes (VIPS)	Domain-independent
Yi et al. [60]	Main Content	St	No	No	Domain-independent
Li et al. [39]	Main Content	St, V, and Lex	Yes	Yes (VIPS)	Domain-independent
Kang and Choi [32]	Main Content	St	No	Yes (VIPS)	Domain-independent
Vadrevu and Velipasaoglu [53]	Main Content	St and V	Yes	Yes	Domain-independent
Liu et al. [41]	Records and Attributes	St and V	No	Yes (VIPS)	Domain-independent
Zhu et al. [65]	Records and Attributes	St and V	No	Yes (VIPS)	Domain-independent
Zheng et al. [64]	Records and Attributes	St	No	No	Domain-independent
Wang et al. [55]	News Title and Body	St and V	Yes	No	Domain-dependent
Maekawa et al. [43]	Eleven classes of images	St and V	Yes	No	Domain-independent
Goel et al. [25]	Maps	V	No	No	Domain-independent
Burget and Rudolfova [9]	Nine classes from the News Domain	St and V	Yes	Yes	Domain-dependent
Song et al. [50]	Three importance levels	St and V	Yes	Yes (VIPS)	Domain-independent
Chen et al. [16]	Six object functions	St and V	No	No	Domain-independent
Yang and Shi [59]	Five block functions	St and V	Yes	Yes	Domain-independent
Kang and Choi [31]	Menu, Content, NV List, Garbage	St and V	Yes	Yes (VIPS)	Domain-independent
Lee et al. [37]	Seventeen classes from the News domain	St, V, and Lex	Yes	Yes	Domain-dependent
Keller and Hartenstein [34]	Breadcrumb Trails	St	Yes	No	Domain-independent

Table 7.2: Taxonomy of web block classification approaches

features, some of which we have re-implemented in our classifiers. [25] proposes an interesting algorithm that relies on the Content Based Image Retrieval (CBIR) technique for distinguishing map images from non-map images on the web. However, it has to download the images and analyse their visual representation, which can be time-consuming and is most likely not scalable to the entire web, a property that is crucial for DIADEM. A more promising approach for image classification is introduced in [43]. It presents eleven categories for image classification, including maps, and employs a machine learning classifier based on 37 features that are defined solely through DOM tree structure and CSS encoding. The precision and recall values for map classification are reported to be 100%, and therefore we will attempt to use this technique in BER$_y$L to classify maps and possibly other types of images. The authors of [9] provide a useful analysis of visual features, which can be employed for web block classification. They present a machine learning algorithm on web pages taken from the News domain to classify pre-segmented blocks (the approach used for segmentation is described in [8]) into nine separate classes. Whilst this method is similar to ours in using a machine learning classifier for block labelling, the F_1 values it achieves (below 90% for all classes and above 80% for two classes out of nine) are unacceptable for the accuracy level we aim for in BER$_y$L.

In [50], the authors propose the assignment of one of the four importance levels (noisy information, useful but not very relevant information, relevant information, and the most prominent part of the page) to pre-segmented web blocks. This is achieved through a machine learning classifier based on content and visual features. However, the F_1 values this approach achieves are relatively low (below 90% for all cases).

[16] and [59] prefer to classify blocks through their functional roles rather than semantic meaning. The authors of [16] introduce an ontological schema for the description of functional roles of different parts of a web page, which distinguishes the following five functions: Presentation, Semanteme, Decoration, Hyperlink, and Interaction. [59] presents a machine learning approach for classification of web blocks according to their functional meaning, which is described by five categories: Information, Interaction, Navigation, Advertisement, and Other. This method uses roles of images contained in the web blocks (described in [43]) as one of the key features for functional classification. These functional-based approaches can be of use in the context of BER$_y$L as coarse-level pre-filters.

The approaches presented in [31] and [37] use a union of co-trained classifiers to enhance the accuracy of prediction. [31] employs a combination of two classifiers, the first based on a tree alignment technique, and the second based on a machine learning 1-NN method that is passed a 14-dimensional feature vector as input. These classifiers are joined

through a machine learning technique called bagging [7]. However, the granularity of distinguished block types is quite coarse (the four block types are Menu, Content, Navigation List, and Garbage), and the F_1 values achieved are unacceptable for the purpose of BER$_y$L (below 80% for all types of blocks). The authors of [37] offer another co-training approach, this time based on a combination of stylistic (structural in our notation) and lexical features. The mutual co-training is performed with `BoosTexter` [49], an ensemble learning method based on combining a set of weak classifiers into one strong one. This method is evaluated on the News domain, for which seventeen block types of different granularity are identified. However, the classification accuracy results are relatively low (F_1 values are below 90% for all block types).

In [34], the authors propose a novel approach to web block classification based on converting the style identifiers acquired from CSS `class` and `id` attributes into block groups and corresponding link graphs, and apply it to classification of breadcrumb trails. They also cite our work on classification of pagination links [22] and mention that we have achieved nearly perfect accuracy for that task.

The proposed web block classification approaches are mostly relevant to our BER$_y$L system with respect to the feature sets they use. We would also like to experiment with the approaches based on non-ML techniques, such as tree alignment. Some of these methods can serve as useful baselines. However, the vast majority of them do not take into account domain knowledge and those that do, use it for one domain only. Hence we think that these approaches can be outperformed in accuracy and block granularity for BER$_y$L's domains of choice.

Web page classification

Web page classification is an area we would like to research, due to the domain-specific nature of our system, since we need to be able to determine accurately whether a given page can be categorised as belonging to any of the domains we consider in DIADEM (Real Estate and Used Cars, for the moment). There has been quite a lot of work done in this field, but for now we limit ourselves to a brief overview of papers we think are most relevant in the context of our research.

[61] proposes `PEBL`, a system for web page classification without negative examples, achieved through the use of a Mapping-Convergence (M-C) Algorithm. The evaluation is performed on the 'Entire Internet' and 'University Computer Science Department' sets, with three distinct types of pages specified for each set. The authors claim that `PEBL` significantly outperforms one-class SVMs and achieves results almost as accurate as traditional

SVMs. Another interesting and computationally cheap approach, based solely on the analysis of web pages' URLs, is presented in [30]. It first splits URLs into tokens, which are then passed on as feature vectors to a machine learning classifier. This solution is very efficient both in time and space, as the data examined is small in comparison to other methods. We can tailor this method to classify pages in our specific domains. Finally, in [42] the authors suggest a method that attempts to combine the fields of web page classification and information extraction through the use of Conditional Random Fields (CRFs). The forward dependency is quite obvious, as knowing which domain a web page belongs to can significantly improve the accuracy of extracting data from it. The backward dependency is less trivial, but the authors provide a strong example for its existence. This framework would be very interesting to evaluate in DIADEM, as one of its modules AMBER is specifically tailored for information extraction from certain domains (Real Estate and Used Cars for the recent prototype), and the current version of our BER$_y$L module is capable of accurately classifying a range of web blocks as well. We will attempt to link these two modules, separately or as a combination, with a web page classification module and evaluate any mutual accuracy gain.

Chapter 8

Conclusion

We propose a two-phase web block classification system, BER$_y$L, where the two phases relate to: **(1)** classification of individual web blocks and **(2)** large-scale classification of domain-independent web blocks. We have completed and validated both phases of the system. The global feature, template, and annotation framework in combination with verification framework allows us to validate all main hypotheses of the BER$_y$L system, in particular the ones listed below.

(i) Employment of domain-specific knowledge enhances accuracy and performance of both domain-dependent and domain-independent classifiers.

(ii) A holistic view of the page implemented through constraints enhances the accuracy and performance of classification.

(iii) Global feature, template, and annotation repositories provide for substantial improvement in the manual effort of defining new features and improve the performance of feature extraction.

We introduce a novel modular-based approach to feature extraction, where complex features are to be engineered in a fast and robust manner through the combination of a component model with the Datalog-based BER$_y$L language. We provide a thorough set of examples to show how this approach is used in practice, as well as its large-scale evaluation.

We also introduce templates and global features, which are used throughout different classifiers to improve the time required to engineer a right feature set and also improve the overall performance of the system. For example, two complex features from different classifiers can follow the same template and only differ slightly in specific parameters, in which case they are both acquired through the instantiation of this general template, or a feature (for instance, the CSS box width) can be common to more than two block types, in

which case it is imported from the global feature repository at the step before computation of features for actual block types takes place.

A holistic view of the page is implemented through a system of constraints on the output classifications produced by the first stage of the BER$_y$L system. The evaluation proves that the holistic approach to web page classification significantly improves the accuracy of BER$_y$L.

Self-evaluation and future research We aim to pursue multiple avenues for future research. Below, we list those upon which we want to focus particularly.

1. Enhancement of the large-scale domain-dependent part of the classification system. In particular, we would like to develop a framework for automatic learning of new features from the generic feature templates on top of the existing component model approach powered by the BER$_y$L language. We would also like to further improve the techniques for the reduction of feature spaces and selection of optimal features from the global feature repository.

2. Exploration of further improvements to our greedy algorithm approach for constraint conflict resolution and optimal execution of classifiers, in order to minimise the overall running time of the BER$_y$L system. We aim to model this through building a graph between constraints and then executing them (and possibly the underlying classifiers as well) in the order dictated by that graph.

3. Further exploration of the α-accuracy problem, which allows the system to find an optimal balance between the accuracy and performance expectations set by the user (as in most applied systems, performance is crucial and sometimes accuracies of individual classifiers need to be slightly lowered in order to reach acceptable overall performance of the system). The key question, then, is how to achieve acceptable overall performance of the system, whilst sacrificing the accuracies of individual classifiers as little as possible.

4. Integration of new sources for input features into our system, such as URLs and outputs generated by performing certain actions on the page (e.g., filling in the search form or clicking on the "Next" link candidate link/button or one of the elements of a candidate navigation menu web block).

5. Integration of our component-based declarative-driven approach to feature extraction into domains that go beyond web block classification. This is a particularly interesting area of future work that, if successful, will result in an API that can be used by other machine learning researchers in their systems.

6. Extension of the holistic approach component of the BER$_y$L system to handle external sources of classification. For example, we would like to integrate BER$_y$L with the OPAL module of the DIADEM system [23] that allows users to accurately identify forms on web pages, as well as the AMBER module of the same system that allows users to detect data records and the main content area of result pages. That will allow us to improve the overall performance of BER$_y$L.

Applications of our system We foresee several practical applications of the approach proposed. In particular, the two most interesting applications we would like to focus on are data extraction and advertisements removal.

1. We can use block classifiers for various data extraction tasks. For example, being effective in identifying navigation blocks, such as "Next" links or navigation menus with high precision and recall, will allow web data extraction systems to effectively navigate between constituent pages of the web site. Our classification system can also be used as a pre-filter for data extraction systems. For example, a system that is concerned with extracting data from the main content area of the page, such as records for an Online Retail website, does not need to consider headers and footers, which get removed by our classifiers. That is likely to improve the classification accuracy of the record extraction, as well as its performance, as irrelevant parts of the underlying DOM tree will not be considered. In particular, we are interested in integrating BER$_y$L as part of broader object-oriented and vertical-oriented data extraction platforms, such as DIADEM[1] and VADA[2].

2. Removal of advertisements is another important application of BER$_y$L. A lot of current ad removers use manually programmed lists of blocked URLs. However, creators of the ads can circumvent these blocker lists by creating new URLs for the same ads. On the other hand, an accurate classifier for advertisements based on visual and structural features is agnostic to the URLs of the ads, and is therefore likely to be more accurate and robust in the task of ad removal.

[1] http://diadem.cs.ox.ac.uk/
[2] http://www.cs.ox.ac.uk/projects/vada/

Bibliography

[1] S. Abiteboul, R. Hull and V. Vianu. Foundations of Databases. *Addison-Wesley Longman Publishing Co., Inc.*, 1995

[2] S. Alcic and S. Conrad. Page Segmentation by Web Content Clustering. *WIMS 2011*, May 25–27, 2011.

[3] G. Almpanidis, C. Kotropoulos and I. Pitas. Combining Text and Link Analysis for Focused Crawling – an Application for Vertical Search Engines. *Information Systems Journal, volume 32, no. 6*, 2007.

[4] S. Baluja. Browsing on Small Screens: Recasting Web-Page Segmentation into an Efficient Machine Learning Framework. *WWW'06*, 2006.

[5] R. Baumgartner, S. Flesca and G. Gottlob Visual Web Information Extraction with Lixto. *VLDB*, 2001

[6] P. D. Bra and R. D. J. Post. Information Retrieval in the World-Wide Web: Making Client-Based Searching Feasible. *Computer Networks and ISDN Systems, volume 27, no. 2*, 1994.

[7] L. Breiman. Bagging Predictors. *Machine Learning, volume 24, no. 2*, 1996.

[8] R. Burget. Layout Based Information Extraction from HTML Documents. *Ninth International Conference on Document Analysis and Recognition*, September 23–26, 2007.

[9] R. Burget and I. Rudolfova. Web Page Element Classification Based on Visual Features. *2009 First Asia Conference on Intelligent Information and Database Systems*, 2009.

[10] D. Cai, S. Yu, J. Wen and W. Ma. Extracting Content Structure for Web Pages Based on Visual Representation. *Proceedings of the 5th Asia-Pacific Web Conference on Web Technologies and Applications*, 2003.

[11] D. Cai, S. Yu, J. Wen and W. Ma. Block-Based Web Search. *SIGIR'04*, July 25–29, 2004.

[12] D. Cai, X. He, J. Wen and W. Ma. Block-Level Link Analysis. *SIGIR'04*, July 25–29, 2004.

[13] Y. Cao, Z. Niu, L. Dai and Y. Zhao. Extraction of Informative Blocks from Web Pages. *International Conference on Advanced Language Processing and Web Information Technology*, 2008.

[14] S. Chakrabarti, M. V. D. Berg and B. Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. *Computer Networks*, 1999.

[15] D. Chakrabarti, R. Kumar and K. Punera. A Graph-Theoretic Approach to Web Page Segmentation. *WWW'08*, April 21–25, 2008.

[16] J. Chen, B. Zhou, J. Shi, H. Zhang and Q. Fengwu. Function-Based Object Model Towards Website Adaptation. *WWW'10*, May 1–5, 2010.

[17] V. Crescenzi and G. Mecca. Automatic Information Extraction from Large Websites. *Journal of the ACM, volume 51, no. 5*, 2004.

[18] O. de Moor, G. Gottlob, T. Furche and A. Sellers, eds. Datalog Reloaded, Revised Selected Papers. *LNCS*, 2011.

[19] M. Diligenti, F. M. Coetzee, S. Lawrence, C. L. Giles and M. Gori. Focused Crawling Using Context Graphs. *VLDB*, 2000.

[20] B. Fazzinga, S. Flesca and A. Tagarelli. Schema-Based Web Wrapping. *Knowledge and Information Systems, volume 26, no. 1*, 2011.

[21] E. Ferrara, G. Fuimara, R. Baumgartner. Web Data Extraction, Applications and Techniques: A Survey. *Knowledge-Based Systems Journal, 70, C*, November 2014.

[22] T. Furche, G. Grasso, A. Kravchenko and C. Schallhart. Turn the Page: Automated Traversal of Paginated Websites. *ICWE'12*, 2012.

[23] T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. J. Sellers and C. Wang. DIADEM: domain-centric, intelligent, automated data extraction methodology. *WWW'12*, 2012.

[24] R. Gaizauskas, H. Cunnigham, Y. Wilks, P. Rodgers and K. Humphreys. GATE: an Environment to Support Research and Development in Natural Language Engineering. *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[25] A. Goel, M. Michelson and C. A. Knoblock. Harvesting Maps on the Web. *International Journal on Document Analysis and Recognition, volume 14, no. 4*, 2011.

[26] G. Gottlob, G. Orsi, A. Pieris and M. Šimkus. Datalog and Its Extensions for Semantic Web Databases. *Springer Berlin Heidelberg*, 2012.

[27] H. Guo, J. Mahmud, Y. Borodin and A. Stent. A General Approach for Partitioning Web Page Content Based on Geometric and Style Information. *Ninth International Conference on Document Analysis and Recognition*, September 23–26, 2007.

[28] S. Gupta, G. Kaiser, D. Neistadt and P. Grimm. DOM-Based Content Extraction of HTML Documents. *WWW'03*, May 20–24, 2003.

[29] M. Hersovici, M. Jacovi, Y. S. Maarek, D. Pelleg, M. Shtalhaim and S. Ur. The Shark-Search Algorithm. An Application: Tailored Web Site Mapping. *Computer Networks and ISDN Systems, volume 30, nos. 1-7*, 1998.

[30] M. Kan and H. O. N. Thi. Fast Web Page Classification Using URL Features. *Proceedings of the 14th ACM International Conference on Information and Knowledge Management CIKM'05*, 2005.

[31] J. Kang and J. Choi. Block Classification of a Web Page by Using a Combination of Multiple Classifiers. *Fourth International Conference on Networked Computing and Advanced Information Management*, September 2–4, 2008.

[32] J. Kang and J. Choi. Recognising Informative Web Page Blocks Using Visual Segmentation for Efficient Information Extraction. *Journal of Universal Computer Science, volume 14, no. 11*, 2008.

[33] J. Kang, J. Yang and J. Choi . Repetition-Based Web Page Segmentation by Detecting Tag Patterns for Small-Screen Devices. *IEEE Transactions on Consumer Electronics, volume 56, no. 2*, 2010.

[34] M. Keller and H. Hartenstein. GRABEX: A Graph-Based Method for Web Site Block Classification and its Application on Mining Breadcrumb Trails. *2013*

IEEE/WIC/ACM International Conferences on Web Intelligence (WI) and Intelligent Agent Technology (IAT), 2013.

[35] A. Kravchenko. DIADEM: Domains to Databases. *35th European Conference on Information Retrieval ECIR 2013. Doctoral Consortium, 7-14*, March 24-27, 2013.

[36] B. Krüpl-Sypien, R. R. Fayzrakhmanov, W. Z. Holzinger, M. Panzenböck and R. Baumgartner. A versatile model for web page representation, information extraction and content re-packaging. *DocEng'11*, September 19-22, 2011.

[37] C. H. Lee, M. Kan and S. Lai. Stylistic and Lexical Co-Training for Web Block Classification. *WIDM'04*, November 12–13, 2004.

[38] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello. The DLV System For Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic (TOCL), volume 7, issue 3*, July 2006.

[39] C. Li, J. Dong and J. Chen. Extraction of Informative Blocks from Web Pages Based on VIPS. *Journal of Computational Information Systems, volume 6, no. 1*, 2010.

[40] H. Liu, J. Janssen and E. Milios. Using HMM to Learn User Browsing Patterns for Focused Web Crawling. *DKE, volume 59, no. 2*, 2006.

[41] W. Liu and X. Meng. VIDE: a Vision-Based Approach for Deep Web Data Extraction. *IEEE Transactions on Knowledge and Data Engineering, volume 22, no. 3*, 2010.

[42] P. Luo, F. Lin, Y. Xiong, Y. Zhao and Z. Shi. Towards Combining Web Classification and Web Information Extraction: a Case Study. *KDD'09*, June 28–July 1, 2009.

[43] T. Maekawa, T. Hara and S. Nishio. Image Classification for Mobile Web Browsing. *WWW'06*, May 23–26, 2006.

[44] R. R. Mehta, P. Mitra and H. Karnick. Extracting Semantic Structure of Web Documents Using Content and Visual Information. *WWW'05*, May 10–14, 2005.

[45] G. Pant and P. Srinivasan. Learning to Crawl: Comparing Classification Schemes. *TOIS, volume 23, no. 4*, 2005.

[46] G. Pant and P. Srinivasan. Link Contexts in Classifier-Guided Topical Crawlers. *TKDE, volume 18, no. 1*, 2006.

[47] R. Romero and A. Berger. Automatic Partitioning of Web Pages Using Clustering. *MHCI*, 2004.

[48] H. Sano, S. Shiramatsu, T. Ozono and T. Shintani. A Web Page Segmentation Method Based on Page Layouts and Title Blocks. *International Journal of Computer Science and Network Security, volume 11, no. 10*, 2011.

[49] R.E. Schapire and Y. Singer. BoosTexter: A Boosting-Based System for Text Categorisation. *Machine Learning, volume 39, nos. 2/3*, 2000.

[50] R. Song, H. Liu, J. Wen and W. Ma. Learning Block Importance Models for Web Pages. *WWW'04*, May 17–22, 2004.

[51] P. Srinivasan, F. Menczer and G. Pant. A General Evaluation Framework for Topical Crawlers. *Information Retrieval, volume 8, no. 3*, 2005.

[52] S. Vadrevu, F. Gelgi and H. Davulcu. Semantic Partitioning of Web Pages. *International Conference on Web Information Systems Engineering*, 2005.

[53] S. Vadrevu and E. Velipasaoglu. Identifying Primary Content from Web Page and its Application to Web Search Ranking. *WWW'11*, 2011.

[54] A. Van Gelder, K. Ross and J. Schlipf. Well-founded Semantics for General Logic Programs. *Journal of the ACM, volume 38, no. 3*, 1991.

[55] J. Wang, C. Chen, C. Wang, J. Pei, J. Bu, Z. Guan and W. V. Zhang. Can We Learn a Template-Independent Wrapper for News Article Extraction from a Single Training Site? *KDD'09*, June 28–July 1, 2009.

[56] C. Wu, G. Zeng and G. Xu. A Web Page Segmentation Algorithm for Extracting Product Information. *Proceedings of the 2006 IEEE International Conference on Information Acquisition*, August 20–23, 2006.

[57] P. Xiang, X. Yang and Y. Shi. Effective Page Segmentation Combining Pattern Analysis and Visual Separators for Browsing on Small Screens. *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, 2006.

[58] P. Xiang, X. Yang and Y. Shi. Web Page Segmentation Based on Gestalt Theory. *2007 IEEE International Conference on Multimedia and Expo*, 2007.

[59] X. Yang and Y. Shi. Learning Web Block Functions Using Roles of Images. *Third International Conference on Pervasive Computing and Applications*, October 6–8, 2008.

[60] L. Yi, B. Liu and X. Li. Eliminating Noisy Information in Web Pages for Data Mining. *SIGKDD'03*, August 24–27, 2003.

[61] H. Yu, J. Han and K. Chang. PEBL: Web Page Classification without Negative Examples. *IEEE Transactions on Knowledge and Data Engineering, volume 16, no.1*, January 2004.

[62] S. Yu, D. Cai, J. Wen and W. Ma. Improving Pseudo-Relevance Feedback in Web Information Retrieval Using Web Page Segmentation. *WWW'03*, May 20–24, 2003.

[63] Y. Zhai and B. Liu. Web Data Extraction Based on Partial Tree Alignment. *WWW'05*, May 10-14, 2005.

[64] S. Zheng, R. Song, J. Wen and C. L. Giles. Efficient Record-Level Wrapper Induction. *CIKM'09*, November 2–6, 2009.

[65] J. Zhu, Z. Nie, J. Wen, B. Zhang and W. Ma. Simultaneous Record Detection and Attribute Labeling in Web Data Extraction. *KDD'06*, August 20–23, 2006.