Automatic Record Extraction for the World Wide Web

by

Yuan Kui Shen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
January 20, 2006

Certified by ....
David R. Karger
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automatic Record Extraction for the World Wide Web

by

Yuan Kui Shen

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2006, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

As the amount of information on the World Wide Web grows, there is an increasing demand for software that can automatically process and extract information from web pages. Despite the fact that the underlying data on most web pages is structured, we cannot automatically process these web sites/pages as structured data. We need robust technologies that can automatically *understand* human-readable formatting and induce the underlying data structures.

In this thesis, we are focused on solving a specific facet of this general unsupervised web information extraction problem. Structured data can appear in diverse forms from lists to trees to even semi-structured graphs. However, much of the information on the web appears in a flat format we call "records". In this work, we will describe a system, MURIEL, that uses supervised and unsupervised learning techniques to effectively extract records from webpages.

Thesis Supervisor: David R. Karger
Title: Professor

Acknowledgments

First and foremost, I would like to thank my advisor, Professor David Karger, for his insightful feedback, constructive criticism, and inspirational ideas.

I am very grateful to my officemates: Alex Gruenstein, Ali Mohammad, and Chen Hui Ong for dedicating their time to proof reading my thesis and providing constructive criticism. Without your comments and insights some of the ideas in this thesis would not have come to fruition. I would also like to thank Harr Chen, Federico Mora, and the many "regulars" of 32-G494 (Jeremy, Meg, Jelani, Gabriel, Punya, Tara and so many more to list all here) for being considerate, engaging, and a constant source of humor and "good" distraction during those especially stressful days. You guys are awesome!

I would like to thank members of the Haystack group for their helpful comments regarding my research: Andrew Hogue for initiating this research problem, Nick Matsakis for his insightful discussions on cluster evaluation and machine learning techniques, Vineet Sinha for his always helpful advice on how to work the *system*, and David Huyhn for teaching me how to write a Firefox plug-in.

I would also like to thank CSAIL, and its wonderful professors: Professor Michael Collins for being my academic advisor and for answering my machine learning related questions; Professor Jaakkola (6.876) for teaching me about machine learning; Professor Kaelbling (6.625) for teaching me about AI; these two classes were especially helpful for my research.

I am most indebted to my family. Mom, Dad, and Granddad: thank you for your concern, encouragement, and support through this difficult time.

This is dedicated to you!

Contents

1 Introduction **17**

1.1 Applications . . . 20

1.1.1 Building Better Web Browsers . . . 20

1.1.2 Applications Within Search . . . 21

1.2 Definitions and Assumptions . . . 23

1.2.1 Scope . . . 23

1.2.2 Web page representation . . . 23

1.2.3 Mapping from tag trees to Records . . . 24

1.2.4 Single versus Sibling Tree Records . . . 24

1.2.5 Page-sized Records . . . 25

1.2.6 Terminology . . . 26

1.3 Basic System Outline and Assumptions . . . 26

1.3.1 Pattern Detection . . . 27

1.3.2 Record Detection . . . 29

1.4 Organization . . . 30

2 Related Work **33**

2.1 Wrapper Induction . . . 33

2.2 User Friendly Wrapper Induction . . . 34

2.3 Unsupervised Record Induction . . . 34

2.3.1 Top down systems . . . 35

2.4 Tree Models . . . 39

3 Pattern Detection **41**

3.1 Clustering Metrics . . . 43

3.1.1 Conventions and Notation 44
3.1.2 Structural Characteristics 45
3.1.3 Content Based Features 51
3.1.4 Contextual Features 52
3.2 Learning Pair Classifiers 53
3.2.1 Training Data 53
3.2.2 Features 55
3.2.3 Classification Algorithms 58
3.3 Evaluating Pair Classifiers 58
3.3.1 Evaluation Metrics 58
3.3.2 Comparison of Classifiers 59
3.3.3 Comparison of Features (Independently) 60
3.3.4 Adjusting Tree Matching parameters 61
3.3.5 TF/IDF versus Frequency Vector Weights 65
3.4 Feature Subset Selection: finding the optimal feature subset 66
3.4.1 Feature Subset Evaluation 66
3.4.2 Exhaustive Search 66
3.4.3 Correlation-based Feature Selection (CFS) 68
3.5 Clustering 69
3.5.1 Prefiltering 69
3.5.2 Distance Matrix 69
3.5.3 Clustering Criteria 70
3.6 Evaluation of Clustering Quality 71
3.6.1 Evaluation Metrics 71
3.6.2 Evaluation and results 72
3.7 Cluster Post-processing 76
3.7.1 Singletons Filter 76
3.7.2 Overlapping Trees 77
3.7.3 Tree Normalization 77
3.7.4 A Greedy Algorithm For Resolving Overlapping Conflicts 79
3.7.5 Re-merging Clusters 79
3.7.6 Post-clustering processing results 80

3.8 Performance Optimizations . . . 81
3.8.1 Tree Alignment Algorithm Optimizations . . . 81
3.8.2 Reducing the number of comparisons . . . 82
4 Record Detection **85**
4.1 Types of non-record patterns . . . 86
4.2 Contiguity . . . 90
4.2.1 Algorithm for computing content distance . . . 92
4.2.2 Comparison of Contiguity-Based Metrics . . . 93
4.3 Variation . . . 93
4.3.1 What Is Variation Defined Over? . . . 96
4.3.2 Analysis of Variation . . . 96
4.3.3 Periodicity . . . 99
4.3.4 Compression . . . 100
4.3.5 Mean Pair-wise Dissimilarity . . . 101
4.3.6 Defining The Sequence . . . 101
4.3.7 A Technicality Involving Piped Trees . . . 102
4.3.8 Variation Difference . . . 103
4.3.9 Distribution of Variation Features . . . 103
4.4 Other Features . . . 105
4.5 Cluster Learning . . . 105
4.5.1 Training Data . . . 106
4.6 Evaluation . . . 106
4.6.1 Individual Features Performance . . . 107
4.6.2 Comparison of Classifiers . . . 109
4.6.3 Feature subset selection . . . 109
5 System Evaluation **113**
5.1 Testing the tree selection algorithm . . . 113
5.2 Comparison of Omini versus MURIEL . . . 115
5.2.1 Failure analysis of Omini . . . 118
5.3 Evaluating MURIEL . . . 119
5.3.1 Failure analysis of MURIEL . . . 120

5.4 Summary 123

6 Conclusion **125**

6.1 Contributions 125

6.2 Near-Term Future Work 126

6.3 Future Work 130

6.4 Final words 131

A Appendix **133**

A.1 Overview of the classifiers 133

A.2 A Comparison of Clustering Algorithms 134

A.3 URLs 136

A.4 Classifier Internals 136

B Tables **145**

B.1 Pattern Detection 145

B.2 Record Detection 145

List of Figures

1-1 A website in a foreign language 18

1-2 An example of an iterative template. In this example, the code is iterating over an array of records: $DATA; the field values in each data record fill holes in the HTML template (the `echo` statements). 20

1-3 Single-tree record 24

1-4 Sibling-tree record 25

1-5 Example of three types of patterns on the CSAIL PI page 30

3-1 A hypothetical example of a good versus a poor pairwise metric 43

3-2 An example tag subtree 48

3-3 Labeling tool, screenshot 54

3-4 Histograms showing the distributions of positive and negative data points under various features. (Note: some features show a 1.0 bucket because of numerical precision; some feature values can be slightly larger than 1.0 due to numerical precision errors. All 1.0 values should properly be in the 0.95 (0.95-1.0] bucket.) 57

3-5 A comparison of different classifiers with all pairwise features 59

3-6 Performance comparison of individual features with an ADTree classifier . . 60

3-7 Tree Alignment configurations with varying w_{gap} 62

3-8 Distribution of Tree Alignment pairs (of the training data) with varying w_{gap} 63

3-9 Tree Alignment feature error rate as a function of varying w_{gap} 64

3-10 Comparison of frequency versus TFIDF vector weighting for BOT, PCP and PCCT. 65

3-11 Training/test errors for all pairwise features versus the optimal 3-feature subset. Note that there is an absolute decrease in the overall test error and a decrease in the training to test error ratio. 68
3-12 Average maximum f-Score, precision, and recall for values for different clustering criteria 73
3-13 Distribution of clusters by precision and recall. 74
3-14 Pair counts (Mean) of *associated record classes* of clusters distributed by the cluster f-score. 75
3-15 Ratio of misclassified pairs (Mean) of the *associated record classes* of clusters as a function of cluster f-score. 76
3-16 Piped Trees, Supernodes, and Most-Specific-Tree Normalization 78

4-1 Distribution of clusters produced by pattern detection as a function of cluster size 85
4-2 Examples of different types of clusters. (Cluster instances are highlighted in yellow.) 87
4-3 Examples of different types of clusters (Continued). (Cluster instances are highlighted in yellow.) 88
4-4 Examples of different types of clusters (Continued). (Cluster instances are highlighted in yellow.) 89
4-5 A tag tree with content numbering, the [*] number in leaf nodes denote the underlying content size. 93
4-6 Examples of different types of clusters (highlighted in yellow). 94
4-7 A tag tree with record collection, records and fields. Different shapes denote differences in formatting. 95
4-8 An example piped tree. All sub-trees that are in piped configurations (that are in the grey region) will have the same variation measure value as the nearest branching point or leaf node. 102
4-9 Comparison of variation metrics 104
4-10 Comparison of individual cluster features using a linear SVM classifier, sorted by test error 108
4-11 Comparison of different classifiers using all cluster features. 110

5-1 Evaluation of MURIEL looking at the top-5 returned clusters. 119
5-2 An example of MURIEL returning a navigational type record. The instances are highlighted in yellow. 121
5-3 Two record like regions. Boxed in red are two instances of a cluster that MURIEL mistaken as a record cluster. 122
5-4 An strange case of hidden records. Boxed in red is a hidden "record" instance that appears only when the menu item is moused-over. 123

A-1 Comparison of K-Means and HAC clustering algorithms. Reported results are over a set of 20 test pages. 135
A-2 List of URLs of labelled web pages 137
A-3 List of URLs of labelled web pages (continued) 138
A-4 List of URLs of labelled web pages (continued) 139
A-5 Comparison of tree/content based cluster metrics 140
A-6 Comparison of tree/content based cluster metrics 141
A-7 The internals of baseline pair-classifier. 142
A-8 The internals of baseline cluster-classifier. 143
A-9 A decision tree constructed using 30 features using the J48 classifier implementation (from Weka). Note that only about 10 features actually make it into the tree, and not all of them need to be any of the top features. 144

List of Tables

3.1 Best performing features subsets using different evaluation criterion 67

3.2 Test error, test f-Score for the optimal CFS-selected feature subset 68

3.3 Fraction of near perfect clustering results 74

3.4 Average maximum purity, recall and f-score of clustering results with and without post-processing. 81

4.1 A summary of expected effect of contiguity based features on various types of clusters 94

4.2 A summary of expected variation-based signatures for relevant types of patterns. 96

4.3 Test results (error, recall, precision and f-score) of training a linear SVM classifier using the various 111

5.1 An evaluation of tree selection heuristics. The values in brackets are percentages relative to the total number of instances. 115

5.2 Evaluation comparison of Omini versus MURIEL 117

5.3 Distribution of the number of clusters returned per page. 120

B.1 Error rates while varying the type of classifer. All errors are reported for all 9 pair features 145

B.2 Error rates for single features trained on a ADTree classifier 145

B.3 Error rates for two-feature combinations trained using an ADTree classifier 146

B.4 Error rates for three-feature combinations trained using an ADTree classifier 147

B.5 Clustering results for different criterion 148

B.6 Error rates for single features trained on a SVM classifier 148

B.7 Varying classifier, using all features, reporting cross-validation, training and testing errors; also with test precision, recall and fscores 148

Chapter 1

Introduction

As the amount of information on the web grows, there is an increasing demand for software that can automatically process and extract information from web pages. Many pages on the web today are dynamically generated, whereby data from a database fill templates to generate HTML. Even though the underlying data on such web pages is structured, the resulting transformation to HTML adds formatting and layout noise making the original data much harder to access by automated programs.

Web information extraction is the field of discovering automated methods for extracting information from web pages. One of the grand goals of web information extraction is to develop methods that can automatically convert human-readable formatting into machine-understandable data structures in a completely unsupervised manner. The state of the art information extraction technologies can learn from human-labelled examples to automatically extract data specific to a single type of web page. This is done by learning surface features of the page and inducing an extraction pattern specific to that page type. However, there are still no systems robust enough to generalize web data extraction to an entire gamut of web pages (thus opening the possibility of completely unsupervised extraction from new types of pages with no human labelled examples). The aim of this thesis is to demonstrate a system that strives to realize that goal.

To better illustrate the problem, imagine visiting a webpage where all the content is written in an unknown foreign language like in figure 1-1. While the content of the page is largely incomprehensible, the information layout and structure is nonetheless discernable. In the example page (figure 1-1), we can clearly determine that the page contains two

Figure 1-1: A website in a foreign language

side panels on the left and a main section on the right containing 5 (fully visible) records, with each record composed of 4 fields. This example illustrates that we can recognize the general layout of the page and understand the logical segmentation of a data presentation without knowledge or understanding of its underlying information. Such intuition is also independent of the syntax of the markup language used. The web page could be rendered from HTML, SGML, or any GUI/language and the same format recognition would exist. Therefore, neither knowledge of content nor the type of markup language affect our ability to successfully recognize the structure of the underlying information. This implies that we do not need to solve hard AI problems like natural language processing in order to solve the general problem of structure understanding. We were able to recognize that there were 5 "records" (in the example) because we notice a regularity in the reoccurrence of an icon or change in the bolding and indentation of text on the page. This recognition of regularity in content or syntax features is critical in allowing us to infer about the original data structure. If we can formalize a method for detecting salient data presentation patterns then we will be able to build systems that can induce the data structures from web pages in a generic manner.

In this thesis, we focus on solving a specific facet of this general unsupervised web information extraction problem. Structured data can appear in diverse forms from lists to trees to even semi-structured graphs. However, much of the information on the web appears in a flat format we call *records*. Our goal is to build a system that will automatically extract records from such web pages in an unsupervised setting. We define records as single units of information each describing a semantically coherent entity. When collections of record instances appear within a web page, for example as search results, directory listings, and shopping catalogues, we call such pages *record pages*. We assume that these record instances share a simple schema (or record template) of the form $R_i = (f_1 \dots f_n)$, where each f_i represents a field value and R_i represents a single tuple extracted from a database table (or query). However not all fields are always required, some may be optional, and some fields will receive more formatting emphasis than others. So the data we receive through the HTML page may be missing some fields and differing in formatting of fields. Record pages are generated by embedding record tuples with some modification into an iterative HTML template. An iterative HTML template is a piece of HTML code that contain "holes" which are filled by the field values of record instances. The template and record

```
<?php
  foreach( $DATA as $rec ) {
    echo "<tr>";
    echo "<td>";
    echo "<b>" . $rec['name'] . "</b><br>"
    echo "<font size=-1>" . $rec['street'] . "<br>";
    echo $rec['city'] . ", " . $rec['state'] . "<br>";
    echo $rec['zip'] . "<br>";
    echo "</td></tr>"
  }
?>
```

Figure 1-2: An example of an iterative template. In this example, the code is iterating over an array of records: $DATA; the field values in each data record fill holes in the HTML template (the `echo` statements).

tuples are combined to generate a sequence of HTML-embedded records. Figure 1-2 shows an example iterative template written using PHP. Our task is to *reverse* that embedding process by inducing or recovering the tuples R_i from the web page.

1.1 Applications

Where is unsupervised record induction applicable? Record induction is most useful in applications where scalability or user convenience are deciding factors.

1.1.1 Building Better Web Browsers

One possible target application is as a web browser enhancement. State-of-the-art web browsing software treats web pages largely as read-only pages: users wishing to manipulate the information on a given page are limited to the functionality provided by the web service host. The most common mechanism for exporting data from web pages is still cut-and-paste. Richer client-side data manipulation is restricted by the difficulty of extracting the data from the formatting. The browser should automatically possess some basic *understanding* of underlying formatting and be able to correctly highlight potential records. The user's role should be more passive than active, adjusting and correcting extractions only in the case of need or error.

Our tool, MURIEL (Mostly Unsupervised Record Information Extraction tooL), will demonstrate ways by which this type of software intelligence could be achieved. Once fully

integrated with a web browser, our tool will show users potential records on a page without user intervention. The tool is *mostly* unsupervised because the system still requires some training examples in order to bootstrap. The training is not per page type but at a higher level; we are teaching the system how to recognize formatted records in general rather than a particular type of record. Also, this training is one-time to enable the system to learn the proper parametizations in order to achieve accurate record recognition.

We will provide a simple browser enabled version of MURIEL as our principal application for demonstrating the algorithms and techniques described in this thesis.

1.1.2 Applications Within Search

Unsupervised record induction has the most potential in applications that need to scale or require batch processing, one such area is web search.

Improving Retrieval Granularity

Most web search engines assume that a web document represents a single concept and is the smallest unit of retrieval. However, many web pages actually represent collections of items (*e.g.* a personnel directory contains information about a collection of individuals). Many times, the user is interested in information related to a single concept instance on the page rather than the entire collection. This problem is especially apparent when search engines return large pages as results, as the users often need to perform an additional search within the retrieved page to look for the instance relevant to their query.[1] An ideal search engine should provide the user with a result at the appropriate level of granularity. *Snippeting* (a form of result summarization) is the current solution to this common problem. However, snippeting can truncate the information at the wrong places, for example cutting a record in half. With good unsupervised record induction, we would be able to know in advance the basic units of information on the page and simply retrieve and return the best matching units.

Furthermore, better information granularity can improve the precision of retrieval. For multiple word queries, results that contain query terms in close proximity are often more

[1]For example, search for "6.170" on Google. 6.170 is a software engineering course taught by MIT. One of the search results points (ranked 8th at time of writing) to http://student.mit.edu/catalog/m6a.html which a page containing a long listing of all the courses offered in the MIT EECS department, with over 50 course offerings. The search result snippet only shows the prefix of the course description; a better search result using our system would include the entire record as the snippet.

precise than those where the words are far apart; hence, phrase queries are more precise than free word queries. If we have knowledge of the containment bounds of a single unit of information, say a record, we can further restrict queries to return results that have query terms that occur within a single unit. Using a record-level notion of word proximity is more flexible than phrase queries but still restrictive enough to preserve some precision.

Object Search Engines and More Structured Queries

Being able to extract the data structure of web pages automatically is a step towards a grander goal of data consolidation and federation of data sources all over the web. Companies like Google have commercial efforts such as Froogle that try to aggregate and organize commercial entities (any sellable item) into one single searchable database. But in order to build such large scale object databases that data must first be extracted from web sources. Manual extraction methods demand enormous human effort and labor costs. With robust unsupervised record induction and a little help from data mining (to clean up the data) it would be possible to build databases containing collections of extracted objects. Instead of web page search engines, we would effectively be building web object search engines.

If we could extract fields from record objects and use extracted field values as attributes then these extracted objects can be manipulated at a richer level than free-text pages. We could formulate more structured ways to query these objects that are based on extracted attributes and attribute labels. These attribute labels and attribute values need not be associated with known ontologies or schema in order for them to be useful. For instance consider the query "producer". A free text search engine will return results that contained the word *producer* in the page content or page title; the top result may be "The Producers[2] (2005)" movie. What if the user wanted to find a *list* of objects that contained an attribute label *producer*; suppose they wanted to find a list of movies for which they can later sort by the name of the producer. Queries on attributed objects will allow both the former and latter type of results while queries on free text will only allow the former. Hence, queries on attributed objects can be more expressive than queries on free text; more restrictions can be applied to text specific to parts of an object (like the label) rather than the entire object.

Furthermore, the returned results could also be dynamically manipulated. Now that the

[2] Producers is equivalent to producer after stemming.

result set is a collection of objects with attributes, so sorting, filtering or grouping by those attributes would come naturally on the client side (via some browser extension). Users would be able to manipulate results by the fields that are interesting to them and not just the ones specified by the web service. For instance, the user could issue a command to sort objects by the attribute with label containing "year" to sort a list of objects by year.[3]

These and many more improvements to the way we access and manipulate information on the web will be possible with unsupervised record extraction.

1.2 Definitions and Assumptions

Before focusing on algorithms and methods, we will first outline our scope and state definitions and assumptions.

1.2.1 Scope

As mentioned earlier, structured data can come in numerous forms from flat-layout records to hierarchical-layout trees. On some web pages, records are not flat but appear nested: records occurring within records. Tree layouts are generalized extensions of nested records. However, to simplify our scope, we will not be covering these more complex types and only concentrate on the most common form, namely flat records. Solving generalized tree layout records could be part of potential future work.

1.2.2 Web page representation

HTML documents can be represented as tag trees where each node in the tree represents an opening/closing tag in the corresponding web page. Tag trees are commonly implemented as DOM (Document Object Model) trees. DOM parsers are freely available through most browser APIs and provide easy construction of tag tree representations from HTML. We designed our algorithms to be free of HTML language specificity so they will work over any pages rendered from a DOM-compatible GUI markup language (*e.g.* HTML, XML/XSLT). For these reasons, we choose DOM-based tag trees as our basic representation of web pages.

[3]These structured queries may appear too technical for the average web user to handle but we believe this is a problem that can be solved by having a better or more natural user interface. It is even possible that we allow natural language queries that are translated into structure queries using technology from question answering systems like START [29][30].

1.2.3 Mapping from tag trees to Records

We assume that a record instance maps to a continuous content-containing region on a web page. This implies that we expect all the fields within a tuple $(f_1, \ldots f_n)$ to be locate within the same region on the page. While it is very possible that fields of a record are scattered through out a page, we believe that these cases are rare and we will ignore them to constrain model simplicity. By assuming continuity within each record instance, we constrain record instances to map to either a single subtree or a set of adjacent subtrees in the tag tree.[4]

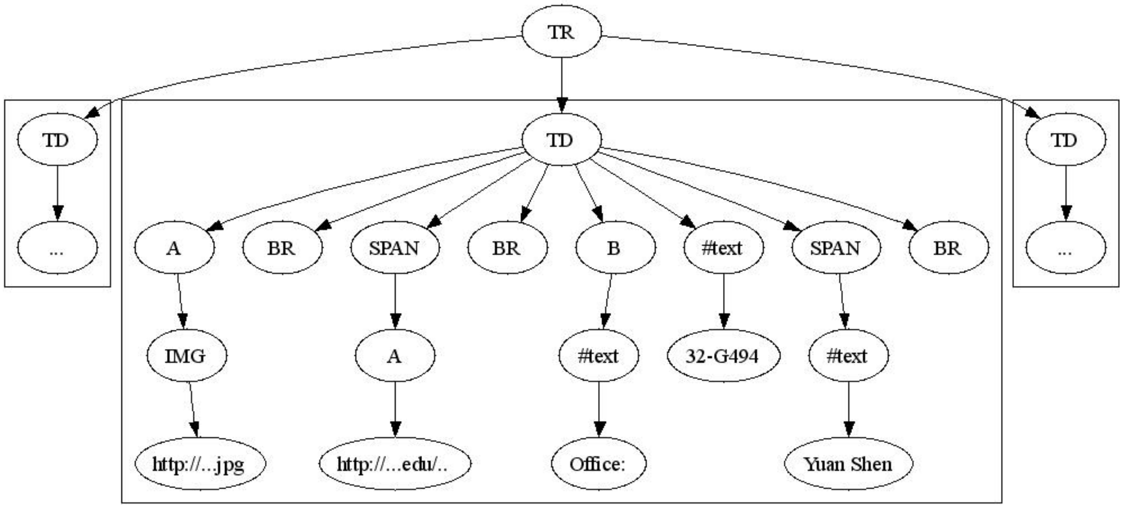


Figure 1-3: Single-tree record

1.2.4 Single versus Sibling Tree Records

A *single-tree record* is a record that has a one-to-one mapping with a single tag subtree. Figure 1-3 shows a single-tree record as represented by a single subtree with a `TD`-root node. The Google search result page[5] and the CSAIL Principal Investigators page[6] contain good examples of these single-tree records. A *sibling-tree record* is a record that spans a contiguous set of subtrees. It does not qualify as a single-record tree under its parent because some of its siblings are NOT part of the same record instance. Figure 1-4 shows two sibling tree

[4]We confine our definition of continuity and adjacency to static DOM trees. With CSS (Cascading style sheets), one may place two *non-adjacent* DOM trees next to each other by using absolute positioning. To account for the possibility of dynamic positioning one will need actual layout information from a browser. However, our general methods will still apply only our definitions of adjacency and continuity will need to be redefined to reflect actual layout information.

[5]http://www.google.com/search?hl=en&q=arts&btnG=Google+Search

[6]http://www.csail.mit.edu/biographies/PI/biolist.php

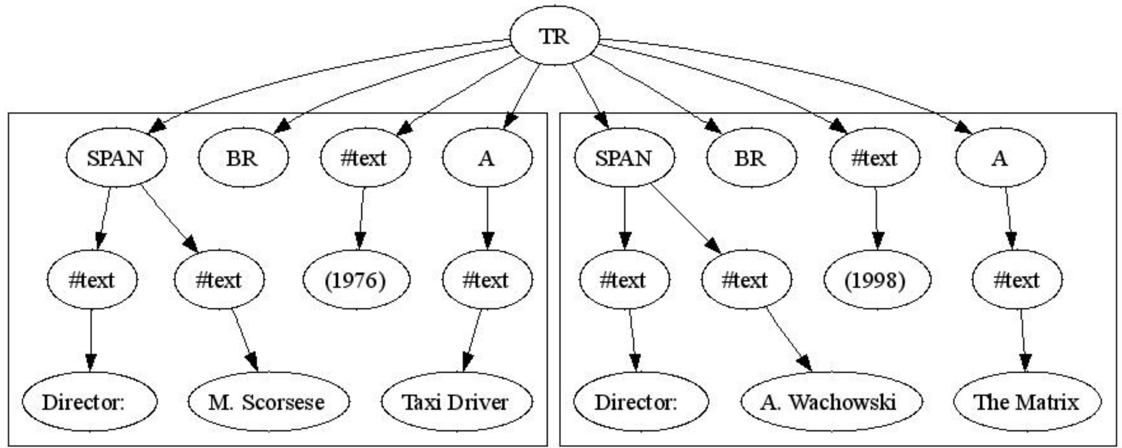


Figure 1-4: Sibling-tree record

records each denoted as a set of four consecutive sibling subtrees. The Slashdot[7] news site has examples of sibling-tree record layouts. Each article on Slashdot is comprised of a set of adjacent subtrees: a header subtree, an article information subtree, an article summary subtree, and finally a "read more" subtree. Sibling-tree records are more difficult to detect. We will need to perform additional analysis to find a mapping from sibling trees to record instances by searching for potential "virtual" nodes to which we can attach subtree sets.

1.2.5 Page-sized Records

A record instance may also be mapped to an entire page or even a set of web pages. IMDB.com contains examples of single page records where an entire page represents a single record instance of a movie or actor. A record can also consist of a set of web pages. For instance, a TV show record on tv.com[8] is composed of a collection of web pages containing cast/crew information, an episode guide, and related news.

Single page or multiple page records are large-scale cases of the single-tree or sibling-tree problems. A plausible solution would be to construct a giant site-wide tag tree by using hyperlinks as connecting branches between page-level tag trees. We could then perform the page-level tag tree analysis on the large site-wide tree.

Page-sized records can be extracted using the same types of techniques used in solving single tree and sibling-trees cases. Therefore, for the remainder of this thesis we will focus

[7]http://www.slashdot.org/

[8]http://www.tv.com/family-guy/show/348/summary.html

mainly on the simplest case: record detection for records that are single-tree instances and whose instances are located entirely within a single page.

1.2.6 Terminology

Next, we define a set of terminology specifically for referring to groups of trees.

- A *pattern* is a collection of trees that share some set of common features. (What these features are will be defined in detail later.)
- A *record class* is a collection of known or labelled record instances where there is a known mapping between record instances and subtrees. A record template is the iterative template that was used to produce the associated record instances of a record class.
- A *record pattern* is a collection of trees that the system identifies as being associated with a record class.

The major difference between record class and record pattern is simply that the former is known to be true as judged by humans and the latter is what our system produces.

1.3 Basic System Outline and Assumptions

The input to MURIEL is a tag tree representation of a web page; the output is a set of record patterns. The goal of our system is to find the set of record patterns that best matches the true record classes. The problem is difficult mainly because the parameters of the source record classes are unknown, namely: the number of instances (size), location, or even the number of classes.[9] Finding these parameters can be made easier with some simplifying assumptions:

We assume:

1. A record-page contains one or more record classes.
2. A record class contains *two* or more record instances.
3. Trees associated with the same record class are similar under a specific set of features.

[9] When the parameters of record classes are given then the problem becomes solvable using wrapper induction (see related work).

4. Trees associated with any two different record classes are different (and are distinguishable) from each other under those same features.

5. Groupings of trees based on similarity as defined by this above set of features form patterns

6. Not all patterns are associated with record classes.

7. There exist features over patterns that can distinguish record patterns from *non-record* ones.

The first assumption makes explicit that we expect more than one type of record per page. Assumption 2 restricts that only patterns with more than one member are significant. Record classes that contain only one record instance per page will not be recognized by our system.[10]

Assumptions 2-7 set up the general framework of our problem, breaking it into two convenient sub-problems. First, assumptions 3-5 tell us that trees, if part of a record class, group together in a specific way based on similarity features. To uncover these natural groupings we must discover what those similarity-based features are. We call this subproblem *pattern detection*.

Assumptions 6 and 7 form the basis of our second sub-problem. Namely, that there are non-record patterns as well as record patterns. So we must find features that most effectively separate the two types. We call this second step *record pattern detection* or just *record detection*.

1.3.1 Pattern Detection

In the first phase, our system must find groupings of trees that would most likely map to record classes. We use clustering as a general method for discovering these optimal patterns. Clustering is the set of general unsupervised learning techniques that is used for discovering hidden relationships (patterns) within unlabelled data (trees). We considered it as a way of converting our feature space from individual trees to collections of trees

[10] Record classes that repeat over multiple pages of the same type but do not repeat within a single page could still be recognized. By extending our system to analyze multiple same-type web pages simultaneously we would be able to recognize single-instance-per-page type record classes as well.

(patterns). Simpler methods like sorting or bucketing based on attributes are also possible alternative techniques, but clustering is in general a superset of those techniques.

To cluster, we first need to define a clustering criterion. For that, we need to define what similarity means. For our problem, similarity can be measured in terms of three general categories of features: structural, content-based, and contextual.

Structural features are features based on the graphical structure of any two subtrees being compared. Depending on the assumptions made about the importance of nodes and edges in the tree, we can get different measures of structural similarity.

Content-based similarities are similarities based on the text (and images) contained under trees. Measures of content-based similarity can range from approximate measures such as the difference in the length of text, to more precise measures such as the string edit distance of the underlying texts.[11]

Finally, contextual similarities are measures of the location and the environment of trees. Are two subtrees positioned near each other on the page? Do they share the same ancestors? The distance between two subtrees can be approximated by the amount of content interleaved between them. We can also measure the commonality between the ancestors of two subtrees by measuring the edit distance of their tag-paths.[12] In chapter 3, we will give a comprehensive analysis and definition of these and many other similarity measures.

On some web pages, structural similarity may be the deciding feature for determining which trees form clusters, whereas on other pages contextual-based similarity may play a stronger role. We can find an optimal combination of these similarity features that will work best with all types of pages. Our clustering metric $\phi(T_i, T_j)$ is some combination of m pairwise feature functions: $\phi_i(T_i, T_j)$. Our target learning function is thus:

$$\phi(T_i, T_j) : \langle \phi_1(T_i, T_j), \ldots, \phi_m(T_i, T_j) \rangle \rightarrow \{0, 1\}$$

$\phi(T_i, T_j)$ is our combined pairwise clustering metric; it is equal to 1 for pairs of trees that should belong in the same cluster, and 0 for pairs that should not be in the same cluster.

We use supervised learning (classification) to optimize the model parametizations. Our

[11] Length of text ignores the actual underlying text and therefore is more approximate. Edit distance considers the number of changes to transform one string to another; it is a measure that depends on both the text and character position in the strings and therefore is much more precise.

[12] A tag-path of a tree is the sequence of the tags starting from the root of the document to the given tree

training examples are derived from labeled web pages. Each training example is derived from a pair of trees; it is composed of an m-dimensional vector of component features values, and an output value of 1 if the tree pair is part of the same record class, and 0 otherwise. In chapter 3, we will discuss which features work and which do not. We will also detail our experiments and evaluate methods used for learning the model parameters.

1.3.2 Record Detection

Once we have patterns, we want to identify which patterns are record patterns. We want to learn some function over the space of patterns that will best separate the record patterns from the non-record patterns.

To do record detection well, we must know what types of non-record patterns exist and what features best differentiate them from record patterns. To get a sense of the types of non-record patterns that exist, let us consider the example of the CSAIL Principal Investigators page[13] (see figure 1-5). The main content area of the page contains a large table. Contained within each cell of that table is a short descriptive record about each PI; this is our target record. Pattern detection returns three types of patterns within this table region: sets of trees positioned at the table cells, sets of trees positioned at the table rows, and sets of trees positioned over the PI names. The table-cell-based trees denote the correct record instances. The table-row-based trees correspond to collections of record regions. The PI-name-based trees denote the fields within the desired records. Here we have three patterns that are related by containment, of which only one represents a record pattern.

Field patterns are generally non-continuous, *i.e.* there is usually content occurring between instances of field patterns. In contrast, record patterns are generally continuous, and usually have no gaps between instances.[14]

Record collections are patterns that aggregate true record instances (each instance denotes a set of records). Because the children of record collection subtrees are records, they

[13] http://www.csail.mit.edu/biographies/PI/biolist.php

[14] One may argue that record patterns need not be continuous and could be scattered on the page. We view that contiguity is necessary to maintain content consistency and flow within a page. A record page where record instances are laid out together is more readable than one with a scattered organization. Readability and consistency are central principles of good web design. Therefore under general design principles records tend not and should not to be scattered (in layout). Certain layouts for pages such as calendars require record instances (events) be scattered because of specific layouts required by monthly, yearly or daily views. These are, however, exceptions rather than the trend.

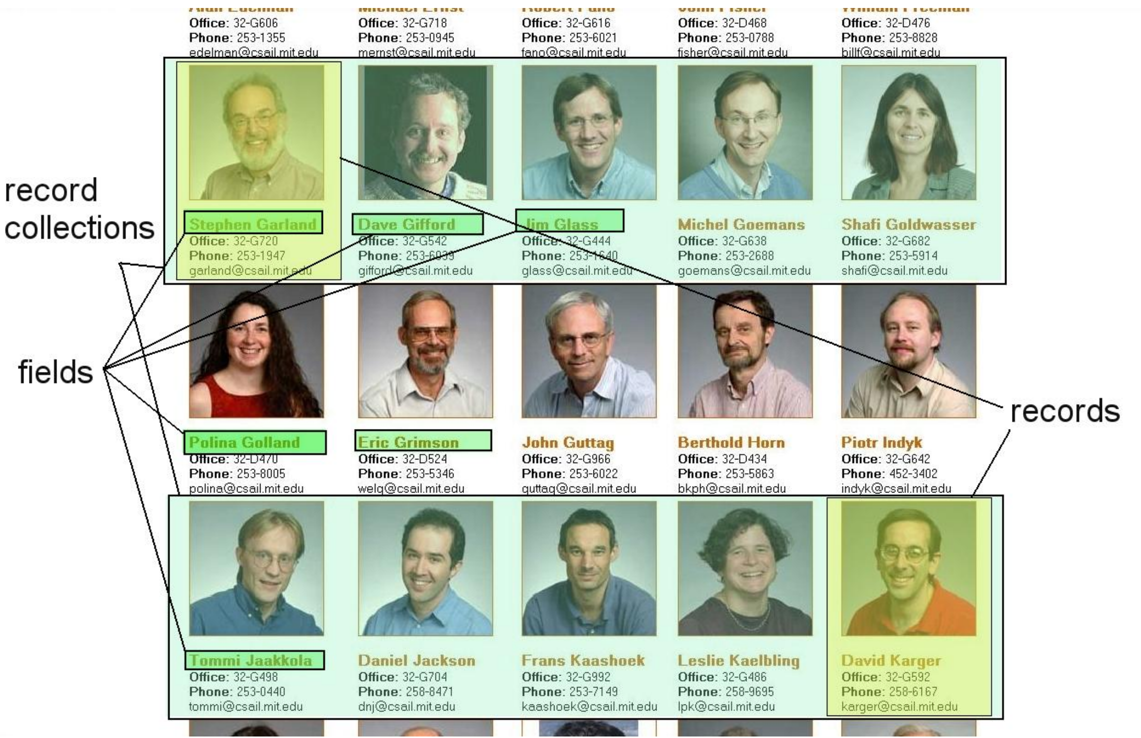


Figure 1-5: Example of three types of patterns on the CSAIL PI page

generally have homogenous internal structure. On the other hand, true record subtrees contain fields as children but fields are usually more varied in formatting; so record subtrees have heterogeneous internal structure. This difference in internal structure variation can be leveraged as a measure for distinguishing record collections from individual records.

As these examples show, there are many types of non-record patterns. Features like continuity and internal structural variation are just some of the features that can distinguish record patterns from non-record patterns. We will visit these and many other features in detail in chapter 4. Just as in pattern detection, we use supervised learning to find an optimal combination of record detection features.

1.4 Organization

In chapter 2, we will provide a summary of related work. In chapter 3, we will give details on building a pattern detector. In chapter 4, we will describe the details of building a record detector. In chapter 5, we will put the two parts together. We will perform a system evaluation and compare our system with another record induction system. In our concluding chapter (chapter 6), we will summarize our results and contributions, and list

possible future work.

Chapter 2

Related Work

The goal of web data extraction has been to better methods for transforming data in web pages into machine accessible forms. The direction of the state of art has been towards more generalizability and more automation. Many of the earlier solutions were largely specialized for single types of web pages. More recently, the research has shifted towards more generalizable solutions that do structured extraction over all (or a large set of) web page types.

2.1 Wrapper Induction

Wrapper induction (WI) is the process of learning pattern extractors (scrapers) for web pages. It removes the need for manual examination of HTML by learning the pattern extractors from examples. Most wrapper induction algorithms begin by defining a model representing possible extraction patterns. A model could be a finite state machine, a rule with wildcards, or variations thereof. Individual instances of records are provided as training examples usually via a specialized labelling tool. A learning algorithm is used to find the most likely model parameters given the training instances. Kushmeric *et al* [31] used a model with four feature slots: HLRT, with (regular expression) pattern matchers for the **H**ead **L**eft **R**ight, and **T**ail of a record sequence. STALKER [37] used a model consisting of a k state *landmark* automaton where each state described a specific field in the record k tuple. Hsu *et al* [24] used *soft* finite state transducers as their model. There are also more general information extraction (IE) systems such as RAPIER [7] and WHISK [45] that operate on more unstructured text data. These systems use models consisting of frames

with pattern matching slots. The accuracy of these myriad IE or WI systems differs largely due to the expressiveness of the underlying model over the target data.

2.2 User Friendly Wrapper Induction

The work of Hogue *et al* [23] focused on building an end-user-friendly wrapper induction system that reduced the amount of training examples required. Unlike previous WI systems where patterns were learned over flat HTML, Hogue's wrapper induction model leveraged the hierarchical structure of HTML and learned patterns over the space of HTML parse trees (tag trees). Their system, Thresher, constructed tree patterns by aligning multiple example trees using a tree edit distance algorithm. The resulting system was able learn the record extraction patterns with only a few user-highlighted examples.

While wrapper induction technology has been in existence for several years, there is still no "killer" application that widely makes available this technology to the mainstream web user. We believe that one of the reasons may be the overhead of training: it is tedious for the user to have to label every web page/site she visits. Wrapper induction technology is also unsuitable for batch processing applications because of the need for manual intervention on each page type.

2.3 Unsupervised Record Induction

Unsupervised record induction is the task of discovering and identifying records (objects) on a web page without explicit user provided examples. It has also been referred to in earlier works as automatic wrapper inference, automatic object extraction or automatic wrapper induction. We use the name unsupervised record induction to emphasize that our method is *unsupervised* per page type and is focused on extracting only record type objects. First we will describe some of the earlier unsupervised record induction systems.

Jiang *et al* [20] built a system for identifying record separators from record-containing web pages. Record separators are the HTML tags that are the most likely delimiters between records in a record-rich region of a page. Their technique combined five tag-identification heuristics: standard deviation of text size between potential tags, occurrence of common record separating tags, matching of the text with a known ontology, repetition of tags, and repetition of tag bi-grams.

Buttler *et al* built Omini [5], a system that used a similar set of heuristics and introduced one more heuristic: a partial path heuristic (which looked for common sub-sequences of tags). Both Jiang and Buttler used a combination of heuristics as the final algorithm. In both systems, each heuristic was assigned some confidence (probability) for generating the top n ranking record separators. These confidence probabilities were estimated from a labelled data set. The algorithm combined the individual confidences $P(Tag_{rank}|H_i)$ by computing a combined confidence $P(Tag_{rank}|H_1 \cup \ldots H_5)$[1] and returned tags with the maximum combined confidence. One can think of their approach as a form of learning using a weighted combination of features.[2] Both systems, reported surprisingly high levels of accuracy when the union of all 5 heuristics was used, 100% [20] and 98% [5] respectively. However we believe that these numbers may only be a reflection of the given training data. Our own evaluation of these systems shows that the high accuracy does not extend to a wider (or different) test data set.

2.3.1 Top down systems

Abstractly, these earlier record induction systems can be thought of as being composed of two subsystems: a *tree selection* subsystem and a *record segmentation* subsystem.

1. The *tree selection* subsystem tries to find a *record-packed tree*: a tree that spans the entire region occupied by record instances. The key assumption is that all instances of a single record class live under the record-packed tree. Also, there is no *non-record* content occupying any space under the record-packed tree. For tree selection, Jiang *et al* used a simple "largest fan out" heuristic to find record-packed trees. Buttler *et al* used a more sophisticated combination heuristic that looked at features such as tree fan-out, enclosing content size, and overall sub-tree size.

2. The *record segmentation* subsystem tries to segment the content of a given record-packing tree into a sequence of record objects. The segments are contiguous regions under the record-packed tree broken up by record separator tags.

We call these systems *top-down* approaches because they search for potential records by first looking for a likely record containing region. In contrast, we call our approach *bottom-up.*

[1] Using the standard definitions for a union of events: $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.

[2] Although, their writings do not mention any machine learning formalisms.

We first assume a general model for each record (with the assumption that each record corresponds to some rooted subtree). Then, we cluster candidate models (subtrees) into groups, and finally apply filters over those groups to arrive at a most likely set of records. We make no initial assumptions about the location or layout of the records.

The major differences between the two types of systems lie in the underlying assumptions. Top-down systems assume:

- There is a record-packed tree under which *all* and *only* record instances live.
- This record-packed tree can be found using heuristics.
- Record instances are always contiguous to one another (*i.e.* one instance follows another without gaps)
- The boundary of record instances is defined by a single HTML tag.

Our system assumes:

- There may be more than one record class per page.
- Record patterns are defined by degrees of similarity (structural, content-based, and contextual).
- Record trees can be constrained by contiguity but are not defined by it.
- The bounds of a record instance is defined by a tree (or sequence of trees).

These basic assumptions factor greatly in how accurate and generalizable these systems can be in extracting records. Our general criticism of top-down systems is that their assumptions are overly restrictive. The top-down assumptions tend to break down when given more general web page inputs.

Next we will outline our four major criticisms of top-down systems. We will follow each criticism by explaining how our approach addresses these problems.

1. *Ambiguity of Regions/Boundaries*

    - Top-down systems assume that record instances are separated by HTML tag separators. First, *tags* may not be good record separators because they are often

ambiguous. The same HTML tags may serve different functions when used in different contexts.

Single tag boundaries are sensitive to small changes in formatting. Web pages may use different tags to indicate indentation or emphasis amongst different records. For example, the Google search result pages use an alternation of `BLOCKQUOTE` (for indented record) and `P` (for normal records) as record separators. On such pages, the top-down system will fail to mark the indented record instances (by picking `P` as record separator).

Single tag boundaries assume that records instances are consecutive and non-record content do not occur between, before or after record instances. This is true for many web page types but definitely not always true. A common practice in web design is to place "spacer" images between record instances. These spacers serve the purpose of adding fixed amount of blank spacing between record instances. Top-down systems will extract these spacers as records because of their contiguity of records assumption.

- Our system models records as regions enclosed by a tree (or sequence of trees). Using a tree (or trees) to model a region defines not only that region's boundaries but also what structures or content are present within that region.

  Tree-based models are more robust against tag alternation. Because we constrain record patterns to be groups of trees with structural similarities, two record trees could still retain structurally similarity even if a small portion of their node labels differed, such as in the tag alternation case.

  Tree-based models are also robust with regard to spacers. Spacers and record trees do not (in general) share structural similarities; therefore spacers will not cluster with record trees to form record patterns.

2. *Not Comprehensive*

   - Top down systems assume one record class per record page. This is certainly not true of all record pages. Search result pages for instance often contain multiple record classes. Some denote news items; while others denote relevant commercial listings.[3]

[3] http://search.yahoo.com/search?p=flower contains three record classes.

- Our approach is more comprehensive: we search and analyze the entire page while looking for record patterns. During our pattern detection phase, we compare all tree pairings, which allows our system to find multiple record patterns if they are present. While a whole-page analysis may seem computationally expensive, we will show several optimizations that can garner an acceptable running time while not sacrificing result quality.

3. *Tree Selection Errors*

   - The tree selection subsystem is major source of errors in top-down systems. In chapter 5 we report our own study on the effectiveness of these tree selection algorithms. We found that max-fan-out heuristic was only around 30% effective in finding the record packing trees (amongst our test pages). For Omini, the combination heuristic performed better at around 75% accuracy. However, that still amounted to over a quarter of the pages having failed due to tree selection error. Also, a significant fraction of failure cases where due to tree selection subsystem finding only *record-containing trees* and not record-packed ones. In those cases, the tree was too general, too far up in the tag tree.[4] The tree selection subsystem error leads to record segmentation failure because the latter assumes that records instances are always packed under a single tree.
   - Our system does not make *assumptions* about where records are located within the page. Our framework analyzes every subtree and assumes that any tree (or trees) could be a potential record. We use contiguity as one of the many features in record detection. However, the contiguity feature only *constrains* our decision on what should and should not be a record, it does not define it. Our solution is formulated around a general machine learning framework that combines multiple features. Therefore, no single feature dominates the overall accuracy of the system.

4. *Lack of "Focus"*

   - A top-down system often fails for a specific type of record-pages, namely those containing record collections. Record collections occur when individual records

[4]For example, choosing the page root would be choosing a record-containing tree but not a record-packed tree.

are grouped together into collections due to layout constraints. As an example, our canonical CSAIL principal investigator page contains records in cells which are in turn organized into rows. The rows of the table are record collections. Failures on these pages occur largely because the features used by these top-down systems do not account for the level of "focus" (whether at rows or the cells of the table) at which records occur.

- Our system accounts for the level in which records occur by considering features that measure the variation inside a record instance relative to its outside. These are the variation-based features that we will discuss in detail in chapter 4. Our variation-based features allow us to better account for the level at which to find record instances. Better "focus" means better ability to discriminate record patterns from record collection patterns.

In summary, our approach provides a more comprehensive search of records, a better and more precise definition of regions, and a better accountability for "focus" than top down systems. Furthermore, we avoid relying on fragile heuristics to find record instances; instead, we use similarities of structure to constrain and group trees into record patterns. In chapter 5, we will evaluate both our system and Omini to provide further support of our claims.

2.4 Tree Models

In our system, we model records as tag trees or DOM trees. Using DOM trees as representation for web pages is a recent trend in web information related applications. Reis *et al* [8] and Hogue *et al* [23] used tree edit distance to find records within web pages. Ramaswamy *et al* [44] used a unique shingles-based algorithm for determining similarities of subtrees within a page and demonstrated the usefulness of such algorithms for segment-based web page caching. No work to our knowledge thus far has used a tree representation in the problem of unsupervised record induction. Previous work has focused exclusively on surface features (html text patterns) or tree statistics such as tree size or tag frequency.

Chapter 3

Pattern Detection

If someone showed you a single snippet of HTML and asked you to identify it, you would not likely immediately identify it as a record snippet. However if some number of the same type of snippets were shown, you might start to see a pattern and declare after a while that the snippets form the basis of a record class. The purpose of this simple thought experiment is to show that a record class is a definition that emerges out of an aggregation of instances. Our goal is to find record classes, so logically it follows that to search for them we should examine the aggregate characteristics of tag trees and not those of single tag trees. We defined a *pattern* as an aggregation of trees that share some common set of characteristics. But what are these common characteristics? How do we derive patterns from such characteristics? In this chapter, we will examine in detail the process of converting a set of individual trees into a set of *patterns* in a process we call pattern detection. Along the way, we will provide the answer to many of these questions.

In a nutshell, the mechanics of pattern detection is actually quite simple: we apply clustering over the set of the subtrees of the page. Clustering groups trees that share common characteristics to form patterns. From those patterns we can then search for record patterns (chapter 4). While the mechanics are simple, what is difficult is coming up with the right features with which to cluster, and finding the optimal method for deriving these clusters. Clustering is an unsupervised machine learning technique that has been extensively studied; excellent surveys of this area can be found in Berkin *et al* [2], Duda *et al* [19] and Jain *et al* [26]. Formally, clustering assumes that we have some data (in our case trees) that were generated by some number of hidden classes (record classes); the

data instances which were generated by the same class exhibit similar characteristics. A clustering procedure operates over the input data and outputs a set of clusters. Different clustering procedures often produce very different clusterings. For instance, some procedures such as classic K-Means tend to form clusters that exhibit equal variance. For our problem, we want the opposite, we are specifically interested in minimizing the variance, such that clusters of items are as "same" or tight as possible. For our purposes, we chose hierarchical agglomerative clustering or HAC. We discuss why we chose HAC in detail in section A.2.

The key to using clustering successfully is defining the proper clustering metric. A clustering metric quantifies the relationship between data instances (in our case HTML tag trees). Without loss of generality we define the following generic functional form as our clustering metric: a function over pairs of trees:

$$\phi(T_i, T_j) : \langle \phi_1(T_i, T_j), \ldots \phi_n(T_i, T_j) \rangle \rightarrow \{0, 1\} \tag{3.1}$$

$\phi(T_i, T_j)$ is a decision function that maps a vector of several primitive clustering metrics ϕ_i (each a real valued function over pairs of trees) into a binary yes or no answer: 1 (yes) when the input tree pair belongs in the same cluster, and 0 (no) if they belong in different clusters. It is no coincidence that the formulation for our distance function has the flavor of a supervised classification problem. Instead of finding the proper weights of the n different $\phi_i(T_i, T_j)$ components or even the function form of $\phi(T_i, T_j)$, we can use supervised machine learning to *learn* the weights and functional form.

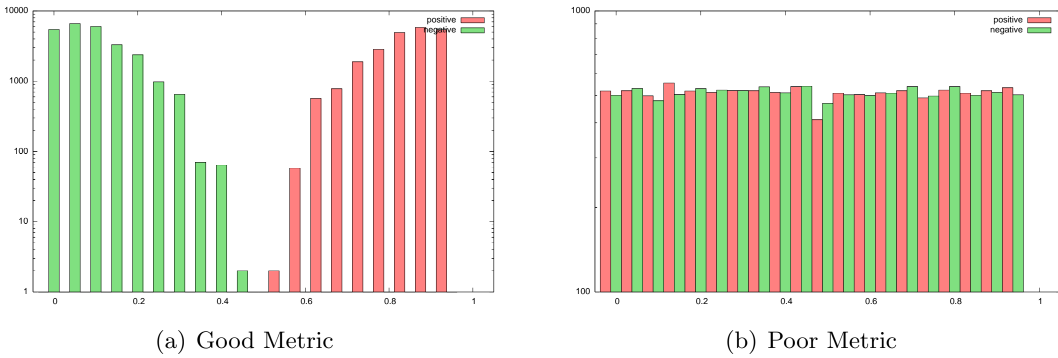
Why use a pairwise function? Why not a triplet or k-set function? Specifying which pairs are in the same cluster is sufficient to completely define the clustering; higher order specifications are not necessary (or will simply overfit the data). Furthermore, most of the relationships between trees that we work with are defined only between pairs of trees such as tree edit distance (section 3.1.2); they are undefined for multiple instances.

Defining a "useful" ϕ function is the central theme of this chapter. To do that we need to first consider what are reasonable primitive clustering metrics (section 3.1) and how to combine them (section 3.2). We also need to know how to properly evaluate our clustering results to determine how useful they are, and what parameters affect their quality (section 3.6). Finally, while accuracy is our main goal, speed and memory performance should not be neglected as they affect the usability of our system; in the last section 3.8 we describe

practical methods for optimizing the running time.

3.1 Clustering Metrics

What is a good clustering metric? Our clustering metric assumes that there are two types of pairs in the universe. Pairs that belong together are linked pairs, and pairs that do not are unlinked pairs. Ideally, a good metric would be a function that produces a distribution over pairs such that linked pairs are clearly distinguishable from unlinked pairs. If we plotted the distribution of metric values of tree pairs as a histogram and labelled linked and unlinked pairs with different colors, a good metric would look something like figure 3-1(a) while a poor metric would resemble something like figure 3-1(b).



(a) Good Metric (b) Poor Metric

Figure 3-1: A hypothetical example of a good versus a poor pairwise metric

Good metrics will distribute the linked (positive) and unlinked (negative) pairs into two distinct modes while poor metrics will uniformly distribute them. In the former we can easily distinguish the two types, but in the latter we can not separate the two types (using a simple linear boundary) without making many errors. Of course, sometimes poor linear separatability implies that we need to look to higher dimensional separators. In those cases we will need to perform additional analysis to determine whether the higher dimensionality is necessary. These histograms are a guideline and not the final word for determining the usefulness of a metric. They, however, provide a valuable visual tool for determining how a metric affects our data. We will revisit these distribution histograms later when we compare different types of metrics in detail.

The clustering metrics or *features* we examined can be categorized into three general types:

- *Structural* - features that are based on the graph-structure of individual trees. Similarity between trees is defined based on the degree of agreement of node labels and edges between the corresponding subtree graphs. (Section 3.1.2)

- *Content-based* - features that are dependent on underlying text or image content. (Section 3.1.3)

- *Contextual* - features that are external to each tree, such as the position or the *environment* surrounding each tree. (Section 3.1.4)

Our main focus will be on structural features, we will start by describing a tree edit distance based feature: Tree Alignment, followed by three vector based structural approximations (Bag of Tags, Parent-Child Pair, and Parent-Child-Child triplets *etc*), and finally two constant time structural approximation features: Tree Hash and Tree Size. For content based features, we will cover a character class based feature that has yet to be implemented and a simple content size feature. Finally for contextual features, we will describe tree context, a feature that measures the similarity of the "environment" between trees.

3.1.1 Conventions and Notation

Terminology

Throughout this chapter, we will use the terms distance and similarity interchangeably. For clarity, *distance* is simply one minus similarity because we normalize our metrics to be in the range [0, 1]. We will always define clustering metrics in terms of similarities (where 1 means exactly the same and 0 means completely different; we will use *distance* (where 1 means different and 0 means the same) when discussing these metrics in the context of clustering procedures to conform to the conventions found in the literature. We will liberally use the term *trees* to describe HTML tag trees. HTML tag trees will also be considered abstractly as *ordered labelled trees* where the labels are the HTML tags. Also, we will use the term *cluster* to describe groupings of trees usually generated by a clustering procedure. Manually labelled groupings of trees we will refer to as *record classes* or in some instances simply *classes*.

Notation

Our discussion will frequently refer to trees and attributes of trees, so we adopt the following notation for convenience:

- T_i denotes a nonspecific (ith) tree.
- $\emptyset$ is an empty (or nil) tree.
- h_i is the height of the ith tree.
- b_i is the degree or branch factor of the ith tree.
- L_i is the number of leaf nodes for the ith tree.
- $T(v)$ is the tree rooted at node v.
- $Ch_i(v)$ is the tree that is the ith child of the node v.
- w, v denote a pair of nodes from the first tree and the second tree respectively.
- $F(v_i \ldots v_j)$ is a *forest*, a sequence of trees $T(v_i)$ to $T(v_j)$ with a specific ordering.
- $F(v)$ is an abbreviated notation for the sequence of trees $F(v_0 \ldots v_n)$ that are the children of node v.

3.1.2 Structural Characteristics

One measure of structural similarity between two ordered labelled trees is their degree of correspondence. Tree correspondence is measured by how many node labels and edges agree between two trees. The problem of computing the optimal tree correspondence has been extensively studied. For ordered labelled trees, optimal correspondence can be found using quadratic time algorithms (with respect to the sizes of the compared trees) using dynamic programming. There are also many variations on the basic algorithm; Bille *et al* [3] provides an extensive survey. For our first structural feature we used a popular variant of the tree edit distance algorithm to compute the correspondence between two tag trees.

Tree Matching (TM)

Tree edit distance is the problem of finding the optimal edit script that incurs the least number of edits to convert one tree to another tree. Edits can be deletions, insertions, and transformations of tree nodes. Each edit operation is assigned a specific cost. The sum of all the edit costs for the minimum-cost script is known as the *edit distance*. The problem of tree edit distance was first described in Tai *et al* [46] as a generalization over trees of the string edit distance problem. Tai's algorithm has a reported running time bound of $O(\mid T_i \mid\mid T_j \mid h_i^2 h_j^2)$ on trees T_i and T_j. The best reported time bound for this problem was found by Chen *et al* [10] with a time bound of $O(\mid T_i \mid\mid T_j \mid + \mid L_i \mid^{2.5} \mid T_i \mid + \mid L_j \mid^{2.5} \mid T_j \mid)$.

While reduce running time is important, we are not obsessively interested in finding the absolute best time bound for computing tree correspondence. Our primary goal is still feature quality. Therefore, we implemented a variation on tree-edit distance algorithm that follows the classic formulation of the tree edit distance algorithm as first stated by Tai *et al* [46]. However, within our implementation, we make two changes. First, insertions and deletions are always for entire subtrees rather than for only tree nodes. This change was done partly for efficiency and partly for simplicity of implementation.[1] Second, we inverted the scoring function and target objectives. Instead of minimizing the number of edits we maximized the number of matching labels. We did this to ensure that all our metrics are similarity based metrics for consistency. Maximizing matches versus minimizing edits are the same after 1-normalization.[2]

We will refer to our implementation as the max-match tree edit-distance to distinguish it from the standard (min-cost) tree edit distance. The max-match score function: $\alpha(T(v), T(w))$ is the function our edit distance algorithm tries to maximize:

$$\alpha(T(v), T(w)) = \max \begin{cases} \alpha(T(v), \emptyset) + \alpha(\emptyset, T(w)) \\ \gamma(v, w) + \alpha(F(v), F(w)) \end{cases} \tag{3.2}$$

The first case reflects the null scenario where the resulting mapping is simply the deletion of the $T(v)$ and insertion of $T(w)$. The second case reflects the scenario where a correspondence actually occurs. $\gamma(v, w)$ is the *node pair score function*, the score associated with alignment

[1] This variation is also the version used by Hogue *et al* in Thresher.

[2] Our implementation can be readily converted to an edit distance formulation by a simple change of scoring weights and optimization objective which is possible by a simple change of settings within our code. We confirmed through (thorough) testing that in fact our maximized implementation produces the same optimal tree correspondences as the minimized edit distance implementation.

of the root nodes, and is defined as:

$$\gamma(v,w) = \begin{cases} w_{match} & \text{if } l(v) = l(w) \\ w_{mismatch} & \text{if } l(v) \neq l(w) \\ w_{gap} & \text{if } v \text{ or } w \text{ is nil} \end{cases} \tag{3.3}$$

The w_{match} parameter weighs the importance of an exact label match (node pairs labelled with the same HTML tag). The $w_{mismatch}$ parameter assigns the importance of a label mismatch (node pairs with different HTML tags). The w_{gap} adjusts the importance of aligning a node to gap; a gap represents an insertion or a deletion. The default setting of $\gamma(v,w)$ is:

$$\langle w_{match} = 2,\ w_{mismatch} = 0,\ w_{gap} = 0 \rangle$$

Under these settings, only node pairs that are exact matches contribute to the alignment score. Gaps and mismatches are penalized due to their 0 weights. Under this setting of weights, $\alpha(T_i, T_j)$ returns exactly the number of nodes (from both trees) that matched in the returned optimal tree alignment. Adjusting any of the scoring weights changes the range for which scores are distributed as well as the resulting configuration of tree to tree mappings. We will present some studies on the effects of adjusting these weights in section 3.3.4.

Coming back to equation 3.2, $\alpha(F(v), F(w))$ defines the score given to the optimal alignment between two ordered forests: the children of roots v and w. The recurrence equation for the optimal alignment of two forest sequences is defined as follows:

$$\alpha(F(v_0 \ldots v_j), F(w_0 \ldots w_k)) = \max \begin{cases} \alpha(F(v_0 \ldots v_{j-1}), F(w_0 \ldots w_{k-1})) + \alpha(T(v_j), T(w_k)) \\ \alpha(F(v_0 \ldots v_{j-1}), F(w_0 \ldots w_k)) + \alpha(T(v_j), \emptyset) \\ \alpha(F(v_0 \ldots v_j), F(w_0 \ldots w_{k-1})) + \alpha(\emptyset, T(w_k)) \end{cases} \tag{3.4}$$

and thus:

$$\alpha(F(v), F(w)) = \alpha(F(v_0 \ldots v_n), F(w_0 \ldots w_m)) \tag{3.5}$$

The first statement in the recurrence (equation 3.4) covers the case of aligning the jth and kth child trees. The second and third recursive cases cover where the jth tree (or kth tree) are aligned to gaps. The $\alpha(\emptyset, F(\cdot,\cdot))$ (and vice versa) terms are scores for aligning a subsequence of trees to a gap, defined as:

$$\alpha(\emptyset, T(v_i)) = w_{gap} \cdot \mid T(v_i) \mid \tag{3.6}$$

In other words, gap alignment score is simply w_{gap} weighted size of the gap aligned tree.

As a final step, we normalize the tree alignment score to [0,1] by computing:

$$\alpha_{norm}(T_i, T_j) = \frac{\alpha(T_i, T_j)}{\mid T_i \mid + \mid T_j \mid} \tag{3.7}$$

The normalized alignment score lies within [0,1] only for the stated default $\gamma(v, w)$ weights. For other weight configurations, normalization does not bound the score within the same [0,1] range. However, the normalization step in those cases still has the effect of discounting away any tree-size dependence. Other normalization formations are possible, for example dividing the average alignment scores of each tree by its size (for which the current normalization is a special case).

Tree matching considers the optimal correspondence between trees, but similarity need not be so constrained. Next, we will look at approximations to optimal tree matching that are less constrained and thus faster to compute.

Bag of Tags (BOT)

Bag of Tags is a simple metric that ignores tree graph edges and focuses only on the node labels. Under Bag of Tags, each tree is represented as a weighted vector of node labels. For example, a

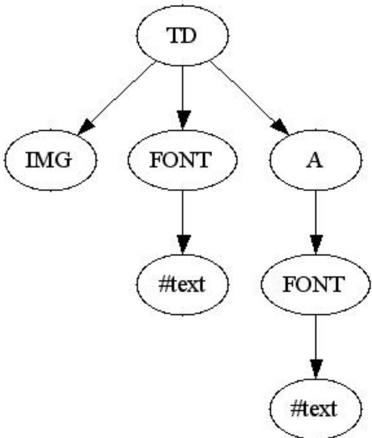


Figure 3-2: An example tag subtree

decomposition of the reference tree (figure 3-2) results in:

$$\langle td, 1/7\rangle, \langle img, 1/7\rangle, \langle font, 2/7\rangle, \langle a, 1/7\rangle, \langle text, 2/7\rangle$$

. Here the weights are simply the frequencies of the label occurrences. The weighted vector representation is reminiscent of the vector space model (VSM) representation used in traditional information retrieval. Just as in VSM, we normalized BOT vectors to unit vectors and compute similarities as the angle cosine between the unit vectors.

There are two ways to which we generate the weights for the vector components. We can use simple frequency-based weights:

$$w(l) = f(l) = \frac{Counts(l)}{\mid T \mid}$$

as shown in our example, or we can integrate the document frequency of labels by using a weight formulation similar to TF/IDF (Term Frequency Inverse Document Frequency):

$$w(l) = f(l) \cdot log \frac{1}{D(l)}$$

Here $D(l)$ is the frequency of html label l within all trees in the entire document.

Parent Child Pairs (PCP)

If we consider a tree as a set of parent-child node pair fragments, we can derive an incremental improvement to the BOT metric. Graphically, instead of looking only at nodes, we also look at the tree edges (with edges independent of other edges). A parent-child-pair decomposition of the reference tree in figure 3-2 results in:

$$\langle td \rightarrow img, 1/7\rangle, \langle td \rightarrow font, 1/7\rangle, \langle td \rightarrow a, 1/7\rangle, \langle a \rightarrow font, 1/7\rangle, \langle font \rightarrow text, 2/7\rangle$$

Like the Bag of Tags representation, each tree is represented as a weighted vector of the parent-child pairs, hence the same mechanisms of comparison and weighting can be used.

Parent Child Child Triplets (PCCT)

We can further constrain the metric by constraining the ordering of the child nodes, creating a triplet consisting of a parent label and a bigram on the children's labels. A triplet is made up of a node, and any of its two consecutive child pairs. Special "\$" nodes denote the beginning and ending of child sequences. A PCCT decomposition of our reference tree yields:

$$\langle td \rightarrow \$\ img, 1/7\rangle, \langle td \rightarrow img\ font, 1/7\rangle, \langle td \rightarrow font\ a, 1/7\rangle, \langle td \rightarrow a\ \$, 1/7\rangle,$$

$$\langle a \rightarrow \$\ font, 1/7\rangle, \langle a \rightarrow font\ \$, 1/7\rangle, \langle font \rightarrow \$\ text, 2/7\rangle, \langle font \rightarrow text\ \$, 2/7\rangle$$

Again the same mechanism for comparison and weighting can be applied to PCCT as well.

Beyond PCCT, we can imagine other ways for decomposing trees into fragments: parent-child-child-child quadruplets, or even parent-child-grandchild fragments. All of these varieties can be formulated as weighted vectors. Increasing the size and hence the complexity of the fragments will result in a metric that is not only progressively more sensitive to specific tree structure but also less flexible to variations in structure. For our purposes, we will only consider these "tree-gram" type features up to the triplet level.

Of the four structural metrics we have considered so far, tree alignment is a quadratic time bound metric and all of the weighted vector representations are linear time bound metrics. Next,

we consider two constant time metrics.

Tree Hashing

If we could hash trees then we would have a very simple constant time metric. A *tree hash* is the string-ified enumeration of the labels of that tree. We apply a standard string hash function over the tree hash to get a constant time comparable hash of each tree. The tree hash label enumeration is produced using the following recursive definition:

$$H(T(v)) = (Label(v)\ H(Ch_0(v))\ \ldots\ H(Ch_n(v))) \tag{3.8}$$

The tree hash of our example tree is simply: `(TD (IMG) (FONT (#text)) (A (FONT (#text))))` The similarity metric for tree hashes for two trees is simply 1 if their tree hashes equate (under a standard hash function) and 0 otherwise. As trees become larger and more complex the likelihood of any two trees having the same tree hash becomes increasingly smaller. This implies that large record trees will not likely share the same tree hash value. We need to modify the definition of tree hash so that same record class trees will have the same tree hashes.

If we performed selective deletions on record trees we can generate a transformed "minimum" tree that is less sensitive to variations between different instances of record trees. The deletions should not remove nodes that are essential to the general structure of the tree but only remove common variable nodes. But how do we know which nodes we should delete? Or conversely how do we know which nodes we should keep? We can derive such a list by manual examining which nodes occur consistently within same-record-class trees. Any nodes that do not consistently appear are not structurally important and we can delete. We did just that, and came up with the following list of structurally "essential" tags: $\langle td\rangle, \langle tr\rangle, \langle p\rangle, \langle a\rangle, \langle table\rangle, \langle li\rangle, \langle ul\rangle, \langle ol\rangle$

Instead of manually examining trees, we could have also automatically derived this "structurally essential" tag list using labelled data. We could have found this essential set by automatically computing the *intersection* of tree instances from the same record class. If trees were represented using a weighted vector metric such as Bag of Tags, then the tree intersection would simply be a vector intersection. We can also use tree edit distance alignment to generate tree pattern templates, in the spirit of Thresher[23]. Tree pattern templates contain only nodes that do not form *conflicts* in the pairwise tree-edit-distance alignments of original trees; a conflict is defined as a gap alignment or a mismatching of nodes. Instead of generating pattern templates for record extraction, we find the tree pattern templates in order to tabulate a list of essential tree hash tags.

To generate a minimal tree, we first identify the nodes that are not structurally essential; these are deleted from the tree. Next, the children of deleted nodes are promoted to the position of their parents. Finally equation 3.8 is applied to generate the string-ified tree hash. The tree hash feature

will almost always correctly pair same record class trees but it may incorrectly pair record and non-record trees; it is a good feature to use to generate "rough" clusters in a common technique called *blocking*. Blocking is a popular technique used in clustering to reduce the number of comparisons. In blocking, a *rough* metric like tree hash is used to partition the feature space and then a finer metric is used to fine-tune the clustering within the initial partitions. We will revisit blocking in section 3.8 of this chapter when we describe running time optimizations.

Tree statistics-based metrics

Constant time metrics can also be derived from simple statistics over tree pairs. For instance, we can define the similarity of two trees based on their tree size ratio:

$$Sim_{size}(T_1, T_2) = \frac{2 \cdot min(\mid T_1 \mid, \mid T_2 \mid)}{\mid T_1 \mid + \mid T_2 \mid} \tag{3.9}$$

The same ratio-based comparisons can be applied for tree height, average branching factor, leaf set size, or any combination of quantifiable tree characteristics. Again, like tree hash, these metric are cheap to compute but usually imprecise. Like TreeHash, these types of features are excellent as rough metrics to reduce on the number of comparisons or as approximations to more precise but more expensive metrics like tree matching.

3.1.3 Content Based Features

Content-based features focus on the text or image underlying trees. For text data, we could compare the string edit distances of the raw text strings. However record instances usually have very different underlying text. Any text alignments that do occur will be due to common field labels (and field labels do not always occur). Furthermore, trees that do align well by text string edit distance may not be records at all. Text may appear multiple times on a page because it fulfills some decorative purpose or as an artifact of the page content. For example, a search result page may have frequent occurrences of the query term (in bold) but none of the occurrences reflect any regular patterns that are part of record formatting.

Content Class Alignment

Instead of looking at direct edit distance of the text we could instead classify the text into character classes. We could segment fixed portions of text into pre-defined characteristic classes. For instance, a segment of characters that are all numbers could be labelled as `NUMERIC`; a segment of all alphabetical characters could be labelled `ALPHA`; and image content labelled `IMAGE`. Bolded text could be labelled with a multi-class label such as `BOLD-NUMERIC`. The labelling could be based on characteristics of fixed size chunks of text or simply the longest running segment consistent with

a label. The content of each tree would be represented as a sequence of character classes such as: $\langle NUMERIC\rangle\langle ALPHA\rangle\langle NUMERIC\rangle$ for "32-G494". With such a labeling, the similarity of two content bearing trees would be defined in terms of the sequence edit distance of the two character-class label sequences.

Content Statistics

Analogous to tree statistics metrics we could also use content statistics as features. The simplest of such statistics-based features is content size. The content size of a tree $\sigma(T)$ is defined as the length of its textual content size (which is simply the string length) plus image content size. Image content size is defined as 1 size unit for every 10x10 image pixel area. This conversion ratio is derived by comparing the pixel area occupied by images with the space occupied by accompanying text. Once the content size of two trees has been determined, the similarity over two trees is simply:

$$Sim_{\sigma}(T_1, T_2) = \frac{2 \cdot \min(\sigma(T_1), \sigma(T_2))}{\sigma(T_1) + \sigma(T_2)} \tag{3.10}$$

Other statistics for content comparison could also be word counts, content class counts *etc.*

For our experiments and evaluations (section 3.3), we only implemented content size as our content-based feature. Due to mainly time-constraints, we did not implement character classes or other content statistics features, and will leave it as part of a future exercise. Furthermore, we obtained sufficient accuracy from using structural and contextual features, so implementing more (complex) content-based features became less of a priority.

3.1.4 Contextual Features

Records are generally located in the same general areas on the page. They will likely be siblings or cousins of one another. Not only will they be located together, they will also likely share the same types of parent nodes or *environment*.

Tree Context

A tag path is a sequence of tags enumerated from the root of the page tree to the root of the target tree. The tag path represents all the labels that are the ancestor of a tree; it is essentially a description of the tree's environment. To use tag path as a pair metric, we used the max-match variation of the standard sequence edit distance algorithm to find an alignment score between any two tag paths. The edit distance between two tag paths is the optimal number of changes (insertion, deletion or transformations) to transform one sequence to the other. The formulation for the edit

distance of two sequences $(v_0, \ldots, v_j)$ and $(v_0, \ldots, v_k)$ is as follows:

$$\alpha((v_0 \ldots v_j), (w_0 \ldots w_k)) = \max \begin{cases} \alpha((v_0 \ldots v_{j-1}), (w_0 \ldots w_{k-1})) + \gamma(v_j, w_k) \\ \alpha((v_0 \ldots v_{j-1}), (w_0 \ldots w_k)) + \gamma(v_j, \emptyset) \\ \alpha((v_0 \ldots v_j), (w_0 \ldots w_{k-1})) + \gamma(\emptyset, w_k) \end{cases} \tag{3.11}$$

It follows the same form as the recurrence used for tree-edit distance for a forest of trees. $\gamma(v_j, w_k)$ is the weight configuration for matches, mismatches, or gaps, same as in equation 3.3. The resulting score after normalization to [0,1] is the TreeContext feature. Because tag paths between sibling trees are always identical, there is ample opportunity for caching when tag path pairs between sibling trees are compared. Also, just as tree size could be an approximation for tree alignment, tree depth (distance from the root) could serve as an approximation for tree context.[3]

3.2 Learning Pair Classifiers

Now that we have introduced several types of features, the next logical question is how do we optimally combine them into a single clustering metric $\phi(T_i, T_j)$? Are all the component metrics important? Are some more important than others? The form of the function that we learn will depend on the type of classifier we use. In all cases however, the optimized function would be a decision boundary in n dimensional space. If we used linear classifiers such as linear regression, logistics regression, or SVM (with linear kernel) we would arrive at a n-dimensional hyper-plane of the form:

$$\phi(T_i, T_j) = w_0 + \sum_{t=1}^{n} w_t \phi_t(T_i, T_j) \tag{3.12}$$

Each type of classifier would find a different weight vector $\vec{w} = \langle w_0, \ldots, w_n \rangle$ that optimized that classifier type's respective objectives. For instance, SVMs would find the boundary that maximizes the margin between positive and negative examples, and linear regression would find the boundary that minimizes the squared error.

But to train our pair-classifier we will first need to obtain some training data.

3.2.1 Training Data

To produce training data, we labelled a total of 82 web pages using a custom labelling tool (figure 3-3). These web pages were hand picked from a variety of commercial, educational, and informational websites. URLs of picked websites are listed in the appendix A.3. The labelled web pages all contained examples of single-root records. Each web page came from a different website (or in some cases a different web service from the same web site, *e.g.* Froogle and Google Images from Google.com).

[3] Another approximation would be to use just the last 3 labels in the tag path to compute the tree context feature.

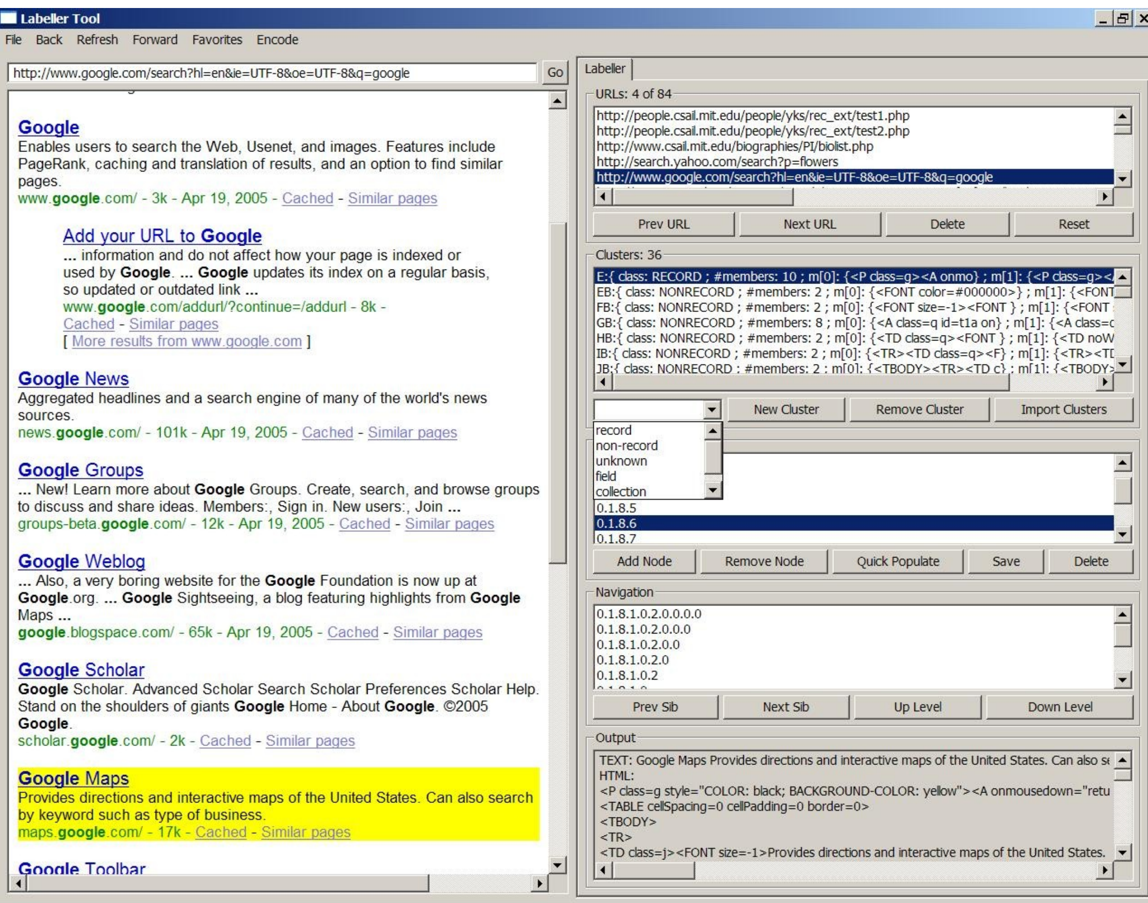


Figure 3-3: Labeling tool, screenshot

Our labeller allows us to create clusters, to select trees within a page to populate a cluster, and to assign a label to each cluster. Clusters are labelled either RECORD or NON-RECORD. For each page there may be multiple RECORD classes. A page may also contain unlabelled records that we simply failed to label. So essentially we have only partial positive data and we cannot trust that all unlabelled data are negative.

We have labelled clusters but our training instances should be pairs of trees. We considered all RECORD-RECORD tree pairs to be (*RR-pos*) positive examples. To generate these positive training points, we enumerated all tree pairs within a single labelled record set and repeated the process for every labelled record set. We considered negative examples of two types: RECORD-RECORD' (RR-neg) pairs and RECORD-NON-RECORD (RN-neg) pairs. The RR-neg pairs only appear when there are multiple record classes per page. RR-neg pairs were included based on the assumption that trees from different record classes should not be clustered together. We generate RR-neg pairs by enumerating pairs between two different record sets. Finally, RN-neg instances are pairings of trees from labelled record sets and unlabelled (and NON-RECORD labelled) trees in the

page. We intentionally excluded NONRECORD-NONRECORD pairs in the training data as these pairs can come from potentially missed (unlabelled) record sets.[4]

Since there was only one source of positive examples and multiple sources of negative examples, the negative training points outnumber the positive instances. If we trained a classifier where the negative instances greatly out-weight the positive instances, it would be very easy to produce a low error classifier that simply misclassifies all positive instances. Therefore, a common practice is to increase the weight of positive instances so as to place more emphasis on positive data. Instead of reweighting, we simply capped the number of negative data instances sampled per page such that the positive to negative ratio was more reasonable. We used all positive instances and randomly sampled 800 per page of the negative instances. The negative training data was randomly sampled to minimize any kind of sampling bias. The reweighting and resampling approaches are equivalent as long as our resampling was done at random. We used this resampling approach rather than reweighting approach in order to cut-down on our training time (since reweighting does not decrease the number of training examples). Some of our later experiments, involving feature subset selection, required training many classifiers, and training time was a bottleneck.[5]

We used half of our 82 web pages (41 pages) to generate the training set. This training set consisted of a total of 100,162 pairs; of those 70,300 (70.2%) were negative tree pairs and 29862 (29.8%) were positive tree pairs. This distribution of the positive and negative pairs places an upper bound on the training error at 29.8% which would be the case where all positive points were misclassified. The distribution of the pairs amongst the 3 sources of origin was as follows: 64156 (64.1%) were RN-neg, 6144 (6.13%) were RR-neg and 29862 (29.8%) were RR-pos.

We saved the other half of our 82 web pages as testing data. We also confirmed that the distribution of positive/negative data (and the distribution amongst the 3 sources) in the test set is close to that of the training set. So we expect that our testing error to also be bounded at around 30%.

3.2.2 Features

We implemented most of the features discussed in section 3.1. The actual nine features we used in experimentation are listed below:

- *structural*: Bag Of Tags (BOT), Parent-Child-Pair (PCP), Parent-Child-Child-Triplet (PCCT), Tree Alignment (TreeAlign), Tree Hash, Tree Size.
- *content-based*: Content Size
- *contextual*: Tree Context.

[4]We can only trust that our labelled data is correct but not complete.
[5]Even with resampling that experiment took 1 week run.

- Tree-Overlap: a simple binary feature that is 1 if two trees are not ancestor or descendant of each other and 0 otherwise. This feature alone is not particularly discriminative, however it may be useful when combined with other features.

How does each of these metrics distribute our training data pairs? We plotted histograms of the pair data distribution under each of the nine features in figure 3-4.

Observations:

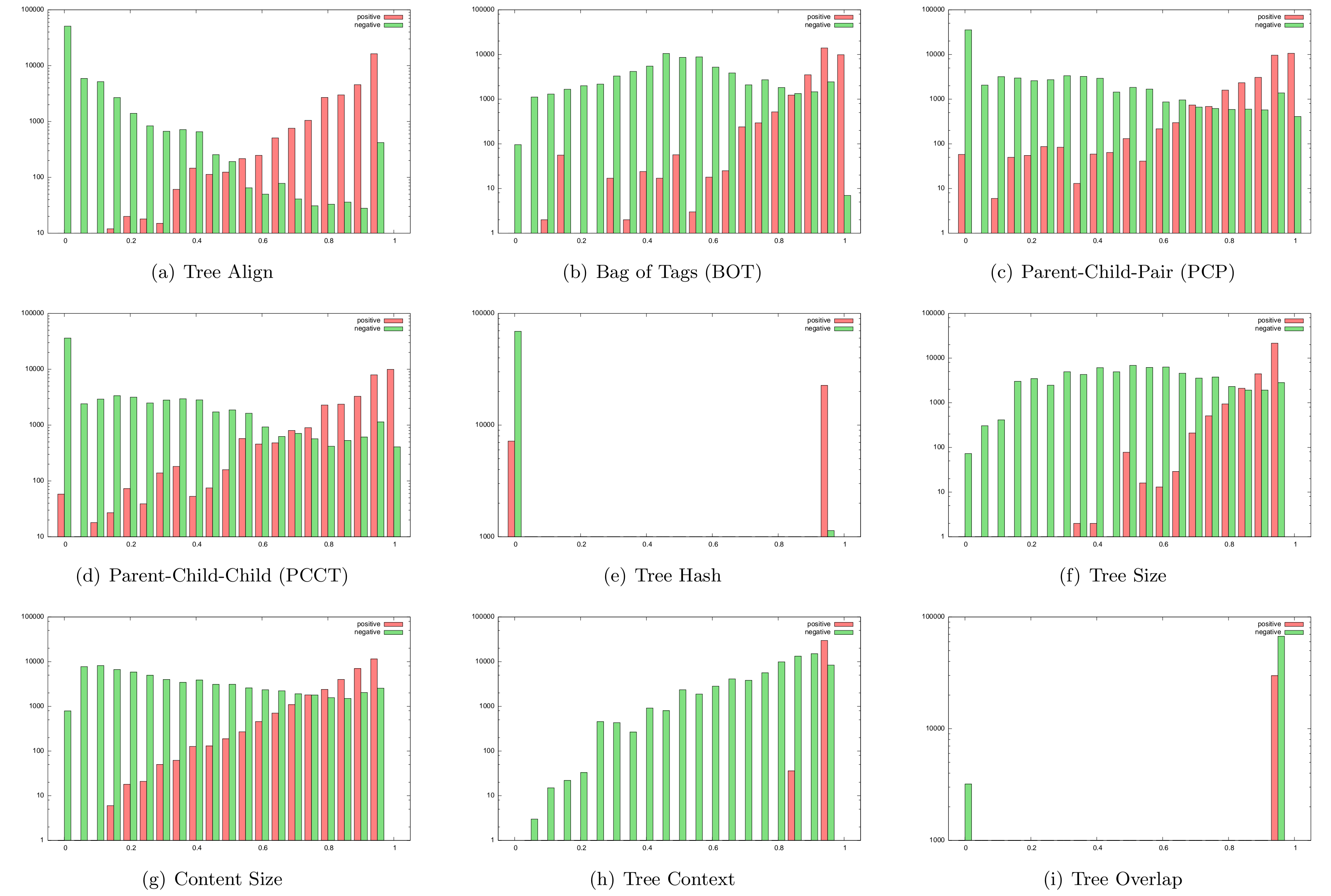
- The Tree Alignment histogram (figure 3-4(a))) showed a very distinct bi-modal distribution. Namely the positive pairs are distributed towards 1-end of the range while the negative pairs distribute towards the 0-end with the concentrations of mass near the ends of the scale. This distribution suggested that this feature is very separable with a boundary somewhere near the middle. Also, the two distributions were plotted in log scale. If the distribution appears linear (triangular) in a log scale then the actual distributions must be exponential.

- For the weighted vector based features: BOT, PCP, and PCCT (figures 3-4(b)-3-4(d)) the distribution of positive points are concentrated in the 1-end of the range like in Tree Alignment but the distribution of negative points is less distinct. For PCP and PCCT, the distribution of negative points appeared almost even across the range (although increasing slightly near the 0-end). For BOT, the distribution of negative points actually peaks near the middle of the range rather than at any of the ends. The general impression from these histograms is that these vector based features would probably perform worse than Tree Alignment on an individual feature basis.

- The Content Size histogram (figure 3-4(g)) showed a bi-modal separation of positive and negative points. However, the amount of overlap between the two classes is very high. This suggests that any separator for the two classes would incur high error rates. The same analysis appears true for the Tree size feature.

- For Tree context (figure 3-4(h)), while the negative data was distributed widely across the range, the positive data points were distinctly collected at the 1-end of the spectrum. This implies that this feature could be highly precise for positive data.

- Other features like Tree Hash, and Tree Overlap (figures 3-4(e), 3-4(i)) are binary features and hence contain only two values. None of these features appeared to distinctly separate the positive from negative. However, these features could be excellent filters for negative data. (They are good false negative detectors). By simply ignoring any data points in the 0-end of the spectrum, we could eliminate a large portion of the negative points. We shall see later that these features worked well when used in conjunction with other features.



(a) Tree Align

(b) Bag of Tags (BOT)

(c) Parent-Child-Pair (PCP)

(d) Parent-Child-Child (PCCT)

(e) Tree Hash

(f) Tree Size

(g) Content Size

(h) Tree Context

(i) Tree Overlap

Figure 3-4: Histograms showing the distributions of positive and negative data points under various features. (Note: some features show a 1.0 bucket because of numerical precision; some feature values can be slightly larger than 1.0 due to numerical precision errors. All 1.0 values should properly be in the 0.95 (0.95-1.0] bucket.)

3.2.3 Classification Algorithms

We used various classifier implementations from the Weka [47] Java machine learning library to conduct our learning experiments. We compared several type of classifiers to determine the most optimal (the lowest error rate) for our clustering metric.

For pairwise learning, we compared 7 types of classifiers: SVMs, linear regression, logistic regression, voted perceptron, ADTree (a classifier that combined decision tree and boosting) and J45 (a classic decision tree). We included a brief overview of these classifiers in appendix A.1.

The key distinguishing quality among these classifier types is that SVM (linear kernel), linear regression, logistic regression and voted perceptron all generated linear decision boundaries (hyperplanes in n-dimensional space), while ADTree and J45 generated decision boundaries that were bounded regions (boxes) in n-dimensional feature space. From a machine learning perspective, a decision tree can split the feature space into multiple regions and is far more powerful (and expressive) than a linear boundary where only 2 regions are defined.

3.3 Evaluating Pair Classifiers

3.3.1 Evaluation Metrics

To evaluate the performance of these classifiers, we performed standard testing and two forms of cross-validation: 1) the standard N-Fold and 2) leave one page out (LOPO). Standard testing involved simply running our test set over the trained classifiers. N-fold (N = 10) cross-validation divided the training data into N partitions. Training was performed using N-1 groups and testing was performed on the remaining partition. LOPO cross-validation on the other hand divided the training data along page boundaries. With 41 training pages, we left one page out and train a classifier on data generated from the remaining 40 pages. For both types of cross-validation, we averaged the error, precision, recall rates over individual runs (10 for N-fold and 41 for LOPO). Definitions of error, recall, and precision are as follows:

$$Error = \frac{fp + fn}{tp + tn + fp + fn} \tag{3.13}$$

$$Recall = \frac{tp}{tp + fn} \tag{3.14}$$

$$Precision = \frac{tp}{tp + tn} \tag{3.15}$$

Here *tp* is the count of the correctly classified positive examples (true positives); *tn* is count of the correctly classified negative examples; *fp* is the incorrectly classified positive examples; *fn* is the incorrectly classified negative examples.

3.3.2 Comparison of Classifiers

As our first experiment, we compared the performance of different types of classifiers. We trained all 7 types of classifiers using all of the nine features. The graph in figure 3-5 summarizes the error rates under each type of classifier. Data for the graphs are tabulated in Table B.1.

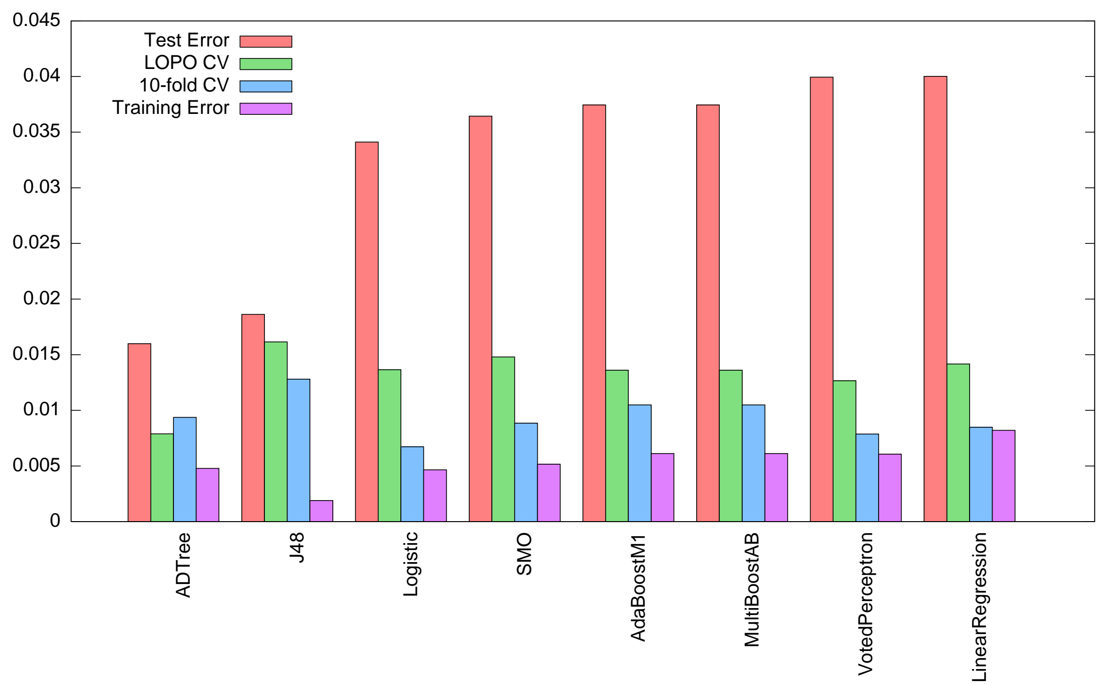


Figure 3-5: A comparison of different classifiers with all pairwise features

The graph shows a clear difference between classifiers that are decision-tree based (ADTree, J48) and those that were not (Logistic, SVM (SMO), AdaBoost, Multiboost, Voted Perceptron, Linear regression). The winner with respect to lowest test error was ADTree.

Most of the non-decision-tree classifiers performed at nearly the same level on cross-validation errors. Worthy of note, all the non-decision-tree classifiers displayed a large test to training error ratio. The test errors were on the scales of 3 to 4 times that of the training errors. Large test error to training error ratio is a classic sign of over-fitting. Over-fitting is a phenomenon where we have specified an overly complex model when the true model is much simpler.[6] Training error will almost always decrease as more complexity (more features) is introduced, but additional complexity does not mean we are getting closer to modeling the true underlying distribution; instead, the classifier is simply fitting to noise. As a result, the test error of an over-fit classifier would greatly exceed the training error.

[6]Over-fitting also occurs when we have too little training data. In such cases, increasing the size of the training data would verify whether over-fitting is occurring. We did not have sufficient training data to verify this. We will likely revisit this once we have labelled more web pages.

The most plausible source of over-fitting is that we are simply using too many features at once. One way to get around feature-over-fitting is to do feature subset selection; that is, we find a smaller (simpler) subset of features where the test error would be more in tune with the training error. We report our results on feature subset selection experiments in section 3.4.

3.3.3 Comparison of Features (Independently)

In our last set of experiments we kept the feature set fixed and varied the classifier. So naturally, in the next set of experiments, we kept the classifier type fixed (ADTree) and trained classifiers using each feature individually. The goal is to determine each feature's individual discriminative power. Histograms of the data distributions under each feature (in figure 3-4) had already given us good preview of what we should be expecting. The graph in figure 3-6 summarizes the trend in errors for our nine features. From these results it appears that Tree Alignment and TreeContext were the

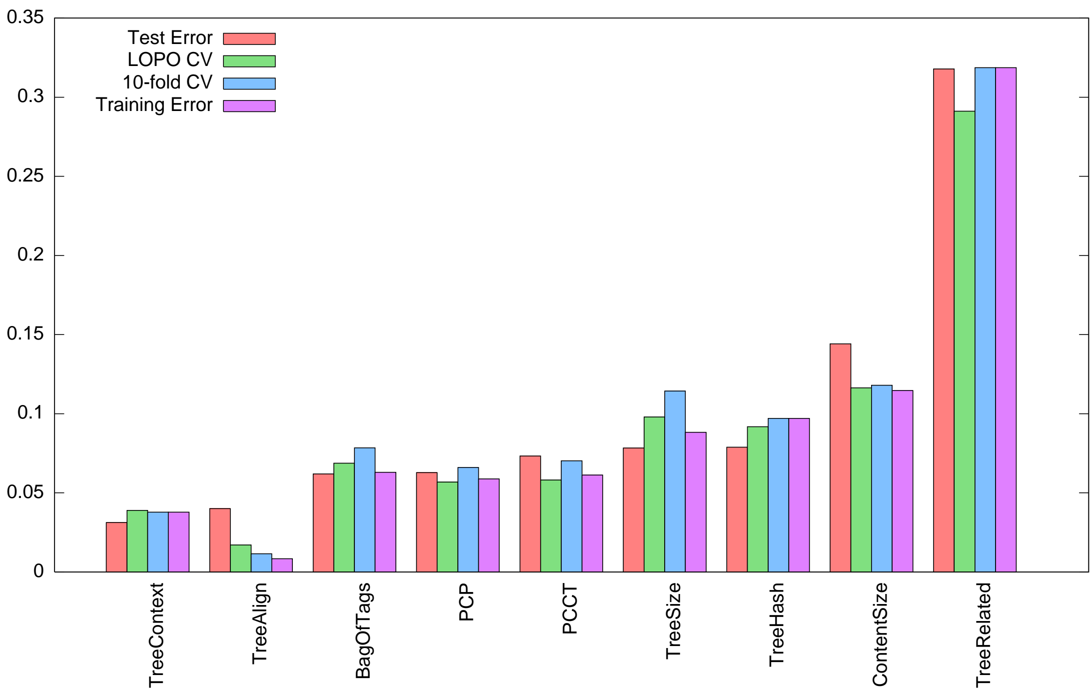


Figure 3-6: Performance comparison of individual features with an ADTree classifier

two top performing features. Weighted vector-based features BOT, PCP, and PCCT were in next place in the performance scale. These three features performed relatively equally in terms of test error rates. Approximate features: Tree hash, Tree size and Content Size all produced large errors as expected. The worst performing feature was Tree Overlap: it maxed out at the error rate upper bound of 30% (the point where all positive data points were misclassified).

3.3.4 Adjusting Tree Matching parameters

Now that we have established that Tree Alignment is a *good* metric, the next question is can we improve it? The max-match tree edit distance score that we described in section 3.1.2 was tuned using three parameters: w_{gap}, w_{match} and $w_{mismatch}$. By default, the weights were set to:

$$\langle w_{gap} = 0, w_{match} = 2, w_{mismatch} = 0 \rangle$$

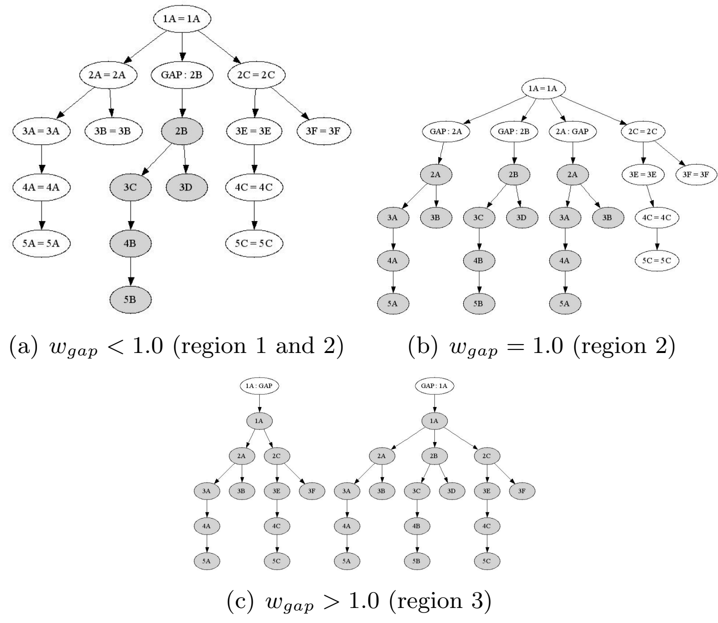
Under this configuration, the final score is equivalent to the count of exact matched nodes, with no preference for either gaps (insert/deletes) or mismatches. Adjusting $w_{mismatch}$ versus w_{gap} will tip the favor towards mismatch-rich alignment configurations or gap-rich alignment configurations.

In our next series of experiments, we kept both w_{match} and $w_{mismatch}$ constant and varied w_{gap}. By varying w_{gap} from -1 to 2 we essentially split the score domain into 3 regions.

- Region 1: $w_{gap} \in [-1, 0)$ and $w_{gap} < w_{mismatch} < w_{match}$. In this region, the optimal alignment favors mismatch-rich configurations.
- Region 2: $w_{gap} \in [0, 1]$ and $w_{mismatch} \leq w_{gap} \leq w_{match}$. At the lower end of this region 2, gaps and mismatch are equally favored (both 0) but at the upper end of this region, gap-rich configurations become predominant.
- Region 3: $w_{gap} \in (1, 2]$ and $w_{gap} > w_{match} > w_{mismatch}$. Gap operations are favored over all types of edit operations. As a result, all tree pair alignment configurations become of a single type, the type where one tree is inserted and the other tree is deleted. (This region is not particularly interesting but we include it for completeness.)

Figure 3-7 illustrates the range of possible tree pair alignments for two hypothetical trees. In the diagrams, the example tree pair undergoes three alignment configurations as w_{gap} varied from region 1 to region 3. The number of configurations that a general pair of trees may undergo will depend more or less on their branching factor, depth and degree of similarity. What we are most interested in though is how the *distribution* of optimal tree alignments for a population of trees change as we vary w_{gap}. In figure 3-8 we show the training data distribution at various values of w_{gap} within [-1,2].

There are several trends worthy of notice regarding these distributions. The width of the distribution becomes progressively narrower as w_{gap} increases; this is expected as normalization is not to [0,1] for non-default weights. In region 3 (figure 3-8(d)), all the alignments are of one configuration, namely the complete deletion and insertions variety, and hence all scores are the same. The data will not be separable in region 3. The score distribution is widest in region 1 when the $w_{gap} < 1.0$. Within both region 1 and 2, while the score range becomes progressively narrower the distribution shape is roughly maintained. This implies that the data is still separable in both regions 1 and 2.



(a) $w_{gap} < 1.0$ (region 1 and 2)

(b) $w_{gap} = 1.0$ (region 2)

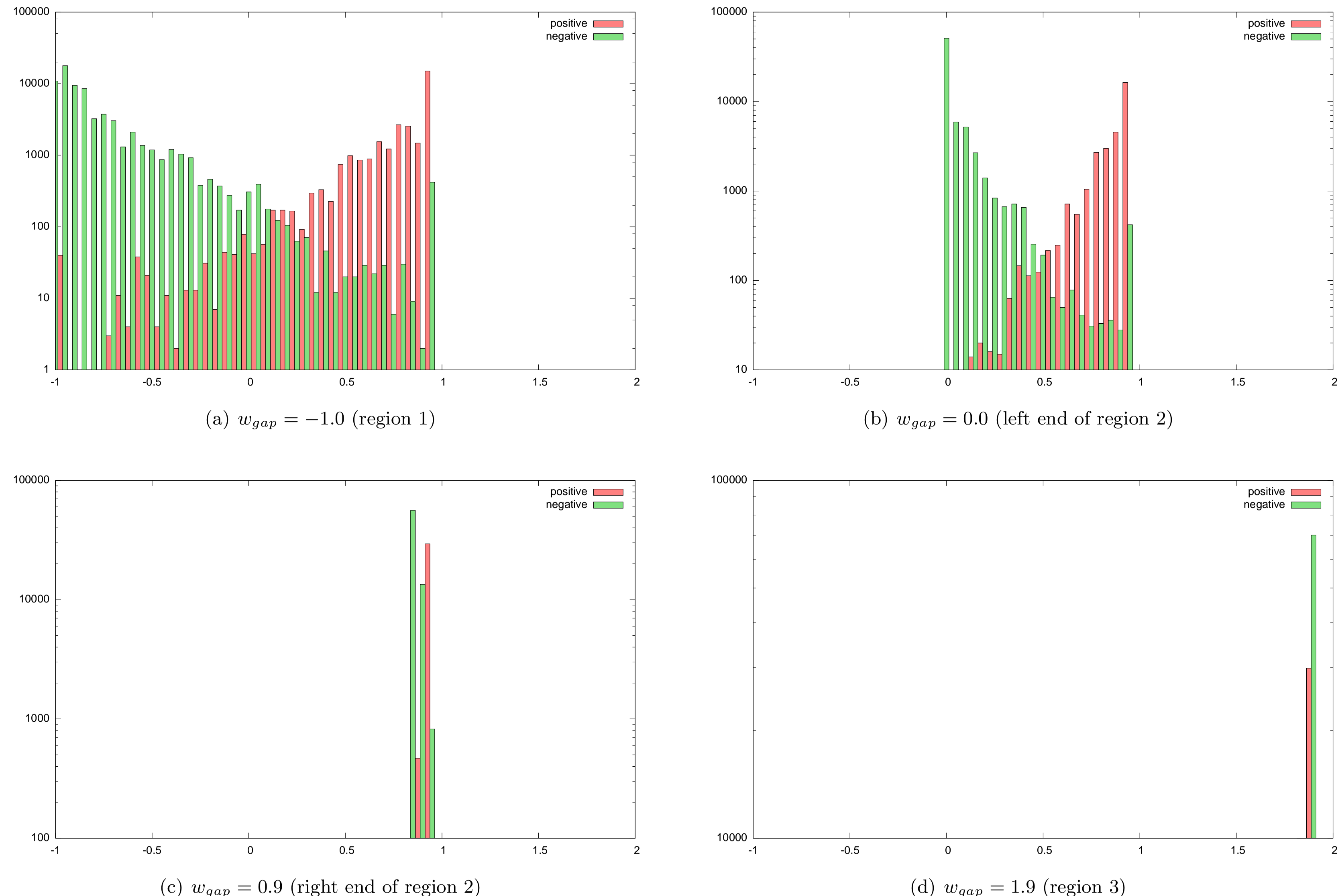
(c) $w_{gap} > 1.0$ (region 3)

Figure 3-7: Tree Alignment configurations with varying w_{gap}

As a subtle observation, notice that in figure 3-8(a) there is a population of pairs distributed in the -1 bucket (left end) indicating a significant number of gap-rich pair alignments. Shouldn't the best alignments in region 1 be mismatch-rich? So shouldn't all the alignments have a minimum score of 0? Yes, they should be mostly mismatch-rich, but for some pairs of trees, gap alignments are inevitable. This is clearly true if one tree is larger than the other tree. Consider the case of two trees T_1 and T_2 where T_1 is a single leaf node and T_2 is an n-node tree. Even in the best alignment, $n-1$ nodes must still be gap aligned. In such pairings, the best alignment score is at most $\frac{(n-1)w_{gap}}{n+1}$ which in the limit for large n approaches -1.0.

Coming back to our original question, how does varying the w_{gap} affect the classifier's error rate? To answer this question, we trained 31 different classifiers each with data generated using a different w_{gap} value (within [-1, 2] at increments of 0.1). Figure 3-9 shows a plot of the error rate as a function of w_{gap}. The error rates are at their maximum possible value for region 3. This is expected from our distribution graphs analysis since all pair alignments in region 3 are of the entire-tree-insert-and-delete variety.

The error rates for regions 1 and 2 appeared mostly flat. However if we zoom in on those regions (figure 3-9(b)) we see the lowest error rate occurs when w_{gap} is around 0 (our default value). The error rate actually trends up as w_{gap} decreases (into region 1). In region 1, mismatch-rich alignments would be the predominant optimal configuration which implied that tree pairs that were structurally similar **only** but maybe labelled differently will attain high scores. These types of alignments run contrary to our familiar notion of structural similarity which requires both structure *and* labels to be similar. So as more structurally-similar but label-dissimilar alignments become dominant, it



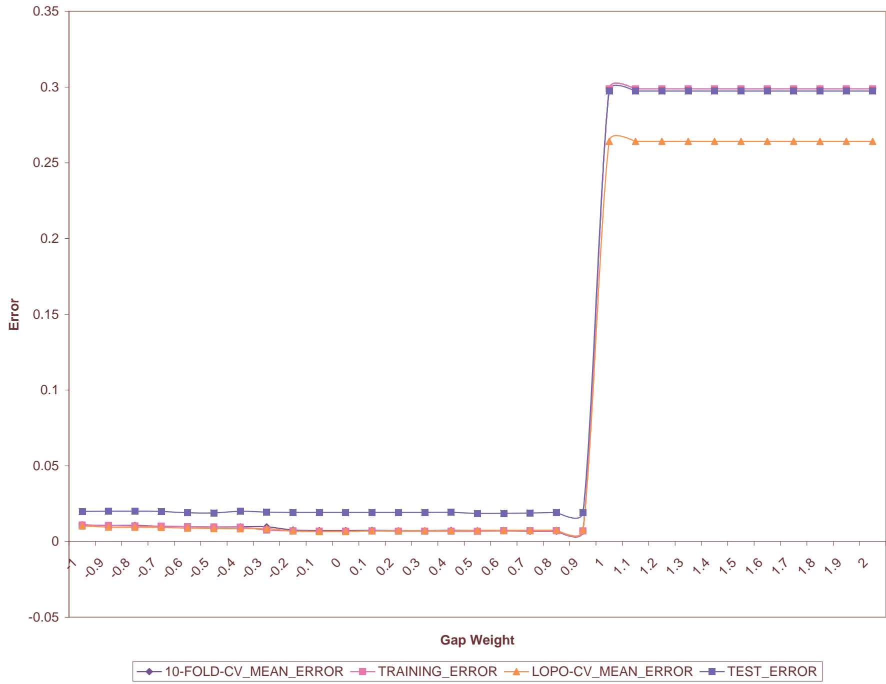
(a) $w_{gap} = -1.0$ (region 1)

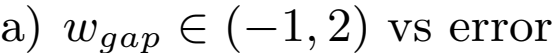
(b) $w_{gap} = 0.0$ (left end of region 2)

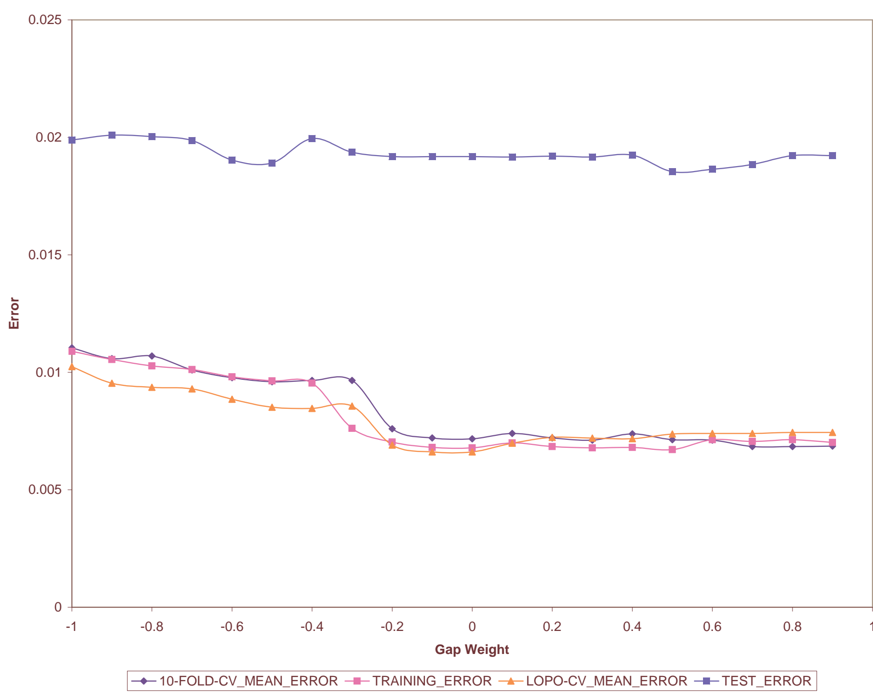
(c) $w_{gap} = 0.9$ (right end of region 2)

(d) $w_{gap} = 1.9$ (region 3)

Figure 3-8: Distribution of Tree Alignment pairs (of the training data) with varying w_{gap}




(a) $w_{gap} \in (-1, 2)$ vs error



(b) $w_{gap} \in (-1, 1)$ vs. error (zoom in of regions 1 and 2)

Figure 3-9: Tree Alignment feature error rate as a function of varying w_{gap}

becomes increasingly difficult to separate positive trees pairs from negative ones, hence explaining the uptrend in the error rate.

In region 2, the error rates remained relatively flat. There is no significant up or down trend in error rates.

These experiments show that varying w_{gap} (and by proxy $w_{mismatch}$) significantly affect the quality of the resulting classifier. Fortunately, the score weight values we chose as default were in fact the most optimal for Tree Alignment.

3.3.5 TF/IDF versus Frequency Vector Weights

When we discussed weighted vector based features such as BagOfTags, Parent-Child Pairs, and PCC Triplets, we noted two ways to compute the vector weights: simple frequency weighting and TF/IDF style weighting. Does the choice of vector weighting affect the discriminative power of the feature? To find out we experimented with both frequency versus TFIDF vector weighting for all three features. Figure 3-10 summarizes the error rates. Ranking the results based on test error (the

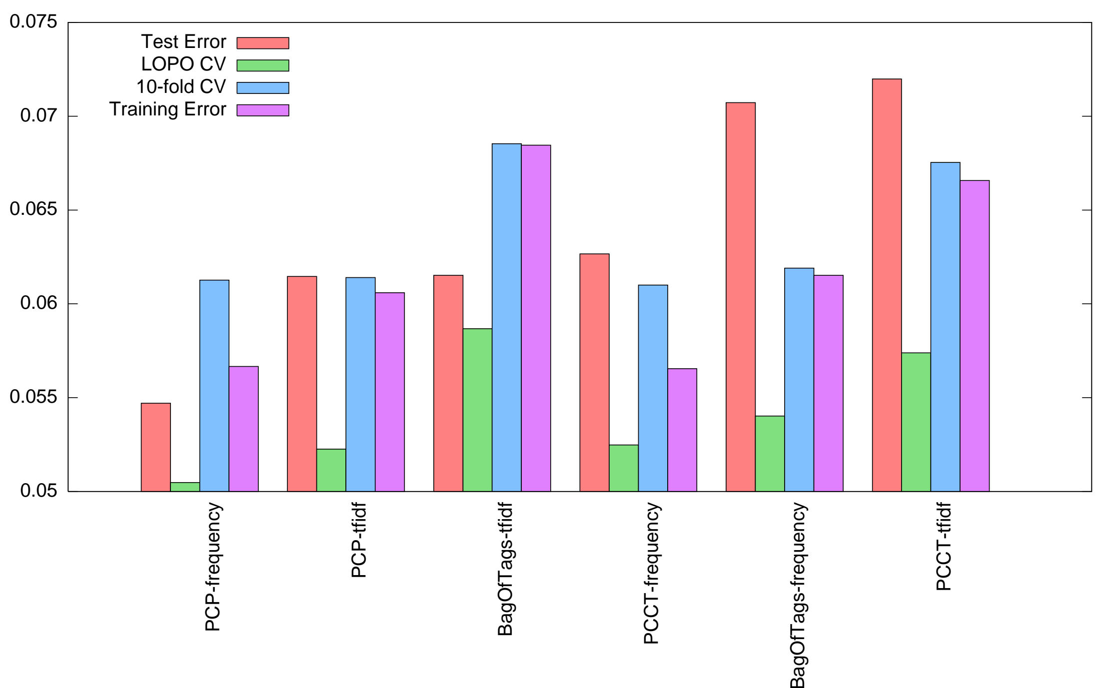


Figure 3-10: Comparison of frequency versus TFIDF vector weighting for BOT, PCP and PCCT.

left-most bar), TFIDF appears better than frequency weighting for BagOfTags, but the inverse is true for PCP or PCCT. Ranking the results based on cross validation errors, frequency weighting for each feature achieved lower errors than all their TFIDF counter-parts. However, in both cases, the difference in error rates was very small. So, in conclusion, it would appear that the choice

of weighting formulation did not significantly improve or degrade the performance of the resulting classifiers. For subsequent experiments we will use a frequency based weighting as default for all weighted-vector-based features.

3.4 Feature Subset Selection: finding the optimal feature subset

In the next set of experiments, we wanted to determine whether we can use less than all nine features to construct an optimal classifier. As we discovered earlier (section 3.3.2), adding too many features introduced a form of over-fitting. One can keep adding features to a classifier to decrease the training error rate but the testing error will increase because the extra features simply provide redundant information. So it is crucial that we find the optimal feature subset that does not exhibit (significant) over-fitting.

Finding an optimal feature subset or *feature selection* is a common problem in machine learning [22]; it is a problem that is fundamentally solved using search. Formally, we are given n features and want to find the optimal k subset that minimizes some evaluation criterion such as classification error. An exhaustive search by trying all $2^n - 1$ feature combinations is certainly one option if n is small, but the process is still resource intensive. A typical strategy is to use heuristic search strategies to cut down on the number of combinations searched. However heuristic searches like best-first are incomplete (it does not guarantee all combinations are evaluated) and cannot guarantee that we will find a globally optimal solution.

3.4.1 Feature Subset Evaluation

In the feature selection literature, there are generally two types of methods for evaluating subsets of features: *wrapper methods* and *filtering*. A wrapper method trains a classifier for each feature subset then use an evaluation metric such as cross-validation error, test error or f-score to guide a search through the space of all feature set combinations. The feature subset with the best evaluation metric value is the optimal feature subset. Wrapper methods are generally resource intensive due to the training/testing/validation overhead required per feature subset. Filtering is the less costly alternative that uses statistical correlation metrics such as information gain (between the current feature subset and the target class) instead as the evaluation criterion.

3.4.2 Exhaustive Search

We conducted an exhaustive search using a wrapper method. We evaluated our baseline classifier: ADTree through all possible 511 ($2^9 - 1$) feature combinations starting from subset size 1 to subset

size 9 (all features).[7] It took *only* about a week to run. Parallelizing the search is definitely a possible future improvement for speeding up the search time.

In the appendix, tables B.2,B.3,B.4 summarizes the performance results for single features, double-features, and triple-features combinations. Depending on which error criterion used, the best performing combination found differed. Table 3.1 summarizes the best feature subsets we found:

Eval. criterion	Feature	Eval. criterion value
Test Error	BagOfTags, TreeSize, TreeContext	(min) 0.013687
Test F-Score	BagOfTags, TreeSize, TreeContext	(max) 0.97880
LOPO-CV Error	PCCT, TreeContext, TreeOverlap	(min) 0.008173
10-fold CV Error	PCP, TreeAlign, TreeHash, TreeContext	(min) 0.006887

Table 3.1: Best performing features subsets using different evaluation criterion

Surprisingly, all the optimal feature subsets included TreeContext, our only contextual feature. TreeAlign (one of the best performing single features) only appeared in the 10-fold CV optimal combination. It would seem from these results that one can train a low error classifier by using feature combinations that were not the most optimal (individually). Note also that no more than two of structural features: TreeAlign, PCP, PCCT, and Bag Of Tags appeared together in any of the optimal feature subsets; features of the same type shared redundant information, therefore adding more of them does not add more value.

The third feature in the optimal subset usually consisted of TreeSize, TreeHash or TreeOverlap. From the histograms of these features, we knew that these were not useful features on their own but were excellent true negative filters (they all had very high true negative rate). Features that were able to reduce a large portion of negative instances (accurately) combined with strongly discriminating features make for good feature combinations.

The trend for optimal subset seemed to consist of some combination of a contextual feature, a structural feature, and a third true-negative-filtering feature. Central to this trend is the idea that best feature subsets contain features with orthogonal sources of information.

Returning to our discussion on over-fitting, we compared the test versus training error ratios for the 3-feature optimum (found through feature selection) and the 9-feature classifier (that we trained for classifier comparison). The testing and training errors are reported in table 3-11.

[7] To be fully convincing we would have needed three sets of data to properly perform wrapper-based feature subset selection, a training data set for training wrapper classifiers, a testing data set for picking the optimal feature subset, and a final evaluation set to evaluate our selection. In our experiments, we only had a training and test data set. So admittedly, if we use test error as our evaluation criterion then we will be using the test data to both select the feature set and evaluate the feature set. This can be construed as mixing training and evaluation data. However, if we picked the best feature subset using the cross validation errors (which are based on the training data) then we would not be as biased. Nonetheless, we expect to revisit this experiment in the future with three data sets once we obtained more labelled data.

Number of features	Training	Test	Train:Test
all 9 features	0.004784	0.015998	1:3
3-feature optimal	0.00897	0.01369	1:1.6

Figure 3-11: Training/test errors for all pairwise features versus the optimal 3-feature subset. Note that there is an absolute decrease in the overall test error and a decrease in the training to test error ratio.

The ratio of testing and training error has vastly diminished as we decreased the number of features from nine to three. Furthermore, the value of the test error has also decreased by 0.23%. We cannot say absolutely that over-fitting has been eliminated but it has certainly been reduced.

3.4.3 Correlation-based Feature Selection (CFS)

We also tried an alternate feature selection method called filtering. CFS or *correlation-based feature selection* [22] is a filtering based method developed by Hall *et al* and is included in the Weka machine learning package[47].

Fundamental to CFS is the hypothesis that the best feature subset is one that has features which correlate best with the output class (relevancy) but do not correlate well with other features (redundancy). Using mutual information (information gain) as a correlation measure between the features and the output class, CFS searches the subset that minimizes the average mutual information between features but maximizes average mutual information between features and the output class. This hypothesis agrees with our empirical observations from our exhaustive wrapper-based search that the optimal feature subsets usually involved features that contained orthogonal sources of information.

We ran CFS under an exhaustive search using mutual information as the criterion. The algorithm returned the following 3-feature combination (table 3.2). We know that TreeContext and TreeAlign

Feature Subset	Test Error	Test F-score
TreeContext, TreeAlign, and TreeHash	0.03517	0.9433

Table 3.2: Test error, test f-Score for the optimal CFS-selected feature subset

are orthogonal features since they came from two different categories of metrics. TreeHash again is one of the approximation features that had a high true negative rate and therefore fits in with our previous empirical observations. The feature subset found via CFS search was unfortunately not the combination with the lowest test error rate as reported by our wrapper based exhaustive search.

3.5 Clustering

Having done a fairly comprehensive exploration of methods to generate a suitable clustering metric (pair classifier), our next task is to apply the metric for clustering.

3.5.1 Prefiltering

Before we began clustering, several additional filtering steps were needed to remove obvious non-record subtrees from consideration. Reducing the number subtrees to cluster reduced the number of unnecessary comparisons we would have to do, as well as the amount of memory used (another important constraint on our system). The types of trees we filter away include:

- Non-content bearing subtrees - Records by definition must contain information; and information is conveyed mostly via images or text. So it is safe to remove all trees that do not contain text or image leaf nodes.

- Text and image (leaf) nodes - Records are composed of content **and** formatting. Text and image nodes are purely content with no attached formatting information and therefore they are not very useful in comparisons where we compared structural features. We imagine that deep structure could be extracted from the underlying text/image with the proper image processing or natural language processing. One possible venue for future work is to derive character classes (as mentioned in our discussion on content similarities) or apply named entity parsing over the text strings. Text and image leaf nodes constituted about 50% of the nodes in a DOM tree so by removing them we dramatically reduced the number of trees we needed to compare.

- Top or root level nodes - Nodes such as HTML or BODY are top level trees and are one-of-a-kind within a page. They do not repeat and therefore cannot form patterns. So immediately they are not considered candidates for records.

- Invisible nodes - Some HTML elements such as $\langle META \rangle$, $\langle SCRIPT \rangle$ or $\langle STYLE \rangle$ serve general tagging functions but do not affect underlying formatting or layout of the underlying text. Furthermore, elements such as $\langle INPUT \rangle$ or $\langle SELECT \rangle$ produce specific HTML user interface elements and serve to configure those elements rather than provide formatting or layout to the underlying content. Trees rooted by all these tags, and in addition any trees under the HEADER section, were excluded from consideration.

3.5.2 Distance Matrix

After pre-filtering, we are left with N trees that we actually perform clustering over. Using the trained pairwise classifier, we fill in an $N \times N$ distance matrix where each i,jth cell contains the

decision {1,0} whether the ith tree should be clustered with the jth tree. The matrix is symmetric, so only $\frac{N^2-N}{2}$ entries were actually computed. This distance matrix represented a graph with edges for any pairs of trees that were likely record tree pairs.

While the literature lists numerous clustering algorithms, we came to an early decision to use one of the simplest: Hierarchical Agglomerative clustering (HAC). We decided on using HAC as result of an earlier study, the summary of which is included in the appendix A.2. HAC starts with N singleton clusters. Two clusters are merged if they satisfy the clustering criterion (which we shall define later). After each iteration, the number of clusters is reduced by one and this merging process continues until no more cluster pairs could be merged.

The distance between any two singleton clusters is simply the value of the pairwise metric over the pair of underlying trees. But for two non-singleton clusters (both size > 1) there were no clear "representative" trees that we can use for representing the entire collection. In certain metric spaces we could define a centroid or medoid: a new point that is the average or the median of the collection. For example, in a vector space representation (applicable for BOT, PCP type features), the centroid is the normalized sum of the cluster member vector. In a representation using a quantifiable statistic (like tree size or content size) the centroid is simply the average of the statistics. In our {0,1} decision-function based metric space we have to define an alternate criterion that is an aggregate metric between two collections of points (tree pairs).

3.5.3 Clustering Criteria

The two most commonly used aggregate clustering criterion are *single link* and *complete link*. In single link, the cluster to cluster distance is defined as the minimum distance between the two cluster sets:

$$d_{min}(C_i, C_j) = \min_{x \in C_i, x' \in C_j} 1 - \phi(x, x') \tag{3.16}$$

If d_{min} is below a threshold t, then we merge C_i and C_j. We set t to be 0.5. So single link means that two collections of points are clustered if there exists at least one pair whose ϕ value is 1.

In single link we asked the question, what is the closest pair of points between two sets of points such that they still satisfy a threshold? We can also ask: what is the *farthest* pair of two points between the two clusters that still satisfies the threshold? Thus we arrive at the other extreme, the complete link criterion:

$$d_{max}(C_i, C_j) = \max_{x \in C_i, x' \in C_j} 1 - \phi(x, x') \tag{3.17}$$

Complete link requires that **all** pairs within a cluster must satisfy the threshold t. In simple graph theoretic terms, the complete link criterion produces clusters that are cliques (with respect to the threshold).

Of course, there are other criteria that lie between those two extremes. For instance, average

link computes the threshold as the average of all distances between pairs of points:

$$d_{avg}(C_i, C_j) = \frac{1}{n_i n_j} \sum_{x \in C_i} \sum_{x' \in C_j} 1 - \phi(x, x') \tag{3.18}$$

At $t = 0.5$, average link guaranteed that at least 50% of the pairs in the final cluster will be connected. Adjusting the threshold t adjusted the percentage of links connected.

Finally, if we want to ensure at least k links within a cluster satisfy the threshold, we can define a *k-link* metric. For two clusters, the kth pair in the sorted pairs by ϕ value between the two clusters determines the threshold value. When there are less than k pairs between two clusters then all pairs must be connected.

3.6 Evaluation of Clustering Quality

3.6.1 Evaluation Metrics

Typically when evaluating cluster quality we compare our produced clustering of instances with a reference clustering. The reference clustering is our labelled testing data. Usually cluster evaluation is done over both positive and negative clusters. However we are interested only in how well our system produces clusters that are most similar to record classes (positive clusters). Furthermore, since we do not have labels for all the non-record clusters, it would make little sense to evaluate how well non-record clustering performed.

To perform our evaluation, we found for each record class (labelled record set), all the clusters that contained instances from that record class. We call this set of clusters the *associated clusters* with respect to class j; the ith associated cluster is denoted C_{ij}. The associated clusters for a class could range from a single cluster where all the class instances are distributed in one cluster, to n clusters with one class instance distributed in each cluster. Within each cluster C_{ij}, we can have *pure* clusters where all members are of the same class, or we can have impure clusters where most of the instances are non-record instances (or members of other record classes). In short, we have two dimensions for evaluation: a measure of *distribution* and another of *purity*.

To quantify these dimensions, we first define *purity* as the precision of C_{ij}:

$$p_{ij} = \frac{|L_j \cap C_{ij}|}{|C_{ij}|} \tag{3.19}$$

Here L_j is the labelled class set, and C_{ij} is the unlabelled cluster. To quantify the *distribution* of class instances we wanted a measure that is independent of the size or number of members of the labelled class. Recall is that size independent quantity that roughly describes instance distribution.

r_{ij} is the fraction of instances from a given jth class that is contained within the ith cluster.

$$r_{ij} = \frac{|L_j \cap C_{ij}|}{|L_j|} \tag{3.20}$$

A clustering with a low average recall would imply that such instances were distributed thinly across different clusters.

A clustering is good if the associated clusters are all pure, and distributed in a concentrated fashion (all in one cluster). Commonly the f-score or the equally weighted average of the two measures is what is reported:

$$f_{ij} = \frac{2 \cdot p_{ij} \cdot r_{ij}}{p_{ij} + r_{ij}} \tag{3.21}$$

So far all our metrics are of a single associated cluster. One common practice is to compute and report the average recall, purity or f-score over all associated clusters. Another common practice is to simply look at the *maximum* values of f-score for associated clusters. Maximums will better suit an application (such as ours) if we intend to eventually pick out a cluster for use (to identify it as a record cluster); we should also ensure that such a cluster exists (not just an average). For these reasons, we chose the maximum f-score as our final evaluation metric.

To compute the max f-score for the jth class, we simply compute:

$$f_j = \max_{1 \leq i \leq N} \frac{2 \cdot p_{ij} \cdot r_{ij}}{p_{ij} + rij}$$

The max precision and max recall values are the precision and recall values of this f_j cluster. N is the number of associated clusters for class j.

Besides maximum and average recall/precision metrics, there have been other metrics suggested for evaluation. One possibility is a variation on edit-distance where we measure the quality of a clustering as the number of changes: cluster member deletions, insertions and merges required to transform the target associated cluster set into the reference record class set. This edit distance evaluation metric is more complicated to define and compute, but is an interesting idea for future exploration.

3.6.2 Evaluation and results

To evaluate our clustering results, we used an ADTree pair classifier trained from the 41 page training set using the feature subset TreeAlign, TreeContext and PCP.[8] The classifier we used is **not** the

[8] We conducted our clustering evaluation in parallel with our optimal feature subset search, so we used the best feature subset known at the time of clustering evaluation. This feature subset combination registered a test error of 0.01706, which was close but slightly higher than the optimal error of 0.01369.

optimal classifier found using feature selection. However, it has a test error rate which is close to the feature-selection optimal error rate. We avoided using the optimal classifier in order to avoid any training/testing data cross contamination since feature selection used the same testing data for selecting the optimal subset as we will be using for cluster evaluation.

Our testing pages, the remaining 41 pages from our 82 labelled set, contained 57 labelled record classes. Most pages (31) had just one record class; nine pages had more than 2 record classes; three pages had more than 3 record classes. We experimented with different clustering criteria (from single-link to k-link to complete-link) and computed maximum f-score, recall, and precision values for the associated clusters of each record class. Figure 3-12 summarizes the results from 8 different clustering criteria. The last criterion is complete-link with post-processing and it will be discussed in detail in section 3.7.5. The reported evaluation results were averaged over the 57 record classes. We include the original data in the appendix B.5.

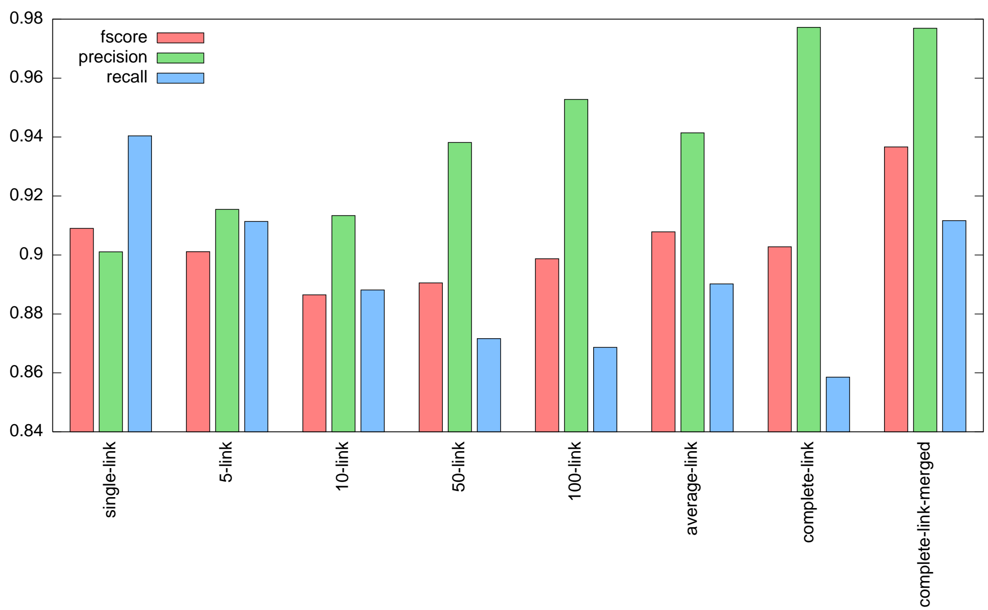


Figure 3-12: Average maximum f-Score, precision, and recall for values for different clustering criteria

The quality of clustering was acceptable for our application. The mean precision and recall values were all above 0.8 for all clustering criteria. The mean max-f-score values for all clustering criteria lay in a range from 0.88 to 0.94. Single and complete-link form the opposite ends of the precision versus recall tradeoff scale. Single link registered a very high recall (and the lowest precision) while complete link registered very high precision (and lowest recall). As the link number increased from 5, 10, 50, and 100, the precision gradually increased and recall decreased. At around 5-link, the precision and recall became close to equal.

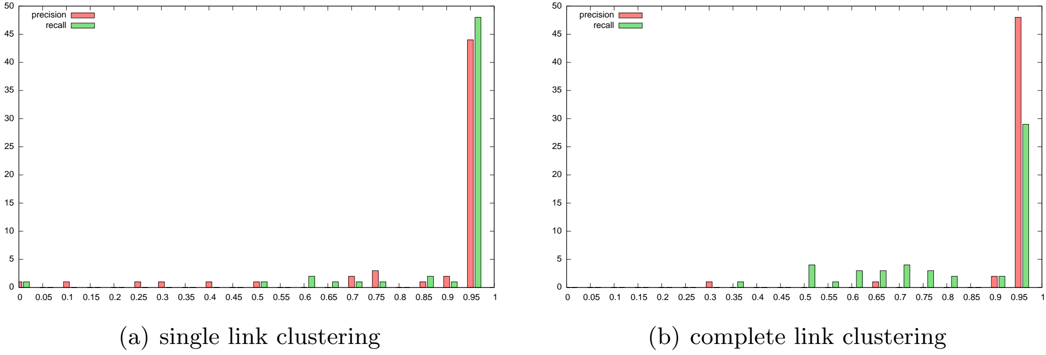
Since our goal is to select the best cluster as the record cluster it makes far more sense to favor precision than recall. If we can guarantee that all our found clusters were pure then we can use other existing methods like Thresher [23] to find more record candidates. These supervised wrapper induction systems take (only) positive examples to generalize the examples into a record pattern template. Therefore, it is crucial that we provide a pure cluster with only positive examples. A high recall cluster, on the other hand, would be less useful since there would be more work needed to filter out the unknown fraction of non-record instances (possibly manually).

What about the distribution of results? Table 3.3 shows a breakdown of clustering results into two quality grades. The f-score-greater-than-0.95 grade contains clusters that are almost perfect to perfect. The greater-than-0.8 grade contains clusters that are on the whole acceptable. An f-score of 0.8 means that the cluster can in the worst case have a precision of 1.0 and recall of 0.66 or vise versa.[9] This breakdown by grades shows that the major fraction of generated clusters was of

clustering criterion	f-score ≥ 0.95	f-score ≥ 0.8
single-link	40/57 [70.2%]	50/57 [87.7%]
complete-link	34/57 [59.6%]	47/57 [82.5%]

Table 3.3: Fraction of near perfect clustering results

excellent to acceptable quality. Naturally, were these near perfect clusters simply easy? And how are unacceptable or "bad" clusters (with f-score less than 0.80) distributed? In figures 3-13(a) and 3-13(b) we plot a histogram showing the cluster distribution by precision and recall.



(a) single link clustering

(b) complete link clustering

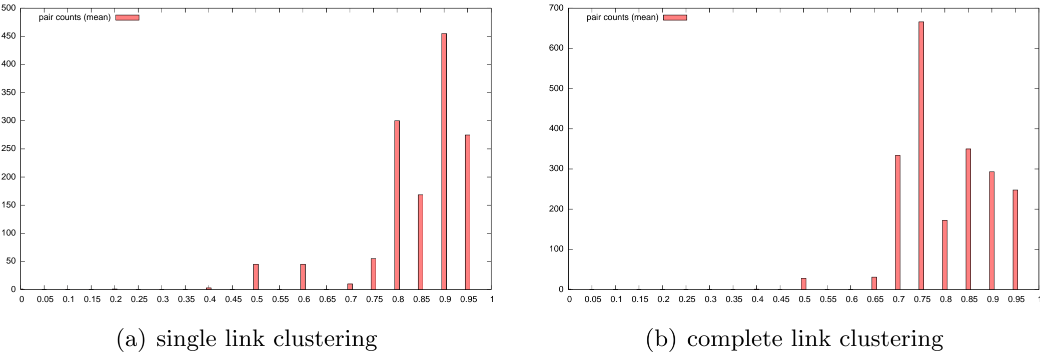
Figure 3-13: Distribution of clusters by precision and recall.

From the figure 3-13(a), it would appear that the tail portions of the distribution for single-link results were largely accounted for by clusters that had low purity. For complete-link (figure 3-13(b)), the tail distribution is attributed to clusters with low recall. This confirms that bad clustering is due to the precision versus recall tradeoff between using a single-link versus complete-link clustering

[9] In retrospect, since we are more concerned about precision, we could have used a version of f-score that favored precision more than recall rather than an equally weighted average.

criteria. To better understand why these “bad” clusters occur we decided also to look into their *associated record classes*.

One possible theory is that “bad” clusters are associated with small-sized (in terms of the number of instances) record classes. Small-sized classes contribute fewer tree pairs to the training data than large-sized classes; thereby they are harder to cluster correctly. We plot the pair-counts of the *associated record classes* for clusters distributed by their cluster’s f-score in figure 3-14. Pair count measures how many positive training pairs each associated record class contributes to the training data.



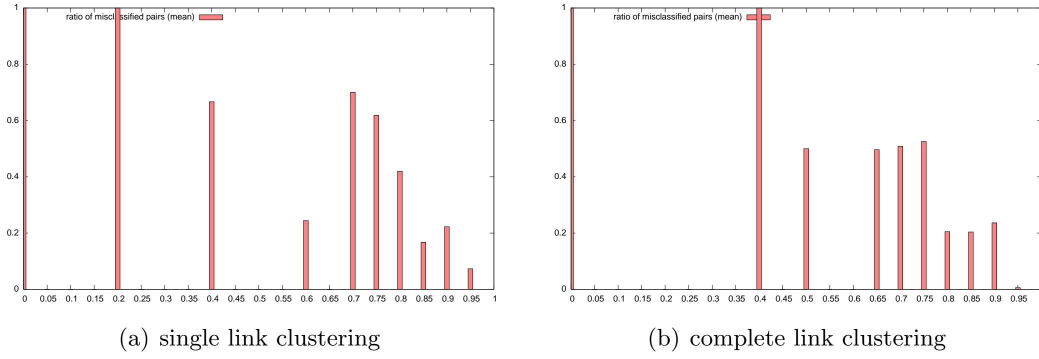
(a) single link clustering

(b) complete link clustering

Figure 3-14: Pair counts (Mean) of *associated record classes* of clusters distributed by the cluster f-score.

We can clearly see a correlation between cluster quality and associated record class size (*i.e.* correlated with pair count). For single-link clusters in the high f-score range (f-score of 0.8 to 1.0), the associated classes were generally large, averaging about 150+ pairs. For clusters in the low and medium f-score regions (f-score of 0.2 to 0.6), the associated classes were small, averaging only about 2-10 pairs. However, for complete-link clusters in the medium f-score range (f-score of 0.7-0.8), the associated record classes had large number of pairs. This last observation can be explained as follows: under complete-link clustering, more pairs a record class has harder it is to form cliques, so larger classes tend to result in fragmented clusters leading to a lower recall and lower f-score.

For clusters in the lowest f-score ranges for both complete-link and single link, their associated clusters are always those with the smallest number of total pairs. Poor clustering seemed to be correlated with small-sized record classes, while good clustering correlated with large-sized record classes. Large-sized record classes composed of more pairs were more robust against pair-classifier misclassifications. Large-sized classes also contributed far more pairs than small-size classes to the training data. Our evidence seems to suggest that our classifier is fitting in favor of the largest sources of training data. However, there is also the possibility that the learned cluster metric is equally accurate on pairs from large and from small clusters but the *clustering algorithm* is more accurate at finding large clusters.



(a) single link clustering (b) complete link clustering

Figure 3-15: Ratio of misclassified pairs (Mean) of the *associated record classes* of clusters as a function of cluster f-score.

To understand the problem better, we looked at the ratio of misclassified pairs within the associated record classes of clusters as a function of f-score (figure 3-15). We definitely observe a trend in the misclassification ratios. Low f-scores are associated with a high ratio of misclassifications; high f-scores are associated with low ratio of misclassifications. We already now that small record classes are in the low f-score range, and large record classes are in the high f-score range. By deduction, small record classes must incur a large ratio of misclassifications, while large record classes must incur a smaller ratio of misclassifications. Hence, we have proof that record class size is a factor in generating "bad" (low f-scoring) clusters.

How do we intend to solve this size discrimination problem? One solution could be to give proportionally more weight to pairs from smaller record classes such that the weight of training data from all the classes are equal; this would be a possible future experiement.

3.7 Cluster Post-processing

After clustering, we apply several post-processing operations over our results to clean up clusters or filter away irrelevant clusters.

3.7.1 Singletons Filter

Clusters with only a single member instance do not (by our assumptions) constitute a pattern and therefore should be filtered. However, some singletons are part of a larger cluster but because we used complete-link criterion the high precision and low recall left them out. To recover some of these singletons, we employed a cluster merging processes that we will describe in section 3.7.5.

3.7.2 Overlapping Trees

In our discussion of tree clustering thus far, we have ignored the relationships among trees (with respect to the page tree). By ignoring relationships, clusters that contain trees that are descendant or ancestor to one another could form. Such tree pairs translate to an overlapping of two regions on a page, hence we call them *overlapping* trees. Record clusters should never contain overlapping trees because records are never overlapping.

To alleviate this problem, our implementation strictly enforces a constraint that the pairwise $\phi(T_i, T_j)$ value for any overlapping tree pairs receive an automatic value of 0. Furthermore, some of the features we used such as *tree overlap* or *tree context* form soft constraints that discourage pairing of overlapping trees. Despite these measures, overlap tree conflicts could still occur. Why and what causes them?

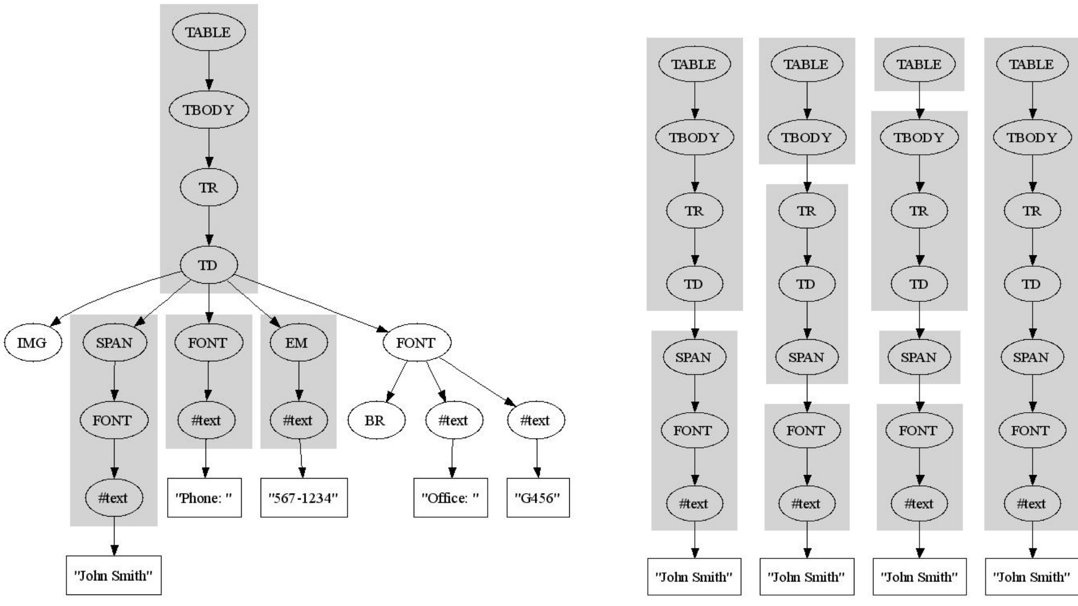
When a clustering criterion such as single link is used, overlap tree pairs could be inadvertently introduced into the same cluster via an effect that we call "chaining." We may initially have a cluster composed of two non-overlapping trees: T_a and T_b. A tree T_c that is a child of T_a, but that is non-overlapping with respect to T_b, can still join this cluster if $\phi(T_c, T_b)$ satisfied the clustering threshold. In this manner a cluster of three with an overlapping pair of trees T_a, T_c could form. The chaining effect is most prominent for single-link criterion but diminishes with k-link and completely disappears if we used complete-link; This is one more reason why we use complete-link over other types as our clustering criterion.

However, for completeness, we will describe two techniques we implemented to remove overlap conflicts from clusters for times when complete link is not used. The first method removes a special type of overlap trees and the second method does a more complete removal.

3.7.3 Tree Normalization

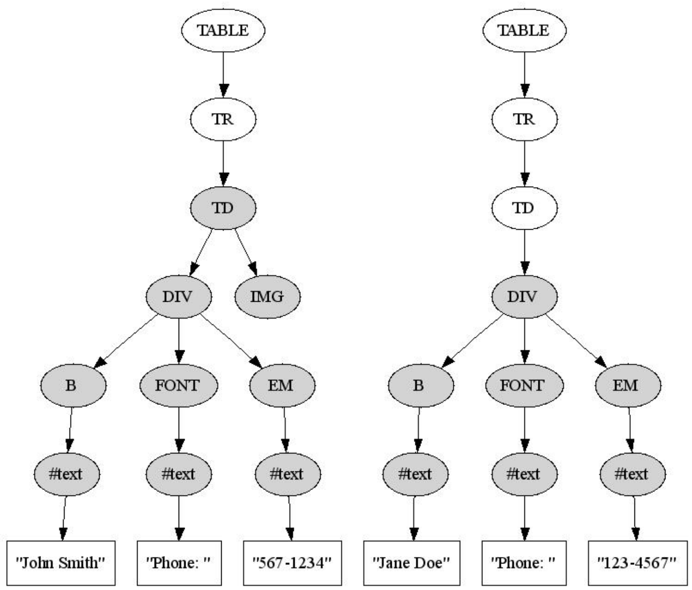
Many overlapping tree pairs are a part of a *piped* tree configuration (see figure 3-16). In fact, we have clusters where the members of each cluster are the same save for an extra node. Trees that are part of a pipe could be considered to be content-equivalent because they *cover* the same underlying leaf nodes. We give the name *most specific root* (MSR) to the tree that is at the deepest part of a pipe. The MSR is usually a node at a fan-out point or a leaf node. In figure 3-16(a), the `TD` node is the most specific root for the given tree. To normalize our clusters, we replace every tree in the cluster with that tree's most specific root. Once all trees and clusters have been thus transformed, we merge clusters that have membership overlap.[10] This normalization procedure guarantees that the resulting clustering does not contain two trees that cover the exact same content. This tree normalization procedure removes a large portion of overlapping trees, and also eliminates duplicate

[10]This membership merging may introduce more overlapping pairs. Therefore, we always apply the greedy algorithm described in section 3.7.4 to guarantee that no overlaps occur within each cluster.



(a) A piped tree. The shaded blocks are potential *supernodes*.

(b) Just four of the many ways (127) to segment this piped configuration into supernode blocks.



(c) Using most-specific-tree normalization before clustering may introduce problems of over-normalization. The shaded nodes depict the MSRs of the two trees. The two trees un-normalized are structurally similar, but their MSRs are not.

Figure 3-16: Piped Trees, Supernodes, and Most-Specific-Tree Normalization

clusters that describe the same record class.

Why normalize in the post-processing step? Why not normalize earlier? We have considered two strategies for normalizing trees prior to clustering.

One normalization scheme is to eliminate pipes by collapsing them into *supernodes*. Making such a change is non-trivial however. First it would require a re-definition of node-matching for "supernodes" in our tree edit-distance algorithm. A more difficult issue is that trees can have multiple supernode segment configurations (see figure 3-16(b)). If we choose the wrong segment configuration, metrics like tree edit distance could fail for the wrong reason. The *correct* supernode tree segmentation may actually need to be dynamically determined based on the structure of the two trees under comparison.

Another, even simpler solution is to cluster only using the most-specific-trees. However, one potential problem is that MSR may over-normalize. Consider the two trees depicted in figure 3-16(c), if we clustered them using their MSR trees (the shaded nodes) then the first tree will likely not cluster with the second. But if we leave them unnormalized, then the two trees are very structurally similar and would likely cluster together. In this case, MSR was too specific, and thereby picked a tree that no longer structurally resembled the original tree. So again, there are unresolved issues even with using a MSR-only clustering strategy.

Supernode and most-specific-tree (MSR) normalization are both potential solutions with still unresolved issues. We plan to experiment with them in our future work as possible replacements for our current solution.

3.7.4 A Greedy Algorithm For Resolving Overlapping Conflicts

Since tree normalization did not guarantee removal of all overlapping trees, we came up with a simple greedy algorithm that is a more foolproof alternative. First, we maintain a *conflict* table of all the overlapping tree pairs within a cluster. We sort cluster members by the number of conflicts in which they are involved. Then, iteratively, we remove the member involved in the most conflicts (overlapping pairs) and place it into a new cluster. If there are only two trees involved in a conflict, then the more ancestral tree is removed. After each iteration, we keep the conflict table up-to-date. We repeat this removal and update process until all the conflict counts in the table become 0. The resulting cluster is guaranteed to have no overlap conflicts. Finally, we check if the new cluster we created is also conflict-free; if not we apply the same algorithm recursively. This simple algorithm may not be optimal, however, we have found it to work well in practice.

3.7.5 Re-merging Clusters

One of the weaknesses of complete-link clustering (or any criterion that produces high precision) is that clusters fail to form if they are not cliques (or well connected). For this reason, our clustering

results often contain a large number of singleton, duplet, and triplet clusters; some are actually parts of record clusters. Our solution to this problem is to remerge clusters by using a less restrictive clustering criterion like single-link. Our cluster re-merging algorithm will merge any two clusters that satisfied all three of the following criteria. Merge two clusters A and B if:

1. There is no overlapping pairs between members of A and members of B.
2. Clusters A and B at least satisfy the single-link criterion.
3. The resulting merged cluster has higher *contiguity* than either A or B (separately) and that the increase is a maximum. Namely, there does not exist any B' such that the merged AB' cluster has a contiguity increase that is greater.

We will describe the contiguity factor in detail in the next chapter (section 4.2). For now, think of *contiguity* as a measure of content continuity between instances. When merging two clusters, contiguity will not increase if A and B are situated on different parts of the page. If two clusters are next to each other then their contiguity will increase by 1, and if they interleave with each other then the increase maybe more than 1. We did not include contiguity as a clustering feature because it is not a pairwise feature; contiguity can only be defined for clusters of trees.

3.7.6 Post-clustering processing results

In summary, our post-clustering processing consisted of the following set of steps:

1. Tree normalization
2. Resolution of overlapping tree conflicts using our greedy algorithm
3. Re-merging clusters by using the three stated criteria
4. Filtering out of singleton clusters
5. Filtering out of text-only clusters - tree normalization can produce text-only clusters. Since text-only trees were filtered out in pre-clustering processing we should also filter them out if we find them post-clustering.[11]

We measured the maximum precision, recall, f-score metrics for complete-link clustering before and after post-processing as a comparison, the results appear in table 3.4. There is a significant improvement in quality as a result of the post-processing. There is a tiny decrease in precision but a significantly large increase in recall, which can be largely attributed to the cluster re-merging

[11]Why did we not filter them out in the pre-clustering phase? These text-only clusters were produced as a result of tree normalization. They were normalized from root-to-leaf piped trees (*e.g.* the four trees in figure 3-16(b) are root-to-leaf piped trees). We did not remove them during pre-clustering for the same reason that we did not perform tree-normalization during pre-clustering; issues like over-normalization prevented us from doing so (see figure 3-16(c)).

clustering criterion	precision (stdev)	recall (stdev)	fscore (stdev)
complete-link	0.9750 (0.106)	0.8494 (0.2023)	0.8926 (0.158)
complete-link with post-processing	0.9632 (0.1215)	0.8902 (0.1819)	0.9139 (0.1548)

Table 3.4: Average maximum purity, recall and f-score of clustering results with and without post-processing.

process.[12] These results are encouraging and further experimentation with post-processing parameters may give us even better improvements.

3.8 Performance Optimizations

Having a good running time is crucial for a user interactive application such as ours. Also if we are considering using our system in a batch process to analyze and extract records from a large number of web pages then improving the speed of each extraction is very much worthwhile. There are mainly two general bottlenecks in pattern detection:

- The running time of each comparison: The running time is especially of concern for edit-distance based metrics such as tree alignment and sequence alignment (for tree context).
- The number of comparisons: We need to do $\frac{n(n-1)}{2}$ pairwise computations to populate the distance matrix. n can be large for some large pages, often in the thousands.

3.8.1 Tree Alignment Algorithm Optimizations

There are two general approaches to optimizing tree edit distance algorithms (and sequence edit distance algorithms). The first approach is to use a cheaper metric as a proxy for the true alignment score. If two trees had noticeable size differences (*e.g.* a 100 node tree versus a 10 node tree) we would not likely care about the accuracy of their alignment scores. Therefore, we could estimate the tree alignment score for such tree pairs by using a size approximation:

$$\alpha_{estimate}(T_1, T_2) = \frac{2 \cdot \min(\mid T_1 \mid, \mid T_2 \mid)}{\mid T_1 \mid + \mid T_2 \mid} \tag{3.22}$$

This estimate would be equivalent to the alignment where all of the nodes in the smaller tree aligned with those in the larger tree. By approximating we saved the work of actually running the tree alignment algorithm for a significant fraction of tree pairs. The approximation need not be as crude as a size estimate, we can even use other structural features such as Bag of Tags to estimate the alignment score.

[12] We repeatedly noted that precision (cluster purity) was important because of our plan to use the resulting clusters as training instances to Thresher. However cluster recall is still important when we evaluate the overall quality of the system - independent of possible applications.

The second approach is caching. Because of the recursive nature of the algorithm, when we compute the alignment of any two trees T_i and T_j, we are also recursively computing the alignment of descendant trees T_k and T_l respectively. If we cached the results for descendant alignments then computing the distance between T_k and T_l (later) becomes a constant time lookup.

What is cached will affect the amount of memory we use and the cache hit rate. The cache key could be: 1) the trees themselves or 2) the labels of the tree pairs. We call the former method tree caching and the latter method label caching. In tree caching, the entire tree (mapped to a unique integer id) becomes the cache key. In label caching the tree hash (as described in equation 3.8) is the cache key. Tree caching is necessary if our application needs to preserve the exact node-to-node mapping of paired trees, for instance when we want to generate a tree pattern template (as in Thresher[23]). Label caching should be used when only an alignment score α is required. Label caching is far more effective because of the finite number of possible HTML label trees. By caching by labels, we are computing the alignments of many (label-equivalent) trees at the same time. If we used tree caching, then the cache would have to be cleared for each page. However for label caching, the cache can actually be carried-on for use in processing subsequent pages.

Tree caching - because of its strict requirement of keeping everything - is consequently very memory intensive. Therefore, our tree cache implementation uses a LRU (Least Recently Used) cache entry replacement policy to keep the cache size capped at a fixed upper bound. In label caching, memory is not a bottleneck; most of cache entries are leaf node pairs and they stay in the label cache for a long time. The effective cache hit rate for label caching is very high, in the range of greater than 90%.

3.8.2 Reducing the number of comparisons

The other way to reduce running time of the system is by reducing the number of comparisons. The most direct way is to reduce the number of items we have to compare. For instance by removing trees that can not be record trees in our preprocessing stage (section 3.5.1) we reduced a very large percentage of trees. However, we still have on the average 500-1000 trees per page to compare. For these we use a second strategy commonly referred in the clustering literature as *blocking.* Blocking, as surveyed in [1], is the general technique of using a computationally inexpensive feature (usually one that has high recall) to decompose the input data into a number of blocks b. Then, a more expensive but more precise metric is used to compare only pairs of instances within each block. The resulting number of computations is $O(\frac{n^2}{b})$.[13] The major problem with blocking however is that it amplifies errors. When instances are separated into blocks if each block did not contain all members of a potential record class (low recall) then that error could easily propagate and become amplified

[13]This is assuming that the blocking procedure breaks the data into b roughly equal blocks. If blocking placed all instances into one huge block, the (worst) running time will still be $O(n^2)$.

in the final clustering results.

The literature also lists alternatives to standard blocking, for instance canopying[34]. Canopying is a form of fuzzy blocking, where any data instance could assume membership in more than one *canopy*. Each canopy is formed by choosing random instances as canopy centers and defining two thresholds, a tight and a loose threshold for each canopy center. If a point falls within the tight threshold then its membership is strictly within that canopy. If a point falls in between the tight and loose threshold then it is a loose member and can have loose membership within other canopies. Comparisons are computed for any pair of points that fall within the same canopy.

Sorted neighborhood is yet another blocking-based method for which trees are sorted by a feature like size. Then, instances within a window of size w are compared using a more precise metric. This method is somewhat reminiscent of our tree-size estimation optimization.

Implementing these and many other clustering optimizations would be interesting directions for future work and exploration. For our base implementation, we implemented a basic blocking algorithm using tree size as the blocking metric.[14] However we found that the errors produced by poor recall of the tree size metric were an unneeded distraction from our goal of improving the overall quality of the system. Therefore we did not activate these optimizations for our experiments. We plan to revisit these optimizations and accuracy versus speed issues in the future when we deploy the system in a large-scale setting.

[14]In retrospect, we believe that a tree-size metric canopying may make more sense since a tree should be allowed to cluster with other trees that are smaller and larger than it (and not just roughly equal in size to it).

Chapter 4

Record Detection

In pattern detection, we trained a pair-wise classifier and used it to cluster sub-trees into *patterns*: groupings of trees that are based on pair-wise structural, content, and contextual similarities. Record clusters are not the only clusters generated by pattern detection however, many clusters form out of patterns that serve roles like highlighting or layout. In fact, the majority of clusters from pattern detection are of non-record variety. Figure 4-1 shows a histogram of clusters produced by a complete-

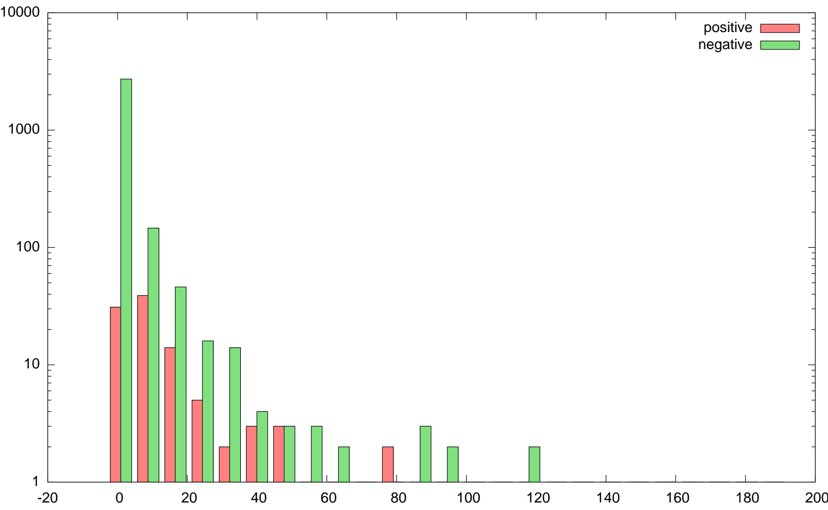


Figure 4-1: Distribution of clusters produced by pattern detection as a function of cluster size

link clustering distributed by the cluster membership size. The positive clusters, as shown by the red bars, are clusters that have f-scores greater than 0.80, these are our "acceptable" clusters. The negative clusters, as shown by the green bars, are clusters with no-overlap (f-score of 0) with labelled record classes. (Clusters with less than 0.8 f-score are not shown here because they are neither fit in as record clusters nor non-record clusters). This plot gives us a rough flavor of the quantities

of record clusters to non-record clusters. The ratio of non-record to record clusters shown is about 29:1. These odds are unacceptable for a user application that advertises to extract records without supervision. Our goal is to filter out as many non-record clusters as possible so that the ratio would be more reasonable (or ideally to have no non-record clusters at all in our results). The general approach to non-record filtering or *record detection* is similar to pattern detection, but instead of training a binary classifier over trees we train a binary classifier over clusters (or patterns).

$$\psi(C_i) = \langle \psi_1(C_i), \ldots, \psi_n(C_i) \rangle \rightarrow \{0, 1\} \tag{4.1}$$

Here, the target function $\psi(C_i)$, is a decision function over the space of *cluster features* $\psi_i(C_i)$. Each cluster feature component is a real valued function over a set of trees. The optimal $\psi(C_i)$ that we hope to produce is a *cluster classifier* trained using standard supervised learning techniques; it should return 1 if C_i is a record cluster and 0 if it is not.

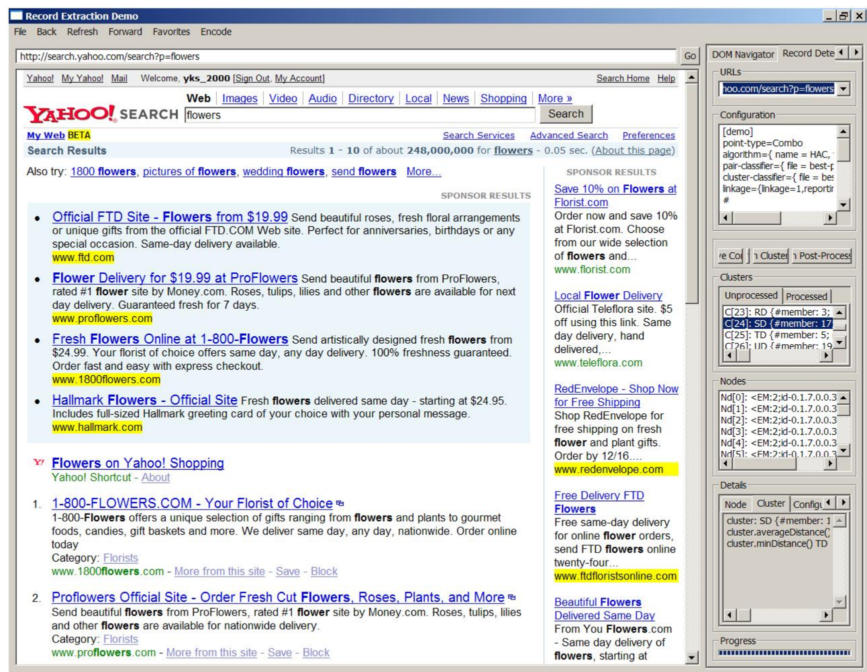
Also critical to our goal is that the cluster classifier must find record clusters with high recall. It need not be completely accurate because our target application has some tolerance for false positives but very little tolerance for false negatives. The occasional non-record cluster appearing in the results could be ignored (or removed) by the user; however, missing record clusters would require more work for the user to manually correct. If the user had to manually label missing records then she would have been better off using a supervised record extractor.[1]

4.1 Types of non-record patterns

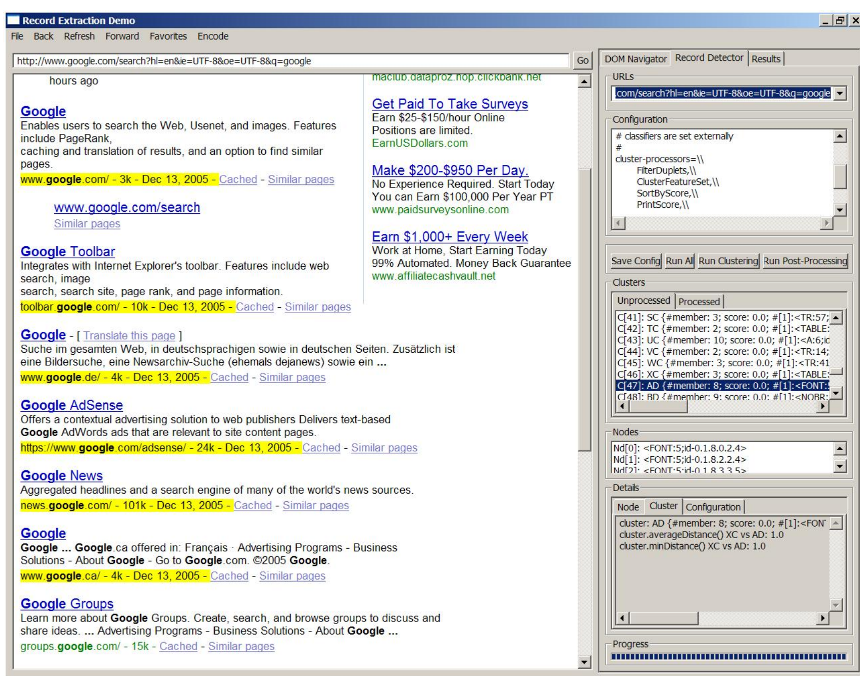
To understand what features make good ψ_i components we must first understand the general characteristics of non-record clusters. We manually analyzed a sample of clusters and identified some common patterns (figures 4-2 - 4-4 show these patterns highlighted in yellow):

- *Formatting patterns*: These are usually large clusters consisting of small trees. Typical cluster members would represent stylistic elements such as **bolding** or differing font styles. A common purpose for formatting patterns is for highlighting a specific type of data. Figure 4-2(a) shows an example where URLs are all formatted in the same way (highlighted in yellow).
- *Field patterns*: These clusters consist of members that could be construed as "fields" or *record fragments* within recognizable records. Field patterns will not always map to semantic field boundaries; some represent blocks of fields or record fragments within records. For instance, figure 4-2(b) shows a field pattern where three semantic fields: the URL, the page size and the date are all grouped into single *record fragment* instances.

[1] This goal for high recall is analogous to goals in information retrieval and web search; users are more accepting if the IR system returned a result set that contained the desired result but less accepting of systems that did not show the desire result at all - any where.

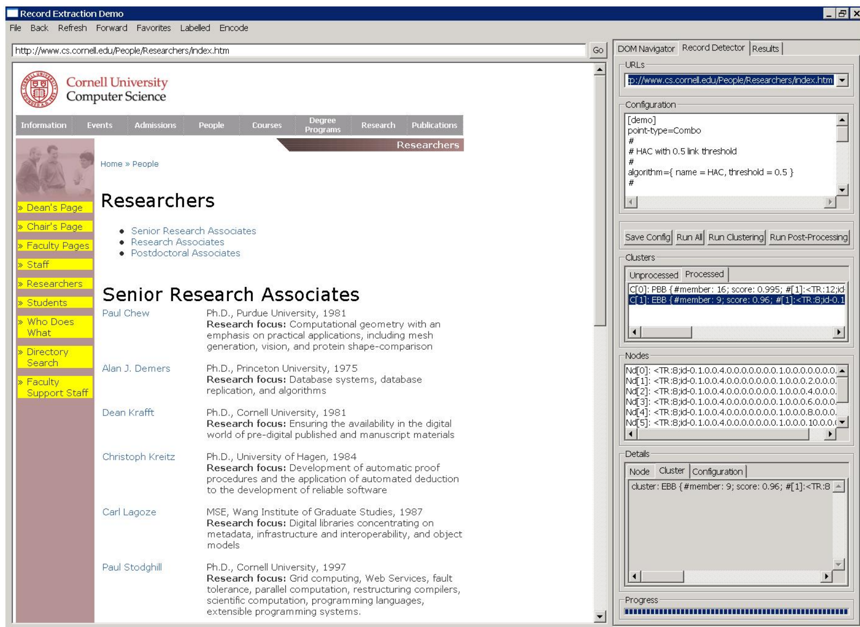


(a) Formatting pattern

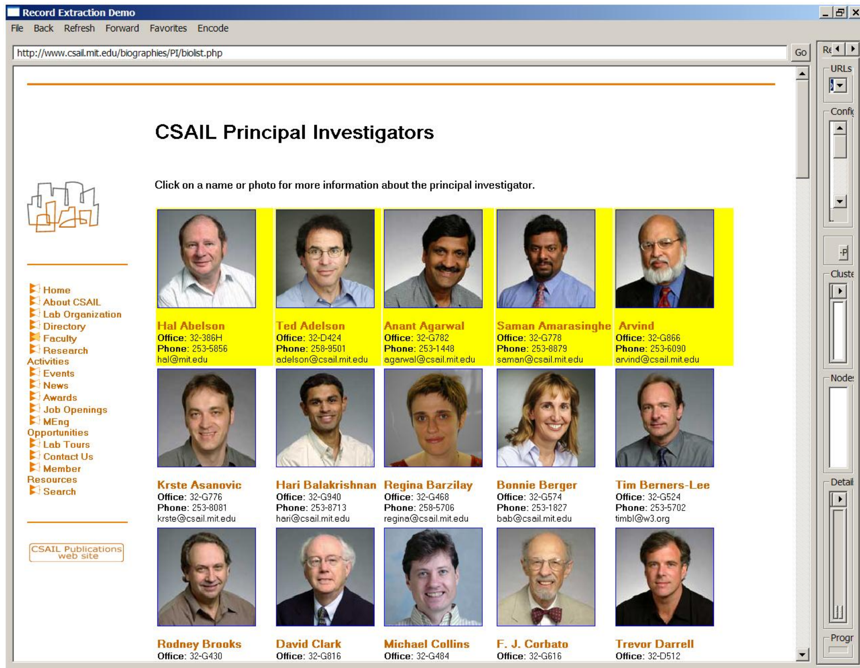


(b) Field pattern (encapsulating 2 fields)

Figure 4-2: Examples of different types of clusters. (Cluster instances are highlighted in yellow.)

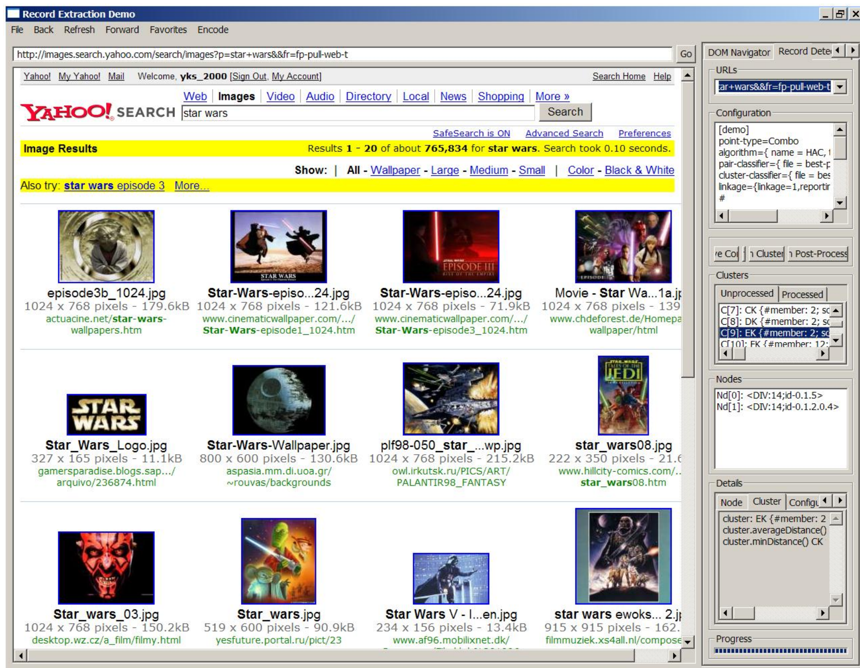


(a) Navigational pattern

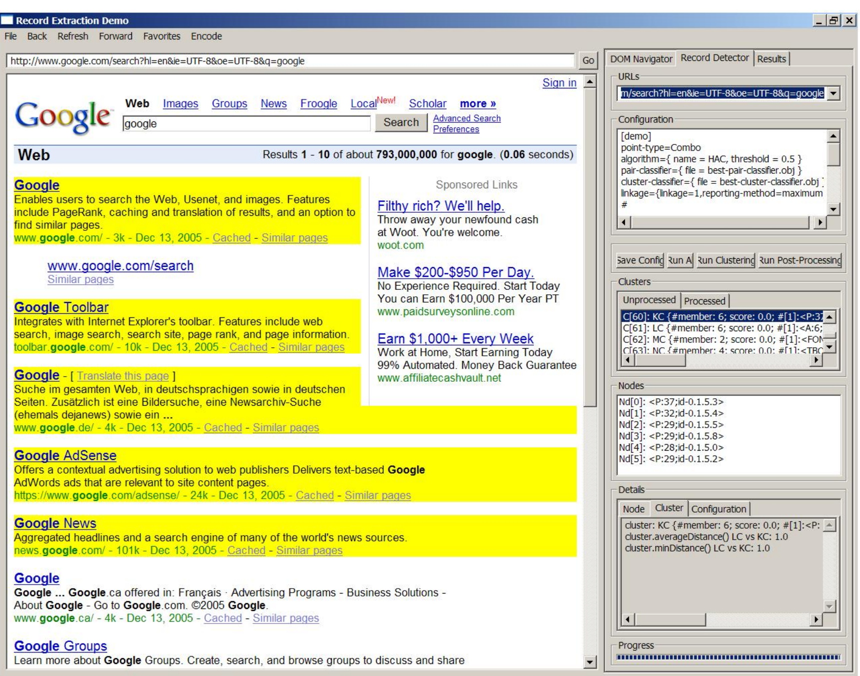


(b) Record Collection

Figure 4-3: Examples of different types of clusters (Continued). (Cluster instances are highlighted in yellow.)



(a) Segment pattern



(b) Record pattern

Figure 4-4: Examples of different types of clusters (Continued). (Cluster instances are highlighted in yellow.)

- *Navigational elements*: These are clusters composed of simple repeating elements that serve a navigational function such as a tabbed menu, browsing menu or a pager. Figure 4-3(a) shows a menu bar (top-left) where each menu item forms a pattern instance.

- *Record collections*: These are typically clusters with large trees. Record collection instances contain under them subtrees that are record instances. Record collection form out of necessity due to layout or organizational constraints. This type of pattern commonly appears as table rows when records appear in table cells. Figure 4-3(b) shows a single record collection instance: a table row that is part of a cluster of an 11-member cluster of row-based record collection instances.

- *Segment-based clusters*: Segment-based pattern are generally similarity based patterns that escapes clear categorization by function or purpose. These usually consist of large non-descript regions of content. Figure 4-4(a) shows an example of a segment pattern; both instances in the example are simply regions on a page that coincidentally share similar formatting. Segment-based patterns tend to contain very few members - usually no more than 3 instances.

- *Record clusters*: This is what we are looking for. Figure 4-4(b) shows an example record cluster.

Now that we have a picture of what we are trying to filter out and what we are trying to keep around, next we will consider possible features that will help us distinguish these pattern types.

4.2 Contiguity

Formatting patterns, field patterns, and segment patterns generally contain instances that are distributed continuously across a web page. Formatting pattern instances generally have a scattered distribution. Figure 4-2(a) shows a formatting pattern where some instances appear within records while other instances appear in the header of the page. Field patterns (such as the ones in 4-2(b)) are usually located within record instances so each field instance is almost guaranteed to be separated from the next field instance by some intervening content. Most segment-based patterns also appear separated such as the ones in the example (figure 4-4(a)).

In contrast to these three types of patterns, record patterns in general tend to be *well-connected*; there is generally no content in between consecutive instances. This notion of instance-to-instance continuity of content is what we term: *contiguity*.

We developed three related measures to formalize contiguity: filled-versus-cover ratio (*FilledCoverRatio*), contiguity indicator, and content-gap standard deviation. Before we can explain these metrics, we need to first clarify some definitions:

- *content gap* or *content distance*: $C_d(u, v)$ is a measure of the amount of content that lie between any two trees u and v (with respect to the page tree). This content does not include content enclosed by either tree. Furthermore, this measure is undefined for related trees.

- $fill(u)$ is our term for the amount of content (text or image) encapsulated by a given tree u.

- $cover(u_1, \ldots, u_n)$ is the term used to describe the overall amount of content covered by a set of instances u_1 to u_n on a page. It is equivalent to the sum of $fill(u_i)$s plus the sum of $C_d(u_i, u_{i+1})$s (the content gaps between consecutive u_i instances).

We further assume that all trees in a cluster can be ordered in a manner consistent with the content flow of a page. This ordering is usually specified by the way in which the content on the page is read, top to bottom and left to right for English. We assume that this ordering can be computed by visiting nodes/trees in a post-order traversal. When we say that two instances in a cluster, $u, v \in C$, are *consecutive* in this ordering then $C_d(u, v)$ has the smallest possible value; there does not exist any other $v' \in C$ where $C_d(u, v')$ is smaller.

Using these definitions, we can now very easily define our three contiguity metrics. First, filled-versus-cover ratio is the ratio of the amount of content filled by all the instances in the cluster versus the amount of space that is spanned or covered by all the instances.

$$fvc(C) = \frac{\sum_{t \in C} fill(t)}{cover(C)} \tag{4.2}$$

Patterns where all instances are contiguous have a *fvc* of 1. No pattern has an *fvc* of 0; but a highly scattered pattern with much less filled content than covered content will have a very low *fvc* value.

Next, the Contiguity Indicator is a metric that thresholds the content distance between consecutive instances. Any two consecutive instances where $C_d(u, v) \leq t$ are "linked" whereas consecutive instances that fail the threshold are "unlinked".

$$ci(C) = \frac{1}{n-1} \sum_{i=1}^{n-1} \begin{cases} 1 & c_d(T_i, T_{i+1}) \leq t \text{ [linked]} \\ 0 & \text{otherwise [unlinked]} \end{cases} \tag{4.3}$$

The contiguity indicator is essentially the ratio of the number of links between *consecutive* instances and over the maximum possible number of links possible (which for n instances is $n - 1$). The threshold t in equation 4.3) is in *content units*. Recall that one text character or 10x10 image pixels represents 1 content unit. For our experiments, we fix t to be 5 content units.[2] Contiguous clusters are expected to have an indicator value of 1; fully dispersed clusters have a contiguity indicator value of 0.

[2]However, a more reasonable t might be one that is fractionally proportional to the average fill (μ_{fill}) of cluster instances. For example, to achieve a 10% threshold we can set $C_d(T_i, T_{i+1}) \leq 0.1\mu_{fill}$.

Finally content-gap standard deviation measures the variance of content distance between consecutive instances:

$$cgstdv(C) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n-1}(c_d(T_i, T_{i+1}) - \mu_{c_d})^2} \tag{4.4}$$

Here μ_{c_d} is the mean content distance over the all consecutive cluster instances. With this metric we are measuring the amount of variance between the content gaps of consecutive instances. Completely contiguous clusters should have no content gap, and hence also zero content gap variance.

4.2.1 Algorithm for computing content distance

Essential to computing these three metrics is a general method for computing the content distance $C_d(T_i, T_j)$ (and consequently, $fill(T_i)$ and $cover(T_1, \ldots, T_n)$). First, we need to define an ordering of trees based on the content flow. During tag tree construction, we assign two numbers to each tree: a *minimum content number*: $C_{min}(T_i)$ and a *maximum content number*: $C_{max}(T_i)$. Content number is assigned starting from 0; the root has a C_{min} of 0 and a C_{max} equal to the content size of the entire page. Here content size is defined in content units (1 text character or 10x10 pixel of image). The content number is assigned as the page tree is traversed in a depth first fashion, the content number is incremented when a content-bearing leaf node is encountered, and the amount of increase is equal to the fill of the leaf node. $C_{max}(T_i)$ is assigned to a node when it is revisited as the depth-first recursion unfolds. In this manner, $C_{max}(T_i) - C_{min}(T_i)$ is equal to the amount of content encapsulated underneath T_i (or simply $fill(T_i)$). Given the $\langle C_{min}, C_{max}\rangle$ assignments of two trees, we can compute their content distance: $C_d(T_i, T_j)$ as:

$$C_d(T_i, T_j) = C_{min}(T_j) - C_{max}(T_i) \tag{4.5}$$

This equation assumes that $\langle C_{min}(T_i), C_{max}(T_i)\rangle < \langle C_{min}(T_j), C_{max}T_j)\rangle$. In other words, T_i and T_j's content numbers must not contain each other which implies the two trees must not overlap. We can guarantee that overlapping trees do not exist within a cluster by applying conflict resolution methods described in the post-clustering processing step of pattern detection (section 3.7). To compute consecutive content distances, we first sort all the trees in a cluster by their assigned $\langle C_{min}(T_i), C_{max}(T_i)\rangle$ then we compute all the consecutive pair-wise content distances by iterating over the sorted tree pairs.

Figure 4-5 provides a concrete example of the content numbering. In the example, the C_d between the first `<A>` and the second `<A>` subtree is 20 and the content distance between the two `<TD>` rooted subtrees is 0.

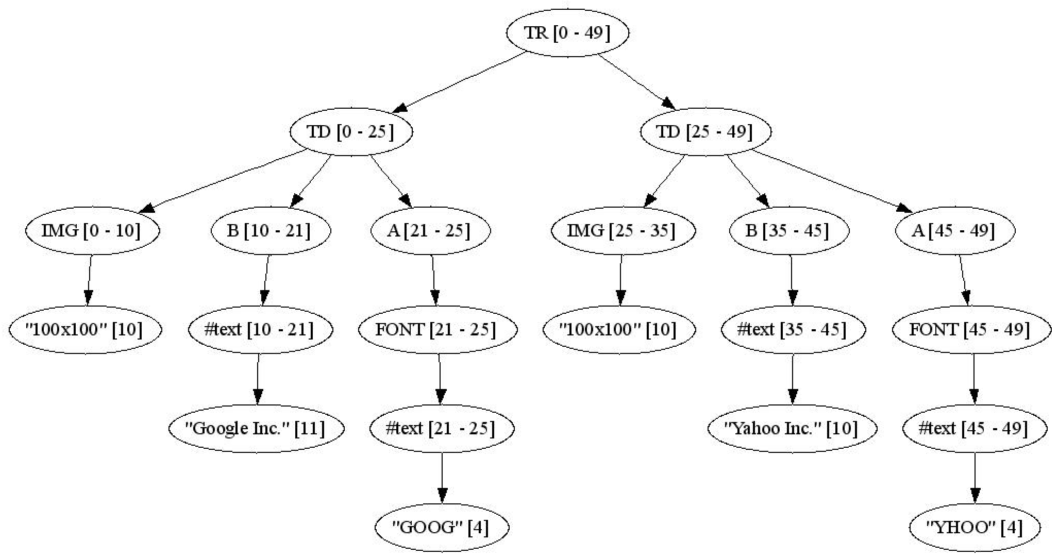


Figure 4-5: A tag tree with content numbering, the [*] number in leaf nodes denote the underlying content size.

4.2.2 Comparison of Contiguity-Based Metrics

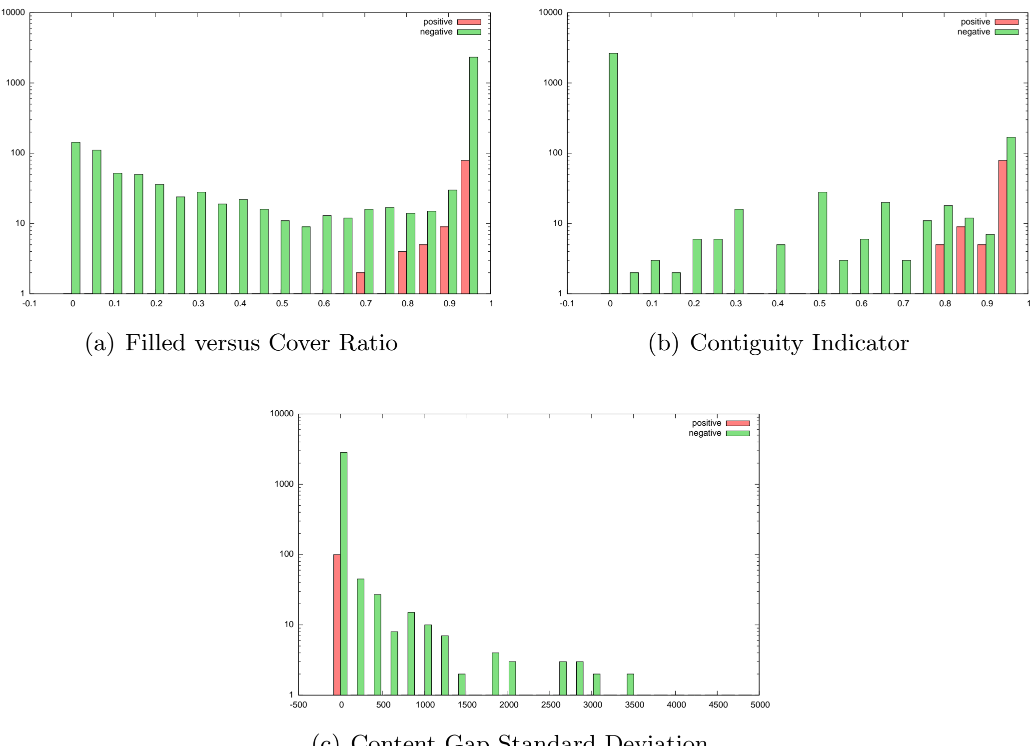
In figure 4-6, we plotted histograms of our training data distributed according to each of the three contiguity metrics. For both filled vs. cover ratio and contiguity indicator histograms, (figure 4-6(a) and 4-6(b)) most of the record clusters are aggregated near 1.0 and most of the non-record clusters are aggregated nearer the 0 end of the spectrum. This distribution agrees with our intuition that record cluster should be contiguous and most non-record clusters (the 3 types we discussed) are non-contiguous. However, there is also an equally large mass of non-record clusters near the 1-end which could be explained by the two types of patterns we haven't examined, record collection and navigational patterns. Like record clusters these two other pattern types also exhibit high contiguity.

For the content gap standard deviation histogram (figure 4-6(c)), both non-record clusters and record clusters are distributed near 0; this indicates that most clusters have low content distance variances. The non-record clusters with *cgsd* near 0 could come from navigational or record collection patterns; they could also come from field patterns. Field patterns, in general, have non-zero content gap *means* but near zero content gap variances because their instances are usually evenly spaced.

In table 4.1, we summarize the expected effect that each of these contiguity factors would have on the different types of clusters we identified.

4.3 Variation

Our analysis of contiguity-base features led us to the conclusion that we needed a different set of features to differentiate record collections and navigational patterns from records; both of which are



(a) Filled versus Cover Ratio

(b) Contiguity Indicator

(c) Content Gap Standard Deviation

Figure 4-6: Examples of different types of clusters (highlighted in yellow).

Type	FillCoverRatio	ContiguityIndicator	ContentGapStdev
Formatting	varies (mostly near 0)	mostly 0	high
Field Patterns	varies (but never 1)	mostly 0	near 0
Navigational	near 1	near 1	near 0
Record Collection	near 1	near 1	near 0
Segment	varies (mostly near 0)	some near 0	varies
Record	near 1	near 1	near 0

Table 4.1: A summary of expected effect of contiguity based features on various types of clusters

indistinguishable by contiguity-based features.

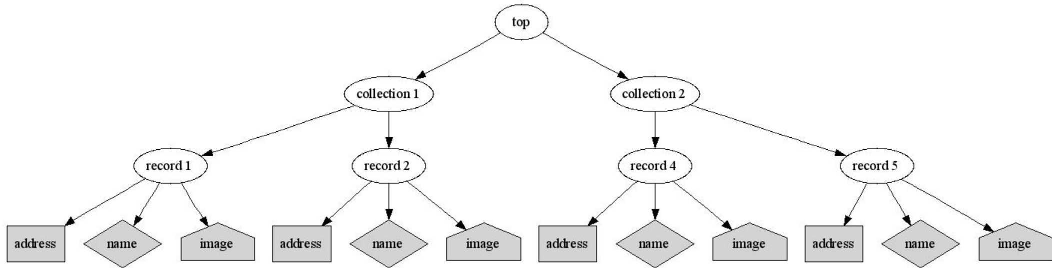


Figure 4-7: A tag tree with record collection, records and fields. Different shapes denote differences in formatting.

The content enclosed within a record has high formatting variation because consecutive fields of record instances are usually formatted differently. Figure 4-7 provides an example tag tree with both records and record collections. Looking under (or internal to) each record instance we see that it is composed of differently formatted fields: an address field, followed by (a differently formatted) name field, followed by an image. Each pair of consecutive fields f_i and f_{i+1} are different in their formatting. On the other hand, looking outside each record instance, we see that between consecutive record instances there is a great deal of similarity because each follows the same *ordering* of internal variations. Thus, we can describe record patterns as being internally varying but externally similar. This signature of internal heterogeny and external homogeny is definitely a discriminating feature for record patterns.

On the other hand, record collection patterns have low internal variation because all their children are record instances. They also have low *external variation* because they themselves form a regular repeating pattern. Therefore record collections have a signature of low external variation *and* low internal variation.

Navigational collections definitely have low external variation. But how about internal variation? Many navigational patterns are simple lists where each pattern instance is terminated at a single leaf. The pager on a Google search page (figure 4-3(a)) is an example of a *simple* navigational pattern. The variation of a single leaf node is zero since there is nothing relative to it for which we can define variation with. So simple navigational patterns will incur a low external and low internal variation signature. There are also *complex* navigational patterns that resemble record patterns. An example would be the menu on the CSAIL PI page[3], each menu instance has a small icon image followed by the menu item text. These complex navigational patterns have a record pattern like signature of low external and high internal variation. To differentiate these patterns, we will need to look to features that measure the amount of content a pattern covers (section 4.4). In table 4.2, we summarize the different types of variation-based signatures for the relevant cluster types.

[3]http://www.csail.mit.edu/biographies/PI/biolist.php

Type	Internal Variation	External Variation
Record	high	low
Record collection	low	low
Simple Navigational	low	low
Complex Navigational	high	low
Field Patterns	low or high	high

Table 4.2: A summary of expected variation-based signatures for relevant types of patterns.

We want to properly formalize variation and these variation-based signatures and use them as features for differentiating these cluster types. We shall denote $V(T_i)$ as this *variation* metric. However, keep in mind, that variation (the degree of heterogeneity) is the 1-norm inverse of homogeneity or regularity. Our metric, is not so much a single-sided measure of "variation" as a mapping of some characteristic of trees to a scale where on one end we have instances that are heterogeneous (variant), and on the other end instances that are homogeneous (regular). Our variation measure is defined over every tree so the variation metric for a cluster of trees is simply the average of the cluster member $V(T_i)$s.

4.3.1 What Is Variation Defined Over?

When we look at a page, if we ignore the content, what we see is a sequence of blocks; some are alike and others are different. If we assume that we can accurately assign labels to blocks so that alike blocks receive the same labels then, abstractly, for each tree there exists an assignment of labels. It is over this abstract sequence $(l_1, \ldots, l_n)$ that we define variation:

$$V(S) : l_1, \ldots l_n \Rightarrow \Re$$

The more heterogeneous this sequence is, the higher is its variation; the more homogeneous this sequence is, the lower is its variation. In section 4.3.6, we will describe our method for computing this sequence.

4.3.2 Analysis of Variation

The best way to quantify variation is to compare examples and determine what qualities makes one sequence more variant than another. Next we will illustrate 4 pairs of sequences. In each pair, the first is considered more variable (or less regular) than the second. Each pair illustrates an important quality of a desirable variation metric.

S1	`A B C D E F G H I`	Completely different labels
S2	`A A A A A A A A A`	All the same label

In our first example, S1 is a sequence where every label is different, and S2 is a sequence where every label is the same; these represent the polar extremes of variation. Any variation measure should be able to differentiate these two extremes. A simple metric that would work would be a count of label-types. A sequence with the greater number of label-types would reflect more variability. However, by only looking at the number of label-types, we ignore any frequency information associated with label-types.

S3	`C A A B A C A B A`	3 types of labels, with distribution A:5, B:2, C:2
S4	`A B C A B C A B C`	3 types of labels, distribution A:3, B:3, C:3

S3 and S4 show two examples where the counts of label-types are equal, but S4 appears more regular. From this second example, it would appear that not only is the label-type important but also the *distributions* of label-types. If the label-type distribution is uniform, namely the counts of As, Bs, and Cs are same (or similar), then the sequence is more regular as we have in S3. However, if the label-type distribution is erratic, *e.g.* with different numbers of As, Bs, and Cs, then the sequence is more variable as we have in S4. A good variation metric should assign S4 a lower variation than S3. So the lesson is, the more *uniform* the label distribution is the less variable the sequence is and vice versa.

S5	`A B C C B A B A C`	No clear pattern in ordering
S6	`A B C A B C A B C`	A strict ordering of `ABC`

The major problem with a single-label distribution is that it ignores the order of the labels; *i.e.* each label is independent of that label's neighbors or other nearby labels. Consider S5 and S6, both have the same uniform single-label distribution but qualitatively S6 appears more regular. It contains *repetitions* of A B C, while S5 is absent of such. Hence, a good metric must also be able to account for label order.

Naturally, label order can be factored in if we looked at the distribution of *consecutive label pairs*, namely bigrams. A homogeneous distribution of bigrams tells us that the sequence will likely contain many repetitions of *two* consecutive labels. However, it makes no guarantees about any longer or more complex repetitions. For example, consider the sequence:

```
B A A A A B A A A A B A A A A
```

This sequence does not have a uniform bi-gram distribution but it does have a more uniform 5-gram distribution. Instead of looking at distributions we should really be looking for *subsequence repetitions*.

In fact, we are not just looking for any kind of regularity but a specific form that conforms to our notion of *record-ness*. Consider the following pair of sequences:

S7	`A A A B B B C C C`	Repetition of A covers only 1/3 of the sequence
S8	`A B C A B C A B C`	Repetition of ABC "explains" the entire sequence

Both sequences have the same label distribution and both are "regular" in their own way. However, S8 is more regular from a "record-like repetition" sense of regularity.

Why are we distinguishing record-like repetitions? Our goal is to be able to distinguish record collections from records. Records should **not** have internal to them *record-like repetitions*. Record collections, on the other hand, internal to them record-like repetitions. By discriminating for record-like repetitions, our variation metric will be able to distinguishing records from record collections.[4]

What exactly is this record-like repetition sensibility? We had made an assumption in our introductory chapter that record instances are flat, with the general form: $(f_1 \ldots f_n)$. A *record-like repetition* is a sequence made up of juxtaposed record instances; the repetition of fields $(f_1 \ldots f_n)$ in the record instance correspond to the repetition of labels $(l_1 \ldots l_n)$. In other words, sequences that are record-like repetitions, obey a *record grammar* of the form:

$$\begin{aligned} S &\Rightarrow RS \\ R &\Rightarrow f_1 \ldots f_n \end{aligned}$$

If a sequence can be shown to be generated from a specific instance of this grammar then that sequence would be considered a record-like repetition. S7 is not a record-like repetition. While the subsequence `A A A` repeats, that repetition only explains 1/3 of the sequence. S8 is a record-like repetition (corresponding to a record grammar of $R \Rightarrow ABC$) and its subsequence repetitions explain the entire sequence. Record-like repetitions require that those repetitions must cover a significant portion of the entire sequence.

We summarize our qualitative observations:

- Variability is directly proportional to the number of different label-types, more label-types more variability.

- The uniformity of the label distribution determines how regular a sequence is. Conversely, the more erratic the label distribution is, the more variable the corresponding sequence is.

[4] One may argue that internal to record collections we should only see a sequence of juxtaposed record instances. So the sequence under a record collection instance should be: a sequence of single repeats, e.g. AAAAA where each A is a record; hence, looking for longer than length one record-like repeats would seem unnecessary. However there are other surface features that may be included into the sequence. For instance, a common web page artifact is the "spacer", an element of HTML that is inserted between records to produce a padding effect. When spacers occur, the sequence under a record collection could become: ABABABABAB, where B is a spacer. Spacers may also be composed of multiple elements, for instance a space plus a new line or horizontal rule *etc*; so the sequence could become repeats of ABC or longer. For the reason of robustness against all such possible sequences, we need to not only look for single repeats but also longer subsequence repeats.

- The distribution of single-labels is insufficient to distinguish different cases of variability. We also need to look into the ordering of labels. The ordering of labels can be accounted for by looking at higher order subsequence distributions: bigrams, trigrams etc.
- Record-like repetition is a desirable form of regularity. If a sequence that can be shown to be generated from a record-grammar, then it would be considered record like. The criteria for record-like-ness involves finding the repeating subsequence that satisfies the record-grammar but also most "explains" the sequence.

Our ideal variation metric should capture all, if not most, of these qualities.

From these stated observations we can come up with a multitude of metrics, however to limit our scope we implemented and evaluated three metrics: periodicity, compression ratio, and average pair-wise dissimilarity.

4.3.3 Periodicity

Periodicity embodies the record-like repetition quality that we described earlier. This metric employs a search strategy to find a repeating subsequence that covers most of the target sequence. All sequences have a finite number of subsequences; so for a sequence S we can generate a finite set of all possible n-grams: G.[5] Our measure is a search within G for the best repeat that satisfies the following equation:

$$V_{srr}(S) = \min_{g \in G} \frac{\mid g \mid + UnCover(S, g)}{\mid S \mid} \tag{4.6}$$

Uncover(S, g) computes the portion of S that is *not* covered by g. Intuitively this minimization is finding whether a repeating g whose repetition accounts for a large portion of S. It tries to minimize the length of g as well as the part of S that is not covered by g. A record-like sequence, would be assigned a value: $\frac{|g|}{|S|}$. A sequence with no-repetitions would be assigned a value of 1.

For illustration here are the V_{srr} value for the sequences we gave as examples:

S1	`A B C D E F G H I`	9/9	No repeat
S2	`A A A A A A A A A`	1/9	Repeat of A
S3	`C A A B A C A B A`	(1 + 4)/9 = 5/9	Repeat of A
S5	`A B C C B A B A C`	(2 + 5)/9 = 7/9	Repeat of AB
S7	`A A A B B B C C C`	(1 + 6)/9 = 7/9	Repeat of A
S8	`A B C A B C A B C`	(3 + 0)/9 = 3/9	Repeat of ABC

The two record-like sequences S2, and S8 naturally receive the lowest periodicity scores.

Another way to interpret this minimization goal is to think of it as a compression strategy. For the repeating subsequence g, we only need to store it once. All portions of the sequence not explained

[5] G is computed by enumerating 1-gram, 2-grams *etc*to $\mid S \mid$-gram (or the entire string).

by g must be stored uncompressed. The quantity that we computed as our variation metric is really the compression ratio of the sequence under our specific compression scheme.

4.3.4 Compression

Periodicity hints at looking at variation as a form of compression or finding the minimum description length of the sequence. A sequence that is highly variable is hard to compress because there is no redundancy to exploit. A sequence that is highly redundant could be easily compressed. Our variation metric could therefore simply be expressed as a compression ratio:

$$V_{compression} = \frac{\mid S_{compressed} \mid}{\mid S \mid} \tag{4.7}$$

But what compression scheme should we use? Finding the minimum description length of a string is known to be undecidable; this problem is also known as the Kolmogorov complexity of a string [33].[6] But it can be approximated using known algorithms like Lempel-Ziv encoding, which is implemented in GZip and Zip (for file compression). The compression ratio metric is literally computable by running gzip over our target sequence!

However, from our examples: S7-S8, we know that optimal compression does not actually guarantee record-like repetitions.

S7	`A A A B B B C C C`	Repetition of A covers only 1/3 of the sequence
S8	`A B C A B C A B C`	Repetition of ABC "explains" the entire sequence

S7 is clearly compressible but is not record-like.

Our implementation of the compression-based metric used GZip. In our implementation, the actual compression ratio was computed as:

$$V_{compression} = \min(1, \frac{\mid S_{compressed} \mid}{\mid S \mid}) \tag{4.8}$$

We capped the largest value of $V_{compression}$ to be 1 in because we needed to compensate for GZip's encoding overhead. We observed that for short sequences, GZip will always return a compressed sequence that was longer than the uncompressed sequence. The overhead is around 10 bytes for the Java version we used (`java.util.zip.GZIPOutputStream`). Any sequences with length longer than 10 bytes would automatically receive a compression score of 1.

[6]The Kolmogorov complexity is the length of the shortest description of that string computable by a Universal Turing machine.

4.3.5 Mean Pair-wise Dissimilarity

Our last metric computes the average *dissimilarity* between consecutive pairs of our abstract sequence. We consider consecutive pairs instead of all pairs because under all pairs the distribution of all pairs would be the same for two sequences with the same (single) label distribution. Consider examples S5, S6:

S5	`A B C C B A B A C`	consecutive pairs: AB: 2, BC: 1, CC: 1, CB: 1, BA: 2, AC: 1
S6	`A B C A B C A B C`	consecutive pairs: AB: 3, BC: 3, CA: 2

Under all pairs, both sequences would have the same distribution of AA, AB, AC ... *etc*; hence the averages of all pairs would be the same. But bigram distributions (as shown) would be different and would be distinguishing for the two sequences.

The mean pair-wise dissimilarity metric we implemented takes the following form:

$$V_{md}(T_1 \dots T_n) = \frac{1}{(n-1)} \sum_{k=1}^{n-1} (1 - \phi(Ch_k(T_i), Ch_{k+1}(T_i))) \tag{4.9}$$

This definition is over a sequence of trees that correspond to the labels of our *sequence*. Here $\phi(\cdot,\cdot)$ is a normalized pair-wise structural similarity metric such as Tree Alignment. We could also have defined dissimilarity based on label dissimilarity. But since labels can only be the same or different $\{0,1\}$, the resulting values are not very interesting. When we replace dissimilarity with a [0,1] real value as returned by Tree Alignment, the dissimilarity thus formulated would also account for the *amount* of dissimilarity.

4.3.6 Defining The Sequence

The *sequence* on which we compute variation has thus far been an abstraction. We compute the *sequence* as a labelling corresponding to the cluster assignment of the descendant trees. Any two descendant subtrees that are part of the same cluster would be assigned the same label (and those of different clusters different labels). Note that we cannot simply use the descendant tree root labels because the root labels tell us nothing about the similarities of underlying trees. Clustering-based labelling guarantees (some degree of) correspondence between same labelled trees.[7] Formally, the sequence for a tree T is defined as:

$$S(F(T)) = \left\{ (l_0 \dots l_n) | l_i = \begin{cases} clusterLabel(Ch_i(T)) & Ch_i(T) \text{ is part of a cluster} \\ nodeLabel(Ch_i(T)) & Ch_i(T) \text{ is a leaf and not part of a cluster} \\ S(F(Ch_i(T))) & \text{otherwise} \end{cases} \right\} \tag{4.10}$$

[7]Because our sequence is cluster based it could not have been computed during the pattern detection phase.

The *sequence* is constructed such that the labels are recursively obtained; the label of an unclustered child is recursively defined over that child's descendants (case 3). Leaf trees, if not clustered, carry the label of the node itself (case 2). S is either a cluster label or the label of a leaf node.

4.3.7 A Technicality Involving Piped Trees

Because we have overlapping trees, many trees will end up covering the same content region. Specifically, trees that are in a piped configuration will share the same content. Because of this sharing, variation is not computed for every subtree; subtrees that cover the same content will also share their variation score. Formally, the variation for any tree takes the following form:

$$V(T_i) = \begin{cases} V(F(T_i)) & \mid F(T_i) \mid > 1 \text{ [branching point]} \\ 0 & \mid F(T_i) \mid = 0 \text{ [leaf]} \\ V(Ch_0(T_i)) & C_d(T_i) = C_d(F(T_i)) \text{ [piped tree]} \end{cases} \tag{4.11}$$

Variation is only a useful measure if a tree has more than one child. Variation changes only at two types of places in the page tree: at the branch-out nodes and at the leaf nodes. Fan-out nodes are covered by the first case and leaf nodes by the second case in equation 4.11. So when a tree is part of a pipe (see figure 4-8), each enclosing tree sees the same amount of content as the next tree in the pipe. Variation of trees that are part of a piped tree is covered by a recursive call.[8]

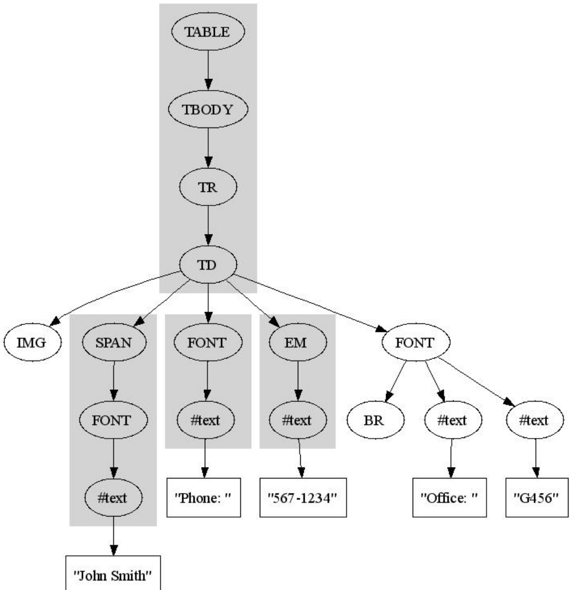


Figure 4-8: An example piped tree. All sub-trees that are in piped configurations (that are in the grey region) will have the same variation measure value as the nearest branching point or leaf node.

[8]Note: this recursive definition of variation becomes unnecessary once we implement a tree-normalization scheme as described in section 3.7.3.

4.3.8 Variation Difference

Having defined two suitable variations metrics, we return to the original task of formalizing "minimized external variation and maximized internal variation." We can rephrase this pattern as a single maximization goal: maximizing the internal variation and the *negation* of external variation. Thus, we have the *variation difference* metric δ:

$$\delta = V_{internal}(T) - V_{external}(T) \tag{4.12}$$

$V_{internal}(T)$ is simply the variation computed over the given tree. $V_{external}(T)$ is the variation value of the tree's parent (or ancestor for a piped tree). *Record collection* patterns as we postulated (see table 4.2) have both low internal and low external variation so it should have a near zero δ. Record patterns, on the other hand, should have a positive δ since they are expected to have high $V_{internal}$ and low $V_{external}$. Field patterns should have high external variation because they are part of a record. However their internal variation could be low if it is at the leaf level, or high if it is a record fragment *i.e.* a grouping of fields. So field patterns would have a δ from near 0 to negative.

We will examine the distribution of this difference in the next section.

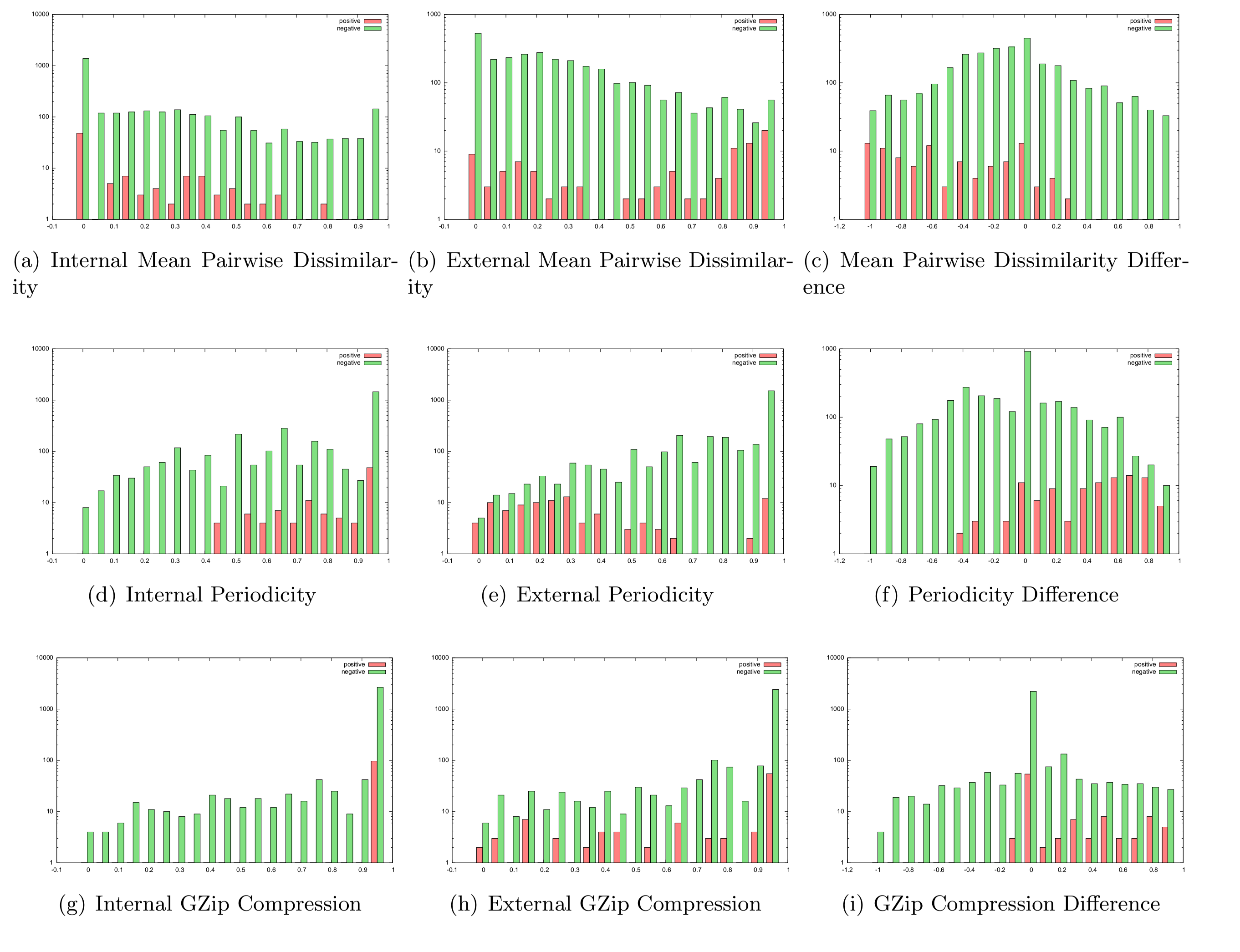
4.3.9 Distribution of Variation Features

Figure 4-9 shows the distributions of the training data under the various formulations of our three metrics: periodicity, gzip compression, and mean pairwise dissimilarity. For each type of variation metric we have three versions corresponding to external variation ($V_{external}$), internal variation ($V_{internal}$), and variation difference (δ).

From the distribution analysis, we could predict that pairwise-dissimilarity would likely be a poor feature (see figures 4-9(a)-4-9(c)). The distribution of positive and negative instances does not appear to be distinctly separated by any of the pairwise-dissimilarity derived features.

Periodicity, on the other hand, shows a promising pattern of positive/negative class separation. Internal periodicity (figure 4-9(d)) distributes much of the record clusters (positives) towards the 1-end (more variation). External periodicity (figure 4-9(e)) distributes the mass of the record clusters towards the 0 end of the spectrum (low variation). As a result, periodicity difference distributes positive instances mostly in the greater than 0 region. These distributions match our expectations that record clusters should have, in general, positive variation differences.

The GZip compression feature distribution had mixed results. Because of the compression overhead, the majority of values are cumulated at the 1-end. For External GZip compression, the distribution of positive and negative instances almost appears random. But for internal Gzip compression, all the positive instances are at the 1-end. This suggests two things. One, record instances all have a relatively short *sequences*. Two, internal Gzip compression is a good filter for



(a) Internal Mean Pairwise Dissimilarity

(b) External Mean Pairwise Dissimilarity

(c) Mean Pairwise Dissimilarity Difference

(d) Internal Periodicity

(e) External Periodicity

(f) Periodicity Difference

(g) Internal GZip Compression

(h) External GZip Compression

(i) GZip Compression Difference

Figure 4-9: Comparison of variation metrics

negative instances because all the positive instances are all distributed at 1.0 under this feature.

From these histograms, we predict that periodicity would likely be our best performing variation-based feature.

4.4 Other Features

In addition to variation-based and contiguity-based features, we also considered features that are based on more standard statistics of clusters, trees, and content. Amongst the features we also considered:

- Tree size mean and standard deviation
- Cluster size: the number of members in each cluster.
- The log tree size and standard deviation
- Content size mean and its standard deviation
- Content coverage:

$$ContentCoverage(C) = \frac{\sum_{i \in C} fill(i)}{fill(root)}$$

  This is the ratio of content filled by instances in the cluster C with respect to the content size of the entire page. This feature is derived from the intuition that record clusters should contain most of a page's content. This feature should help us differentiate complex navigational clusters from record clusters.

In addition to these features we also considered two different modifications over some of these features.

- *Ranked* is the transformation of a feature into a ranking. We rank the clusters by the value of some given feature, the resulting assigned ranking then become the actual feature. Clusters that tie on some feature value are given the same rank.
- *Inverse Ranked* is simply $\frac{1}{Rank}$. Inverse ranking has the property that as rank increases the separation between successive ranks becomes smaller. This follows the intuition that the difference between items ranked 1 and 2 is important but the difference between items ranked 100 and 101 is less important.

The class distribution histograms of these features appear in figure A-5.

4.5 Cluster Learning

Including all the feature modifications, we had a total of 30 features in our cluster-classification feature set.

4.5.1 Training Data

To generate our training data, we considered three options.

1. *Using the labelled data directly*: In our labelled data set, we labelled record clusters as well as many non-record clusters, some imported from clustering results and others manually constructed. We could simply use the labelled record and non-record clusters as the positive and negative examples for our training set. However the potential risk is that the manually labelled clusters may not have the same distribution as the clusters produced by pattern detection.

2. *Using unlabelled data*: We could take the clusters generated by pattern detection and annotate a labelling over these clusters. We could take the raw clusters and compute their content overlap with our labelled clusters. If the content overlap is greater than 80% then the cluster becomes a positive example. If the cluster had no overlap with a labelled class then the cluster becomes a negative example. Clusters with greater than 0% and less than 80% degrees of overlap are ignored. Labelling these clusters (that contain some record and non-record instances) either way would create a bias.

3. *A mixed approach*: We could throw away all raw clusters (generated by pattern detection) that have any overlap with labelled classes. To the resulting list of non-record only clusters we add back the labelled classes. In this manner the positive examples are simply the labelled classes but the negative examples are non-record clusters derived via clustering. This way at least the negative clusters would have the same distribution as produced by pattern detection.

After carefully considering these options, we decided to construct training data using purely unlabelled data (method 2). We felt strongly that the training data should have a distribution as close as possible to the actual input, and method 2 generated data closest to that distribution. We used our baseline pair classifier with a complete-link clustering criterion (with post-processing) to generate our training and testing data set.

4.6 Evaluation

Weighting training data

We were surprised on our first attempt at training a cluster classifier using our data set. The trained classifier returned the same error rates for any combination of features. After some detailed investigation, we discovered that the recall of the trained classifier was zero. The cause was that the negative examples greatly outweighed the positive ones. The classifier sacrificed recall for accuracy by simply misclassifying all positive examples. (In addition, these classifiers were also useless because they assigned 0 weights to the given feature.)

In pattern detection, we solved a similar problem by sampling the negative data to compensate for positive/negative class skew. However, in record detection we have a limited amount of training data[9]. So instead, we reweighed our training data; we gave the sparser positive examples much more weight, 30 times more weight than the negative examples (as discussed in this chapter's introduction the positive to negative data ratio was roughly 1:29). The weight is slightly biased towards positive data since we wanted the resulting classifier to favor misclassifying negative instances over misclassifying positive instances.[10] In this manner, our classifier will always achieve high recall.

4.6.1 Individual Features Performance

As our first experiment, we trained a linear SVM classifier to evaluate the performance of each feature individually. While we expected that the best classifier will likely consist of some combination of features, we felt that it was important to compare the usefulness of features individually.

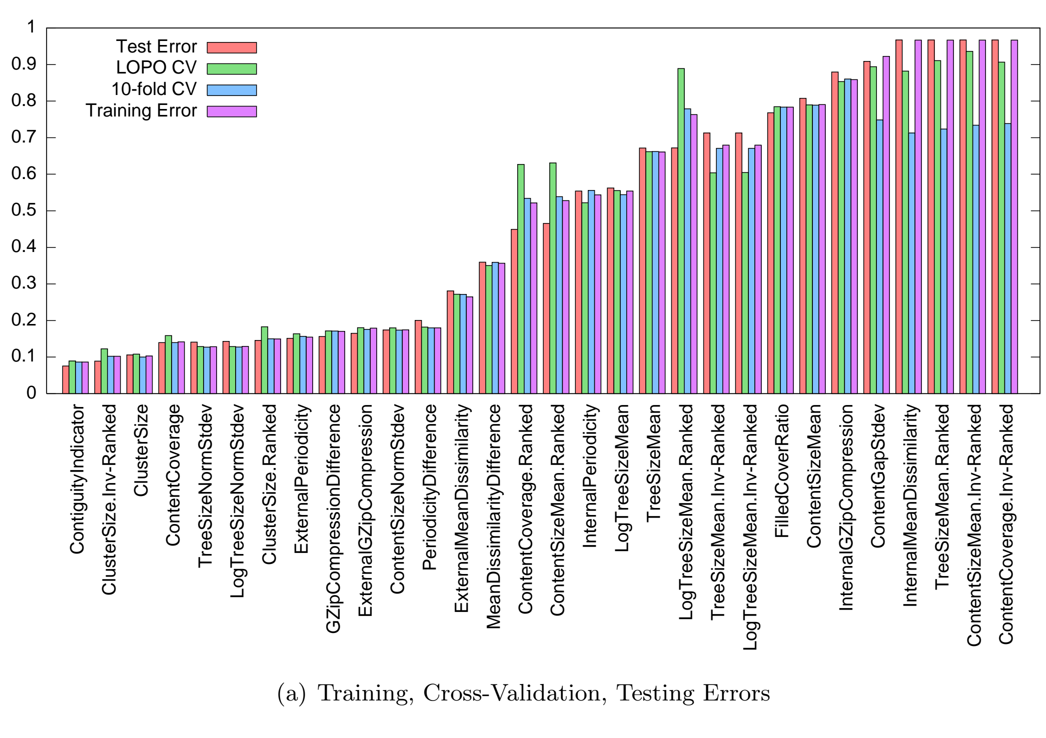
Figure 4-10(a) summarizes the errors (LOPO and 10-fold cross validation, test and training), and figure 4-10(b) summarizes the precision recall for all 30 features. Table B.6 contains the corresponding raw data.

In the precision and recall graphs (figure 4-10(b)) note that most features reported high recall rates due to our weighting scheme. We summarize some key observations from these graphs:

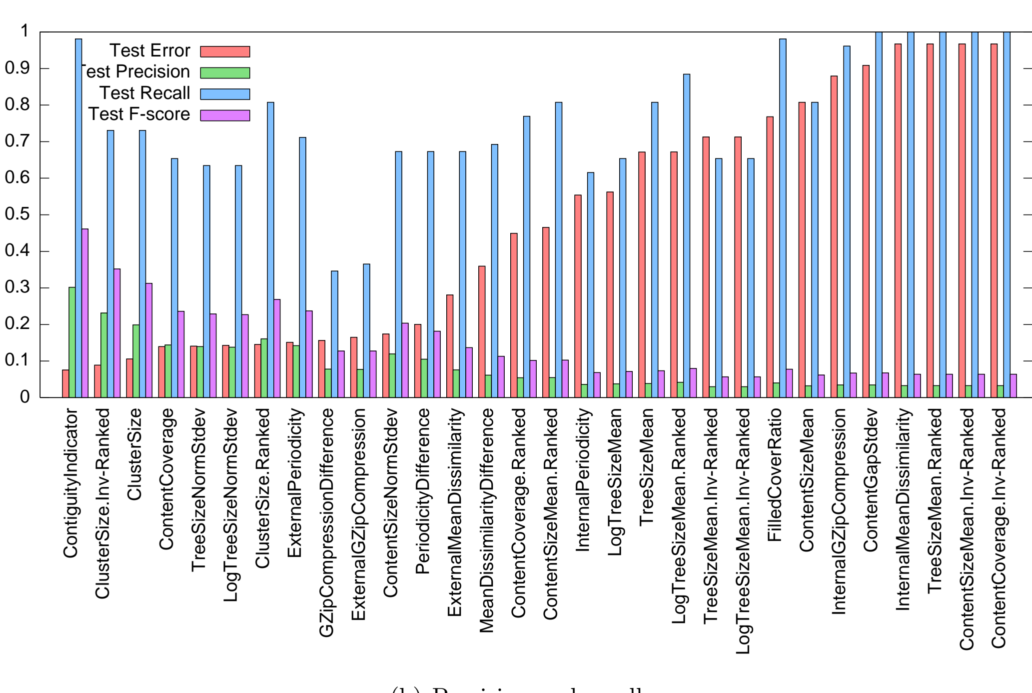
1. No single feature appear to perform very well (or nearly perfect) in this record detection task. Even the lowest achieved test error (by Contiguity indicator) was high around 10% error rate.
2. Features that incurred less than 30% error, could be grouped:
    - Contiguity based (Contiguity indicator)
    - Periodicity based variants (external periodicity , and periodicity difference)
    - GZip compression based variants (GZip compression difference, external GZip compression)
    - Standard deviation derivatives of Log tree size, tree size and content size
    - Variants of cluster size: cluster size (inverse ranked), cluster size, and cluster size (ranked)
    - Content coverage
3. For contiguity-based features, the contiguity indicator proved to be the best performing whereas FilledCoverRatio and ContentGapStdev both stood at the lowest performing end of the spectrum.

[9] We had about 1000 training points rather than 10,000 for pattern detection.

[10] For some classifiers, like linear regression, it is also possible to adjust the recall by changing the decision boundary threshold.



(a) Training, Cross-Validation, Testing Errors



(b) Precision and recall

Figure 4-10: Comparison of individual cluster features using a linear SVM classifier, sorted by test error

4. For variation-based features, the external variation metrics out-performed internal variation metrics. In terms of external variation, periodicity was the best, followed by gzip compression, and pairwise-dissimilarity came last. This ordering matched our predictions from examining the data histograms.

From these observations, it would appear that we could not rely on any single feature to carry us all the way; We will need to look at how combinations of features performed in order to make a final judgment on what optimal subset of features to use for record detection.

4.6.2 Comparison of Classifiers

As our second experiment, we compared different types of classifiers on an all-features set. Our goal was to determine whether the type of classifier used had any significant impact on performance. Figures 4-11(a) and 4-11(b) summarize the error rates and precision/recall results of our experiments over eight classifier types with all 30 features. The original data can be found in table B.7. We summarize our key observations:

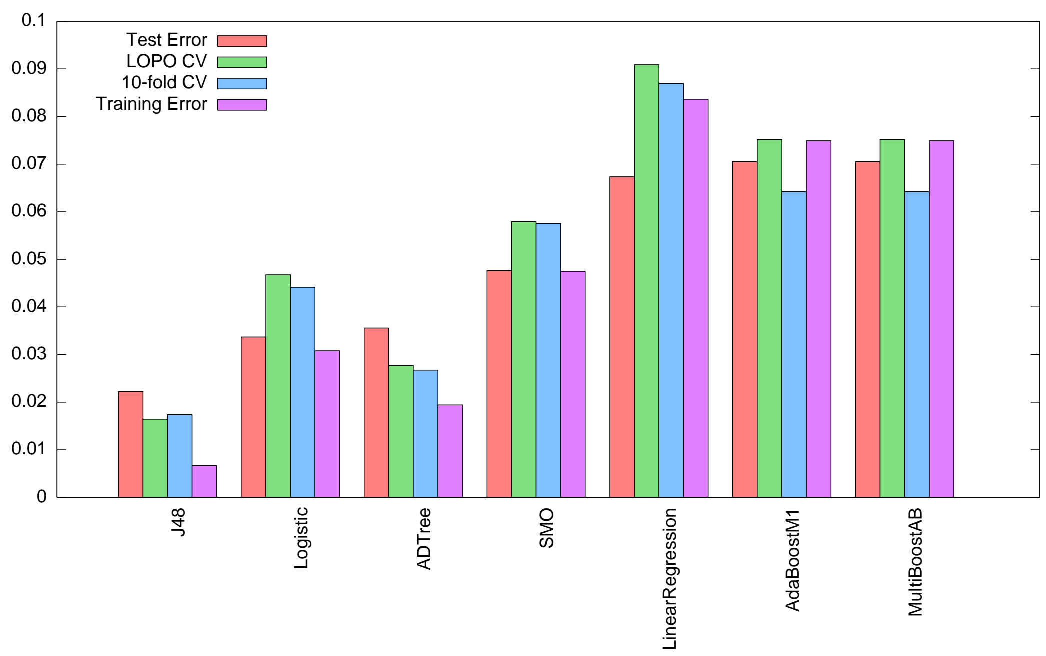
- The best performing classifier in term of lowest test error was J48, a decision tree based classifier. Decision trees have the advantage producing an interpretable result; figure A-9 in the appendix shows the resulting decision tree. The notable quality about the decision tree is that it has built-in feature subset selection. While the training data contained 30 features the resulting tree only used less than a dozen.

- While most of the linear classifiers (Linear Regression, SVM and Logistic Regression) reported poorer overall error rates, they were also the ones with the highest recall rates (figure 4-11(b)).

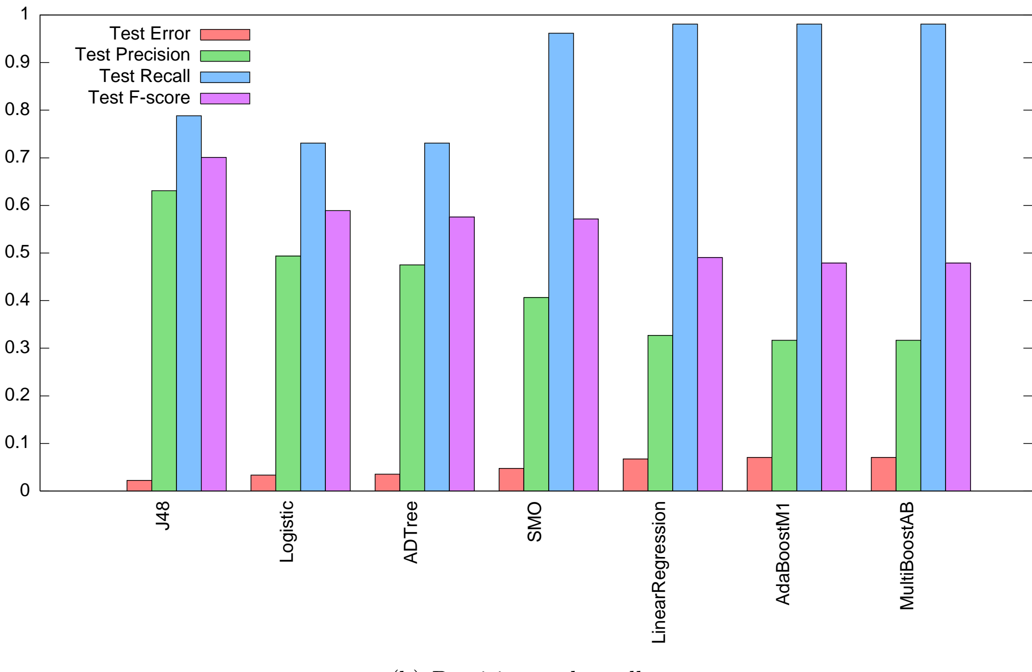
These results show that the type of classifier did significantly affect the accuracy of the classification task. In terms of error rates, the decision-tree-based classifiers (ADTree and J48) were clearly the winners. However their low error rates also came with a lower than desirable recall rate. Since our goal is to aim for high recall, we settled on using a linear SVM (SMO) as our baseline classifier. For one, it had the lowest error for a nearly perfect recall. Also its cross-validation and test errors did not deviate far from the training error, a sign that the classifier was not over-fitting.

4.6.3 Feature subset selection

In record detection, we have 30 features to analyze; naturally, we were interested in finding a reduced optimal subset of features. We first tried using CFS[22] as in pattern detection to find our optimal feature subset. Recall that CFS used an information gain criteria to search for an optimal feature subset. After running CFS, we were surprised to find that under both best-first and exhaustive search strategies, CFS returned “Contiguity Indicator” as the sole best feature subset.



(a) Training, Cross-Validation, Testing Errors



(b) Precision and recall

Figure 4-11: Comparison of different classifiers using all cluster features.

We already knew the test error rates for contiguity indicator was not optimal. In fact we, tried all two-feature subset combinations and found a lower error rate. In table 4.3, we compare the test error rates of using the best 1 feature, 2 feature and all feature combinations.

Feature Subset	Test Error	Test Recall	Test Precision	Test F-score
Best CFS-selected[1]	0.07560	0.98077	0.30178	0.46154
Best Wrapper selected 2-feature[2]	0.06925	0.9807	0.3208	0.4834
All features	**0.04765**	0.96154	0.40650	0.57143
Features $<$ 30% error[3]	0.05337	0.98077	0.38060	0.54839

Table 4.3: Test results (error, recall, precision and f-score) of training a linear SVM classifier using the various

Since we did not do an exhaustive search of feature combinations, it is very possible that the optimal feature subset lay in some 3-30 sized feature subset. Running an exhaustive wrapper-based search to pin-point a better optimal subset would be a lengthy proposition with $2^{30} - 1$ combinations to test.[11]

We know that the all-feature subset has the lowest *known* test error, and signs do not indicate that the over-fitting is occurring. Under these auspices, we decided to use the all-features subset as our baseline feature subset for record detection.

Have we succeeded in achieving our stated goals? For one, our classifier has a high recall rate (96%) and an acceptable test error of $\approx 5\%$. The precision of the classifier is low at $\approx 40\%$. This means that on average for every one record cluster we expect to also find 1.5 non-record clusters; this is certainly a significant improvement from the previous odds of 1 in 30! Our result is certainly not perfect but they are acceptable and fulfill our stated goals. In the next chapter, we will evaluate our complete record extraction system that combines our baseline pair classifier (we found from pattern detection) and this baseline cluster classifier.

[11]If our $2^9 - 1$ combos took a week then $2^{30} - 1$ combos would take roughly 160,000 years!

[1]Best CFS-selected feature subset: ContiguityIndicator

[2]Best wrapper-selected 2-feature subset: ContiguityIndicator, ClusterSize.Ranked

[3]Features less than 30% error: ContiguityIndicator, ClusterSize.Inv-Ranked, ClusterSize, ContentCoverage, TreeSizeNormStdev, LogTreeSizeNormStdev, ClusterSize.Ranked, ExternalPeriodicity, FilledCoverRatio, GZipCompressionDifference, ExternalGZipCompression, ContentSizeNormStdev, PeriodicityDifference

Chapter 5

System Evaluation

By combining the pattern detection and record detection subsystems from our last two chapters we now have a complete system for unsupervised record extraction. More specifically, we have a general process for converting a DOM tree of a web page into a set of tree clusters that are plausible record clusters. How does our system, MURIEL, compare with previous record extraction systems like Omini [5]? In our related work (chapter 2), we noted several differences in our approaches. How do these differences factor into the final system level performance evaluation? Did they matter? In this chapter, we describe our comparison evaluation and analysis of the problems that we found within each of the two record extraction systems.

5.1 Testing the tree selection algorithm

In our discussion of related work, we described Omini [5] as a system that employed a *top-down* strategy to solving the record extraction problem. Namely, its system architecture was partitioned into two main components: a tree selection module and a record segmentation module. The tree selection module was responsible for finding the record-packing tree in a page, often using a heuristic. The record segmentation module was responsible for segmenting the record boundaries from the record bearing tree. Under this architecture, the tree selection module was a potential accuracy bottleneck; if it failed to find the right tree then the entire system failed. Omini (and other top-down systems) assumed that its tree selection algorithm was always sufficiently accurate, but just how accurate?

We evaluated two tree selection heuristics to find out.

1. The maximum fan-out heuristic was used in Jiang et al's [20] system. This heuristic was founded on the intuition that records cover a large portion of the page therefore their parent or ancestor trees must also likely be trees with the largest fan-out.

2. Omini's best tree selection heuristic was more sophisticated. Omini combined five features in a weighted linear combination tuned to select a best tree. The features used were:

   (a) Tag Frequency (TF) - Trees rooted with certain "container" tags like $\langle TABLE \rangle$, $\langle DIV \rangle$ were likely candidates for record covering trees.

   (b) LargestTagCountTree - Select trees that have the largest repeating tag count.

   (c) HighestTagFanoutTree - Select trees with the largest fan-out.

   (d) LargestSizeTree - Select trees that have the largest overall size (or content size).

   (e) Size Increase - Select trees that maximized:

$$| T | - \frac{1}{n} \sum_{i}^{n} | ch_i(T) |$$

   i.e. the difference between the size of the tree and average size of its children trees. This feature has a hint of similarity with our variation difference metric.

In our evaluation, we took labelled tag trees and applied each heuristic over the page tree and examined the returned candidate record-packing tree. To determine whether these candidates were indeed *record-packing*, we tested each candidate using two metrics: ancestry and content overlap.

- *Ancestry* determines whether a selected tree is a common ancestor to all (or most) of the instances of a record class. (A record class is a set of labelled record subtrees).
- *Content overlap* determines whether the content spanned by the candidate tree covers the same region covered by the record subtrees.

The content overlap metric was necessary to ensure that the tree selection algorithm did not simply select a trivial solution, namely the root of the page (since it was the ancestor to all trees).

To compute ancestry, we determine the fraction f of trees in a given record class that are the descendant of the selected tree: T_i. For convenience we adopt the following definitions:

- *full ancestry*: The selected tree is ancestor to all the members of a record class (or $f = 1$).
- *no ancestry*: The selected tree cannot be associated with any record class (or $f = 0$).
- *partial ancestry*: The selected tree is ancestor to only a subset of members from a record class (or $f \in (0, 1)$).

In most cases, we have either full ancestry or no ancestry.

To compute content overlap, we computed: $overlap = \frac{cover(L)}{fill(T_i)}$. $Cover(L)$ is the number of content units[1] spanned by members of a given labelled record class L, and fill(T_i) the amount of content spanned by the candidate tree.

[1] A content unit is 1 text character or a 10x10 pixel in an image.

Table 5.1 summarizes the results of our evaluation.

Ancestry	total	> 75% overlap	50 – 75%	< 50%
Max fan-out				
Full	37 [31.62%]	29 [24.79%]	5 [4.27%]	3 [2.56%]
No	78 [66.67%]			
Partial	2 [1.71%]			
Total #	117 [2]			
Omini's best-tree				
Full	71 [86.59%]	62 [75.61%]	7 [8.53%]	2 [2.44%]
No	8 [9.76%]			
Partial	3 [3.66%]			
Total #	82			

Table 5.1: An evaluation of tree selection heuristics. The values in brackets are percentages relative to the total number of instances.

We consider a selected tree that has full ancestry and greater than 75% content overlap with that record class as a correct result. In order for a tree to be truly record-packing the content overlaps should be 100%. But we instead use a more lenient threshold of 75% to allow some room for error.[3] From our reported results (table 5.1), we can conclude that:

- Max fan-out is a poor heuristic for selecting record packed trees. Most largest-fan-out trees are not record covering, we found only about 32% were. Of those trees that did contain records, 29/37 or 78% had greater than 75% content-coverages. So only for 24.79% of the pages examined did the heuristic pick a correct record packed tree.

- Omini's tree selection heuristic faired much better. The percentage of trees that it found were actually record covering was much higher: 86.59%. Of those that did contain records about 62/71 or 87.32% had greater than 75% content coverage. Therefore, Omini's tree selection heuristic was able to pick a correct record packing tree for 75.6% of the pages.

Our results implied that Omini can at best achieve a 75.6% accuracy (on our data set); it is limited by the best that its tree selection algorithm can do.[4]

5.2 Comparison of Omini versus MURIEL

Next, we performed a comparison evaluation of our system MURIEL versus Omini.

[3]Some record-packing regions that are formed within tables may contain table headers that are not considered part of record content. Content like these extraneous headers are what our lenient threshold tries to compensate for.

[4]Omini came with its own configuration settings; it was not clear from their documentation how those settings were obtained or how their system was trained.

To evaluate Omini, we downloaded and ran the available source code from SourceForge[5]. We did not use Omini's provided data set. The reason was that their data set contained many examples of sibling-root records - the case that our system was not designed to handle at the moment.[6] Therefore we used our labelled data set as the evaluation data set since it contained only single-root records. We will need to revisit the Omini data set once we have a suitable sibling root record solution. Omini consumed as input the raw HTML of a page, and produced as output a custom markup text file with extracted record instances as fragments of HTML. To evaluate Omini results, we measured the precision/recall/f-score between Omini-extracted records and our labelled records:

$$r_{system} = \frac{| C_{system} \cap C_{labelled} |}{| C_{labelled} |} \quad (5.1)$$

$$p_{system} = \frac{| C_{system} \cap C_{labelled} |}{| C_{system} |} \quad (5.2)$$

$$f_{system} = \frac{2 \cdot r_{system} \cdot p_{system}}{r_{system} \cdot p_{system}} \quad (5.3)$$

We used text match as the main criteria for determining the intersection $C_{system} \cap C_{labelled}$. The extracted objects returned by Omini were string normalized. String normalization involved stripping away HTML tags, lower-casing, and collapsing any white-space. To make text match even more lenient we further relaxed the definition of string equality to be a rough edit distance. Two strings were considered the same (or matched) if they had a normalized edit distance of less than 0.10 (or less than 10% difference). To evaluate MURIEL results, we perform the same text normalization.

Our labelled data can contain multiple record classes per web page but Omini only returned one record set per page. So we graded Omini using the best scoring labelled record set for each page.

MURIEL on the other hand returned several clusters that could be plausible records, however in no particular order. We needed some way to rank our results, so as to pick a single top result like Omini. We found two ways to rank our clusters. One cluster ranking strategy is to using the Weka [47] library's *logistic distribution fitting functionality*. Weka implemented a feature that fits a logistic distribution over the decision boundary; each test instance could be assigned a probability according to this distribution. This strategy is equivalent of computing the distance to the decision boundary. Our second cluster ranking strategy is simply to use a cluster feature; We used content coverage as that ranking feature. Recall that content coverage is defined as:

$$ContentCoverage(C) = \frac{\sum_{i \in C} fill(i)}{fill(root)}$$

Sorting by content coverage means clusters that represent more visual space area on the page will be

[5]http://sourceforge.net/projects/omini/

[6]Sibling-root records are records that are defined over a set of neighbouring trees, see figure 1-4 for an example.

favored. Note that content coverage is not the same as cluster size since a large cluster may contain many small members that overall do not cover a lot of content on the page. The top-scoring cluster under either regime was used to compare against the Omini results.[7]

To simplify the discussion of results we defined 5 quality grades.

1. Perfect - $f_{system} = 1$ for the extracted records; extract records matched one-to-one with labelled records

2. Mostly perfect - $f_{system} \geq 0.8$

3. Mostly correct - $f_{system} \geq 0.5$

4. Mostly incorrect - $f_{system} < 0.5$

5. All incorrect - $f_{system} = 0$ - no match at all

In order for the results to be considered acceptable we deem that it must perform well in the *mostly correct* grade.

We evaluated Omini and MURIEL using our test set of 41 labelled pages (MURIEL was trained on the other 41 pages); the results are summarized in table 5.2.

Grade	Omini	MURIEL (w/ logit)	MURIEL (w/ content-cover)
perfect	6 [0.1463]	20 [0.4878]	19 [0.4634]
mostly perfect	24 [0.5854]	31 [0.7561]	31 [0.7561]
mostly correct	24 [0.5854]	31 [0.7561]	31 [0.7561]
mostly incorrect	17 [0.4146]	10 [0.2439]	10 [0.2439]
all incorrect	16 [0.3902]	10 [0.2439]	10 [0.2439]

Table 5.2: Evaluation comparison of Omini versus MURIEL .

These results showed that MURIEL significantly performed better than Omini for all quality grades, a 17% difference for the *mostly correct* grade.

Naturally, what were the reasons behind the failures in both systems? We did a manual failure analysis of results in the *mostly incorrect* grade for both systems. First, we will present our analysis for Omini; we will re-visit the failure analysis for MURIEL in section 5.3.1.

[7] One may argue that content coverage scoring and the max-fan-out heuristic are very similar; we could have devised a top-down system that simply used content coverage as a tree-selection heuristic. The difference between the two approaches is that in MURIEL content-coverage is used as a tree selection criteria **after** filtering by record detection (not before as it was used in top-down systems). Had we simply used content-coverage as the tree selection criteria, the top-scoring result would not likely be a record cluster. Before record detection, the ratio of non-record to record clusters was 29:1, and afterwards, 1.5:1. So in our system, we are selecting a tree based on filtered data.

5.2.1 Failure analysis of Omini

We classified the failure cases for Omini into the following categories. We include the web page references in brackets after each category; the full URL to these references can be found in the appendix A-3.

- No results [0, 2] - Omini returned no record sets for our two smallest sized pages, both of which contained a record set of three record instances. Checking against our tree selection evaluation we confirmed that Omini's tree selection system did not return a record covering tree for these pages. MURIEL is less susceptible to these pages because we make no limiting assumptions about the size of record instances; we do not specify how large or small record instances maybe only that they should be similar enough to be clustered; Our only major restriction is that record instances repeat at least once.

- Navigational Menu: [17, 21, 58, 85, 128, 132] - In many cases, Omini returned record sets that contained instances that were part of a navigational menu.

- Advertising Record: [27] - Omini extracted a set of objects that was not labelled as a record set in our test data set. Upon closer examination these could be construed as advertisment records. We can not count this case against Omini since advertisement type records are ambiguous. It is a type of record by syntax but not by semantics. (Can you tell for instance something is an advertisement in a foreign language?)

- Entire page: [60] - In one instance, Omini considered a tree covering a large part of a web page as the record set. The page header, some menus, a record containing segment, and the page trailer were all individual instances in the extracted record set. This seemed to be a case of where Omini's tree selection algorithm failed. The algorithm selected a tree that was much too high above the actual record-packing region; it was closer to the root of the tree than to the records.

- Record Collection: [88, 94, 105, 109, 110] - Omini often identified the rows of tables as the record rather than the individual table cells.

- Records: [123] - This is an unusual case where Omini extracted plausible records but also included all the *spacers* as records. A spacer is a table cell in which a blank image was added to maintain spacing between instances. For this reason, while most of the correct record instances were found, the cluster also contained many non-record instances (low precision).

- Other reasons: [117] - For one page, Omini apparently generated a bizarre parse of the page. As a result, records that were extracted were a combination of the tail portion of one instance joined with the head portion of the next instance. We suspect this is not a quality problem with the system but a bug in their tag tree parsing.

From this analysis, we see that the navigational and record collection cases were the majority causes of problems. The navigational menu, empty, entire-page cases where all examples of where tree selection algorithm failure cascaded into a systemic failure. Record collection cases were problems related to incorrect *focus* - a problem in the record segmentation system. Even though the right record bearing location was found, the system segmented records at the wrong level in the tree. This is where MURIEL's variation-based features (like periodicity) would be useful in discerning the difference.

5.3 Evaluating MURIEL

In the comparison study, MURIEL returned results that were *mostly perfect* for 75% of the pages. We were curious as whether the remaining 25% was due to problems with the cluster results or simply due to poor ranking. To find out, we conducted a different type of evaluation. Since our system was designed to return as many plausible record clusters as possible we should really be asking how good are the top-n results? We did a top-5 returned clusters evaluation of MURIEL, the results are summarized in figure 5-1 (the ranking is by content coverage).

Grade	Number of Instance [percent]
perfect	25 [0.6098]
mostly perfect	34 [0.8293]
mostly correct	40 [0.9756]
mostly incorrect	1 [0.0244]
completely incorrect	1 [0.0244]
Other stats	
average rank	1.2927
average #clusters	3

(a) Result evaluation looking at the top-5 clusters produced by MURIEL

rank	1	2	3	8
Content-coverage	31	6	3	-
Logistic ranked	31	6	2	1

(b) Distribution of ranks of the found record cluster ranked using content coverage vs. logistic distribution. Each cell contains the number of pages with the first record cluster being that rank.

Figure 5-1: Evaluation of MURIEL looking at the top-5 returned clusters.

Our system returned results in the *mostly perfect* grade for 82% of the test pages. Better yet, our system returned *mostly correct* clusters within the top 5 results for 97% of the test pages. These

results imply that we will almost certainly find a record cluster by looking within the top 5 results.[8]

We also analyzed the rank distribution of the found record clusters (see table 5-1(b)). The rank distribution shows that all of the found record clusters were in the top 3 ranks (under content coverage ranking). This distribution also explained away our failure cases in our comparison study. In the comparison study, MURIEL missed the 10 pages; the correct results for these 10 pages were simply ranked second or third! Had we done a comparison using a top-5 (or even top-3) results evaluation, our system would have passed with flying colors.

As a side note, if the content-coverage ranking was able rank record clusters into the top-3 then further precision improvements can be made by simply returning the top 3 clusters! For completeness, table 5-1(b) also shows the distribution from ranking using the logistic distribution method. The distribution is mostly the same except for a single outlier case (at rank 8).

The rank of record clusters does not tell us about the number of clusters returned. Recall that one of our goals was to reduce the number of returned clusters if possible. If the system returned too many clusters then in an (interactive) application setting, the user would have to examine many record cluster results. Thus to find out whether our system was returning too many clusters, we tabulated the distribution of number-of-results in table 5.3. This number-of-results distribution

# of clusters	1	2	3	4	5	6	7	9
# pages	**10**	**9**	**11**	3	2	4	1	1

Table 5.3: Distribution of the number of clusters returned per page.

shows that the mode (and mean) is around 3 which implied that for most pages (that have records) we should expect to find two (false positive) non-record clusters for each (true positive) record cluster. [9]

5.3.1 Failure analysis of MURIEL

What caused many of MURIEL's false positive results? We did a manual failure analysis of MURIEL results looking at a sample of clusters graded as being *mostly incorrect*. We found the some general problems:

[8] One may argue that if Omini were adjusted to return the top 5 results - from the say top 5 selected trees - it will do just as well. This would need to be confirmed by a future evaluation. However from our examination of Omini failure cases, we believe increasing the number of selected trees only improves the odds of selecting the correct record-packing tree. Omini also fails because can not distinguish records from record collections. Therefore, we predict that even under a top-5 results comparison our system will still perform better.

[9] It is very possible that some clusters in the distribution are record clusters that were split into two or more smaller clusters. However we did not observe these split clusters in our detected clusters. Split record clusters exhibit low contiguity and therefore are filtered out as non-record clusters.

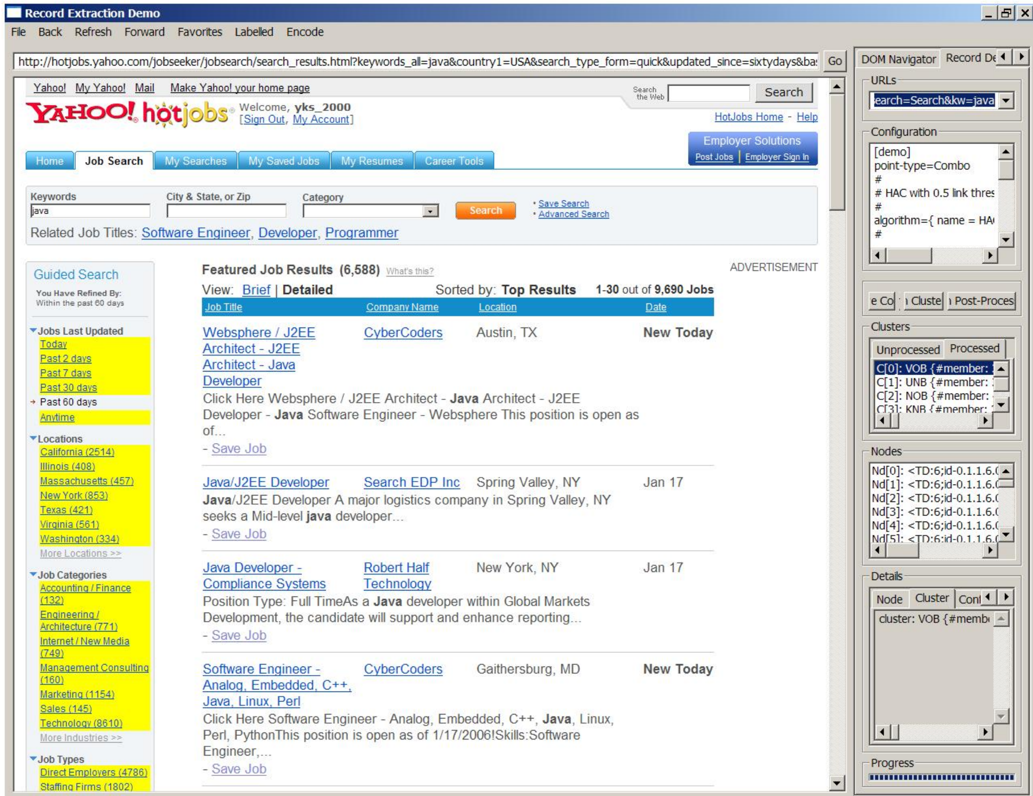


Figure 5-2: An example of MURIEL returning a navigational type record. The instances are highlighted in yellow.

- Navigational [32, 55, 82, 94, 50] - Some returned clusters were of navigational nature, *i.e.* menus, pagers. Figure 5-2) shows an example.

- Mistaken Record: [58] - In this particular page the top cluster contained two large blocks (see 5-3). Each block contains a list of items. Without looking at the content, one can se that each block has some record-like characteristics. The list of items are not completely homogenous but varies similar to how fields vary.

- Partial record sets [58] - On closer examination of the mistaken record cases we found that the main reason for failure was that only partial record clusters formed during pattern detection. These partial record sets were largely non-contiguous. Our cluster re-merging also failed to re-merge these clusters. So lacking contiguity they classified as non-record and were filtered out by record detection. Had we succeeded in merging partial record sets into one cluster, we might have found the expected record clusters in our results.

- Floaters [85] - MURIEL extracted a cluster whose elements were visually hidden; Figure 5-4 shows this example. The web designer placed a menu element on the page that was visible only when the corresponding menu item was moused-over. In the figure, we show one cluster

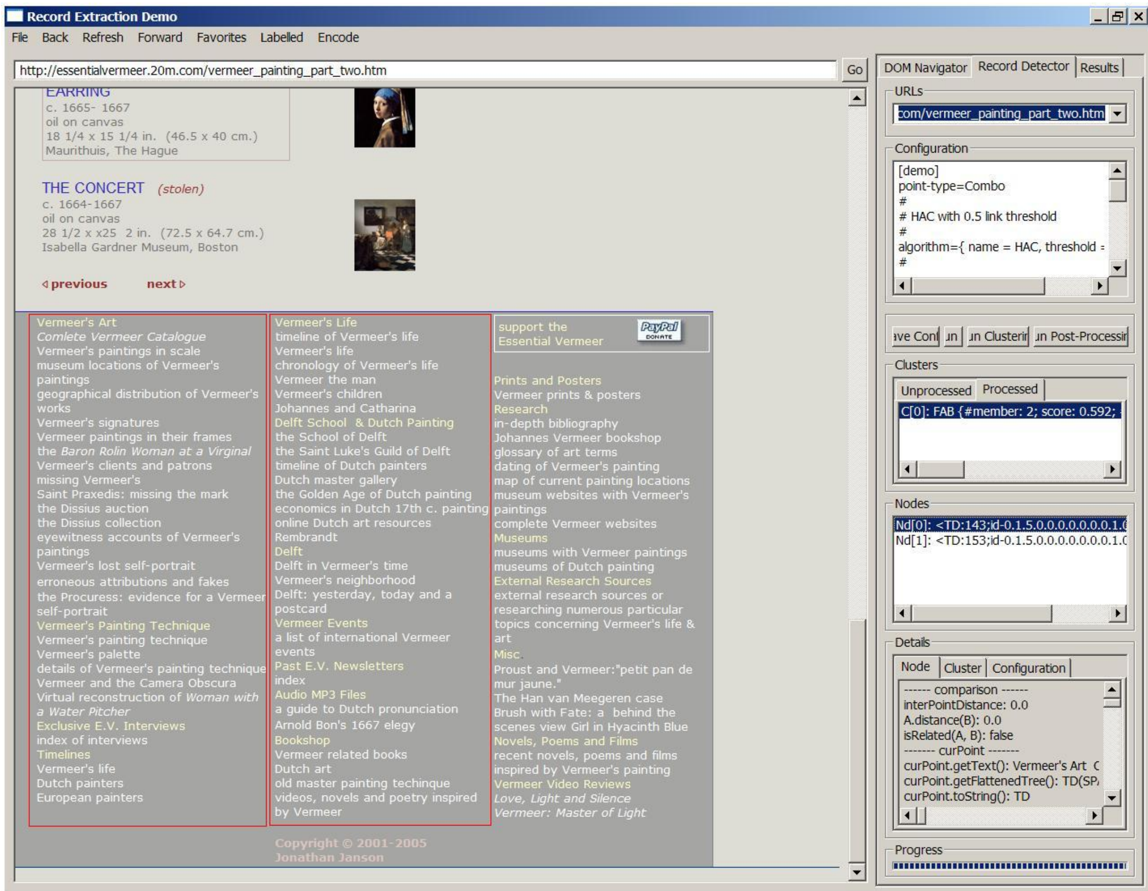


Figure 5-3: Two record like regions. Boxed in red are two instances of a cluster that MURIEL mistaken as a record cluster.

instance that appeared visible when we moused-over the appropriate menu element; it is highlighted in red. These types of invisible clusters could be filtered out given knowledge of whether an element is visible when it is rendered to the user. This requires better handling of style sheets by our system.

- Record collection [88] - Despite our use of variation-based features to filter out record collection type clusters, some inevitably still fall through. However, we only had one case of record collection failure whereas Omini had five cases. Clearly our variation-based features is having some effect.

Some of these failure cases could be fixed through tweaking and further improvement of our system. For example, partial record sets can be fixed in two ways. We can try to further tweak the clustering parameter or the cluster merging algorithms to further improve recall, or we can generate a tree pattern from these partial record sets and use the pattern to increase the number of record instances. However, failure cases such as unlabelled advertising and navigational type patterns are much harder to fix. To differentiate them, one would need some semantic knowledge.

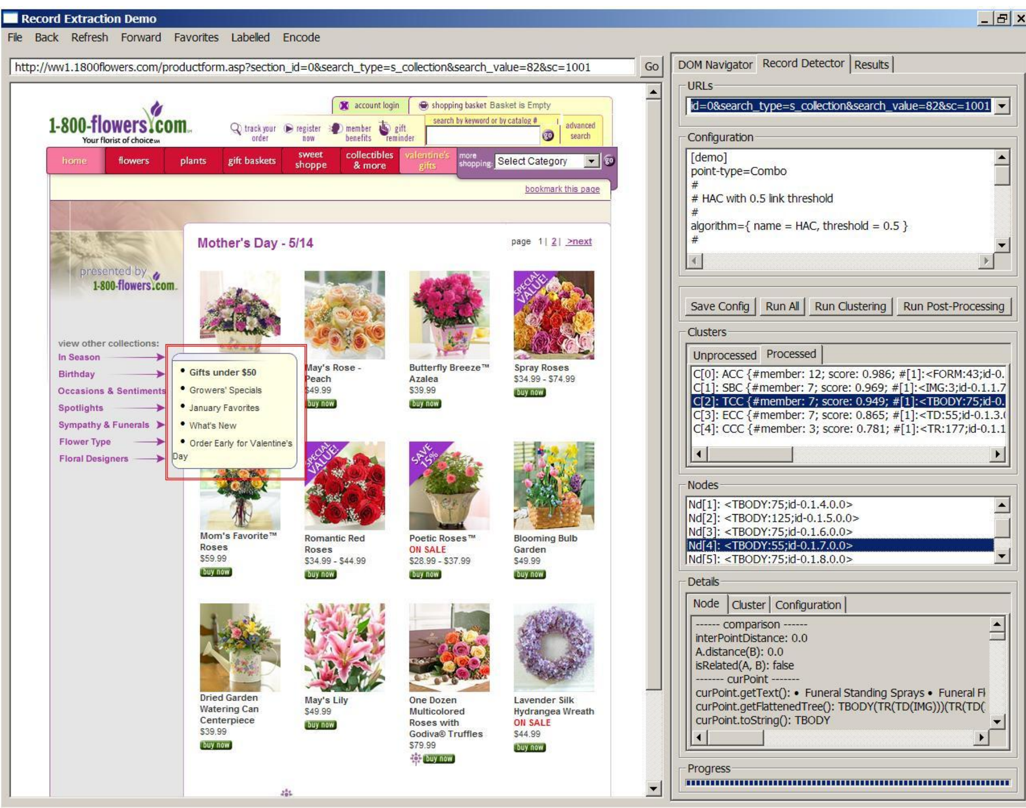


Figure 5-4: An strange case of hidden records. Boxed in red is a hidden "record" instance that appears only when the menu item is moused-over.

Differentiating advertising from non-advertising requires textual analysis and possibly also layout analysis. Looking only at the tag tree, we cannot tell whether a cluster of content is located centrally on the page or at the margins of the page. If we had layout position information then perhaps we could come up with features that would differentiate some of these patterns. Navigational patterns could be also be identified better if we knew that they occurred across different pages within a website but their content stayed the same. These types of cross-page cues and semantic features are interesting future extensions to explore for our system.

5.4 Summary

We have shown through our evaluations that, our system MURIEL, performed better overall than predecessor systems. Using a top-result comparison, we showed that our system was more accurate than Omini by a margin of about 17%. Furthermore, if we compared systems based on a top-3 result evaluation then MURIEL would yield an even better rating with an acceptable accuracy for more than 90% of the test pages.

Detailed analysis pin-pointed two major weaknesses in the Omini system, the first being the

fragility of the tree selection module in failing to find the right record containing tree, and the second being a failure to find the focus for records; stopping at the record collection level rather than at the record level. In a similar detailed analysis of MURIEL, we found that the ranking of results was one area that we needed more improvements. Further tweaking of our system and the introduction of semantic knowledge could also potentially improve our system accuracy.

Chapter 6

Conclusion

If a human can discern the structure of information from a web page why can't a computer program? We started this investigation asking this exact question. Within the course of this thesis, we have demonstrated a method for building a system that could extract a specific type of structure, flat records, from web pages without any external supervision. Our domain was limited to flat records and our record model limited to single root trees. But within this limited scope, we can claim success. As our results have demonstrated, our system was able extract records accurately in a top-5 results test for over 97% of the cases that we tested. In a top-1 test, our system out-performed a predecessor record extraction system, Omini, by a margin of 17%.

6.1 Contributions

Within our work we have made several contributions to this research area, we summarize these below.

1. We introduced a novel system for inducing records from web pages without explicit user training.
2. In our system we modeled records as trees under a tag tree representation of web pages. Previous record induction systems did not define specific models for records and focused on discovering record boundaries. Our representation is better constrained and more specific because of the introduction of features such as tree structure. Our tree-model based formulation improved the precision of extraction results as compared to previous systems.
3. Our approach to the record induction problem is *bottom up.* We first define a model for record instances; anything that fits our model is a record candidate (in our case, a single root tree is the model). Next, we apply constraints to restrict our set of record candidates. Clustering (or pattern detection) constrains records to be sets of trees with common structural, content-based

or contextual features. Record detection further subjects cluster candidates to yet another set of variation-based and contiguity-based constraints. We argue that our approach is an improvement over previous top down approaches. Our approach is built on solid machine learning principles whereas top-down systems is more heuristics based. Our system is less likely to breakdown as long as we have sufficient and accurate training data, where as a top-down system is more fragile to critical heuristic failures.

4. We formalized and determined a large set of features for both the pattern detection and record detection phases of our record extraction task. These features included structural, contextual, and content based features for pair-wise learning, and contiguity and variation based features for cluster learning. We did detailed studies and measurement of these feature and their performances both on an individual basis and in combination.

5. We formalized the record extraction problem as a multi-step machine learning problem. While earlier systems could claim some use of machine learning techniques, there was little evidence of a clear formulation as a machine learning problem. In our work, we made explicit our use of supervised and unsupervised learning formalisms. Our framework allowed us to easily compare the performance of various types of classifiers for our task. With a more solid theoretical foundation, record detection could be studied and extended using more theoretically sound methods.

6. We have provided a reusable data set, available at `http://muriel.csail.mit.edu/`, for use in training, validation, and evaluation of future record extraction systems (or for future extensions of this project).

6.2 Near-Term Future Work

While we have explored the many parts of this problem extensively there still remain many important and unexplored issues where near term work is necessary. Next we list possible extensions and issues that could be further explored or extended in the near future.

- *Sibling trees*: We restricted our record instances to be presented in single rooted trees; sibling trees are the other common form of representation of flat records. The main problem with sibling trees is finding the proper subsequence that represents a single record instance from a sequence of trees. We could search for this proper subsequence by computing the periodicity (section 4.3.3) of a given sequence of trees and using the shortest repeat found as that proper subsequence. Where should we start searching? At locations where field patterns occur! We noted that field pattern cluster instances are separated by non-zero amounts of content. We could use this as a feature to train a *field pattern detector*. Once we find field patterns and

we can then apply our periodicity metric over the areas where field patterns appear. If we find significant subsequence repeats within those areas then we can convert the repeating subsequences into single root trees by attaching "virtual" nodes to those subsequences. Once the virtual nodes are in place, then the mechanisms used for single-root trees could be applied to solve the sibling trees case.

- *Page Level Records*: As noted in our introduction, records do not have to be trees within the page; they could also be trees at the page level or even be multiple-page-spanning. There are numerous issues at the page level that make record extraction more difficult than within-page records. Primarily, we no longer have a clear tree structure from which we can discern parent-child relationships. We will need to rebuild the tree structure from the web graph. The tree structure for page level records could be reconstructed using clues from the path component of the URLs of the pages. Trees that share the same site or the same path prefix should be grouped under a common tree. We may also look for "table of content" pages, single pages that contain links to many similarly formatted pages, as potential internal nodes in this supertree. Once we have a tree structure for page level records, we could apply our system to this web(site) scale tree.

- *Field extraction*: In this thesis we focused exclusively on extracting records from web pages. We did not explore the process of extracting fields from those records. The major reason for overlooking field extraction was that we lacked labelled field data for which we could do training, analysis or evaluation.

  In field extraction we need to correctly determine how many fields there are in a record and where the field segments occur. We can think of the problem as like a mini-record extraction problem: fields are subtrees of a record tree. Field subtrees will cluster (based on similarity) and they will appear within individual instances of a record class. These distinguishing characteristics can help us determine which subtrees are the fields given we know which trees are record instances.

  We did implement a rudimentary field extraction procedure that borrowed from the idea of tree pattern templates from Hogue *et al* [23]. A tree pattern template is special "wildcard" tree produced from the merging of several similar trees using a tree edit distance algorithm. Within a template tree, nodes that mismatched or aligned to gaps in the tree alignment become "wildcard" nodes. Our implementation makes the assumption that each field segment corresponded to a wildcard node in the tree pattern template. Our implementation worked well small set of pages we tested. However we did not have time or sufficient data to properly evaluate our implementation or test our assumptions.

  To implement field extraction properly, we may also need to consider cases where field segmen-

tation occur below the level of a text node (*e.g.* when two consecutive fields are formatted the same way). Hence, we may need to parse the syntactic structure within unformatted text. Once we have a sufficiently large manually label field pattern set, we plan to evaluate our current tree pattern-based approach as well as explore other possible field extraction methods.

- *Using External Data*: While we have demonstrated a method for extracting records without deep analysis of underlying content, the accuracy of our system could be further improved with the availability of external structured data. For instance suppose that we are tasked with extracting data from a directory page[1]. If we have in advance a database of names and email address of several professors then we could leverage that external data to constrain the area to which record instances could occur. This would immediately eliminate many false positive clusters like navigational menus or advertising. Trees matching two or more external data instances would be poor candidates for records; whereas trees that match data from single external instances would be strong candidates. External data in these cases can be integrated into our learning processes as a form of prior knowledge.

  External data could come from sources other than just databases. Domain specific information that occurs with high redundancy on the web could also be used as a source of external data. For example, there are many movie information websites. If we aligned the data for the same movie instance from these redundant sources we can arrive at a partial schema (a list of possible labels for fields). Such a partial schema could be another source of external data useful for identifying records or fields in pages from the same (movie) domain.

- *Record Page Identification* We have assumed that the inputs to our system are pages that contained records. What if such were not the case? In a batch application, the system should discover on its own whether a given page contains records. The trivial solution to this problem of *record-page identification* would be to run our pattern detection and record detection processes over each page. Answer true if the processes return any cluster and otherwise the answer false. Of course because our record detection subsystem is set to high recall, the system may return many non-record clusters. We can solve this by favoring precision more over recall. In other words, the system will identify a page as a record page, only if the system is highly confident of the existence of at least one record cluster.

  If we use a record extraction system for record page identification that system will have to be very efficient in terms of run-time performance. In chapter 3, we detailed some ways in which the pattern detection process can be optimized. However, even with current optimizations the system will still need on the order of minutes to processes each page. To quickly determine

[1]http://www.csail.mit.edu/biographies/PI/biolist.php

record pages quickly we will need further improvements. Also we will need to evaluate the speed versus accuracy trade off, since many optimizations make rough approximations that may negatively impact the accuracy of the detection process.

- *Leveraging Natural Language Processing*: We have actively avoided using natural language cues to aid our record extraction. We wanted to show that record extraction was possible without semantic understanding of the underlying content. However, we do not deny that additional natural language information could further improve the quality of extractions. In chapter 3, we stated a way to use character classes as a method for comparing trees based on their content. One can think of character classes as rough approximations of field types in the underlying information. We can do better than character classes by using state-of-the-art named entity recognition techniques to determine field types. Named entity extractors are specialized pattern recognizers used for detecting specific types of information, for instance whether a segment of text is an email, a telephone number or a common person's name. We could break the text data down into a more structured parse tree based on these tagged named entities. For instance a text segment such as "550 Memorial Drive, Cambridge MA" would become `<STREET-ADDRESS>, <CITY> <STATE CODE>`.

- *Unifying Clustering and Record Detection*: Could we have combined pattern detection and record detection into one process? We decided on having a two step process because some of the features we used were only applicable as aggregate features. For instance, contiguity was only meaningful if you had a cluster of trees. What if we used contiguity (or any of the record detection features) as a tie breaker during the clustering phase? For instance, given a choice of merging several possible pairs of clusters (that are tied), we pick to merge the pair that results in a cluster with the best improvement in contiguity. This idea is not too far from the cluster merging idea we used for post-clustering processing.

- *Exploiting relationships within the tree*: One angle we did not explore actively is relationship between clusters. Recall that our cluster function $\psi(C)$ is a function of a single cluster; should it not be instead $\psi(C_1, \ldots, C_n)$? Our current system needs to capture the inter-dependencies that may exist between different clusters.

  A common interdependency is that record clusters should never overlap. In other words, if one cluster has been determined to be a record cluster then no descendant or ancestor of that cluster's members should belong to another record cluster. Another possible interdependency is between fields and records. If we know that a cluster is a field pattern (maybe by training a field pattern detector) then that should increase the likelihood that an ancestor cluster would be a record cluster. Similarly if we know that a cluster is a record collection then that should increase the likelihood that a descendant cluster would be a record cluster.

If we are able to formalize these inter-relationships between clusters then we could potentially model cluster relationships using a Bayesian network[2]; we can use probabilistic inference to deduce whether a cluster should be a record cluster given information about other clusters.

- *Incremental Learning*: In our experiments, we trained our pair and cluster classifiers in batch mode. In a more realistic application setting, for example a browser extension, we would need to train classifiers with incremental data. A user may make adjustments to the application output; those adjustments feedback into our system as *incremental* training data. Numerous interesting issues arise in such an environment. How do we properly weigh user adjustments versus previous training data? Should training data expire over time? How do we and should we convert user-implicit UI interactions into training data? For example, if the user chooses to export a cluster set, does that imply that the cluster set is of high quality and should be introduced as future training data?

- *Growing the training set semi-automatically*: The dataset we used for experimentation could be greatly expanded. However, manually identifying record pages and labelling them is in itself a tedious process. Since we already have a moderately sized training set, a quick way to increase that dataset is to crawl the web for pages and use our system to identify new record bearing pages. A second way to accumulate training data is to productize the system as a browser plug-in and allow users provide additional training data from direct usage. A third way to find training data is by looking at restricted domains. For instance, searching for `<institution name> directory` on a search engine will provide a set of seeds leading to web pages that contain records of people.

6.3 Future Work

Apart from these possible near term extensions, there are numerous more "blue sky" ideas we consider interesting for future extensions.

- *Web objects database*: We mentioned that a target application of MURIEL would be for building an object database from extracted web pages. Such a project will require a record extractor that is robust against all types of record objects and pages. Furthermore, we will require a robust framework for extracting fields from extracted data. Also we will need a robust categorization framework for organizing the extracted data into same or similar schema groups. Once we realize these prerequisites we will be one step closer to building such a web objects database/search engine.

[2]Bayes nets allow us to better model the relationships between trees. Also it provides a framework for evidence handling; *i.e.* we can compute the likelihood of a node being a record given evidence about other nodes.

- *Towards automating feature engineering*: In our thesis, we explored numerous features for producing classifiers for tree pairs and for clusters. Most of our effort was focused on finding good features and essentially doing feature engineering. As machine learning technology and implementations mature could we find ways to even automate the feature engineering process? An ideal system should take examples of record vs. non-records and automatically "figure-out" useful features.

  We can start by providing a formal system for describing feature atoms. A feature atom is a feature that could not be further broken down, for instance a node label or tree edge. Given feature atoms, we would need to provide specification for combining atoms into feature molecules. These specifications are rules that specify how to "connect" feature atoms. For instance, one rule could be that two labels must be connected by a tree edge; this for instance would be a specification for a parent-child pair feature. With these specifications of rules and basic feature building blocks, larger features can be constructed from smaller blocks systematically. We will hence have a framework for exploring a meta-feature space automatically. The system could be set lose to construct features, evaluate them on a test data set, and report successful features that it finds[3]. In this manner, we would have an automated feature engineering system.

6.4 Final words

Unsupervised record extraction is an interesting area of research with important implications for the future of the web and information processing. In the near term, this technology will provide useful advancements in terms of more intelligent software for end users. In the longer term it will part of a framework of technologies for automatically organizing and structuring information on the web. We believe that this research will bring us closer to the day when we will have software agents that can parse your information and also *understand* it.

[3]A cool way to have a program write your thesis for you!

Appendix A

Appendix

A.1 Overview of the classifiers

- **J48** is the Weka implementation of the classic Quinlan C4.5 decision tree [42]. A decision tree is essentially a tree composed of two types of nodes: internal nodes represent that decisions (that divide input instances into two halves based on one feature) and leaf nodes that represent the classifier outcomes. Decision tree learning usually involves a greedy algorithm that adds the most discriminating feature (using mutual information as criterion for most discriminating) each time. Decision nodes are added until all the features are exhausted or the information gain of any larger tree becomes negative. Decision trees are widely popular because of their interpretability. Furthermore, because the tree essentially splits the input space into two spaces at each node, the output classifier can form a "decision regions" regions in n-dimensional space which is often more expressive than linear separators.
- **ADTree** is a combination generalized decision tree, a voted decision tree, and a voted decision stumps tree[21]. Unlike a standard decision tree where the leaves denote the classifier outcome, for ADTrees it is the sum of the signs of decision nodes (weighted) along the path from the root to a leaf that determines the ultimate classifier outcome.
- **Support Vector machines** [17] are a class of classifiers that tries to find an optimal hyper-plane between the negative and positive classes. The decision boundary is guaranteed to be maximum margin. Namely, the distance between the closest negative and positive examples near the separation boundary is maximal. The algorithm uses quadratic programming to solve for the maximum margin hyper-plane. The implementation in Weka follows Pratt's SMO (sequential minimal optimization) [41] algorithm.
- **Linear regression** finds an optimal separation boundary which minimizes the squared error of misclassified points.

- **Logistic regression** solves the linear regression problem in a probabilistic framework. The decision boundary is still linear. However, the training instances are fitted under a logistic distribution (a sigmoid) around the boundary. The linear boundary that fits the training data with the maximum likelihood under the logistic distribution is the resulting decision boundary.

- **AdaBoost** is a popular meta classification method that is based on the principal of boosting. The classifier is composed of a series of decision stumps (linear classifiers of single dimension (or single feature)). Stumps are added at each iteration to help reduce the error (classify previously misclassified points) from in the last iteration.

- **Voted perceptron** is another meta classification method. It uses simple perceptrons as components. A simple perceptron consists of a weight vector that is updated at each step as training data is fed (in an online fashion) to the algorithm. A weight update occurs when the perceptron makes an error on the incoming training point. A weight adjustment is made to minimize that error. After all the training data have been seen, the final weights become the weights of the trained perceptron. In the voted perceptron, each time the weight vector is adjusted the previous value is saved. If the training of the simple perceptron produced k weight adjustments then for the voted perceptron we would have k saved weights; this is equivalent to having k differently trained perceptrons. The voted perceptron classifies a new instance by feeding the test instance to all of its k perceptrons and returning the answer which received majority vote from its k components.

A.2 A Comparison of Clustering Algorithms

We performed an earlier study evaluating the performance of several pair-wise features, Bag of Tags (BOT), Parent Child Pairs (PCP), and Tree Alignment under two clustering algorithms K-Means and Hierarchical clustering.

In our early study, we collected 20 web pages and tested various metrics to compare the performance of HAC versus K-Means. In each case, we varied the parameters to adjust the affects of the clusters. For K-Means we adjusted the number of starting clusters, and for HAC we adjusted the stopping threshold (the point at which clusters should merge). Table A-1(a) shows the results for the 3 tree features when using HAC. Table A-1(b) shows the results when using K-Means. The HAC thresholds were not classifier trained values but rather raw feature values (normalized to [0, 1]). These results show that HAC was the better choice of algorithm than K-Means.

HAC out-performed K-means under all three metrics. HAC produces clusters of minimal variance flavor because at each iteration only the closest subtrees pairs are merged to form clusters. K-Means is highly dependent on starting state. It also favors equal co-variances over minimum covariance because at each iteration only cluster means are re-estimate but not the variances. A natural

Feature	Threshold	Precision	Recall	F-Score
BOT	0	0.961	0.493	0.589
	0.025	0.236	1	0.38
	0.03	0.479	0.905	0.573
	0.05	0.16	1	0.273
	0.07	0.358	0.92	0.466
	0.1	0.223	0.906	0.315
PCP	0	1	0.487	0.607
	0.025	0.537	0.95	0.627
	0.05	0.346	0.966	0.48
	0.075	0.49	0.95	0.635
	0.1	0.39	1	0.56
Tree Alignment	0	1	0.215	0.339
	0.025	1	0.55	0.59
	0.05	1	0.8	0.875
	0.1	1	1	1

(a) Hierarchical Agglomerative Clustering (HAC)

Feature	K	Precision	Recall	F-Score
BOT	10	0.255	0.9	0.385
	20	0.464	0.9	0.545
	30	0.475	0.55	0.505
	40	0.5	0.65	0.565
PCP	10	0.485	1	0.6
	20	0.515	1	0.62
	30	0.52	0.7	0.57
	40	0.555	0.6	0.57
Tree Alignment[1]	20	0.77	1	0.87
	30	0.03	1	0.05
	40	0.03	1	0.05

(b) K-Means Clustering

Figure A-1: Comparison of K-Means and HAC clustering algorithms. Reported results are over a set of 20 test pages.

extension is to use a Mixture of Gaussian approach to cluster, in such case variance and means will be part of the clustering criterion. As possible future work, a mixture of Gaussians (MOG) model could be applied to cluster the trees. However, there still leaves open the question as to the proper k to choose. However since HAC is simple to implement and performed very well, for these reasons we chose to use it as our primary clustering algorithm.

A.3 URLs

The following is the list of 82 web pages we labelled for training and testing our system.

A.4 Classifier Internals

```
0=http://people.csail.mit.edu/people/yks/rec_ext/test1.php
2=http://people.csail.mit.edu/people/yks/rec_ext/test2.php
3=http://www.csail.mit.edu/biographies/PI/biolist.php
4=http://search.yahoo.com/search?p=flowers
5=http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=google
6=http://images.search.yahoo.com/search/images?p=star+wars&&fr=fp-pull-web-t
8=http://reviews.search.com/search?q=digital+cameras&tag=srch
10=http://autos.yahoo.com/newcars/acura.html;_ylt=Au1Uu_UXbWpRVkTOlOaxEWQ_c78F
11=http://froogle.google.com/froogle?q=machine+learning
12=http://news.search.yahoo.com/search/news/?ei=ISO-8859-1&c=&p=iraq
13=http://shopping.yahoo.com/search;_ylt=AgOmQbirQtOu66vPM46TNkOEgFoB;_ylu=X3o
DMTBhNjRqazhxBHNlYwNzZWFyY2g-?p=camera&did=
16=http://news.google.com/?ned=us&topic=n
17=http://web.ask.com/web?q=flowers&qsrc=0&o=0
18=http://search.msn.com/results.aspx?FORM=MSNH&q=robots
20=http://www.wired.com/
21=http://www.theonion.com/
22=http://www.theregister.com/
24=http://newssearch.bbc.co.uk/cgi-bin/search/results.pl?scope=newsifs&tab=news&q=breaking+news
25=http://www.csail.mit.edu/directory/directory.php
27=http://www.google.com/search?hl=en&lr=&q=flowers
28=http://groups-beta.google.com/groups?q=machine+learning&hl=en&btnG=Google+Search
29=http://search.ebay.com/ipod_Portable-Audio_W0QQfromZR3QQfsopZ1QQsacatZ15052
32=http://www2.newegg.com/ProductSort/Category.asp?Category=15
36=http://www.alltheweb.com/search?cat=web&cs=iso88591&q=search+engines&rys=0&_sb_lang=pref
37=http://search.cpan.org/search?query=wrapper+induction&mode=all
39=http://www.rottentomatoes.com/movies/box_office.php
40=http://search.tv.yahoo.com/tvtitlesearch?p=%2b{family+guy
+%2blineup%3aMA61097+%2butn%3a1114240161/1115449761&s=-$s,utn&lineup=us_MA61097
&range=14&title=family+guy&lineup_tz=America/New_York&sort=score&srch=true
48=http://scholar.google.com/scholar?q=record+extraction&ie=UTF-8&oe=UTF-8&hl=en&btnG=Search
50=http://www.npr.org/
```

Figure A-2: List of URLs of labelled web pages

```
51=http://www.bookpool.com/ss?qs=information+retrieval&x=0&y=0
55=http://hotjobs.yahoo.com/jobseeker/jobsearch/search_results.html?keywords_all=java&
country1=USA&search_type_form=quick&updated_since=sixtydays&basicsearch=0&advancedsearch=0
&metro_area=1&search=Search&kw=java
57=http://story.news.yahoo.com/news?tmpl=index&cid=1682
58=http://essentialvermeer.20m.com/vermeer_painting_part_two.htm
60=http://search.gallery.yahoo.com/search/corbis?p=skyscraper&strip=2
62=http://coppermine.sourceforge.net/demo/thumbnails.php?album=3
65=http://www.csd.cs.cmu.edu/people/faculty.html
67=http://www.cs.uiuc.edu/directory/directory.php?type=faculty
69=http://www.cs.utah.edu/dept/who/faculty.html
70=http://www.amazon.com/exec/obidos/tg/browse/-/1067694/ref%3Dbr%5Fbx%5Fc%5F1%5F4/102-9942286
-8736122l/002-2188614-1242441
74=http://physicsweb.org/events
76=http://video.google.com/videosearch?q=machine+learning&btnG=Search+Video
77=http://video.search.yahoo.com/search/video?ei=UTF-8&fr=sfp&p=machine+learning
78=http://www.alltheweb.com/search?cat=web&cs=iso88591&q=record+extraction&rys=0&_sb_lang=pref
79=http://www.alltheweb.com/search?cat=img&cs=iso88591&q=record+extraction&rys=0&_sb_lang=pref
80=http://www.alltheweb.com/search?cat=vid&cs=iso88591&q=mit&rys=0
82=http://search.ebay.com/bicycle_W0QQfrppZ50QQfsopZ1QQmaxrecordsreturnedZ300
84=http://www.ftd.com/528/catalog/category.epl?index_id=occasion_birthday
85=http://ww1.1800flowers.com/productform.asp?section_id=0&search_type=s_collection&search_value=82&sc=1001
87=http://www.bluenile.com/product_catalog.asp?catid=07&oid=355&track=c2m2&elem=hdr&mod=basic
88=http://www.overstock.com/cgi-bin/d2.cgi?PAGE=CATLIST&PRO_SUB_CAT=457&PRO_SSUB_CAT=1200
91=http://www.washington.edu/students/crscat/cse.html
93=http://www.tvtome.com/Office_US/eplist.html
94=http://www.netflix.com/BrowseSelection?hnjr=3&sgid=gnr
95=http://www.cs.ubc.ca/people/faculty.jsp
96=http://www.cs.cornell.edu/People/Researchers/index.htm
97=http://web.mit.edu/bcs/people/faculty.shtml
98=http://web.mit.edu/bcs/people/gradstudents.shtml
99=http://www.cs.caltech.edu/people.html
```

Figure A-3: List of URLs of labelled web pages (continued)

```
100=https://addons.update.mozilla.org/extensions/showlist.php?application=firefox&numpg=10&category=Search%20Tools
102=http://ifc.mit.edu/summer_housing.php
103=http://www.csail.mit.edu/events/eventcalendar/calendar.php?show=series&id=11
104=http://www.csail.mit.edu/events/eventcalendar/calendar.php
105=http://www.nato.int/cv/hsg/cv-hos.htm
107=http://www.parl.gc.ca/information/about/people/key/pm/index.asp?lang=E&param=mp
109=http://www.cs.mcgill.ca/people/faculty.photo.html
110=http://www.cs.brown.edu/people/faculty/
111=http://www.towerrecords.com/Music/Default.aspx?free_text=beck&a_uid=USP828_i03-050504-00%3a48%3
a13-509470&a_qid=USP828_i03-050504-00%3a48%3a13-509471&genre=Blues&
112=http://jobsearch.monster.com/jobsearch.asp?q=information+extraction&cn=&sort=rv&vw=b&cy=US&re=14
&brd=1%2C1862%2C1863
114=http://www.dogpile.com/info.dogpl/search/web/machine%2Blearning
117=http://www.google.com/search?sourceid=mozclient&ie=utf-8&oe=utf-8&q=hotel+rwanda
118=http://searchenginewatch.com/
119=http://search.barnesandnoble.com/booksearch/results.asp?WRD=harry%20potter&userid=GI14GSOBSX
120=http://search.barnesandnoble.com/booksearch/results.asp?WRD=hitchhiker&userid=GI14GSOBSX
122=http://search.looksmart.com/p/search?qt=clustering&tb=web&search=Search
123=http://www.download.com/Voice-Recognition/3150-7239_4-0.html?tag=dir
124=http://music.download.com/3605-8741_32-0.html?tag=head_edpix&ep=1
125=http://www.download.com/Strategy-Games/2001-10189_4-0.html?tag=more_cgr
126=http://reviews.cnet.com/4502-6501_7-0.html?tag=ont.8mp&501375id=9255772
128=http://www.cs.byu.edu/faculty/
129=http://www.cs.vt.edu/site_pages/facultystaff/facultystaff_faculty.php
131=http://www.cs.princeton.edu/people/fac.php
132=http://www.audible.com/adbl/store/homepage_wg.jsp?BV_SessionID=@@@@0009314672.1115196928@@@@
&BV_EngineID=ccceaddeidmhiidcefecegedfhfdhfm.0&uniqueKey=1115196935328&hwtab=1&switchTo=browseAudio
134=http://movies.yahoo.com/mv/boxoffice/daily/
135=http://answers.google.com/answers/browse?catid=1201
```

Figure A-4: List of URLs of labelled web pages (continued)

(a) Cluster Size

(b) Cluster Size (Ranked)

(c) Cluster Size (Inverse Ranked)

(d) Content Coverage

(e) Content Coverage (Ranked)

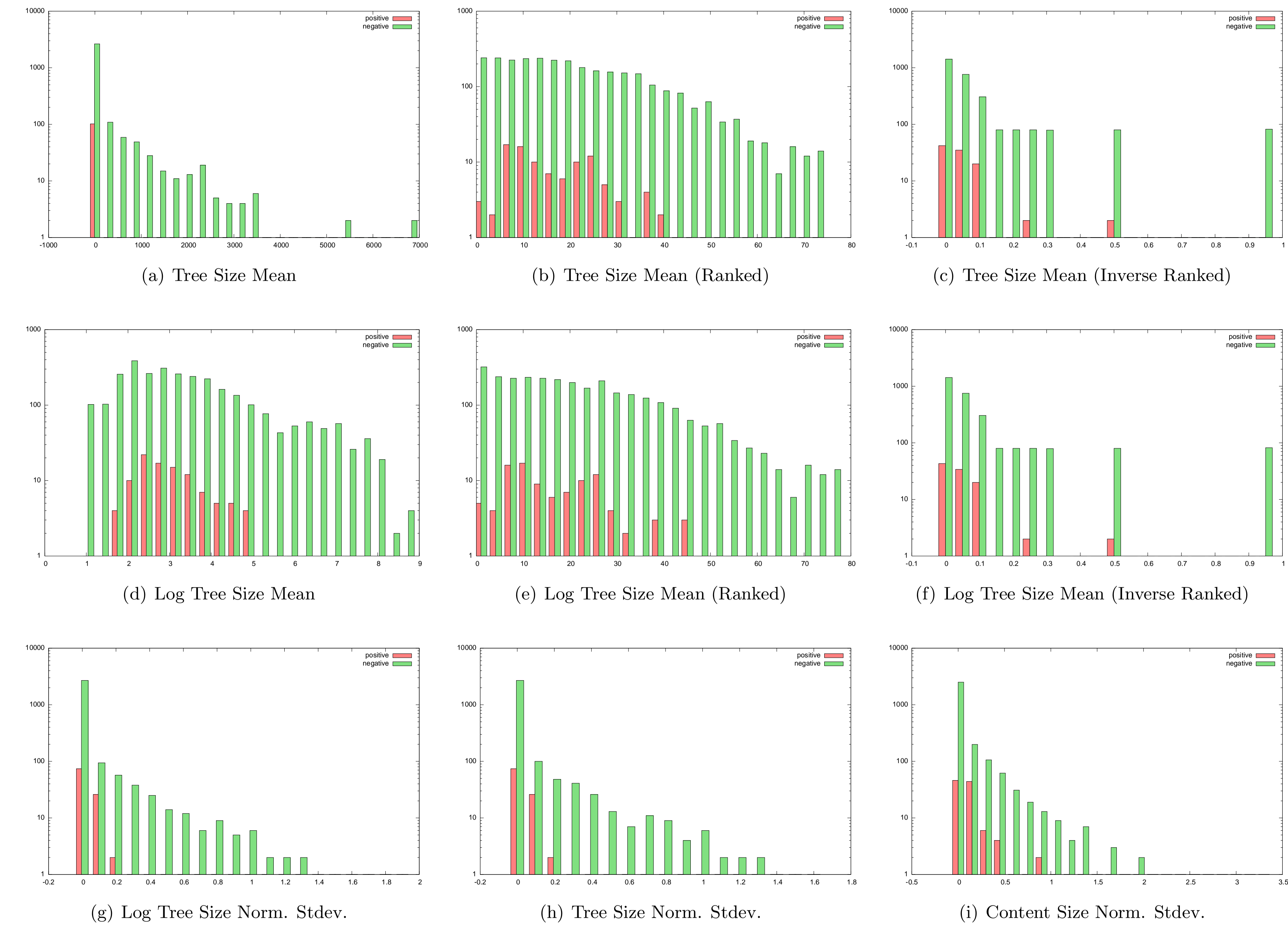
(f) Content Coverage (Inverse Ranked)

(g) Content Size Mean

(h) Content Size Mean (Ranked)

(i) Content Size Mean (Inverse Ranked)

Figure A-5: Comparison of tree/content based cluster metrics



(a) Tree Size Mean

(b) Tree Size Mean (Ranked)

(c) Tree Size Mean (Inverse Ranked)

(d) Log Tree Size Mean

(e) Log Tree Size Mean (Ranked)

(f) Log Tree Size Mean (Inverse Ranked)

(g) Log Tree Size Norm. Stdev.

(h) Tree Size Norm. Stdev.

(i) Content Size Norm. Stdev.

Figure A-6: Comparison of tree/content based cluster metrics

```
: -0.38
|  (1)TreeAlign < 0.471: -2.497
|  |  (3)PCP < 0.787: -1.009
|  |  |  (6)TreeAlign < 0.394: -2.545
|  |  |  (6)TreeAlign >= 0.394: 1.618
|  |  |  |  (10)TreeContext < 0.954: 1.808
|  |  |  |  (10)TreeContext >= 0.954: -1.531
|  |  (3)PCP >= 0.787: 1.561
|  |  |  (5)TreeContext < 0.988: -2.644
|  |  |  (5)TreeContext >= 0.988: 2.127
|  (1)TreeAlign >= 0.471: 2.276
|  |  (4)TreeAlign < 0.582: -1.948
|  |  |  (8)PCP < 0.675: -0.376
|  |  |  (8)PCP >= 0.675: 1.64
|  |  (4)TreeAlign >= 0.582: 0.043
|  (2)TreeContext < 0.951: -3.32
|  |  (9)PCP < 0.829: -1.621
|  |  (9)PCP >= 0.829: 1.574
|  (2)TreeContext >= 0.951: 0.575
|  |  (7)PCP < 0.995: -0.276
|  |  (7)PCP >= 0.995: 0.6
Legend: -ve = non-record, +ve = record
Tree size (total number of nodes): 31
Leaves (number of predictor nodes): 21
Num Features Trained On: 3
Features Trained On: PCP, TreeAlign, TreeContext
```

Figure A-7: The internals of baseline pair-classifier.

```
         2.7127 * (normalized) ContiguityIndicator
 +       0.2169 * (normalized) FilledCoverRatio
 +      -0.1073 * (normalized) ContentGapStdev
 +      -0.6993 * (normalized) InternalPeriodicity
 +      -0.7654 * (normalized) ExternalPeriodicity
 +       0.0297 * (normalized) PeriodicityDifference
 +       1.3083 * (normalized) InternalGZipCompression
 +       0.0434 * (normalized) ExternalGZipCompression
 +       0.6324 * (normalized) GZipCompressionDifference
 +      -0.9867 * (normalized) InternalMeanDissimilarity
 +      -0.3502 * (normalized) ExternalMeanDissimilarity
 +      -0.3183 * (normalized) MeanDissimilarityDifference
 +      -0.1839 * (normalized) TreeSizeMean
 +       0.4991 * (normalized) TreeSizeMean.Ranked
 +      -2.2973 * (normalized) TreeSizeMean.Inv-Ranked
 +      -2.2859 * (normalized) TreeSizeNormStdev
 +      -0.9016 * (normalized) LogTreeSizeMean
 +      -0.0472 * (normalized) LogTreeSizeMean.Ranked
 +      -2.2633 * (normalized) LogTreeSizeMean.Inv-Ranked
 +      -1.857  * (normalized) LogTreeSizeNormStdev
 +       0.2184 * (normalized) ContentSizeMean
 +       1.8033 * (normalized) ContentSizeMean.Ranked
 +      -0.9532 * (normalized) ContentSizeMean.Inv-Ranked
 +       0.5431 * (normalized) ContentSizeNormStdev
 +       2.3481 * (normalized) ContentCoverage
 +      -1.087  * (normalized) ContentCoverage.Ranked
 +      -0.4659 * (normalized) ContentCoverage.Inv-Ranked
 +      -1.9317 * (normalized) ClusterSize
 +      -4.3132 * (normalized) ClusterSize.Ranked
 +      -0.8062 * (normalized) ClusterSize.Inv-Ranked
 -       1.2055

Number of kernel evaluations: 64592 (75.793\% cached)

Logistic Regression with ridge parameter of 1.0E-8
Coefficients...
Variable      Coeff.
       1      -2.7522
Intercept      0.7165

Odds Ratios...
Variable         O.R.
       1       0.0638
```

Figure A-8: The internals of baseline cluster-classifier.

```
J48 pruned tree
------------------

ContiguityIndicator <= 0.333333: non-record (1300.0)
ContiguityIndicator > 0.333333
|   ContentSizeMean <= 17.2: non-record (55.0)
|   ContentSizeMean > 17.2
|   |   ClusterSize <= 2: non-record (29.0)
|   |   ClusterSize > 2
|   |   |   LogTreeSizeNormStdev <= 0.213708
|   |   |   |   ContiguityIndicator <= 0.681818
|   |   |   |   |   ContiguityIndicator <= 0.4: record (30.0)
|   |   |   |   |   ContiguityIndicator > 0.4: non-record (13.0)
|   |   |   |   ContiguityIndicator > 0.681818
|   |   |   |   |   InternalMeanDissimilarity <= 0.790013
|   |   |   |   |   |   ClusterSize.Ranked <= 5: record (1390.0/10.0)
|   |   |   |   |   |   ClusterSize.Ranked > 5
|   |   |   |   |   |   |   PeriodicityDifference <= 0.645492: non-record (6.0)
|   |   |   |   |   |   |   PeriodicityDifference > 0.645492: record (30.0)
|   |   |   |   |   InternalMeanDissimilarity > 0.790013
|   |   |   |   |   |   FilledCoverRatio <= 0.985915: record (30.0)
|   |   |   |   |   |   FilledCoverRatio > 0.985915: non-record (9.0)
|   |   |   LogTreeSizeNormStdev > 0.213708
|   |   |   |   ContentSizeMean.Ranked <= 5: record (30.0)
|   |   |   |   ContentSizeMean.Ranked > 5: non-record (23.0)
Number of Leaves  :     12
Size of the tree :      23
```

Figure A-9: A decision tree constructed using 30 features using the J48 classifier implementation (from Weka). Note that only about 10 features actually make it into the tree, and not all of them need to be any of the top features.

Appendix B

Tables

B.1 Pattern Detection

Classifier	10-fold-CV	training	lopo-CV	Test
ADTree-1	0.009361	0.004784	0.007887	0.015998
J48-1	0.012805	0.001897	0.016151	0.018619
Logistic-1	0.006722	0.004660	0.013653	0.034106
SMO-1	0.008846	0.005155	0.014803	0.036439
AdaBoostM1-1	0.010495	0.006124	0.013613	0.037439
MultiBoostAB-1	0.010495	0.006124	0.013613	0.037439
VotedPerceptron-1	0.007877	0.006062	0.012663	0.039949
LinearRegression-1	0.008474	0.008207	0.014162	0.040016

Table B.1: Error rates while varying the type of classifer. All errors are reported for all 9 pair features

Features	10-fold-CV	Training	lopo-CV	Test
TreeContext	0.03786	0.03786	0.03893	0.03120
TreeAlign	0.01155	0.00843	0.01710	0.04008
PCP	0.06603	0.05885	0.05685	0.06279
BagOfTags	0.07842	0.06293	0.06876	0.06201
PCCT	0.07027	0.06131	0.05812	0.07332
TreeSize	0.11434	0.08822	0.09798	0.07837
TreeHash	0.09710	0.09710	0.09183	0.07881
ContentSize	0.11803	0.11461	0.11632	0.14413
TreeRelated	0.31869	0.31869	0.29113	0.31786

Table B.2: Error rates for single features trained on a ADTree classifier

B.2 Record Detection

FEATURE-NAMES	10-FOLD-CV	TRAINING	LOPO-CV	TEST
BagOfTags,TreeContext	0.00920	0.00858	0.01015	0.01513
TreeSize,TreeContext	0.04477	0.01905	0.03253	0.01846
PCP,TreeContext	0.01544	0.00949	0.01435	0.01886
PCCT,TreeContext	0.01116	0.00810	0.01237	0.01900
ContentSize,TreeContext	0.02412	0.01301	0.01638	0.02508
TreeContext,TreeRelated	0.03786	0.03765	0.03893	0.03102
TreeHash,TreeContext	0.03869	0.03741	0.03980	0.03120
TreeAlign,TreeContext	0.00823	0.00454	0.01263	0.03704
BagOfTags,TreeAlign	0.01035	0.00654	0.01449	0.04010
TreeAlign,ContentSize	0.01113	0.00804	0.01642	0.04119
PCP,TreeAlign	0.01027	0.00728	0.01299	0.04144
PCCT,TreeAlign	0.00854	0.00561	0.01728	0.04262
TreeAlign,TreeHash	0.01169	0.00843	0.01613	0.04248
TreeAlign,TreeRelated	0.01037	0.00843	0.01537	0.04248
TreeAlign,TreeSize	0.01182	0.00883	0.01916	0.04402
BagOfTags,TreeHash	0.08512	0.04702	0.07580	0.04677
PCP,TreeSize	0.05598	0.04260	0.03956	0.05224
PCP,TreeRelated	0.05930	0.05190	0.05108	0.05406
PCP,ContentSize	0.05310	0.05048	0.04307	0.05315
PCCT,TreeSize	0.06685	0.04879	0.05283	0.05519
PCCT,ContentSize	0.05821	0.05366	0.05505	0.05661
BagOfTags,ContentSize	0.05390	0.04920	0.04624	0.05712
PCP,TreeHash	0.06939	0.04965	0.06546	0.06026
BagOfTags,TreeRelated	0.06689	0.05646	0.05633	0.06092
PCCT,TreeHash	0.07225	0.05803	0.07185	0.05852
PCP,PCCT	0.06464	0.05724	0.05652	0.06308
BagOfTags,PCP	0.07240	0.05254	0.05836	0.06535
TreeSize,ContentSize	0.07811	0.06298	0.06896	0.06894
PCCT,TreeRelated	0.06302	0.05636	0.05046	0.06552
BagOfTags,PCCT	0.06953	0.05502	0.05462	0.06921
TreeSize,TreeRelated	0.11094	0.08628	0.09378	0.07721
BagOfTags,TreeSize	0.05704	0.04413	0.04612	0.07463
TreeHash,TreeRelated	0.09673	0.09673	0.09151	0.07861
TreeHash,TreeSize	0.12182	0.08234	0.12579	0.08194
TreeHash,ContentSize	0.11190	0.09331	0.12141	0.11029
ContentSize,TreeRelated	0.11467	0.11139	0.11320	0.14162

Table B.3: Error rates for two-feature combinations trained using an ADTree classifier

Features	10-fold-CV	Training	Lopo-CV	Test
BagOfTags,TreeSize,TreeContext	0.01295	0.00897	0.01401	0.01369
BagOfTags,TreeContext,TreeRelated	0.00918	0.00858	0.01010	0.01373
BagOfTags,TreeHash,TreeContext	0.01126	0.00749	0.01218	0.01469
BagOfTags,PCP,TreeContext	0.01398	0.00656	0.01313	0.01549
PCP,TreeAlign,TreeContext	0.00812	0.00334	0.01166	0.01706
TreeHash,TreeSize,TreeContext	0.04491	0.01798	0.03289	0.01844
PCP,TreeContext,TreeRelated	0.01180	0.00953	0.00947	0.01831
PCP,TreeSize,TreeContext	0.01171	0.00491	0.00951	0.01851
PCP,TreeHash,TreeContext	0.01349	0.00916	0.01149	0.01853
BagOfTags,PCCT,TreeContext	0.02615	0.00810	0.01237	0.01900
PCCT,TreeHash,TreeContext	0.01021	0.00843	0.01014	0.01900
PCCT,TreeContext,TreeRelated	0.01068	0.00810	0.00817	0.01900
PCCT,TreeSize,TreeContext	0.01144	0.00623	0.01087	0.01913
PCCT,ContentSize,TreeContext	0.01177	0.00582	0.01185	0.01962
PCP,PCCT,TreeContext	0.01054	0.00487	0.00886	0.01946
PCP,ContentSize,TreeContext	0.00893	0.00614	0.01033	0.01995
BagOfTags,ContentSize,TreeContext	0.00973	0.00697	0.01100	0.02046
TreeSize,TreeContext,TreeRelated	0.04415	0.02204	0.02823	0.02122
BagOfTags,TreeAlign,TreeContext	0.00810	0.00328	0.01518	0.02193
TreeSize,ContentSize,TreeContext	0.01709	0.01190	0.01762	0.02269
ContentSize,TreeContext,TreeRelated	0.02439	0.01281	0.01638	0.02491
TreeHash,ContentSize,TreeContext	0.01664	0.01299	0.01932	0.02511
PCCT,TreeAlign,TreeContext	0.00705	0.00328	0.00880	0.02664
TreeHash,TreeContext,TreeRelated	0.03864	0.03720	0.03976	0.03102
PCP,TreeAlign,ContentSize	0.00984	0.00726	0.01374	0.03317
TreeAlign,TreeHash,TreeContext	0.00819	0.00466	0.01260	0.03517
TreeAlign,TreeSize,TreeContext	0.00823	0.00466	0.01195	0.03517
TreeAlign,TreeContext,TreeRelated	0.00823	0.00454	0.01263	0.03704
TreeAlign,ContentSize,TreeContext	0.00928	0.00526	0.01292	0.03735
BagOfTags,TreeAlign,TreeRelated	0.00965	0.00654	0.01606	0.04008
BagOfTags,PCP,TreeAlign	0.00907	0.00652	0.01613	0.04008
BagOfTags,PCCT,TreeAlign	0.01396	0.00594	0.01610	0.04008
BagOfTags,TreeAlign,TreeHash	0.00986	0.00654	0.01649	0.04010
BagOfTags,TreeAlign,TreeSize	0.00981	0.00654	0.01466	0.04010
BagOfTags,TreeHash,TreeRelated	0.06499	0.04687	0.04808	0.04206
PCCT,TreeAlign,ContentSize	0.01379	0.00716	0.01762	0.04079
BagOfTags,TreeAlign,ContentSize	0.01097	0.00765	0.01600	0.04106
PCP,TreeAlign,TreeRelated	0.01035	0.00707	0.01292	0.04099
TreeAlign,ContentSize,TreeRelated	0.01113	0.00804	0.01668	0.04119
PCP,PCCT,TreeAlign	0.01035	0.00728	0.01364	0.04144
PCP,TreeAlign,TreeHash	0.00948	0.00728	0.01300	0.04144
PCP,TreeAlign,TreeSize	0.00967	0.00728	0.01310	0.04144
TreeAlign,TreeHash,ContentSize	0.01078	0.00885	0.01681	0.04250
TreeAlign,TreeSize,ContentSize	0.01142	0.00837	0.01654	0.04264
PCCT,TreeAlign,TreeHash	0.00957	0.00561	0.01738	0.04262
PCCT,TreeAlign,TreeSize	0.00981	0.00561	0.01661	0.04262
PCCT,TreeAlign,TreeRelated	0.01043	0.00561	0.01707	0.04262

Table B.4: Error rates for three-feature combinations trained using an ADTree classifier

	max fscore	stdev	max precision	stdev	max recall	stdev
single-link	0.90903	0.19754	0.90109	0.22595	0.94043	0.1659 4
5-link	0.90110	0.18709	0.91546	0.20206	0.91138	0.18961
10-link	0.88645	0.18979	0.91335	0.20429	0.88813	0.20012
50-link	0.89054	0.17837	0.93817	0.17920	0.87164	0.20592
100-link	0.89874	0.17712	0.95278	0.16510	0.86867	0.20579
average-link	0.90784	0.17069	0.94143	0.17381	0.89017	0.188 68
complete-link	0.89521	0.17769	0.96194	0.16000	0.85556	0.21 017
complete-link-merged	0.93667	0.11507	0.97693	0.09839	0.91168	0.14761

Table B.5: Clustering results for different criterion

Feature	10-fold-CV	Training	lopo-CV	Test	Recall	Precision	f-score
ContiguityIndicator	0.08627	0.08629	0.08944	0.07560	0.98077	0.30178	0.46154
ClusterSize.Inv-Ranked	0.10239	0.10234	0.12251	0.08895	0.73077	0.23171	0.35185
ClusterSize	0.10026	0.10301	0.10793	0.10610	0.73077	0.19895	0.31276
ContentCoverage	0.13979	0.14181	0.15876	0.13977	0.65385	0.14407	0.23611
TreeSizeNormStdev	0.12702	0.12843	0.12884	0.14104	0.63462	0.13983	0.22917
LogTreeSizeNormStdev	0.12769	0.12910	0.12884	0.14295	0.63462	0.13808	0.22680
ClusterSize.Ranked	0.14994	0.14983	0.18271	0.14549	0.80769	0.16092	0.26837
ExternalPeriodicity	0.15655	0.15452	0.16365	0.15121	0.71154	0.14231	0.23718
GZipCompressionDifference	0.17127	0.17057	0.17159	0.15629	0.34615	0.07826	0.12766
ExternalGZipCompression	0.17598	0.17926	0.18035	0.16518	0.36538	0.07724	0.12752
ContentSizeNormStdev	0.17379	0.17458	0.17979	0.17408	0.67308	0.11986	0.20349
PeriodicityDifference	0.18001	0.17993	0.18192	0.20013	0.67308	0.10511	0.18182
ExternalMeanDissimilarity	0.27156	0.26488	0.27163	0.28081	0.67308	0.07609	0.13672
MeanDissimilarityDifference	0.35915	0.35652	0.35031	0.35959	0.69231	0.06143	0.11285
ContentCoverage.Ranked	0.53400	0.52174	0.62697	0.44917	0.76923	0.05442	0.10165
ContentSizeMean.Ranked	0.53872	0.52776	0.63111	0.46569	0.80769	0.05490	0.10282
InternalPeriodicity	0.55579	0.54381	0.52195	0.55400	0.61538	0.03620	0.06838
LogTreeSizeMean	0.54389	0.55385	0.55544	0.56226	0.65385	0.03774	0.07135
TreeSizeMean	0.66229	0.66087	0.66156	0.67154	0.80769	0.03857	0.07362
LogTreeSizeMean.Ranked	0.77894	0.76321	0.88888	0.67217	0.88462	0.04189	0.08000
TreeSizeMean.Inv-Ranked	0.67076	0.67960	0.60409	0.71283	0.65385	0.02988	0.05714
LogTreeSizeMean.Inv-Ranked	0.67076	0.67960	0.60463	0.71283	0.65385	0.02988	0.05714
FilledCoverRatio	0.78340	0.78328	0.78476	0.76811	0.98077	0.04051	0.07780
ContentSizeMean	0.78874	0.79064	0.78980	0.80750	0.80769	0.03223	0.06199
InternalGZipCompression	0.86021	0.85886	0.85339	0.87929	0.96154	0.03492	0.06739
ContentGapStdev	0.74855	0.92241	0.89402	0.90851	1.00000	0.03509	0.06780
InternalMeanDissimilarity	0.71288	0.96656	0.88179	0.96696	1.00000	0.03304	0.06396
TreeSizeMean.Ranked	0.72344	0.96656	0.91080	0.96696	1.00000	0.03304	0.06396
ContentSizeMean.Inv-Ranked	0.73399	0.96656	0.93615	0.96696	1.00000	0.03304	0.06396
ContentCoverage.Inv-Ranked	0.73869	0.96656	0.90664	0.96696	1.00000	0.03304	0.06396

Table B.6: Error rates for single features trained on a SVM classifier

Classifier	10-fold-CV	Training	lopo-CV	Test	Recall	Precision	F-score
J48	0.01738	0.00669	0.01642	0.02224	0.78846	0.63077	0.70085
Logistic	0.04411	0.03077	0.04673	0.03367	0.73077	0.49351	0.58915
ADTree	0.02673	0.01940	0.02771	0.03558	0.73077	0.47500	0.57576
SVM	0.05753	0.04749	0.05791	**0.04765**	0.96154	0.40650	0.57143
LinearRegression	0.08692	0.08361	0.09086	0.06734	0.98077	0.32692	0.49038
AdaBoostM1	0.06421	0.07492	0.07515	0.07052	0.98077	0.31677	0.47887
MultiBoostAB	0.06421	0.07492	0.07515	0.07052	0.98077	0.31677	0.47887

Table B.7: Varying classifier, using all features, reporting cross-validation, training and testing errors; also with test precision, recall and fscores

Bibliography

[1] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage, 2003.

[2] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[3] Philip Bille. Tree edit distance, alignment distance and inclusion.

[4] D. Buttler, L. Liu, and C. Pu. A fully automated object extraction system for the world wide web. In *Proceedings of the 2001 International Conference on Distrubuted Computing Systems (ICDCS'01)*, pages 361–370, Phoenix, Arizona, May 2001.

[5] David Buttler, Ling Liu, and Calton Pu. A fully automated extraction system for the world wide web. In *IEEE ICDCS-21*, April 2001.

[6] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. Vips: a vision-based page segmentation algorithm, November 2003.

[7] M. E. Califf and R. J. Mooney. Relational learning of pattern-match rules for information extraction. In *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pages 6–11, Menlo Park, CA, 1998. AAAI Press.

[8] Davi De Castro. Automatic web news extraction using tree edit distance. In *WWW*, 2004.

[9] James Caverlee, Ling Liu, and David Buttler. Probe, cluster, and discover: Focused extraction of qa-pagelets from the deep web. In *ICDE'04*, 2004.

[10] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40(2):135–158, 2001.

[11] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40(2):135–158, 2001.

[12] Weimin Chen. New algorithm for ordered tree-to-tree correction problem. *J. Algorithms*, 40(2):135–158, 2001.

[13] Michael Collins and Nigel Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *ACL*, 2002.

[14] Michael Collins and Scott Miller. Semantic tagging using a probabilistic context free grammar. In *6th Workshop on Very Large Corpora.*, 1998.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[16] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.

[17] N. Cristianini and J. Shawe-Taylor. *An Introduction to support vector machines (and other kernel-based learning methods)*. Cambridge University Press, 2000.

[18] M. Dash and H. Liu. Feature selection for classification, 1997.

[19] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[20] D. Embley, S. Jiang, and Y. Ng. Record-boundary discovery in web documents, 1999.

[21] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm,. In *Proc. 16th International Conf. on Machine Learning*, pages 124–133. Morgan Kaufmann, San Francisco, CA, 1999.

[22] Mark Hall. Feature subset selection: A correlation based filter approach. Doctoral Thesis, April 1999.

[23] Andrew Hogue and David Karger. Tree pattern inference and matching for wrapper induction on the world wide web. Master's thesis, Massachusetts Institute of Technology, May 2004.

[24] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. In *Information Systems*, volume 23, pages 521–538, 1998.

[25] D. Huynh, D. Karger, and D. Quan. Haystack: A platform for creating, organizing and visualizing in-formation using rdf, 2002.

[26] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[27] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees — an alternative to tree edit. *Journal of Theory of Computation*, 143(1):137–148, May 1995.

[28] Thorsten Joachim. Java wrapper for thorsten joachim's svmlight.

[29] B. Katz, S. Felshin, D. Yuret, A. Ibrahim, J. Lin, G. Marton, A. McFarland, and B. Temelkuran. Omnibase: Uniform access to heterogeneous data for question answering, 2002.

[30] Boris Katz and Jimmy J. Lin. Start and beyond.

[31] Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[32] Kristina Lerman, Lise Getoor, Steven Minton, and Craig Knoblock. Using the structure of web sites for automatic segmentation of tables. In *Proc. of SIGMOD-2004*, 2004.

[33] Ming Li and Paul M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, Berlin, 1993.

[34] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge Discovery and Data Mining*, pages 169–178, 2000.

[35] Vipin Kumar Michael Steinbach, George Karypis. A comparison of document clustering techniques, 2001.

[36] Krisztian Monostori, Arkady Zaslavsky, and Heinz Schmidt. Efficiency of data structures for detecting overlaps in digital documents. In *Proceedings of the 24th Australasian conference on Computer science*, 2001.

[37] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.

[38] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(48):443–453, 1970.

[39] C Nevill-Manning and I Witten. Identifying hierarchical structure in sequences: A linear-time algorithm, 1997.

[40] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm, 1997.

[41] John C. Platt. *Fast training of support vector machines using sequential minimal optimization.* MIT Press, Cambridge, MA, USA, 1999.

[42] J. R. Quinlan. Induction of decision trees. In Jude W. Shavlik and Thomas G. Dietterich, editors, *Readings in Machine Learning.* Morgan Kaufmann, 1990. Originally published in *Machine Learning* 1:81–106, 1986.

[43] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann. San Mateo, CA, 1993.

[44] Lakshmish Ramaswamy, Arun Lyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically, May 2004.

[45] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[46] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[47] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools with Java implementations.* Morgan Kaufmann, San Francisco, 2000.

[48] Shipeng Yu, Deng Cai, Ji-Rong Wen, and Wei-Ying Ma. Improving pseudo-relevance feedback in web information retrieval using web page segmentation, 2002.