DESIGN MINING THE WEB

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ranjitha Sampath Kumar
September 2013

This dissertation is online at: http://purl.stanford.edu/fm826nq2964

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Scott Klemmer, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Terry Winograd

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The billions of pages on the Web today provide an opportunity to understand design practice on a massive scale. Each page comprises a concrete example of visual problem solving, creativity, and aesthetics. In recent years, data mining and knowledge discovery have revolutionized the Web, driving search engines, advertising platforms, and recommender systems that are used by more than two billion people every day. However, traditional data mining techniques tend to focus on the *content* of Web pages, ignoring how that content is *presented*. What could we learn from *mining design*?

This thesis introduces design mining for the Web, and presents a scalable software platform for Web design mining called *Webzeitgeist*. Webzeitgeist consists of a repository of pages processed into data structures that facilitate large-scale design knowledge extraction. With Webzeitgeist, users can find, understand, and leverage visual design data in Web applications. In this dissertation, I demonstrate how software tools built on top of Webzeitgeist can be used to dynamically curate design galleries, search for design alternatives, retarget content between page designs, and even predict the semantic role of page elements from design data.

As more and more creative work is done digitally and shared in the cloud, Webzeitgeist illustrates how design mining principles can be applied to benefit content creators and consumers.

Acknowledgments

None of the work described in this thesis would have been possible without the patient and unwavering support of my advisor, Scott Klemmer. Scott challenged me to reach beyond low-hanging fruit and believed in the promise of my ideas even when I was unsure if I would ever see them realized. Few of the items on my research agenda resulted in immediate payoffs, but Scott always had my back, insisting that he would rather publish the *right* paper next week than *a* paper today. I am deeply and sincerely grateful to him for role modeling the kind of relationship I hope to have with my own graduate students: in my five years as his advisee, I never once saw him make a decision that wasn't in his students' best interests.

I would also like to thank my committee members — Pat Hanrahan, Terry Winograd, Noah Goodman, and Michael Bernstein — for the generosity they exhibit with their time, as well as their useful feedback, advice, and encouragement. I will never forget the email Pat wrote to me as an undergraduate, inviting me to stop by sometime and talk about research, which launched my academic career.

This thesis is the product of the blood, sweat, and tears of a group of talented, passionate, and dedicated comrades. Salman Ahmad worked a full-time internship one summer and came into the office almost every night to help ship Bricolage. Arvind Satyanarayan juggled two other research projects half a world away in Paris, but still made the time to implement many of the core components of Webzeitgeist. Maxine Lim ran user studies for one paper while simultaneously leading the implementation effort on another, and somehow still had enough steam left to stay up for forty-eight hours before the CHI deadline polishing writing and tweaking figures. Cesar Torres implemented the design-based search interface in between preparing

his senior show for his art degree. It has been an honor and a privilege to call these people my collaborators, along with Lingfeng Yang, Noah Goodman, Radomír Měch, and Tim Roughgarden.

There are many others to whom I'm indebted: Ron Fedkiw for my first research experience; Hector Garcia-Molina and Andreas Paepcke for teaching me about databases when I knew embarrassingly little; Andrew Ng, Chuong Do and Richard Socher for walking me through any number of machine learning algorithms; David Karger for stimulating conversations on distance metrics and semantifying the Web; James Landay for taking his academic grandfather duties to heart and looking out for me on the job market; Katherine Breeden for always being there, including the night she showed up in the office at 1AM with snacks and groceries sixteen hours before the CHI deadline; Ewen Cheslack-Postava and Siddhartha Chauduri for many memorable moments (drunk) and fixing my makefiles (sober); all my fellow students in the HCI lab for creating such a wonderful environment to learn and work; John Gerth for patiently and repeatedly helping me understand computers; and Jillian Hess, Monica Niemiec and Verna Wong for the thousands of times that they have saved my posterior in ways administrative and otherwise.

Most importantly, I would like to thank my family for their unconditional support and love. My thesis is dedicated to my parents, Asha and Sampath, for keeping all of graduate school's trials and tribulations in perspective and sacrificing so much of their own lives to ensure that I would never have to sacrifice at all. To see the origins of my burning desire to understand the nature of creative work, one need look no further than the countless hours I spent with my mother landscaping the garden or designing jewelry and clothes.

Lastly, I would like to thank Jerry Talton, who is my co-conspirator in all creative endeavors. The best part of graduate school has been sharing this journey with him, through thick and thin, and every minute of the day.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Design is driven by data. Whether building a Web page, programming a Mars rover, or planning the menu for a dinner party, designers draw on prior work to solve new problems [105, 156, 141]. Both novices and experts alike turn to examples of previous work to reduce barriers to entry, lower the cognitive burden of routine tasks, and foster inspiration when innovation is required [80, 69, 159]. In fact, using examples to explore design variations is what Hofstadter called the very "crux of creativity" [86].

The Web provides an opportunity to use data to inform design practice on an unprecedented scale. The ready availability of creative work online has engendered a new culture of *remixing*, in which content producers in diverse fields routinely leverage, adapt, and repurpose existing artifacts in their own creations [118, 180]. For Web design in particular, each of the billions of pages on the Web today comprises a concrete example of visual problem solving, creativity, and aesthetic preference [138, 103], all in a form that can be easily accessed and shared [98].

The means and methods Web designers employ to draw on prior work, however, are largely informal and ad hoc [83]. While search engines like Google have revolutionized the process of locating *information* on the Web, no such broad support exists for finding relevant *designs* amongst hundreds of millions of extant pages. Similarly, while end-users can "view source" to inspect a page's implementation, they still shoulder most of the burden of parsing a page's code to understand its

design, and manually adapting that design to the task at hand. Without better tools to help users locate, understand, and leverage existing work, much design data on the Web remains underutilized.

This dissertation introduces *design mining* for the Web: using data mining and knowledge discovery to index, analyze, and adapt the design of Web pages. It demonstrates how many of the same technologies that drive search engines, advertising platforms, and recommender systems on the Web today [107, 123] can be repurposed to support new, useful design interactions. By focusing not on the information *content* of Web pages but the way that content is *presented*, design mining enables the development of data-driven design tools and rigorous statistical analysis of Web design patterns and trends.

This thesis makes two major contributions. First, it presents a set of principles and practices for Web design mining, embodied in a software platform called *Webzeitgeist*. Webzeitgeist consists of a repository of more than one hundred thousand Web pages processed into data structures that facilitate large-scale design knowledge extraction. Second, it demonstrates how software tools built on top of Webzeitgeist can be used to support a diverse set of design applications, including scalable search for design alternatives, automatic retargeting of content between page designs, and predicting the semantic role of page elements from design data.

1.1 Overview

This dissertation is divided into seven chapters, setting out the origins of design mining in the literature, motivating and chronicling development of a platform for design mining the Web, describing a few key applications that design mining enables, and sketching a future for how knowledge discovery and data mining may eventually transform the nature of creative work on the Web and in other domains.

Chapter 2 examines the technologies that contribute to the design of a Web page, provides a brief overview of the history of data mining, and discusses the challenges and requirements in adapting traditional Web content crawling and indexing techniques to capture design information. The chapter also surveys the

rich history of data-driven design in human computer interaction and cognitive science.

Chapter 3 introduces Webzeitgeist, a scalable platform for Web design mining. The chapter discusses the principles that underlie the Webzeitgeist architecture, the implementation of those principles in software, and the repository of more than one hundred million design elements captured by crawling more than a hundred thousand pages from the Web.

The next three chapters introduce applications built using design mining principles and data. Chapter 4 demonstrates how information retrieval technologies applied to design data can be used to enable interactions like real-time design search and dynamic curation of design galleries. Chapter 5 introduces Bricolage, a structured prediction algorithm that learns how users create correspondences between Web designs to automatically transfer the content from one page into the style and layout of another. Chapter 6 explores how design information can be used in semantic Web applications to predict the structural role of page elements from design data.

Finally, Chapter 7 summarizes the lessons learned from early design mining research, and sketches future directions for the field.

1.2 Statement on Prior Publications and Authorship

The material presented in this thesis is the result of several years of work with my advisor, Scott Klemmer, and a number of other talented and dedicated researchers. Although I initiated and led all of the projects described herein, none of them would have been possible without the efforts of my collaborators. Both Bricolage, which was published at CHI 2011 [111], and the flexible tree matching algorithm published at IJCAI 2011 [113] were joint work with Jerry Talton and Salman Ahmad; Tim Roughgarden contributed to the complexity proof in the latter paper. Similarly, both Webzeitgeist, published at CHI 2013 [110], and the work on structural semantic classifiers published as a Stanford Technical Report [119] were joint work with Jerry Talton, Arvind Satyanarayan, Maxine Lim, Cesar Torres, and Salman Ahmad.

Chapter 2

Related Work

This chapter provides a broad overview of the technologies related to the visual presentation of Web pages. It also presents a survey of prior work in Web data mining to ground the discussion of Web design mining, summarizes the literature in Web design reuse and remixing to motivate the applications described later in this dissertation, and briefly discusses the theory of example-based design.

2.1 The Design of a Page

Most Web pages are written in Hypertext Markup Language (HTML) [92]. Content (*e.g.,* text, images, etc.) is embedded in a set of nested HTML tags, which are identified by a *name* optionally followed by a list of *attributes*. For example, the HTML tag <`img src`="picture1.jpg" /> is an `img` tag with a `src` attribute that specifies the image's location (Figure 2.1, *left*).

HTML pages are displayed in Web browsers, which use a layout engine to parse the source HTML into a Document Object Model (DOM) tree. The DOM encodes a page's render-time content, style, and layout [48]. When computing a page's DOM, the browser often needs to fetch several types of external resources — images, style sheets, scripts — that are specified in the source HTML. For example, to render the HTML in Figure 2.1, the browser would need to request the image file `picture1.jpg` and Cascading Style Sheet (CSS) file `gallery.css`.

```
<html><head>
  <title>Photo Gallery</title>
  <link rel="stylesheet"
   href="gallery.css"
   type="text/css"/></head>
 <body>
  <div class="photo">
   <h3>My first photo</h3>
   <img src="picture1.jpg"/>
  </div>
  ...
 </body>
</html>
```

gallery.html

```
@import url("base.css");

img {
  border:1px solid black;
}
.photo {
  width:300px;
}
.photo h3 {
  font-weight:bold;
}

...
```

gallery.css

Figure 2.1: *Left:* A snippet of Hyper Text Markup Language (HTML) code from an image gallery Web page. *Right:* An external Cascading Style Sheets (CSS) file that describes the presentation of `gallery.html`. CSS *rules* are comprised of *selectors* that map to HTML, and *declarations* that specify style properties. Selectors can match to HTML elements tag names (`img`), `class` and `id` attributes (`.photo`), or ancestry paths in the document hierarchy (`.photo h3`).

CSS is a language for specifying the presentation of Web documents [181]. CSS *rules* describe how markup should be rendered, including the visual properties of elements (*e.g.,* dimensions, colors, fonts) and their positioning in a page's layout. Every rule has *selectors* identifying a set of document elements, and *declarations* specifying the display properties for those elements. Selectors can match HTML elements by tag name (*e.g.,* `div`, `img`), `class` and `id` attributes, or ancestry paths in the document hierarchy (Figure 2.1, *right*). Every declaration consists of a *property* (*e.g.,* `opacity`) and *value*, which can be absolute or defined relative to the value of the parent element.

CSS allows multiple and even conflicting style rules to be applied to the same element. These conflicts are resolved by the layout engine at render-time through a process called *cascading*: rules are ranked according to a priority system and the rule that has the highest priority controls the display. For example, inlined CSS (*i.e.,* style embedded in HTML) has priority over rules defined in external style sheets. Therefore, rendering a page involves cascading style rules and computing the final set of display properties for each page element.

Taken together, HTML and CSS provide a complete specification for how a layout engine should display a Web page. Display information can be accessed through a page's DOM by querying an element for all the HTML and CSS properties that are computed at render-time. Moreover, for many pages, the DOM's structure provides a close approximation to a page's visual hierarchy. As we will see, this structured representation is useful for bootstrapping page segmentation algorithms [28], identifying semantically similar elements [52], and comparing page designs.

2.2 Web Data Mining

Data mining refers to the discovery and extraction of useful knowledge from large datasets [75]. Web data mining often involves finding relevant Web documents, automatically extracting information from those documents, and generalizing patterns from that information [55]. Broadly, data mining techniques can be classified by the types of data they mine from the Web [107].

2.2.1 Content Mining

Because there are many different types of content on the Web (*e.g.* text, images, video, etc.), Web *content mining* comprises a wide array of techniques. Text-based mining approaches are often categorized into two different views on representing Web documents [107]: *information retrieval* and *databases*.

Since most of the information on the Web is unstructured [55], the *information retrieval* view typically models Web documents as *bag-of-words* vectors, treating text as an unordered collection of words [44]. Each vector component corresponds to a distinct word from a fixed dictionary, often encoding the frequency of the word in a particular document inversely weighted by how commonly it occurs in all documents [99]. These term frequency vectors are often modulated by techniques such as latent semantic analysis, which map similar documents that do not necessarily share terms closer together in "semantic" space [45]. To compare documents, retrieval systems compute cosine similarity measures between vectors [75].

The *database* view of text mining treats Web sites as databases, inferring underlying semantic schemas from semi-structured HTML data [59]. Web documents are commonly represented as Object Exchange Models (OEMs), which are edge-labeled graphs describing the semantic structure of information [4]. OEMs are useful for modeling Web information because they do not require that similar objects possess identical schemas, facilitating structure extraction [137] and data integration across multiple sites [37].

2.2.2 Structure Mining

Structure mining involves storing and analyzing the graph structure of the Web imposed by the hyperlinks between pages [44]. The HITS algorithm (Hyper-Link Induced Topic Search) introduced the concepts of *authorities* — pages that provided high quality content for a particular topic — and *hubs* — pages that cataloged good resources for a particular topic [102]. Through link analysis, HITS iteratively computes authority and hub values for each page in a Web network until an equilibrium is reached, identifying the most relevant sources for a particular topic. Google's

PageRank algorithm generalizes these ideas by modeling a random walk over the Web graph, and ranking pages proportionally to the likelihood that they will be visited in this walk [25].

2.2.3 Usage Mining

The usage data of a Web page comprises the information generated by user interactions such as text input and mouse clicks. Usage data can typically be found in user profiles and server and browser logs [107]. Through *usage mining*, individual behavior patterns can be discovered and analyzed to enable "mass customization" of Web applications [167]. For example, by tracking browsing and purchasing history, Amazon can build recommender systems that make personalized product suggestions [41]. Similarly, through *collaborative filtering*, Netflix can predict a person's media preferences by mining other people's ratings for films and TV shows [169]. Usage mining has also transformed the way people build and evaluate Web sites: companies frequently rely on A/B tests to compare alternative user interface designs, optimizing for measures such as user conversion, retention rates, and generated revenue [104].

2.2.4 Hidden/Deep Web Mining

In addition to usage data, user interactions often generate dynamic Web pages that are not directly accessible to Web crawlers which follow hypertext links [149]. For example, many Web pages are hidden behind Web forms or AJAX calls. To mine the *hidden* Web, systems often program robots to simulate user actions [128] or save usage traces from specific inputs [17]. By indexing these pages, content in hidden databases can be extracted and combined across multiple sites [59, 17].

2.2.5 Temporal Web Mining

Web mining can also be used to understand how pages have changed over *time*. The Internet Archive's Wayback Machine periodically recrawls pages to create a

temporal archive of the Web [183]. *Zoetrope* allows users to temporally query and filter Web pages through user-specified *lenses*, which track spatial regions, DOM elements and content through time [5]. *DiffIE*, a browser plug-in, caches visited pages and highlights regions of new text when they are revisited [176], helping users quickly detect when pages change [6].

2.3 Design Mining

Although prior work in Web data mining examines many different features of Web pages, most data mining systems explicitly discard style and rendering data [189, 171], deeming it too expensive to maintain and a confounding factor in content analysis. In recent years, however, researchers have begun to build systems to evaluate Web *designs* along axes such as visual styles [153], visual aesthetics [191, 151], perceived trustworthiness [121], and quality [96]. Many of these models are based on manually harvested repositories of a few dozen or hundred pages [191, 121, 153, 151]. A few systems have been built for crawling and indexing specific design attributes [96, 79], but their architectures cannot easily be repurposed for other design applications.

The goal of this thesis is to develop a *general* platform for design mining, lowering the barriers to learning different data-driven models of Web design. Webzeitgeist archives the entire DOM structure and all of the render-time properties for each DOM element of every page it crawls. For the first time, client applications can stream design descriptors for page elements from a central repository, just like text descriptors are streamed from Web content repositories [85]. Moreover, Webzeitgeist's extensible architecture allows new data to be collected and integrated with the repository for supervised learning applications, for instance via crowdsourcing.

2.3.1 Crawling and Archiving Design

Most content and structure-based Web crawlers such as the Internet Archive's Heritrix [82] only perform static analysis of a page's source, indexing content streams

and other metadata referenced directly in the page's HTML [85, 82]. However, external resources specified in source HTML can have their own resource dependency chains, which can only be resolved by a browser at render-time. In order to compute the DOM tree of a Web page — the data structure containing the complete display specification — the page must actually be rendered. If the page is not rendered when it is crawled, the crawler cannot be certain that all the relevant resources have been requested and stored, raising the possibility that the page's layout will be broken when it is locally re-loaded. Therefore, design crawlers must render every page that they crawl [96, 79], unlike their content-based counterparts.

Having to render pages to access design information raises two issues. First, rendering is much slower than static source analysis: the layout engine has to request and load external resources, cascade style information, compute the DOM, and rasterize the page. For many machine learning applications, re-rendering pages in the inner-loop of an algorithm to access design information would be prohibitively expensive. Second, with the advent of dynamic HTML and client-side scripting, the design of a page may change between accesses even if its source does not. This ephemerality can frustrate client applications that require consistent data. To produce a complete, static record of a page's design, design crawlers must version and save all requested resources, and snapshot the entire DOM tree when a page is crawled [144, 5].

2.4 Design Reuse and Remixing on the Web

There are many benefits to leveraging examples in design: designers find, understand, and adapt relevant prior work to frame problems, generate alternative solutions, and evaluate those solutions [106]. Repurposing successful elements from prior ideas can be more efficient than reinventing them from scratch [70]: a designer still must evaluate whether the adapted solution is appropriate for a given problem, but often "validation is much easier than generation" [105]. This section surveys current tools and technologies that support design reuse on the Web.

2.4.1 Retrieval Systems

Tools developed to help users find *information* on the Web do not always naturally extend to *design*. For instance, if a designer searches for "colorful Web designs," Google will return pages that *discuss* colorful Web designs, instead of finding actual pages with lively and exciting color schemes.

Like the memex proposed by Vannevar Bush in 1945 [27], retrieval systems are intended to augment people's memories, helping users recollect and locate appropriate examples for a given task [155]. Two key challenges in retrieval systems are determining how to index examples, and how to expose stored information to users [105].

Retrieval interfaces for Web designs have investigated a variety of query mechanisms [79, 115, 153]. *Adaptive Ideas* allows users to search for designs that are *similar* to a selected example or that represent *variety* along a low-level style dimension, and borrow from them throughout the design process [115]. In addition to example-based search, *d.tour* allows users to query by high-level style concepts such as "minimal" or "clean" [153]. Retrieval systems in vision and graphics have explored the usage of relative attributes to iteratively refine searches [108, 34], for instance allowing users to request designs that are "more feminine" or "less scary." Igarashi *et al.* present a sketch-based Web design search interface, where users draw boxes to represent content and the system returns pages with similar layouts [79]. This thesis seeks to unify and extend this line of research with a common framework for prototyping design retrieval mechanisms.

2.4.2 Templates

Templates represent one of the most popular mechanisms for design reuse on the Web, offering pre-defined styles and layouts which can be applied to user-specified content [72]. Fundamentally, templates separate the *content* of a document from the way that content is *presented*. This separation is often invaluable in addressing the challenges of the modern Web, where designs must adapt to dynamically aggregated content, different viewing devices, and variable user abilities [93].

CSS-based Templates

CSS is perhaps the most basic type of design template on the Web: the same style sheet can be applied to multiple documents, and conversely, the same markup can be presented in radically different ways by different style sheets [162].

Adapting CSS files to new content, however, can be difficult. CSS written for specific markup often depends on that markup's information architecture and can only be adapted to other documents with similar structure. Moreover, even markup with similar structure must be mapped to the appropriate CSS rules, for instance by setting the `class` and `id` attributes of HTML elements to match corresponding CSS selectors. Since manually adapting markup to existing CSS can be tedious and operates at the level of code, GUIs have been developed to inspect and transfer element-level styles between pages [57, 33].

Templating systems that hide the complexity of CSS entirely may be less burdensome to users, offering a trade-off between design automation and content customization. One-click page generators such as Google's Blogger [20] attract novice Web designers who simply want to pick a template design and drop-in content. However, these templates impose a rigid information architecture, making it difficult for users to repurpose a blog template to build a business site. On the other hand, popular CSS frameworks like Twitter Bootstrap [22] and Zurb's Foundation [60] allow users to more flexibly define the content structure of their pages. By wrapping content with CSS selectors defined by these frameworks, users can create Web pages that are automatically styled and adapt to different viewing conditions. However, these frameworks usually employ a constraint-based grid system to compute page layouts, which requires users to author complex markup.

Template Authoring

While such templates and frameworks enable content producers to take advantage of existing designs, authoring generic templates can be challenging. CSS rules prevent simple design constraints like "always left align the navigation bar with the article heading" from being reliably enforced in code, prompting some researchers

to use sophisticated tools like linear arithmetic and finite domain constraints to gain control over layout and style [11, 24].

Similarly, the limitations of CSS frequently necessitate the use of "presentational" HTML to achieve layout effects such as vertically-aligned text. Benson *et al.* introduced Cascading Tree Sheets (CTS) to factor out presentational structure from HTML for better separation of content from presentation and design reuse [13]. With CTS, Web authors write content-only HTML files and link to CTS templates via CSS selectors to inject presentational HTML where needed.

Automatic Document Layout and User Interface Generation

Several researchers have proposed new templating languages to more flexibly author Web documents and overcome HTML and CSS limitations [97, 158]. Jacobs *et al.* introduced constraint-based templates that render content into grid layouts much like in print media [97]. Although CSS3 supports multi-column layout within a single element, it does not allow content to flow between arbitrary elements or around images as commonly seen in newspapers and magazine layouts. Jacobs *et al.* use a dynamic programming algorithm that chooses the layout from a set of grid-based templates to best satisfy content and viewing constraints. Schrier *et al.* extended this work by developing templates that adapt to different types of content and viewing conditions, reducing the number of individual templates that need to be enumerated [158].

Adaptive document layout is closely related to model-based user interface generation, in which high-level specifications are used to automatically configure UI designs [140, 170, 172, 147]. Some model-based systems such as SUPPLE [63] generate user interfaces that adapt to different devices and user abilities.

While constraint and model-based systems offer expressivity and adaptivity, authors face the burden of learning new specification languages, enumerating possible models, and understanding how constraints will interact with each other under a variety of conditions [132]. To make these systems easier to use, constraint and model specifications can be inferred from examples constructed with GUIs [61, 97, 129], designer demonstrations [134, 133, 172], and usage data [62, 65, 66, 64].

This thesis proposes a method by which *existing* Web designs can be automatically leveraged as templates. The Bricolage algorithm (Chapter 5) provides a one-click solution for adapting the entire content of one page to the style and layout of another. For content producers, the algorithm offers the ease of use of templates along with the diversity of the Web. For template authors, it obviates the need to learn new specification languages and enumerate templates for different types of content. Moreover, the technique can also be used to adapt designs to different viewing conditions (*e.g.*, from desktop to mobile).

2.4.3 Mashups

Another popular form of design reuse on the Web are *mashups*, which allow users to synthesize data from multiple sources, often obtained through scraping and public APIs [76].

To lower the barrier to creating mashups, researchers have built end-user tools that partially automate the selection and transformation of relevant content. Many systems create extraction patterns from user-specified page elements to automatically collect similar content [87, 51, 50, 120, 178]. Graphical tools such as Yahoo! Pipes [143] and *Marmite* [185] use a dataflow approach (similar to Unix pipes) to aggregate and process data from Web services. Given a set of user-selected page elements, *d.mix* generates service calls that return those elements via a site-to-service map [77]. After collection, these mashup tools import the content into interfaces such as spreadsheets [185, 120] or page layouts [51, 50, 77] to help users visualize and manipulate the aggregated data.

While researchers have built a variety of tools for remixing Web content from different sites and services, only a few systems scaffold design mashups. Systems such as CopyStyler [57] and WebCrystal [33] allow users combine stylistic elements like fonts, colors, and backgrounds from different pages in a piecemeal fashion.

2.5 Examples, Creativity, and Expertise

While design reuse confers great benefits, there are pitfalls to relying too much on examples [105]. Although people naturally draw upon examples to aid them in new situations, they may not always identify the most relevant and useful prior solutions [89]. Users may also demonstrate *conformity* when exposed to prior ideas in creative generation tasks [164]: fixating on the wrong analogs can lead to incorrect reasoning about the current situation or cognitive dead-ends [159]. Novices are especially vulnerable to the negative implications of cognitive priming because they are often unable to see past the surface features of a problem to make connections based on more abstract, underlying principles [39].

However, expertise can also contribute to cognitive fixation. A professional can become mired in routine and repetition and "miss important opportunities to think about what he is doing" [157]. As a professional deals with the same types of situations over and over again, he can become a *routine expert*: "outstanding in terms of speed, accuracy, and automaticity of performance, but [lacking] flexibility and adaptability to new problems" [80]. When routine experts are confronted with a new situation, they may map it blindly to an established category of problem even when it is inappropriate to do so.

Adaptive expertise, in contrast, "involves learning not just how to perform a procedural skill efficiently, but also know when variations to previous approaches are necessary – how and when to apply their heuristics – and thus results in experts being able to handle novel situations" [141]. Unlike routine experts, adaptive experts are able to optimize for both efficiency and innovation [159]. When adaptive experts encounter new situations they are able to recognize routine subproblems and solve them efficiently with prior knowledge. This know-how "frees attentional bandwidth and enables people to concentrate on other aspects of the new situation that may require nonroutine adaptation" [159]. Marsh *et al.* studied how exposure to examples affects both

conformity and novelty [127], and found that while exposure to examples increases conformity, conformity does not preclude creativity: conforming elements may more often be used to replace mundane aspects of design than novel ones.

Thus, examples can boost efficiency by informing standard parts of a design problem. Gentner *et al.* describe this as "analogy as recipe: the analogist uses analogy to avoid hard thought, as when we fill out our tax form by cribbing for last years" [69]. At the same time, innovation and conceptual change can occur from making connections at a more abstract, structural level, possibly in a completely different domain: "local analogies are useful for filling in an established framework, whereas distant analogies are used for creating a new framework" [69]. Here, distance can be in concept space or time: people have a tendency to fixate on recently seen examples, but adaptive experts draw from *all* prior experience [141].

Although building on example-based design theory is beyond the scope of this thesis, design mining provides a rich platform with which to study how Web technologies for design reuse affect practice. Can we build design tools that allow novices and professionals alike to behave more like adaptive experts during creative tasks? Lee et al. demonstrated that exposure to examples leads to higher quality Web design [115]. At what point in the design process should users be shown examples to optimize for quality? For efficiency? For creativity [109]?

Chapter 3

A Scalable Platform for Design Mining the Web

Advances in data mining and knowledge discovery have transformed the way Web sites are designed. However, while visual presentation is an intrinsic part of the Web, traditional data mining techniques ignore render-time page structures and their attributes. This chapter introduces *design mining* for the Web: using knowledge discovery techniques to understand design demographics, automate design curation, and support data-driven design tools. This idea is manifest in Webzeitgeist, a platform for large-scale design mining comprising a repository of over 100,000 Web pages and 100 million design elements. This chapter describes the principles driving design mining and the implementation of the Webzeitgeist architecture.

3.1 Introduction

Web knowledge discovery and data mining [123] have transformed the way people build and evaluate Web sites [104], and the way that consumers interact with them. The information gained from Web mining drives search, e-commerce, interface development, network architectures, online education, social science, and more [107].

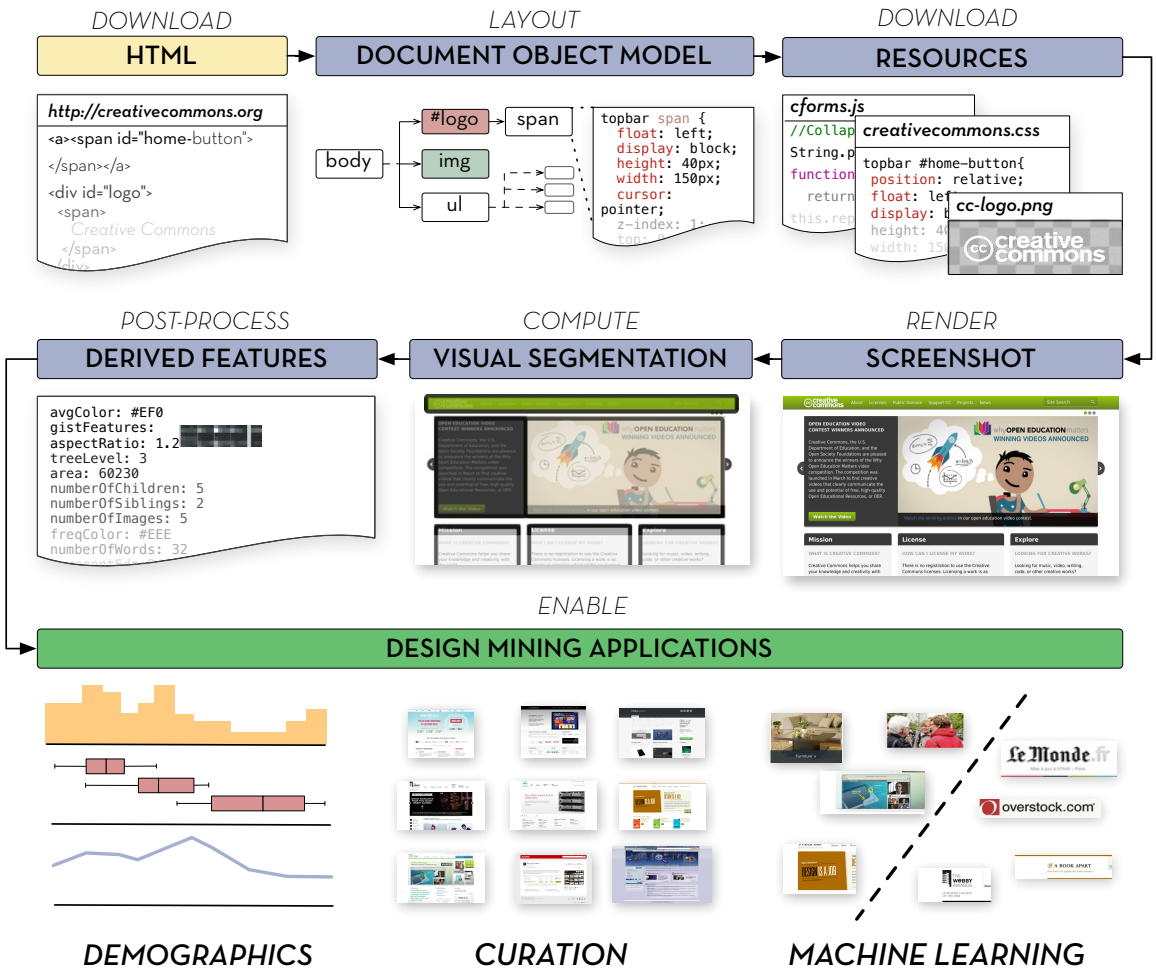


Figure 3.1: Webzeitgeist, a scalable platform for Web *design mining*, supplements the data used in traditional Web content mining (yellow) with information about the visual appearance and structure of pages (blue) to enable a host of new design applications (green).

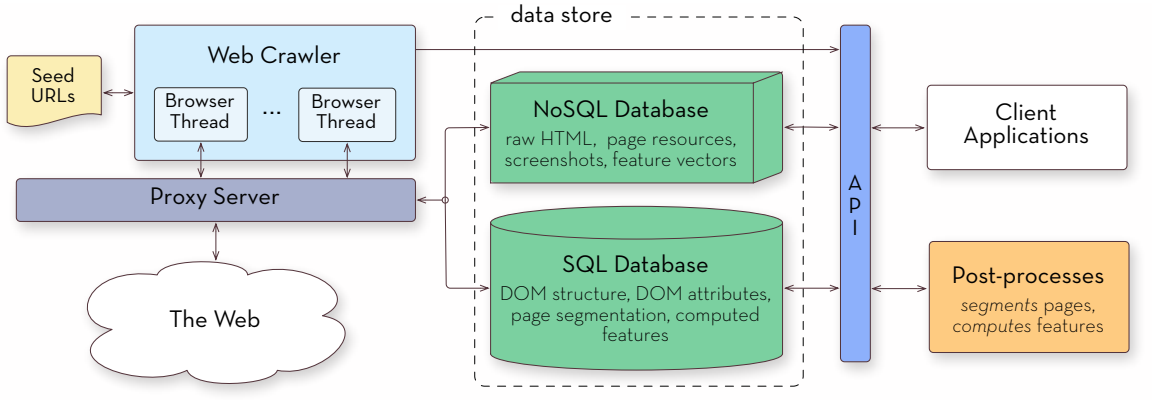


Figure 3.2: The Webzeitgeist architecture. A bespoke Web crawler requests pages through a caching proxy, and renders them in a set of browser threads. The proxy saves requested resources in a NoSQL key-value store, while the crawler writes the complete DOM tree for each page into a relational SQL database. Then, a set of post-process scripts are run to compute high-level features and data-structures over the stored pages. Client applications access the repository through a RESTful API.

Web data mining typically comprises three domains: usage mining, or click analysis [167]; content mining, or text analysis [124]; and structure mining, or link analysis [71]. Together, these techniques mine the *content* contained in a Web page, but ignore that content's *presentation*. In fact, most mining and knowledge discovery systems *discard* style and rendering data [189, 171]. This raises the question: what could we learn from mining design?

Webzeitgeist comprises a repository of Web pages, processed into data structures that facilitate large-scale design knowledge extraction. The Webzeitgeist architecture is based on four underlying principles — scalability, extensibility, completeness, and consistency — and optimized for three common use cases: *direct access* to specific page elements, *query-based access* to identify a set of page elements which share common properties, and *stream-based access* to the repository as a whole for large-scale machine learning and statistical analysis [85].

Webzeitgeist's repository is populated via a bespoke Web crawler, which requests pages through a specialized caching proxy backed by a flexible data store. As each page is crawled and rendered, its resources are versioned and saved, and its Document Object Model (DOM) tree is snapshotted to produce a complete, static record

of the page's design. Then, a set of semantic and visual features describing each DOM node are computed in a post-process and stored. Client applications access the repository through a RESTful API [152].

This chapter discusses the principles that enable large-scale Web design mining, the implementation of the Webzeitgeist architecture, the repository crawled from the Web, and the API that exposes it.

3.2 Principles for Design Mining

To support design mining applications, the Webzeitgeist architecture is predicated on four underlying principles.

3.2.1 Scalability

In rich, visual domains like Web design, the utility of data mining critically depends on the size of the corpus: in a space with thousands of parameters, millions of examples are necessary to extract meaningful statistics or find relevant examples during search [81]. Webzeitgeist, therefore, is designed to scale to millions of distinct page elements. Visual and semantic features are precomputed for fast access, stored in a relational database to facilitate complex queries, and duplicated in a key-value store for efficient streaming. To eliminate redundant storage of shared page resources, Webzeitgeist employs Rabin fingerprint hashing [148]. Additionally, the Webzeitgeist server uses a large memory pool to minimize disk access, backed by a hardware RAID controller with striping to make disk access fast when it does occur.

3.2.2 Extensibility

Since Webzeitgeist provides a general platform for design mining, not all of its eventual uses can be foreseen. Thus, a modular architecture facilitates the addition of new data, features, and functionality. To support transparent updates, two versions of the data store exist at all times: a *production* version that is exposed

to external applications, and a *staging* version where new pages are added during crawling and features are computed. To minimize code and data dependencies, individual features are implemented as independent `C++` dynamic plugin libraries. The post-process communicates with the data store through the public API, allowing implementation details to change as long as interfaces are preserved.

3.2.3 Completeness

Most Web mining employs static page analysis: issuing an HTTP `GET` request for a given URL, storing the returned HTML, and parsing it [116]. To mine the *design* of Web pages, Webzeitgeist must identify and capture every resource and DOM property that contributes to a page's visual appearance. Since render-time visual properties cannot be determined through static page source analysis, Webzeitgeist uses a layout engine to process retrieved HTML into a DOM tree, and a proxy server to dynamically intercept all the resource requests made by the engine during this process.

3.2.4 Consistency

Dynamically-generated content poses another complication for design mining. DHTML and client-side scripting allow arbitrary code to modify the DOM based on requests made to external resources. Thus, it is nearly impossible to archive pages in a format that guarantees reproducible rendering in a browser without altering their source [144, 5]. This ephemerality frustrates many machine learning and statistical analysis applications, which expect data to remain consistent between accesses. Webzeitgeist, therefore, renders a canonical view of each page during crawling, and serializes the resultant DOM and all of its properties to the data store. Client applications and feature computations access this static snapshot of a page's design instead of interacting with the layout engine directly, and can query render-time properties without having to re-render the page.

3.3 Implementation

The Webzeitgeist architecture comprises five integrated components: the Web crawler, the proxy server, the data store, the post-process, and the API (Figure 3.2). The crawler loads pages through the proxy, which writes them to the data store. Post-processes then run on the stored pages to compute high-level features and data structures, after which client applications can access the repository through the API.

3.3.1 Web Design Crawler

The Web crawler consists of a set of parallel browser processes, which load pages from the Web to add them to the Webzeitgeist repository. The crawler builds a queue of URLs from a seed list. At each stage of the crawl, a browser process dequeues a URL, checks that it is not already in the repository, and requests the corresponding page. Once the page is downloaded, its HTML is parsed, and all external links are extracted and added to the queue. The Webzeitgeist crawler loads each page in a Webkit browser window [177], computes its DOM tree, and renders it. This rendering and the computed DOM are then saved in the *staging* store.

3.3.2 Caching Proxy Server

To identify and store a page's resources, the system routes all browser requests through a custom Web proxy. The proxy sits between the Web and the crawler, and connects directly to the Webzeitgeist data store. The proxy intercepts each resource request made by a page, downloads it from the Web, and hashes its contents. If the file does not already exist in the store, it is added.

The proxy server is also responsible for storing the graph structure of page-resource relationships. Since HTTP is a stateless protocol, this requires using custom HTTP headers to associate each resource request with the page that originated it. When the crawler requests a page, the data store generates a unique identifier

and passes it back in the response header. The crawler then uses this ID to label each subsequent request the page makes to the proxy.

Two additional headers determine how the proxy services requests: adding them to the store during the crawl, or serving them directly from the database during retrieval. In the event that an application tries to retrieve a URL that does not exist in the data store, the proxy server responds with a 404 - PAGE NOT FOUND error.

3.3.3 Post-process: Visual Segmentation

Once a page has been downloaded, converted to a DOM, rendered, and stored, a set of post-processes are run. First, Webzeitgeist canonicalizes the DOM, so that client applications can efficiently work with a structured, visual representation of designs that are directly comparable between pages.

Existing page segmentation algorithms begin by partitioning the DOM into discrete, visually-salient regions [28, 30, 100]. These algorithms produce good results whenever a page's DOM closely mirrors its visual hierarchy, which is the case for many simple Web pages. However, these techniques fail on more complex pages. Modern CSS allows content to be arbitrarily repositioned, meaning that the structural hierarchy of the DOM may only loosely approximate the page's visual layout.

This dissertation introduces Bento, a page segmentation algorithm that "re-DOMs" the input page in order to produce clean and consistent segmentations (Figure 3.5). The algorithm comprises three stages. First, Bento extracts all the DOM elements that contribute to a page's visual appearance, while maintaining the DOM's hierarchical relationships. Next, the hierarchy is reshuffled so that parent-child relationships in the tree correspond to visual containment on the page. Each DOM node is labeled with its rendered page coordinates and checked to verify that its parent is the smallest region that contains it. When this constraint is violated, the DOM is adjusted accordingly, taking care to preserve layout details when nodes are reshuffled. Third, redundant nodes are removed so that there is exactly one node per unique page region. These three steps canonicalize the DOM to more closely match the visual hierarchy of the page.

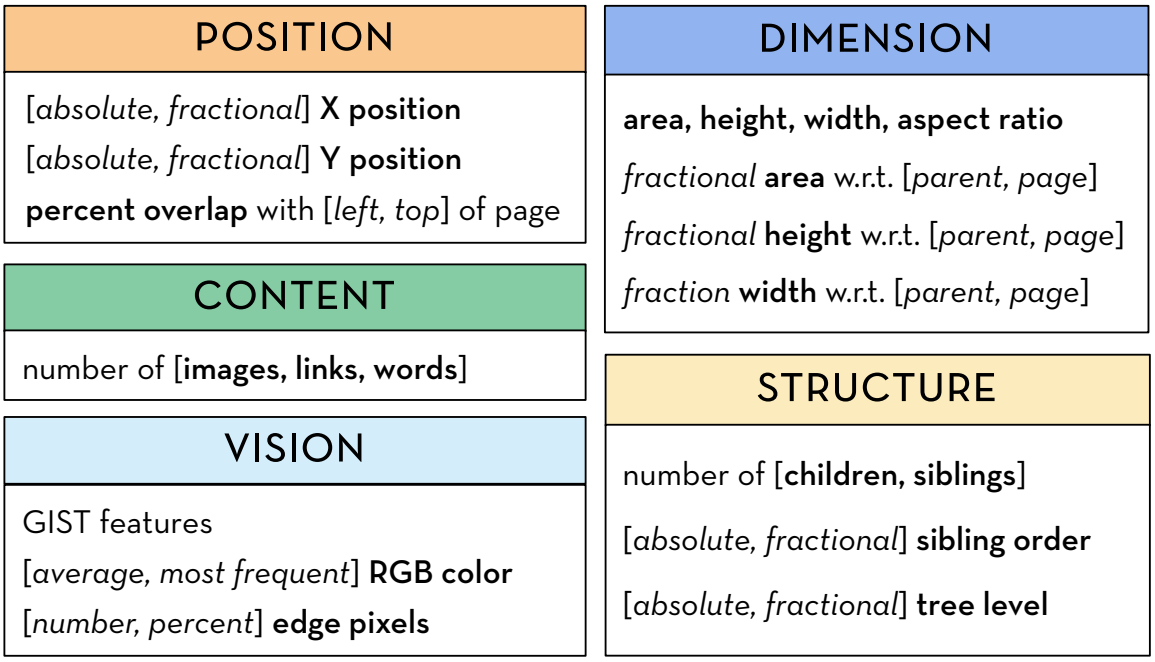


Figure 3.3: The semantic and computer vision features that Webzeitgeist computes over page elements.

Bento segmentations serve as the base representation for algorithms described in chapters 4, 5, and 6. Bento is available as a web service and a BSD open-source C++ library at `https://code.google.com/p/lib-bento`.

3.3.4 Post-process: Visual Element Descriptors

For each element in the Bento segmentation, the system computes a set of semantic and computer vision features and stores them (see Figure 3.3).

Next, the post-process coalesces each node's visual, semantic, and render-time features into a vector descriptor, exposing page properties in a convenient form for design mining applications. Numeric features are normalized to the range $[0, 1]$, and string-based attributes are binarized based on their possible values to generate dictionary-length bit vectors. After this conversion, each page node is associated with a 1679-dimensional descriptor consisting of 691 render-time HTML and CSS properties computed by the DOM, 960 GIST scene descriptors

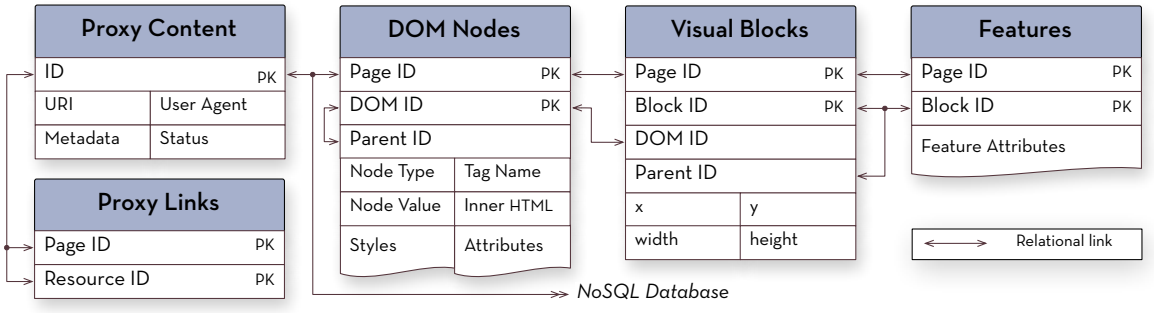


Figure 3.4: The schema for the Webzeitgeist data store, showing the five tables that comprise the SQL database and their contents. This de-normalized structure, with replicated Page, DOM, and Block IDs, facilitates fast retrievals.

computed on the node's rendering [139], and 28 structural and computer vision properties.

After the feature vectors are computed, the post-process restructures the table in which DOM properties are stored. During the crawl, DOM tree data is added to a wide table which facilitates fast insertions but results in slow retrieval; partitioning this table into a *star schema* reduces retrieval times by an order of magnitude [168]. After this restructuring, the post-process migrates the data store from the staging environment to production.

3.3.5 Data Store

The Webzeitgeist data store comprises two databases: a NoSQL database for page resources, binary data, and the vector descriptors for page nodes; and a relational SQL database for DOM nodes and properties, the visual page hierarchies, and the associated vision and semantic feature values.

The NoSQL database is a MongoDB instance [130], which provides fast access to binary files and structures too large to be efficiently stored in SQL tables, while simultaneously supporting dynamic queries and aggregation. This database contains the full HTML of every crawled page, its resources, and the high-dimensional feature descriptor for each visual hierarchy node.

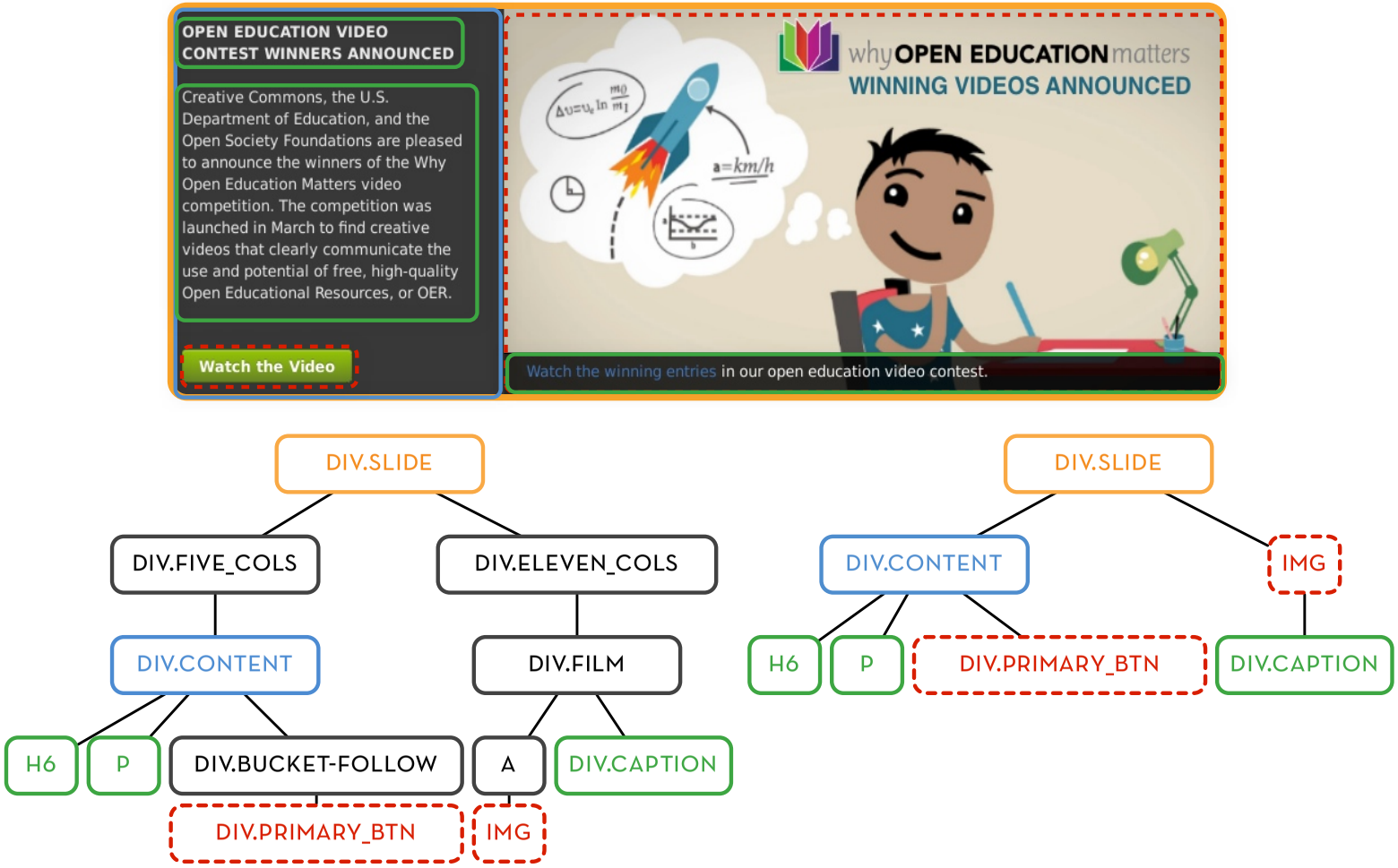


Figure 3.5: The Bento segmentation algorithm provides a representation that closely matches the visual hierarchy by post-processing the DOM. First, it identifies all the visual nodes in the DOM (left tree). Then, it restructures the hierarchy so that every element is the child of the smallest enclosing region, removing any redundant nodes that occupy the same bounding box (right tree).

The relational database is a MySQL instance [135], comprising five tables: PROXY CONTENT, PROXY LINKS, DOM NODES, VISUAL BLOCKS, and FEATURES (Figure 3.4). The PROXY CONTENT table contains metadata for every URL requested by pages during crawling, describing where the resource is from, its identifier in the NoSQL store, when it was retrieved, and a Rabin fingerprint hash of its contents. The PROXY LINKS table associates pages with a list of the resources upon which they depend. The DOM NODES table contains a record of each DOM node encountered in the crawl, along with pointers to its parent page and node; its type, name, value, and inner HTML; and all 258 render-time DOM attributes defined by the HTML4 and CSS3 standards [91, 181]. The VISUAL BLOCKS table contains the page elements that result from the visual segmentations performed during post-processing. The FEATURES table stores the visual and semantic features computed for each such block. For fast retrieval, tables are denormalized with replicated Page, DOM, and Block IDs.

3.3.6 API

Clients access the Webzeitgeist repository through a RESTful API, loading the appropriate endpoint URL and receiving JSON data in return [43]. Three modes are available for requesting data. The first allows *direct access* to design properties based on unique identifiers. The second allows clients to *stream* batches of data from the repository with a single request.

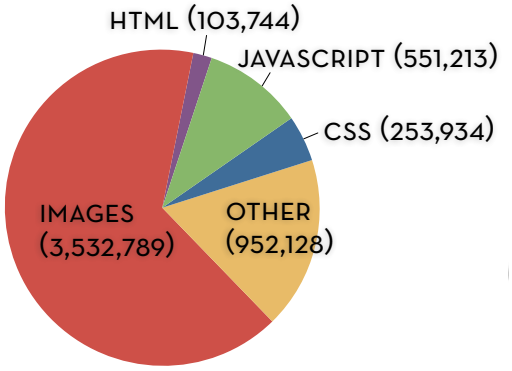
For more complex access patterns, the API also provides a custom JSON-based design query language (DQL). DQL predicates allow for filtering based on combinations of DOM attributes and computed visual features. When issuing queries, the client can also specify a list of properties that should be returned by the API call, keeping result sets succinct. The API transparently converts DQL queries into SQL and Mongo Query Language, sanitizes them to prevent injection attacks, and returns the results in JSON.

3.3.7 Server Hardware

The Webzeitgeist repository is hosted on a twelve-core 2.4GHz Intel Xeon server to allow complex DQL queries to be executed efficiently. The server contains 48 GB of RAM to facilitate caching and ensure that the SQL index fits in memory. The server's 4TB data store consists of fourteen 600GB 15K RPM SAS drives in a RAID 10 configuration, backed by a hardware RAID controller with a 1GB cache. The drive array is capable of sustaining 6GB/s throughput when data is being streamed from disk.

3.4 The Webzeitgeist Dataset

The Webzeitgeist crawl selection policy was optimized for quality, diversity, and providing a holistic view of Web design practice. To guarantee that the crawl included high quality designs, it was seeded with pages from the Alexa Top 500, the Webby Awards gallery, and popular design blogs. To build a diverse repository, Webzeitgeist crawls pages in a breadth-first order and limits downloads to 10 pages per domain. Sampling several pages from each visited domain can help designers analyze how individual page elements are reused and adapted across a site.



Each downloaded page is rendered in three different browser window sizes to identify responsive Web designs, or pages with layouts that adapt to the viewing environment : desktop (1600x1200), mobile (320x480) and tablet (768x1024). This type of mining could help users understand design patterns across different form factors. Mobile and tablet pages are only saved to the database if they are different from the desktop version, which is determined by comparing change in screenshot aspect ratio between the three sizes. Pages designed for just desktop

viewing, serve the same content and design regardless of browser window dimensions, and therefore, their desktop, mobile, and tablet screenshots usually all have a similar aspect ratio. If pages are non-responsive, the crawl does an additional test for mobile- and tablet-specific designs by spoofing user-agent headers.

The resultant dataset contains 103,744 Web pages, of which 5644 are mobile designs and 5528 are tablet designs. These pages span 43,743 domains and are comprised of 143.2 million DOM nodes and 12.7 million visual blocks. The raw HTML content of these pages and their referenced resources together require 425GB of disk space; the SQL database requires 187GB. The pages reference over 5.3 million HTML, CSS, JavaScript, image, and other resources, including Flash, movie, and audio files (see inset figure).

The Webzeitgeist crawl was a computationally intensive task, requiring more than 35 CPU days of processing. As a representative example, the CHI 2013 homepage `http://chi2013.acm.org` references two style sheets, four JavaScript files, and five images for a total of 480KB of raw content. The crawler downloaded the page on September 13th, 2012, in 3.5 seconds. The DOM, which comprised 251 nodes, was computed in 0.1 seconds and stored via the API in 0.47 seconds.

The visual segmentation algorithm ran for 2ms, producing 61 visual blocks; visual feature computation ran for 13.53 seconds. Writing this segmentation and the associated features to the database took another 1.04 seconds, for a total processing time of 18.64 seconds.

3.5 Capabilities

Webzeitgeist's data structures and API allow users to efficiently aggregate many types of design data, which can easily be imported into visualization or statistical analysis software.

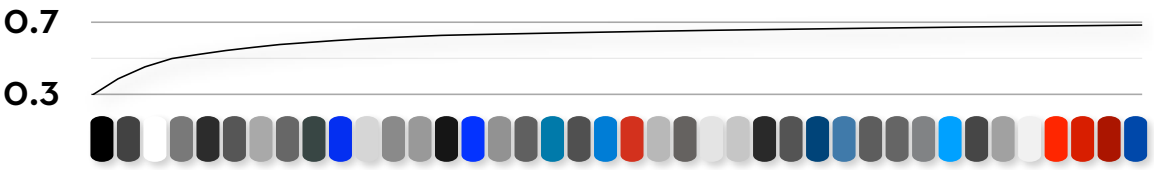
For instance, we can query Webzeitgeist to return all the distinct cursors in the repository by writing the following DQL query:

Figure 3.6: The 295 distinct cursors in the Webzeitgeist repository, fetched in 47.8 seconds. This query searches the CSS `cursor` property on DOM nodes, looks up the ID in the PROXY CONTENT table, and fetches the associated file from NoSQL.

```
POST, /v1/dom.json
query = {
  "$select": [
    {"@styles": {"$distinct": "cursor"}
  ]
}
```

This DQL query — which executed in 47.8 seconds — first finds cursors by examining the CSS `cursor` property across DOM nodes, and then fetches the corresponding files from the NoSQL database. A "cursory" inspection of the 295 results shows that arrows, hands, cartoon characters, and celebrity faces are all popular choices (Figure 3.6).

Webzeitgeist can answer similar questions about popular text color choices, for instance by computing a cumulative distribution function over the CSS `color` property:



The forty most popular text colors in the database account for nearly 70 percent of all text color; most are shades of grey.

Webzeitgeist also affords the ability to examine distributions over both page-

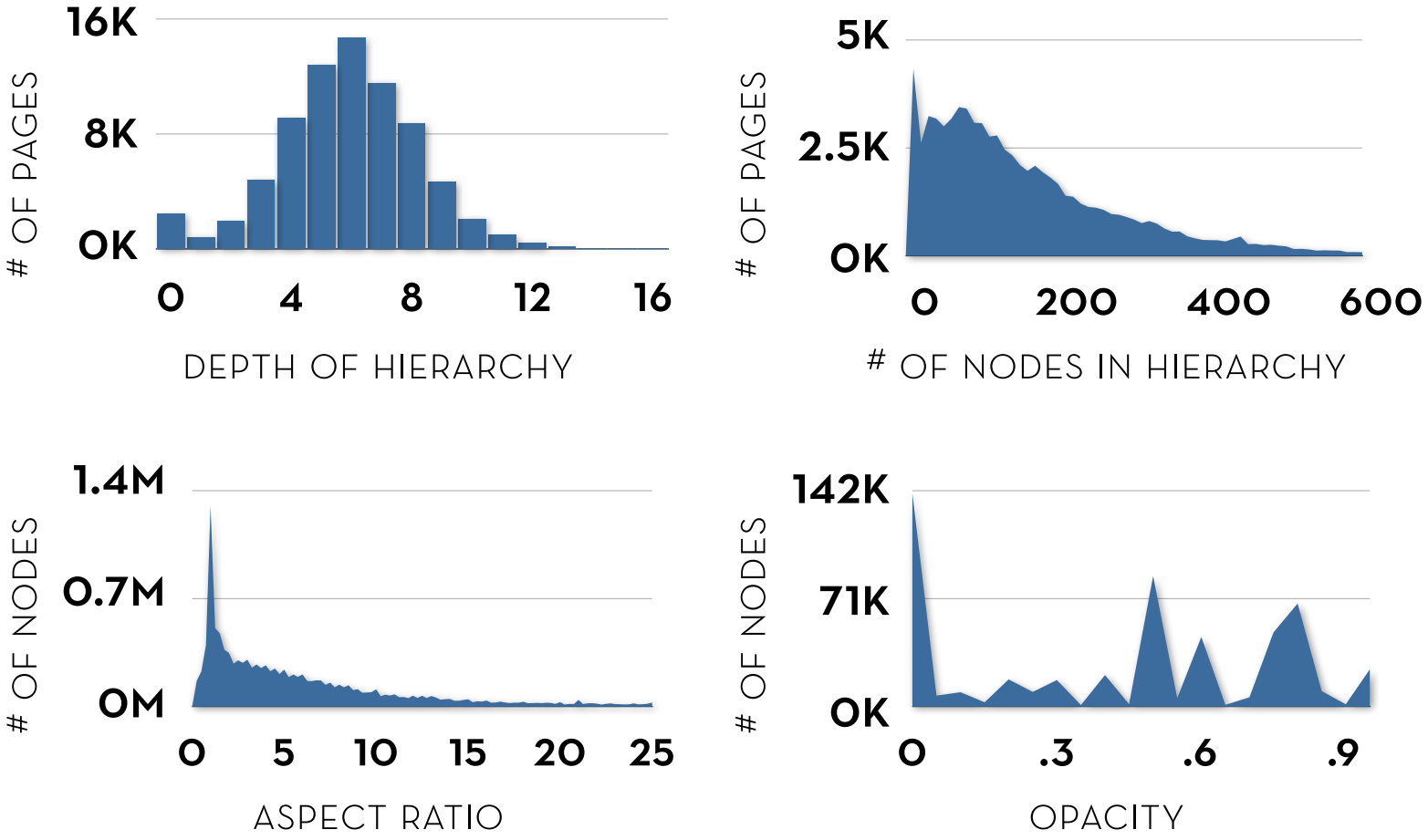


Figure 3.7: *Top row*: distributions over page-level properties: depth (left) and number of nodes (right) in a page's visual hierarchy. *Bottom row*: distributions over node-level properties: *aspect ratio* feature (left) and CSS `opacity` (right).

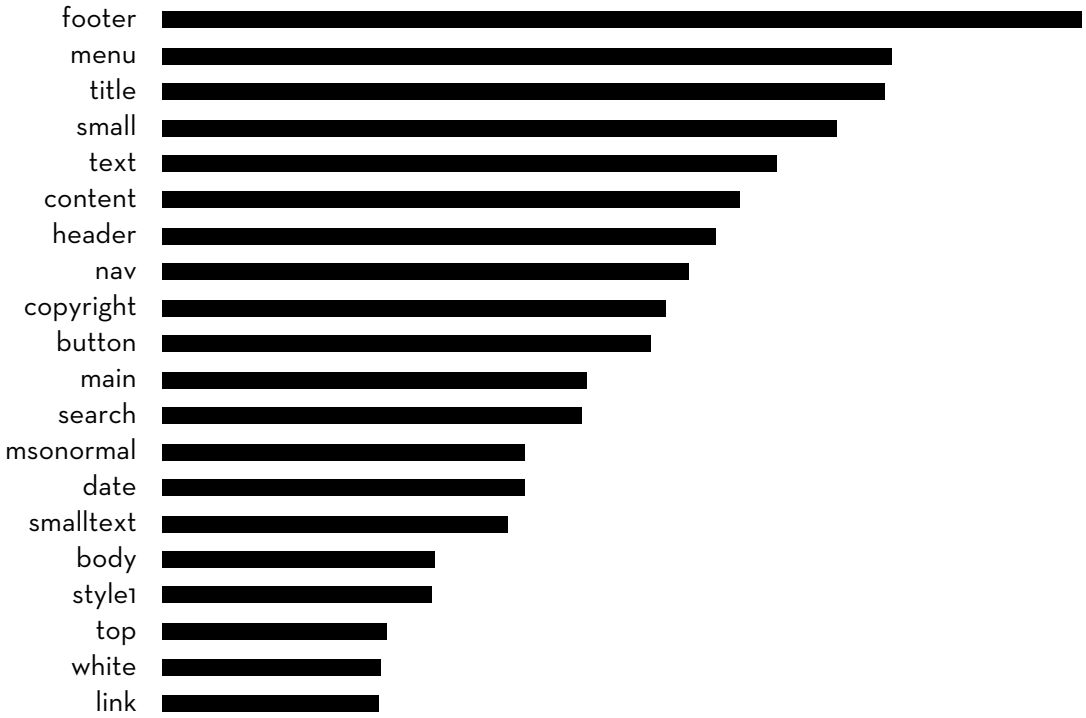


Figure 3.8: Graph reproduced from Google Web survey of popular HTML `class` names from 2005 [184].

and node-level Web properties. For instance, Webzeitgeist can be used to compute statistics on the visual complexity of pages. Figure 3.7 (*top row*) shows that the mode depth of a page's DOM tree is six, and that most pages contain between 50 and 200 DOM nodes. To investigate the cause of the sharp spike in the latter histogram, users can request IDs for pages with only a single DOM node and inspect their HTML: unsurprisingly, these pages are predominantly Flash-based.

Similarly, users can inspect common properties for individual page assets. Calculating a histogram over the *aspectRatio* of visual nodes reveals that there are many square elements, but that page elements, on average, are wider than they are tall (Figure 3.7, *bottom left*). Computing a histogram over the CSS `opacity` property and examining values less than 1, reveals sharp peaks at .5, .65, .75, and .8 (Figure 3.7, *bottom right*).

Since Webzeitgeist stores HTML properties in addition to design data, we can also use the repository to revisit HTML demographics in a new way. In 2005, Google released the results of a large-scale survey of popular HTML `class` names [184] (Figure 3.8). Webzeitgeist allows us to take this study one step further, and understand the relationship between static HTML properties and dynamic render-time ones. Since Webzeitgeist records the render-time bounding box for each DOM node, we can compute spatial probability distributions for the most popular CSS selectors (Figure 3.9). The striking patterns that result indicate that the visual positions and semantic roles of some page elements are highly correlated.

3.6 Discussion and Future Work

There are a number of directions for future work. Scaling the database by several orders of magnitude would increase the accuracy and utility of many design-mining applications. While the current indexing strategy for Webzeitgeist should scale to about five million pages, crawling a more substantial portion of the Web would require porting the infrastructure to a distributed computing and storage platform.

In addition to crawling more pages, altering the crawl's selection policy to capture additional information from each visited site could provide new ways of analyzing Web design practice. Currently, the crawl only stores a single copy of each resolved URL; however, aggregating multiple versions of pages over time could allow users to build data-driven models of Web design evolution [5, 176]. Additionally, designers might want to study responsive design at a finer granularity. We currently check for mobile and tablet versions of pages, but the Time's online magazine has a style sheet detailing layout changes triggered by over 100 different browser widths [126]!

Perhaps the most exciting avenues for future work are using the repository to realize new machine learning applications and reimplementing existing methods at scale [96, 153, 175]. The Webzeitgeist platform could be leveraged to take advantage of recent advances in unsupervised learning [114] or used to bootstrap crowdsourced data collection for supervised applications.

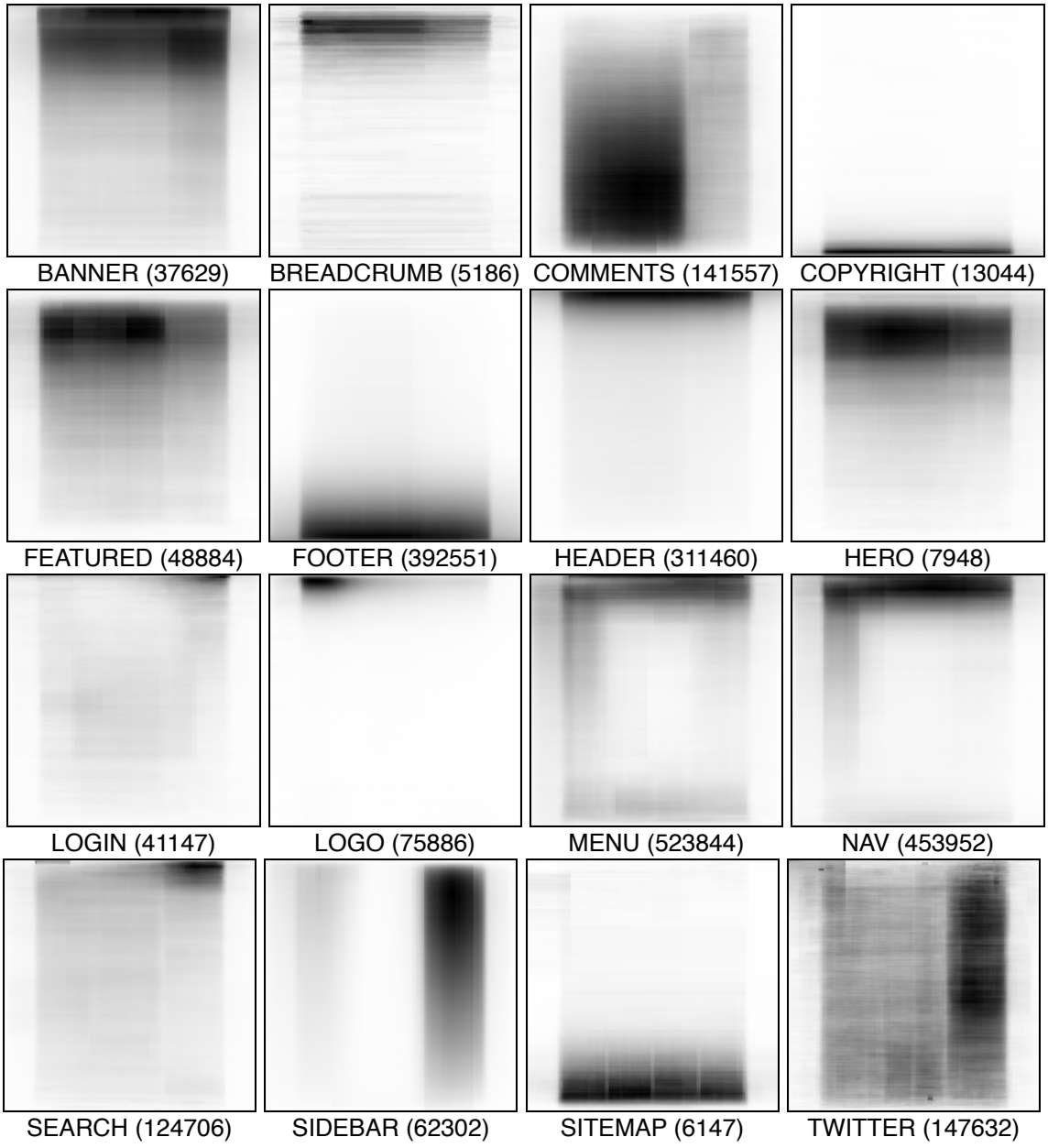


Figure 3.9: Spatial probability distributions for frequently occurring HTML `id` and `class` attributes demonstrate striking visual correlations.

Chapter 4

Design Search

The Webzeitgeist design mining platform enables content producers to answer questions about design practice and software developers to build next-generation design tools. Webzeitgeist significantly lowers the barrier to data-driven Web design, facilitating analysis on a scale 50–300 times larger than prior work [96, 153]. This chapter illustrates how Webzeitgeist can be used to implement a diverse set of design-based retrieval interactions, including constraint-, keyword-, and example-based search. Designers can query Webzeitgeist to search for examples of design patterns and trends [54, 83], without relying on manual curation. Application developers can apply machine learning techniques to learn classifiers and similarity metrics without incurring the overhead of crawling, rendering, and sanitizing Web data.

4.1 Design Queries

Designers are often interested in understanding the context of particular patterns and trends [54]. Many design blogs maintain small, curated sets of examples showcasing trending Web design techniques. For instance, in 2013, examples of "flat" page design (*i.e.,* pages that do not use drop shadows, gradients, and textures) were featured in many design blogs [2, 1].

To give designers more powerful search and collection capabilities, Webzeitgeist introduces the ability to quickly create dynamic collections that exhibit particular

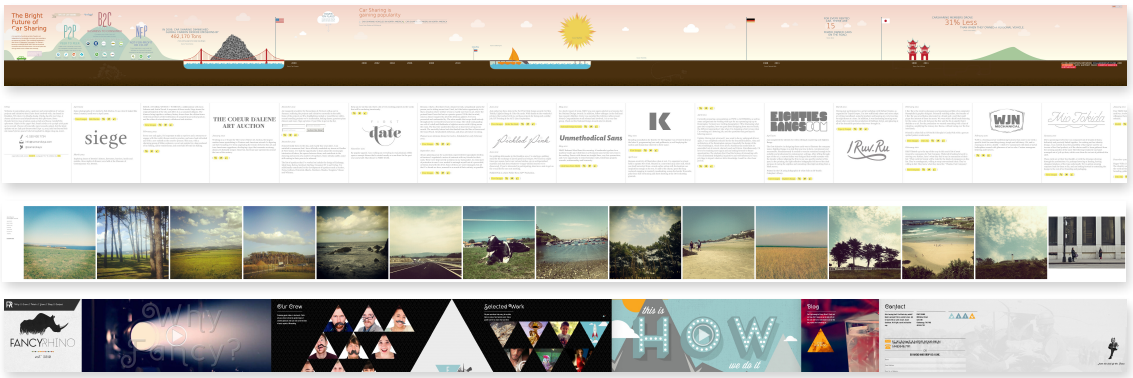


Figure 4.1: Four of the 68 query results for pages with horizontal layouts. Blogs, image/photo galleries, and vector art pages are a few of the representative styles in the results set.

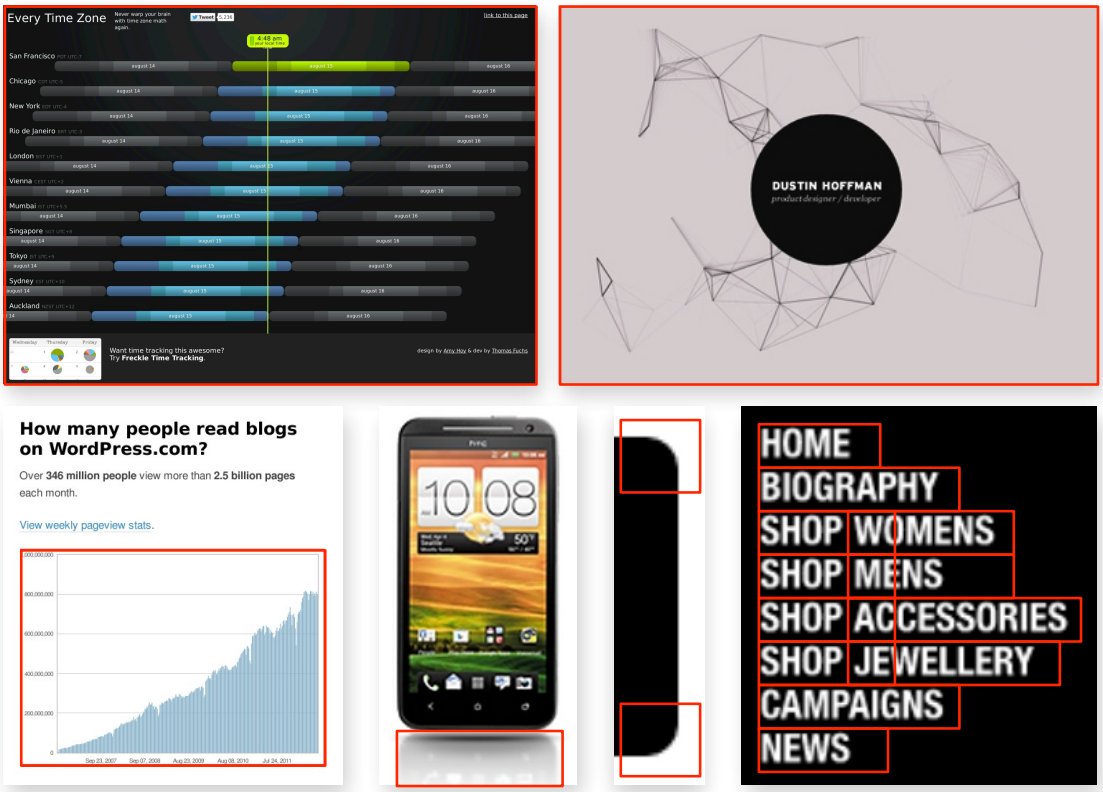


Figure 4.2: Selections from some of the 4943 pages containing <CANVAS> elements in the Webzeitgeist repository, demonstrating interactive interfaces, animations, graphs, reflections, rounded corners, and custom fonts.

design characteristics. For instance, one distinctive technique discussed on design blogs is the use of long, scrolling horizontal layouts. To find such pages, we queried Webzeitgeist for pages with *aspectRatio* greater than 10.0:

```
POST, /v1/pages.json
query = {
  "$select": [{"$distinct": "page_id"}],
  "$where" : [
    {"@visual": {"aspectRatio": {"$gt": 10} } }
  ]
}
```

This query produced 68 horizontally-scrolling pages in 1.1 seconds. Figure 4.1 shows a few representative results.

Querying Webzeitgeist with constraints based on HTML markup can also shed light on design trends. The W3C describes the <CANVAS> element — introduced in HTML5 — as a scriptable graphics container. The specification, however, gives little insight into how the tag is actually used. Webzeitgeist returns all 201,658 <CANVAS> elements in the database in 2.4 minutes. Figure 4.2 shows representative uses.

Webzeitgeist allows us to investigate another problematic aspect of Web design: typography. Although the CSS `@font-face` rule was introduced in 1998 [29], technical and licensing issues with embedding custom Web fonts have traditionally relegated complex typographic effects to images. We can use Webzeitgeist to search for examples of prominent Web font typography, querying for nodes with a CSS `font-size` property greater than 100 pixels. Figure 4.3 shows a few of the 6856 results, which occur in only 1657 distinct pages.

We can build more complex queries by specifying more constraints. To learn the different ways in which semi-transparent overlays are used, we queried for nodes with a solid background color where the *alpha* value of the CSS `background-color` property is between 0 and 1 (Figure 4.4).

We can also use Webzeitgeist to construct queries over high-level design concepts. Suppose a designer wants to browse “search engine-like” pages. This concept is loaded with design constraints, but we can approximate it in DQL as a query for pages with a centered <INPUT> element and low visual complexity:

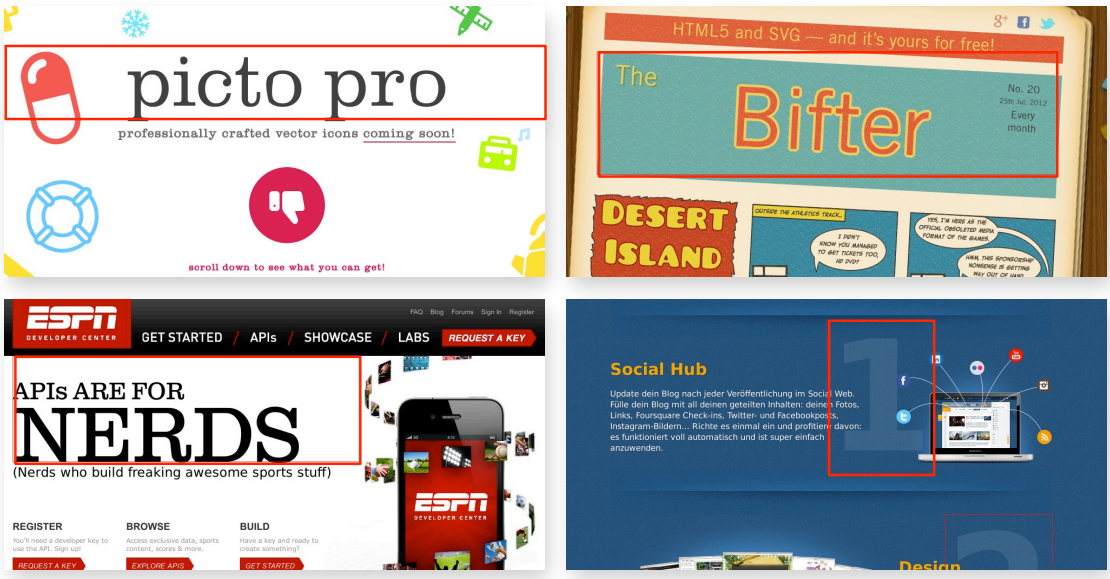


Figure 4.3: Query results for nodes containing large typography, demonstrating large text in logos, site titles, hero graphics, and background effects. The query identified 6856 DOM nodes from 1657 distinct pages, and executed in 56 seconds.

```
POST, /v1/pages.json
query = {
  "$select": [{"page_id": 1}],
  "$where": [
    {
      "@dom": {
        "tagName": "INPUT",
        "type": "text",
      "@visual": {
        "leftSidedness": {"$or": {"$gte": 0.4}, {"$lte":0.6}},
        "topSidedness":  {"$or": {"$gte": 0.4}, {"$lte":0.6}}
      }
    },
    {"@visual": {"$cnt": {"$lt": 50} } }
  ]
}
```

Figure 4.5 shows a few results from this query.

Similarly, attribute queries can be composed to search for pages with specific visual layouts. Figure 4.6 shows a sample layout with a large header, a top navigation bar, and a large body text node. We can encode this layout in a DQL query that searches for pages with a header that takes up more than 20 percent of the page's

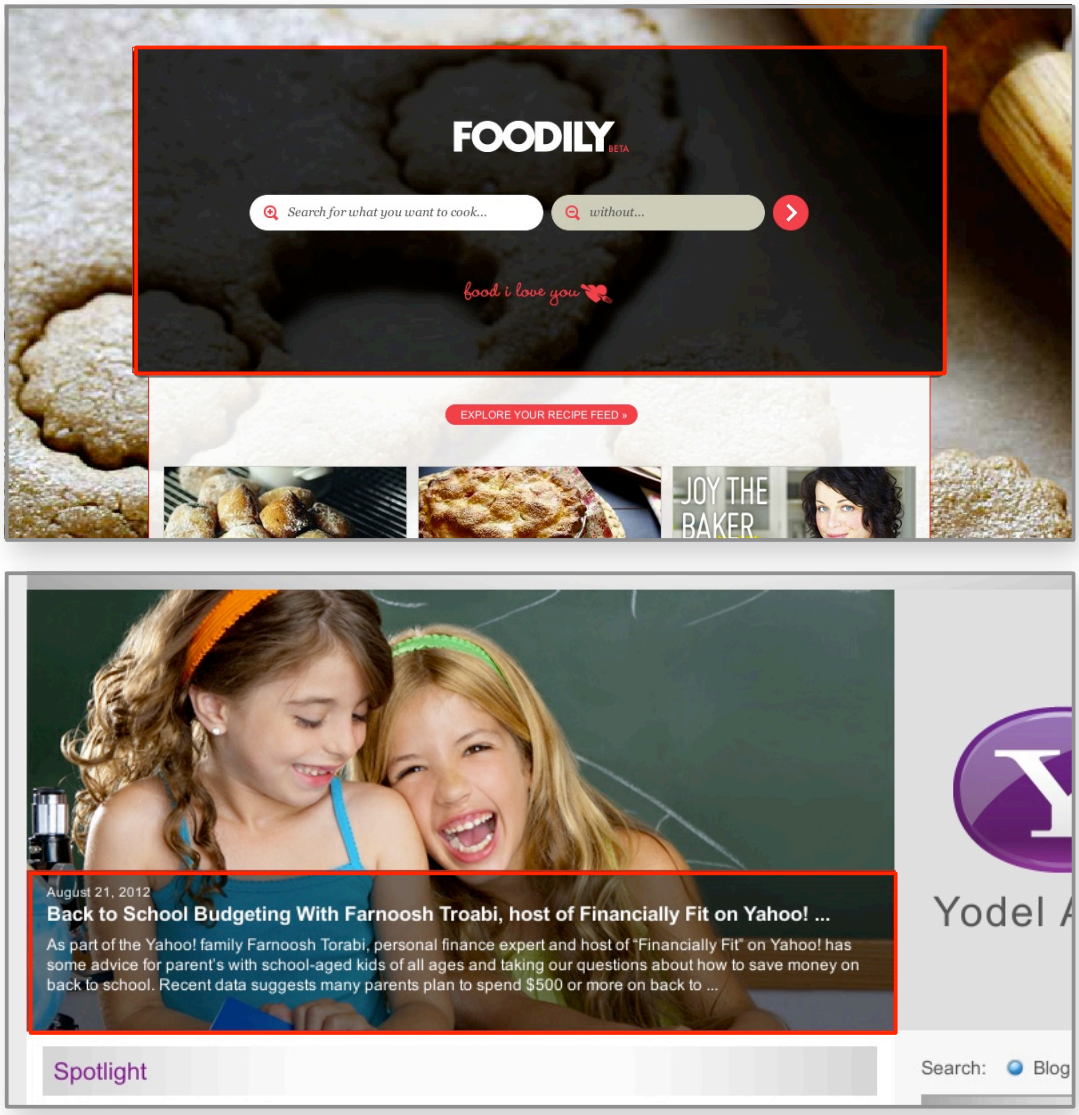


Figure 4.4: Query results for semi-transparent overlays. The query identified 9878 DOM nodes from 3435 distinct pages, and executed in 48.1 seconds. Semi-transparent overlays are often used on top of photographs, either as a way to display content over of a busy photographic background (top) or to frame captions (bottom).

Figure 4.5: Query results for "search engine" pages: roughly centered (vertically and horizontally) text `INPUT` elements, and fewer than 50 visual elements on the page. This query produced 209 pages and executed in 3.9 minutes. Some login and signup pages are also returned (bottom right).

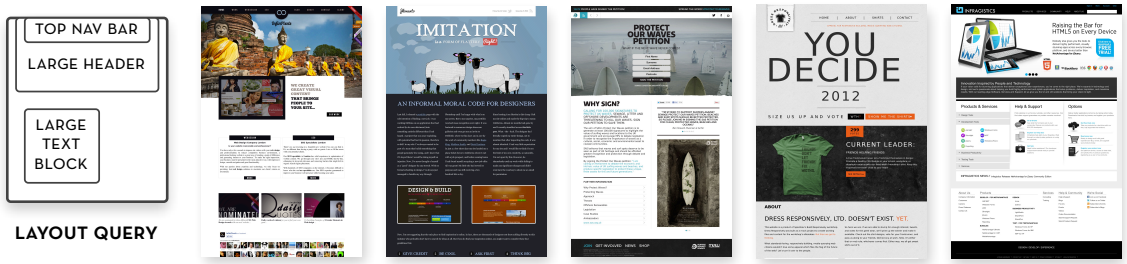


Figure 4.6: Five of the 20 search results for the three-part DQL layout query visualized on the left. The query, which executed in 2.1 minutes, returns pages that share a common high-level layout, but exhibit different designs.

area, a navigation element that is positioned in the top 10 percent of the page's height, and a text node that contains more than 50 words. This example illustrates the kinds of applications that Webzeitgeist might engender: imagine a search interface that automatically formulates queries from sketches like the one shown in the figure.

4.2 Keyword and Example-based Search

While DQL allows designers to query based on specific visual properties, not all users will want to interact with Webzeitgeist at the level of code. Webzeitgeist also enables a new kind of design-based machine learning, which can support higher-level search techniques like keyword and example-based search. For the first time, applications can stream structured visual descriptors for page elements from a central repository. Moreover, Webzeitgeist's extensible architecture allows new data to be collected and integrated with the repository for supervised learning applications, for instance via crowdsourcing.

4.2.1 Keyword Search

We used Webzeitgeist as a backend to train structural semantic classifiers for concepts like ARTICLE TITLE, ADVERTISEMENT, and PRODUCT IMAGE (discussed in more detail in Chapter 6). In an online study, we collected a set of more than 20,000 semantic labels over more than 1000 distinct pages. We then used the descriptors associated with page elements to train 40 binary Support Vector Machine classifiers, reporting an average test accuracy of 77 percent. By running these classifiers over the entire Webzeitgeist data set, we enable keyword search for page elements like "advertisement", "logo", or "blog post."

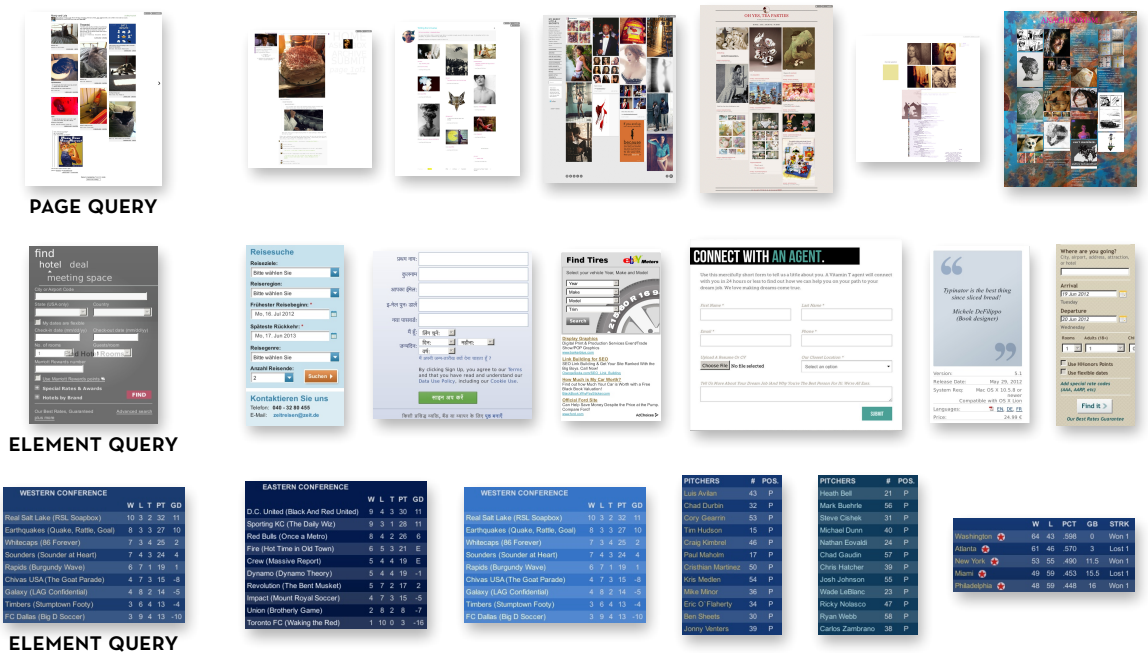


Figure 4.7: Top search results from querying the repository using the learned distance metric and three query elements. These results demonstrate how Webzeitgeist can be used to search for design alternatives (top, middle), and to identify template re-use between pages (bottom).

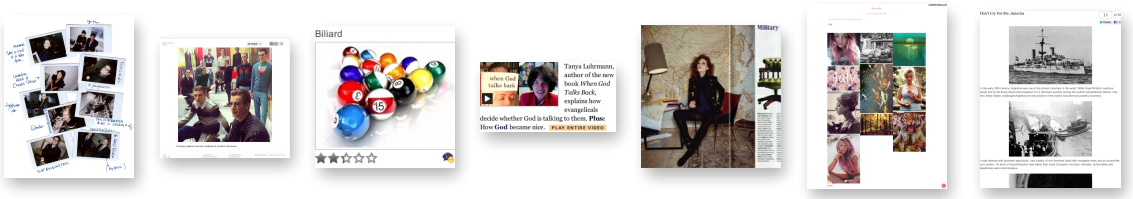


Figure 4.8: The top matches for the first (page) query in Figure 4.7 using *only* the vision-based GIST descriptors. While these elements are visually reminiscent of the query, most of them bear little structural or semantic relation to it.

4.2.2 Query-by-example

Machine learning techniques can also be used to enable example-based search over the repository. Using the structural semantic label data, we induced a distance metric in the 1679-dimensional descriptor space using OASIS, a metric-learning algorithm originally developed for large-scale image comparison [38]. The method takes as input sets of identically-labeled page elements, and attempts to learn a symmetric positive-definite matrix that minimizes interset distances. Once learned, this metric can be used to perform query-by-example searches on page regions via a nearest-neighbor search in the metric space. These nearest-neighbor computations can be performed in realtime via locality sensitive hashing [95].

Example-based search provides a powerful mechanism for navigating complex design spaces like the Web [153]. Figure 4.7 shows three example queries and their top results, demonstrating how Webzeitgeist can be used to search for alternatives for a given design artifact, and to identify template reuse between pages. The utility of this search interaction critically depends on the full Webzeitgeist feature space. For comparison, Figure 4.8 shows nearest-neighbor results for the top query in Figure 4.7 using only the vision-based GIST descriptors. While these elements are visually reminiscent of the query, they bear little structural or semantic relation to it.

4.3 A Design-based Search Engine

We hypothesize that designers will utilize these different retrieval strategies in concert during the design process. Thus, we built a real-time design search engine that supports all of the search interactions described in this chapter: design curation via DQL, query-by-example, and keyword search.

To see the value of this engine, consider the following scenario. Dave is about to kick off a new marketing startup and wants to scout out the competition. Doing a keyword search for “marketing,” he finds that the French company Milky has an attractive Web site (Figure 4.9). To find examples of other pages with similar

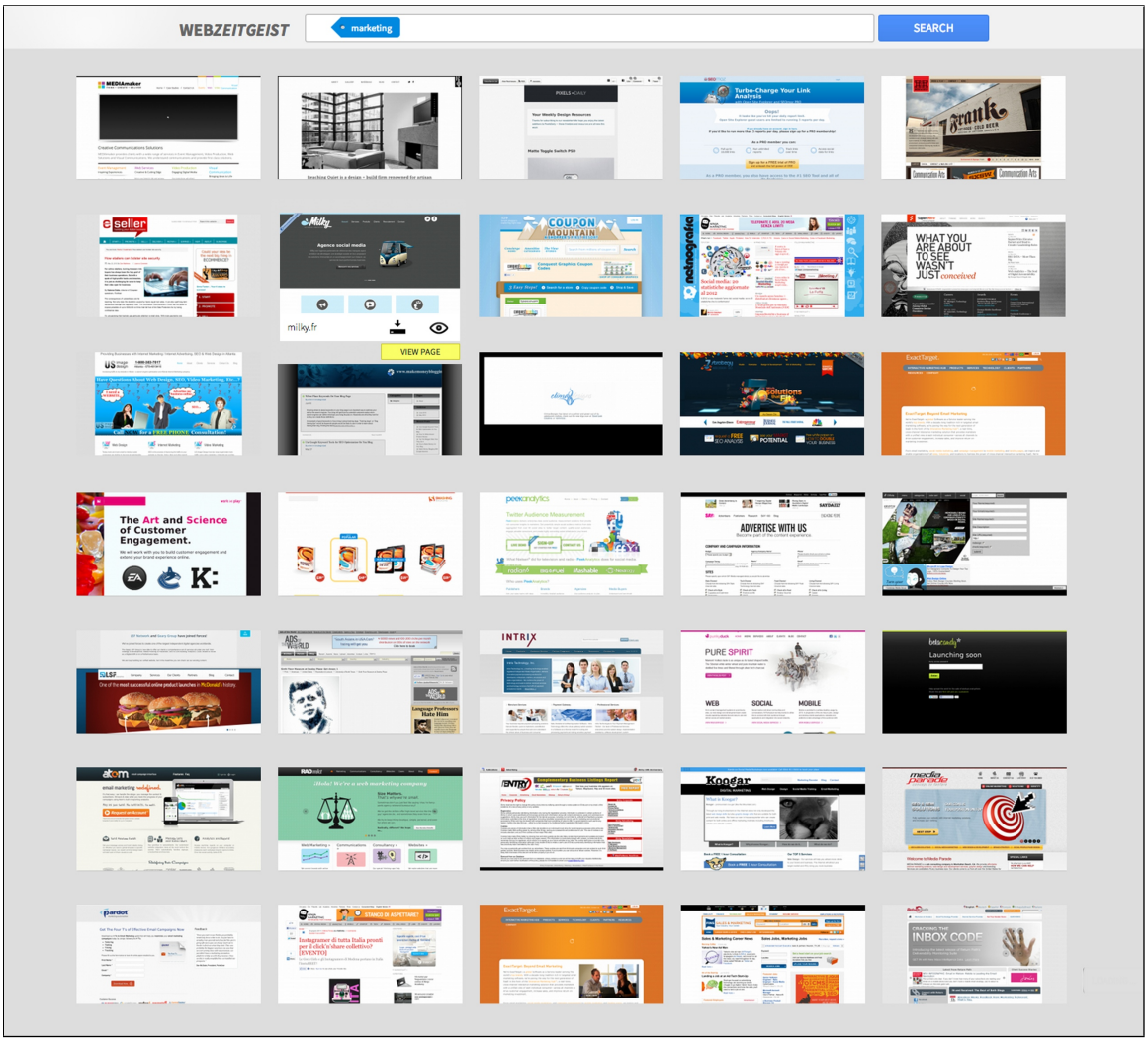


Figure 4.9: The Webzeitgeist search engine supports three types of keyword queries: domain (*e.g.,* blog, news), style (*e.g.,* colorful, funky), and structural semantics (*e.g.,* logo, advertisement). Structural semantic labels are assigned to elements based on a set of classifiers trained on crowdsourced tags. These are a few of the crowdsourced results for the domain label "marketing."

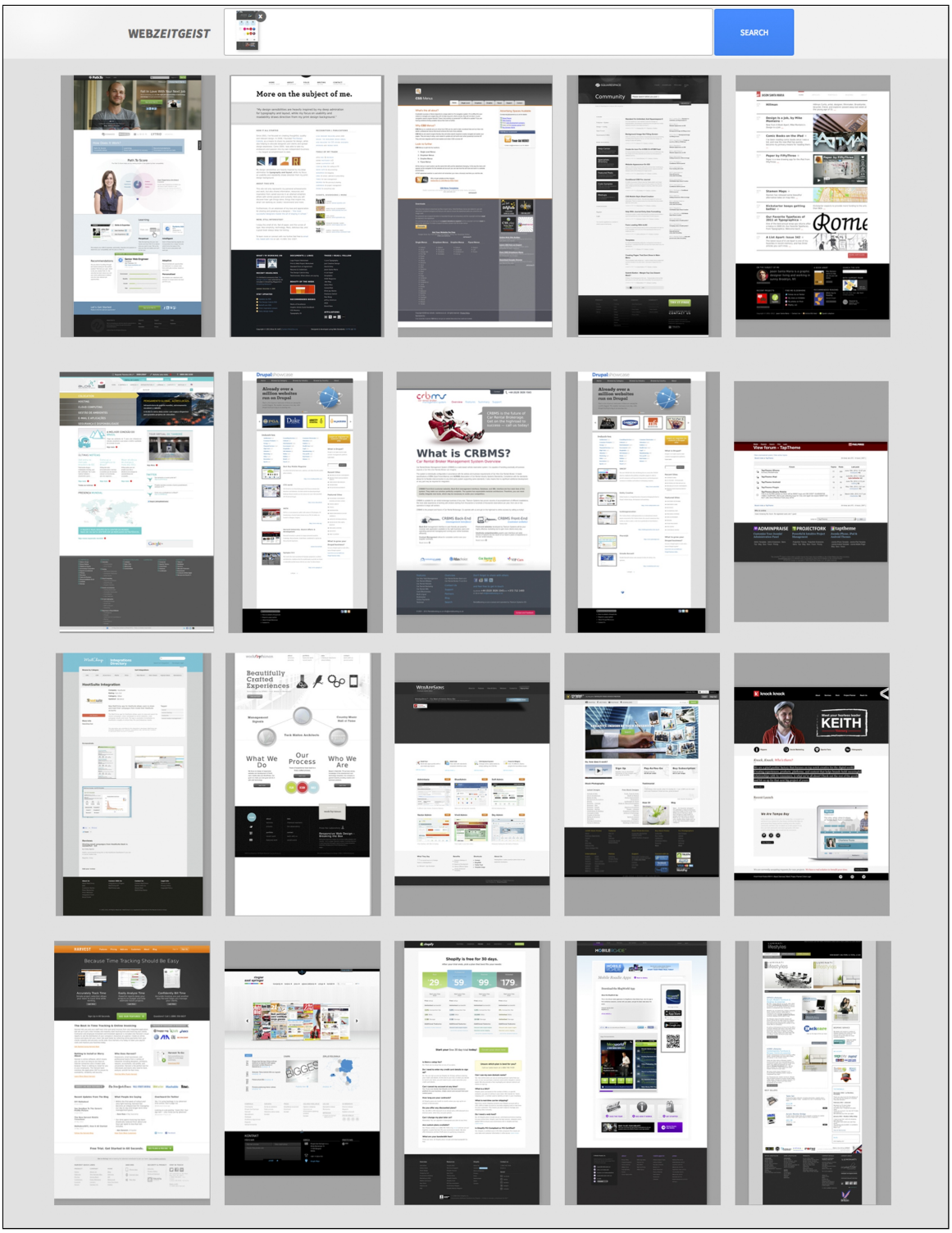


Figure 4.10: The design search engine also supports query-by-example searches. Users can select pages and page elements as queries to find similar design elements in the repository. The results shown here share the same dark-light-dark striping exhibited by the query page. Locality sensitive hashing is used to speed up nearest-neighbors computations in the learned metric space.

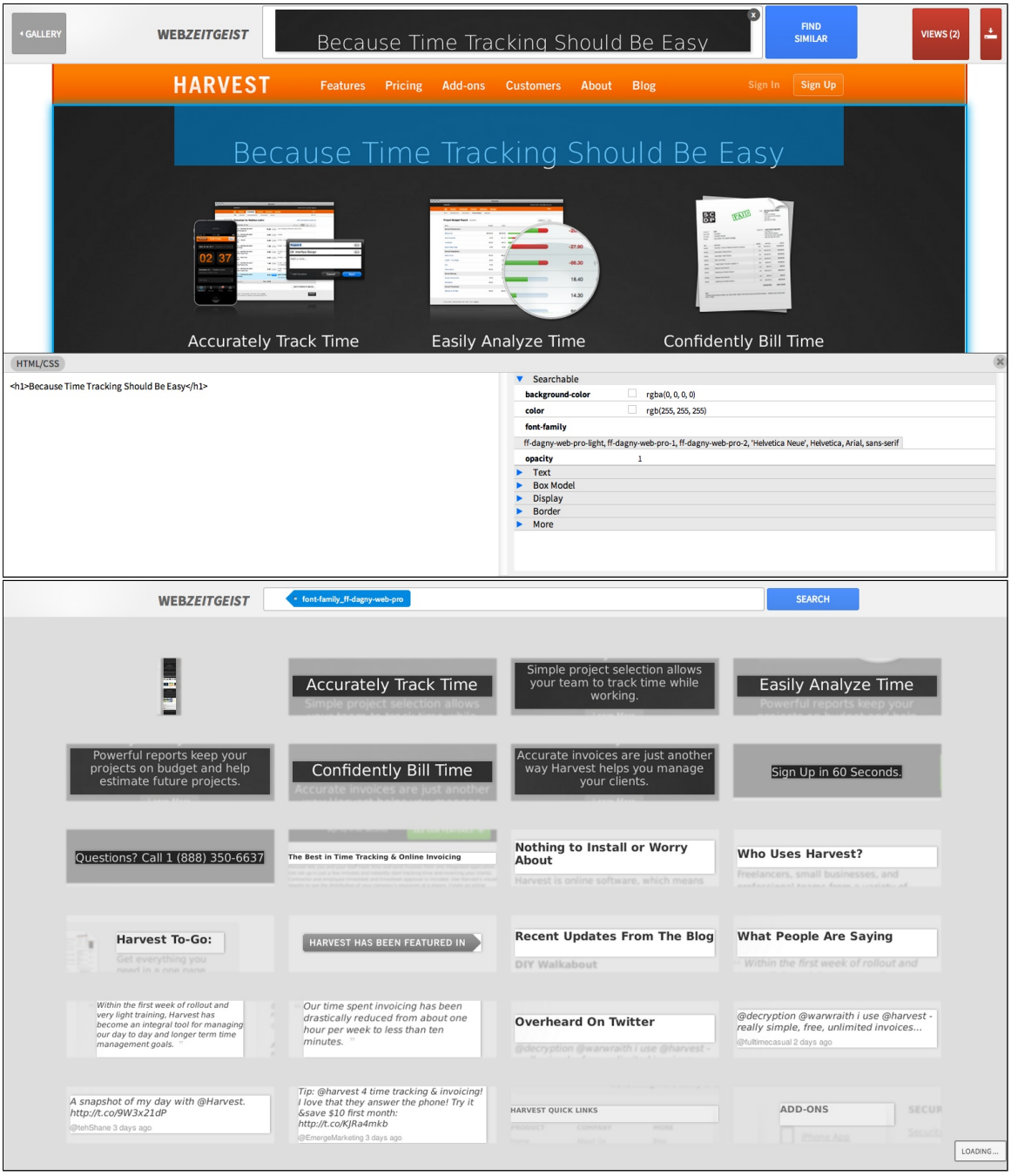


Figure 4.11: By zooming in on a page in the search engine, the user can inspect and copy HTML and CSS properties assigned to particular page elements (top). Moreover, users can also select properties in the inspector (*e.g.*, `font-family`) to perform DQL queries that return other page elements that have the same properties (bottom).

designs that could inform his brand, he selects Milky's page and does a query-by-example search for other similar design elements in the database (Figure 4.10). Once Dave finds a page that he likes, the search engine allows him to close the loop and inspect the page's implementation (Figure 4.11, top). Moreover, if Dave likes a particular aspect of the design (*e.g.,* a font that was used), he can select that field in the search engine's inspector and issue a DQL search to find other design elements that share its properties (Figure 4.11, bottom).

The query-by-example search results are computed using the trained distance metric combined with locality sensitive hashing (LSH) to make nearest-neighbor queries in the feature space run in real-time [95]. A drawback of LSH is that it requires a large number of hash tables to produce high-quality search results, all of which must be kept in memory during runtime. In the future, we may apply more advanced techniques such as multi-probe LSH [125] or algorithms for efficient K-Nearest Neighbor Graph construction [49] to mitigate these memory requirements.

For this search interface, we also crowdsourced more than 25,000 style (*e.g.,* "minimal", "clean") and more than 20,000 domain (*e.g.,* "business", "education") labels over more than 3000 distinct pages. In the future, we hope to use these labels to bootstrap classifiers that can generalize over the entire dataset.

4.4 Discussion and Future Work

This chapter demonstrates how Webzeitgeist enables design-based search at unprecedented scale, and how it can be used to rapidly prototype many different search interactions. An important direction for future work is understanding when different search strategies are useful during the design process. What types of search mechanisms support exploratory tasks? Do design professionals and novices exhibit distinct search patterns?

By logging user interactions in the search engine, we hope to understand how to better index examples (i.e., what features and metadata to store) and how to expose information to users (i.e., search strategies). Other retrieval mechanisms such as sketching [188] and faceted metadata search [79] may also prove useful.

Chapter 5

Automatic Retargeting

While the retrieval systems proposed in Chapter 4 can help users identify, aggregate, and understand design data, they do little to help designers leverage existing designs to produce new pages. This chapter introduces the *Bricolage* algorithm for transferring design and content between Web pages. Bricolage employs a novel, structured-prediction technique that learns to create coherent mappings between pages by training on human-generated exemplars. The produced mappings are then used to automatically transfer the content from one page into the style and layout of another. We show that Bricolage can learn to accurately reproduce human page mappings, and that it provides a general, efficient, and automatic technique for retargeting content between a variety of real Web pages.

5.1 Introduction

Despite the ready availability of design examples on the Web — Google had indexed more than one trillion unique URLs by 2008 [8] — little tool support exists for adapting existing designs to new purposes. While some prior work focused on facilitating easier copy-paste-adapt workflows for individual Web elements [57, 115, 33], adapting the gestalt structure of Web designs remains a manual, tedious process, or relies heavily on templates which homogenize page structure and yield cookie-cutter designs [72].

Figure 5.1: Bricolage computes coherent mappings between Web pages by matching visually and semantically similar page elements. The produced mapping can then be used to guide the transfer of content from one page into the design and layout of the other.

In this chapter, we introduce Bricolage to answer the question: what if *any* page on the Web could be a design template? The word "bricolage" refers to the creation of a work from a diverse range of available things. The Bricolage algorithm matches visually and semantically similar elements in pages to create coherent mappings between them. These mappings can then be used to automatically transfer the content from one page into the style and layout of the other (Figure 5.1).

Bricolage uses structured prediction [40] to learn how to transfer content between pages. It trains on a corpus of human-generated mappings, collected using a Web-based crowdsourcing interface. In an online study, 39 participants with some Web design experience specified correspondences between page regions and answered free-response questions about their rationale.

These mappings guided the design of Bricolage's matching algorithm. We found consistent structural patterns in how people created mappings between pages. Participants not only identified elements with similar visual and semantic properties, but also used their location in the pages' hierarchies to guide their assignments. Consequently, Bricolage employs a novel tree-matching algorithm that flexibly balances visual, semantic, and structural considerations. We demonstrate that this yields significantly more human-like mappings than using any one criteria alone.

This chapter presents the data collection study, the mapping algorithm, and the machine learning method. It then shows results demonstrating that Bricolage can learn to closely produce human mappings. Lastly, it illustrates how Bricolage is useful for a diverse set of design applications: for rapidly prototyping alternatives, retargeting content to alternate form factors such as mobile devices, and measuring the similarity of Web designs.

5.2 Collecting and Analyzing Human Mappings

To learn how people map content between pages, we created the *Bricolage Collector*, a Web application for gathering human page mappings from online workers (Figure 5.2).

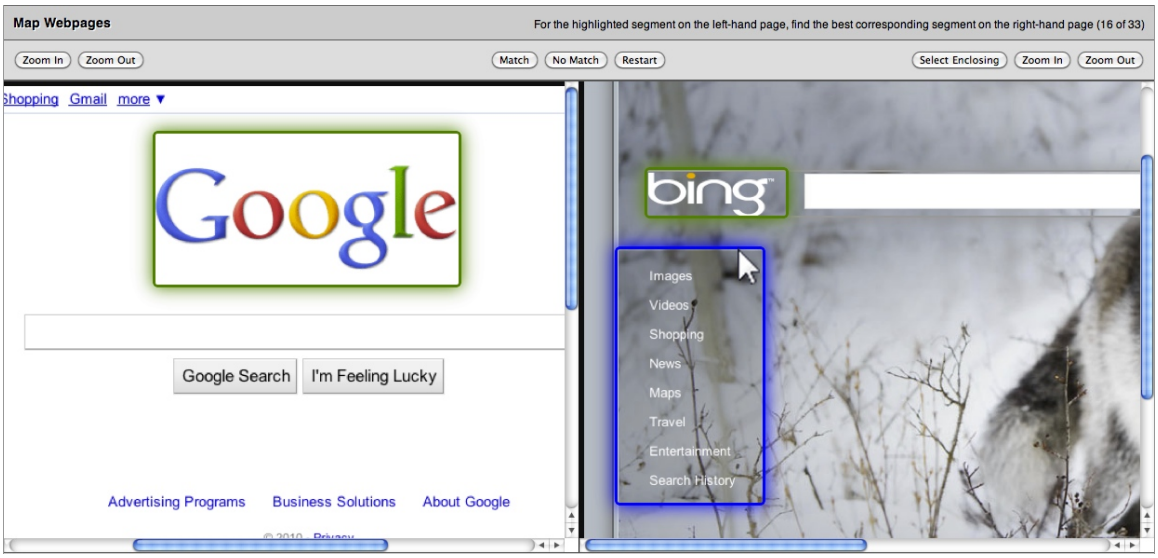


Figure 5.2: The Bricolage Collector Web application asks users to match each highlighted region in the *content* page (left) to the corresponding region in the *layout* page (right).

5.2.1 Study Design

We selected a diverse corpus of 50 popular Web pages chosen from the Alexa Top 100, Webby award winners, highly-regarded design blogs, and personal bookmarks. Within this corpus, we selected a focus set of eight page pairs. Each participant was asked to match one or two pairs from the focus set, and one or two more chosen uniformly at random from the corpus as a whole. The Collector gathered data about how different people map the same pair of pages, and about how people map many different pairs. We recruited 39 participants through email lists and online advertisements. Each reported some Web design experience.

5.2.2 Procedure

Participants watched a tutorial video demonstrating the Collector interface and describing the task. The video instructed participants to produce mappings for transferring the left page's *content* into the right page's *layout*. It emphasized

that participants could use any criteria they deemed appropriate to match elements. After the tutorial, the Collector presented participants with the first pair of pages.

The Collector interface iterates over the segmented regions in the content page one at a time, asking participants to find a matching region in the layout page. The user selects a matching region via the mouse or keyboard, and confirms it by clicking the MATCH button. If no good match exists for a particular region, the user clicks the NO MATCH button. After every fifth match, the interface presents a dialog box asking,"Why did you choose this assignment?" These rationale responses are logged along with the mappings, and submitted to a central server.

5.2.3 Results

Participants generated 117 mappings between 52 unique pairs of pages: 73 mappings for the 8 pairs in the focus set, and 44 covering the rest of the corpus. They averaged 10.5 seconds finding a match for each page region ($\min = 4.42s$, $\max = 25.0s$), and 5.38 ($\sigma^2 = 4.23s$, $\min = 4.42s$, $\max = 25.0s$), and 5.38 minutes per page pair ($\min = 1.52m$, $\max = 20.7m$). minutes per page pair ($\sigma^2 = 3.03m$, $\min = 1.52m$, $\max = 20.7m$). Participants provided rationales for 227 individual region assignments, averaging 4.7 words in length.

Consistency

We define the consistency of two mappings for the same page pair as the percentage of page regions with identical assignments. The average inter-mapping consistency of the focus pairs was 78.3% ($\sigma^2 = 10.2\%$, $\min = 58.8\%$, $\max = 89.8\%$). Moreover, 37.8% of page ($\min = 58.8\%$, $\max = 89.8\%$). 37.8% of page regions were mapped identically by all participants.

Rationale

Participants provided rationales like “title of rightmost body pane in both pages." We analyzed these rationales with Latent Semantic Analysis (LSA), which extracts contextual language usage in a set of documents [45]. LSA takes a bag-of-words approach to textual analysis: each document is treated as an unordered collection of words, ignoring grammar and punctuation. We followed the standard approach, treating each rationale as a document and forming the term-document matrix where each cell's value counts the occurrences of a term in a document. We used Euclidean normalization to make annotations of different lengths comparable, and inverse document-frequency weighting to deemphasize common words like *a* and *the*.

LSA decomposes the space of rationales into semantic “concepts.” Each concept is represented by a principal component of the term-document matrix, and the words with the largest projections onto the component are the concept's descriptors.

For the first component, the words with the largest projections are: *footer*, *link*, *menu*, *description*, *videos*, *picture*, *login*, *content*, *image*, *title*, *body*, *header*, *search*, and *graphic*. These words pertain primarily to visual and semantic attributes of page content.

For the second component, the words with the largest projections are: *both*, *position*, *about*, *layout*, *bottom*, *one*, *two*, *three*, *subsection*, *leftmost*, *space*, *column*, *from*, and *horizontal*. These words are mostly concerned with structural and spatial relationships between page elements.

Structure and Hierarchy

Two statistics examine the mappings' structural and hierarchical properties: one measuring how frequently the mapping preserves *ancestry*, and the other measuring how frequently it preserves *siblings*.

We define two matched regions to be *ancestry preserving* if their parent regions are also matched (Figure 5.3, left). A mapping's degree of ancestry preservation is the number of ancestry-preserving regions divided by the total number of matched

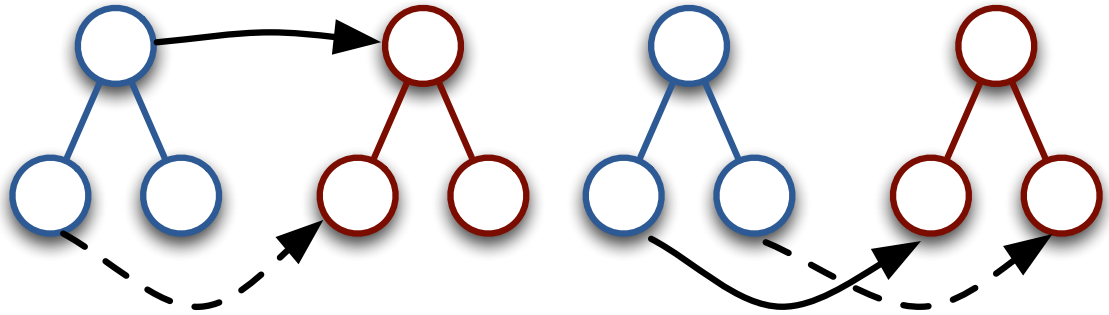


Figure 5.3: Examples of ancestry preservation (left) and sibling preservation (right) in page mappings.

regions. Participants' mappings preserved ancestry 53.3% of the time ($\sigma^2 = 19.6\%$, $\min = 7.6\%$, $\max = 95.5\%$).

Similarly, we define a set of page regions sharing a common parent to be *sibling preserving* if the regions they are matched to also share a common parent (Figure 5.3, right). Participants produced mappings that were 83.9% sibling preserving ($\sigma^2 = 8.13\%$, $\min = 58.3\%$, $\max = 100\%$).

5.3 Computing Page Mappings

The study's results suggest that mappings produced by different people are highly consistent: there is a "method to the madness" that may be learned. Moreover, the results suggest that algorithmically producing human-like mappings requires incorporating both semantic and structural constraints, and learning how to balance between them. The visual hierarchy computed by Webzeitgeist for each page provides a convenient starting point to compute page mappings, since the representation encodes both ancestry and sibling relationships as well as per-element visual and semantic features.

The problem of comparing trees arises naturally in diverse fields, including computational biology, compiler optimization, natural language processing, and computer vision, and even HTML pattern matching [18, 87, 101]. The most common measure for gauging the similarity of two labeled trees is the edit distance metric,

first introduced by Tai [173], which computes the cost of transforming one tree into another through a sequence of elementary node operations such as insertion, deletion, and renaming. To calculate this distance, a minimum-cost correspondence is established between the nodes of the two trees in a process known as *tree matching*.

In the classical formulation, these correspondences are *rigid*: they are not allowed to violate ancestor-descendant relationships between nodes, nor the left-to-right order of a node's children. As a result of these structural requirements, the problem admits an efficient dynamic programming algorithm, and the optimal matching can be found in cubic time [46]. If the ordering requirement is removed (but the ancestry requirement maintained), the edit distance computation becomes $\mathcal{NP}$-complete [190], and approximation algorithms are necessary [161].

In some domains, the most appropriate matchings may not strictly preserve ancestry. For instance, while reparenting even a single node in a phylogenetic tree of bacteria would destroy its validity, the ancestry relationships in the Document Object Model tree of a Web page are less prescriptive: moving a search bar from header to footer results in a different — but largely equivalent — page. This pattern follows for many other tree structures in design and data management, in which hierarchy plays an important — but not definitive — role [36, 117, 163].

We introduce *flexible* tree matching, which relaxes the requirement that the produced correspondence strictly preserve ancestry relationships. Instead, the algorithm provides a parameterized framework for controlling the relative import of labeling and hierarchy. The algorithm uses an edge-based cost model to match nodes with similar labels while simultaneously penalizing matchings which induce violations of the hierarchy or break up sibling groups. Determining the minimum edit distance between trees then reduces to finding a minimum-cost matching under this model.

We prove that flexible tree matching is $\mathcal{NP}$-complete in the strong sense, and give a corresponding stochastic approximation algorithm. We also show how to learn the parameters of the model from a corpus of examples via standard structured prediction techniques, and summarize results from applying the method to automatic retargeting in Web design.

5.4 A Flexible Model for Tree Matching

A *tree matching* is an injective binary relation defined between two labeled trees T_1 and T_2. The relation can be viewed as a bipartite graph between the trees' nodes, with edges representing editing operations for transforming one tree into the other. Edges identifying nodes with dissimilar labels represent *relabeling* operations, while nodes which are not mapped correspond to *insertions* or *deletions*.

To differentiate between mappings, classical tree matching requires a model that defines the relabeling cost between nodes and the insertion/deletion cost for nodes which are not matched. Given such a model, the tree-matching problem is to find a lowest-cost mapping between trees which preserves ancestry: once two nodes $m \in T_1$ and $n \in T_2$ are matched, the descendants of m can only be matched to descendants of n, and vice versa.

Flexible matching relaxes this rigid ancestry requirement in favor of a tunable edge-based cost model. It forms a complete bipartite graph G between $T_1 \cup \{\otimes_1\}$ and $T_2 \cup \{\otimes_2\}$, where $\otimes_1$ and $\otimes_2$ are auxiliary *no-match* nodes. Each edge in G is assigned a cost comprising three terms: a relabeling term c_r, penalizing edges that connect nodes with different labels, an ancestry term c_a, penalizing edges that violate ancestry relationships, and a sibling term c_s, penalizing edges that break up sibling groups. The cost of an edge $c(e)$ is the sum of these three terms, and the goal of flexible tree matching is to produce a set of edges $M \subset G$ such that every node in $T_1 \cup T_2$ is covered by precisely one edge and the total mapping cost $c(M) = \frac{1}{|T_1|+|T_2|} \sum_{e \in M} c(e)$ is minimized.

5.5 Exact Edge Costs

We define the cost of an edge $e \in T_1 \times T_2$,

$$c(e) = c_r(e) + c_a(e) + c_s(e).$$

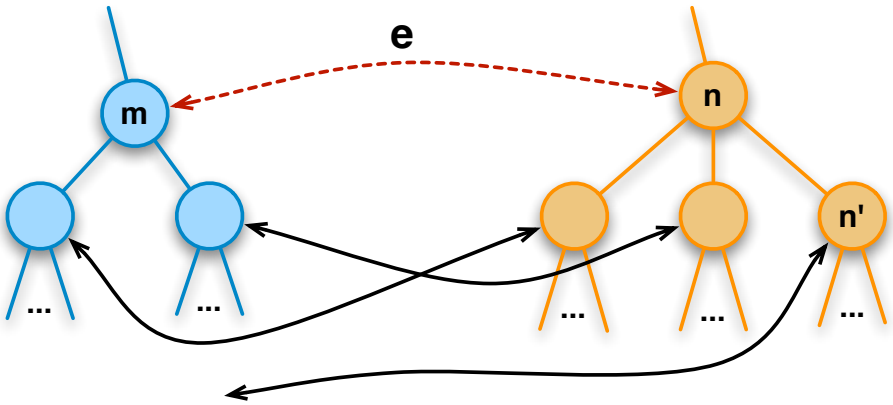


Figure 5.4: Flexible tree matching determines the ancestry penalty for an edge $e = [m, n]$ by counting the children of m and n which induce ancestry violations. In this example, n' is an ancestry-violating child of n because it is not mapped to a child of m; therefore, n' induces an ancestry cost on e.

For edges in G connecting tree nodes to no-match nodes, we fix the cost $c(e) = w_n$, where w_n is a constant no-match penalty weight.

The relabeling term $c_r([m, n])$ defines the cost of swapping the labels of nodes m and n. This function is domain-dependent, and user-specified. If the labels are identical, $c_r([m, n]) = 0$.

The ancestry cost $c_a(\cdot)$ penalizes edges that violate ancestry relationships between the trees' nodes. Consider a node $m \in T_1$, and let $C(m)$ denote the children of m and $M(m)$ denote the image of m in the matching. We define the *ancestry-violating children* of m, $V(m)$, to be the set of m's children that map to nodes that are not $M(m)$'s children, *i.e.,*

$$V(m) = \left\{ m' \in C(m) \mid M(m') \in T_2 \setminus C(M(m)) \right\},$$

and define $V(n)$ symmetrically. Then, the ancestry cost for an edge is proportional to the number of ancestry violating children of its terminal nodes

$$c_a([m,n]; M) = w_a \left(|V(m)| + |V(n)| \right),$$

where w_a is a constant ancestry weight (see Figure 5.4).

The sibling cost $c_s(\cdot)$ penalizes edges that fail to preserve sibling relationships between trees. To calculate this term, we first define a few tree-related concepts. Let $P(m)$ denote the parent of m. The *sibling group* of a node m comprises the children of its parent: $S(m) = \{C(P(m))\}$. Given a mapping M, the *sibling-invariant subset* of m, $I_M(m)$, is the set of nodes in m's sibling group that map to nodes in $M(m)$'s sibling group, *i.e.,*

$$I_M(m) = \left\{m' \in S(m) \mid M(m') \in S(M(m))\right\};$$

the *sibling-divergent subset* of m, $D_M(m)$, is the set of nodes in m's sibling group that map to nodes in T_2 not in $M(m)$'s sibling group,*i.e.,*

$$D_M(m) = \left\{m' \in S(m) \setminus I_M(m) \mid M(m') \neq \otimes_2\right\};$$

and the set of *distinct sibling families* that m's sibling group maps into is

$$F_M(m) = \bigcup_{m' \in S(m)} P(M(m')).$$

We define all corresponding terms for n symmetrically, and then compute the total sibling cost

$$c_s([m,n]; M) = w_s \left(\frac{|D_M(m)|}{|I_M(m)||F_M(m)|} + \frac{|D_M(n)|}{|I_M(n)||F_M(n)|} \right),$$

where w_s is a constant sibling violation weight. The two ratios in the cost increase when siblings are broken up by the matching (*i.e.,* their images have different parents), and decrease when siblings groups are maintained (see Figure 5.5). The $F_M(\cdot)$ terms are included to guarantee that the total sibling penalty contributed by a tree is bounded by the number of nodes it contains.

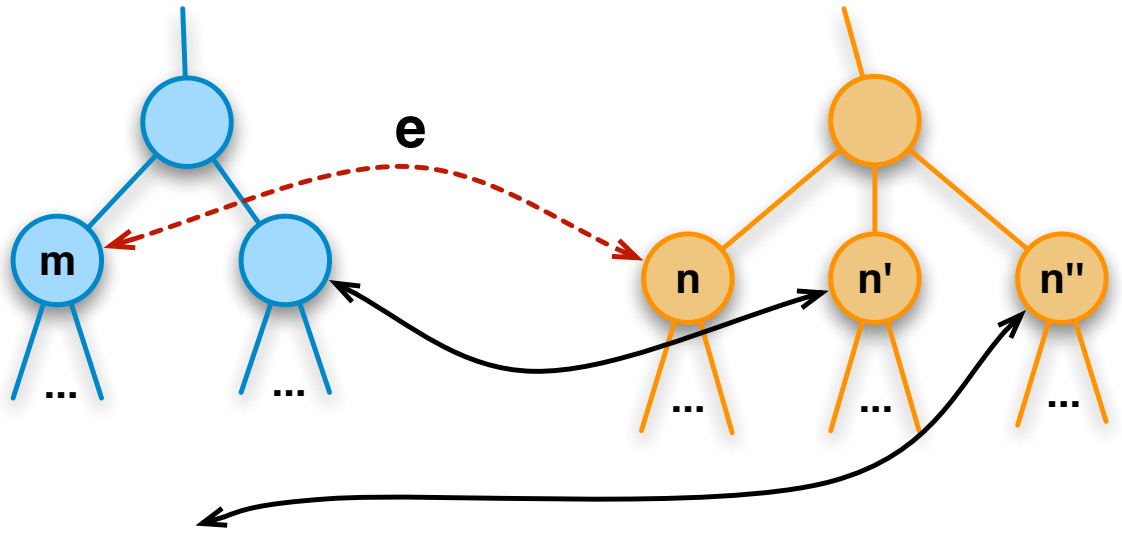


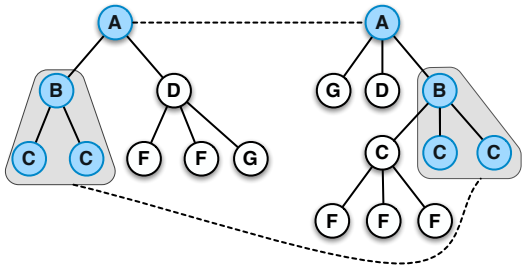
Figure 5.5: To determine the sibling penalty for an edge $e = [m, n]$, the algorithm computes the sibling-invariant and sibling-divergent subsets of m and n. In this example, $I_M(n) = \{n'\}$ and $D_M(n) = \{n''\}$; therefore, n' decreases the sibling cost on e and n'' increases it.

5.6 Example Matchings

Figure 5.6 compares ordered, unordered, and flexible tree matching. In these examples, a simple relabeling function assigns a constant weight w_r to all edges between tree nodes with differing labels. In ordered and unordered matching, the rigid preservation of ancestry leaves many nodes unmatched. In flexible matching, more common structure is preserved between the trees. By varying the terms in the cost model, different mappings can be achieved.

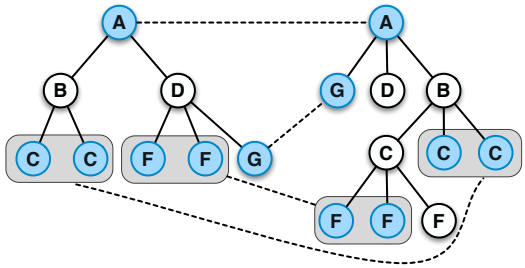
5.7 Flexible Tree Matching is $\mathcal{NP}$-complete

We prove that flexible tree matching is strongly $\mathcal{NP}$-complete. We employ a simple version of the flexible cost model from Section 5.6, where $w_r = 1$, $w_n = 1$, $w_a = 0$, and $w_s = 1$. That is, ancestry violations are forgiven but all relabelings, no-matches, and sibling violations are costly. Given this model and two labeled trees T_1 and T_2,



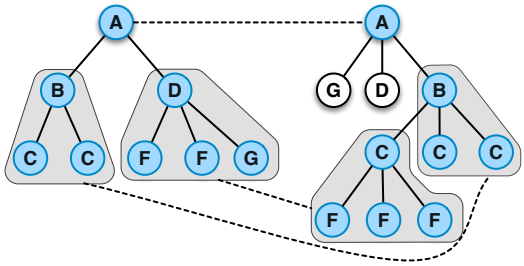
Ordered Tree Matching

$w_r = 2.0; w_n = 1.0$



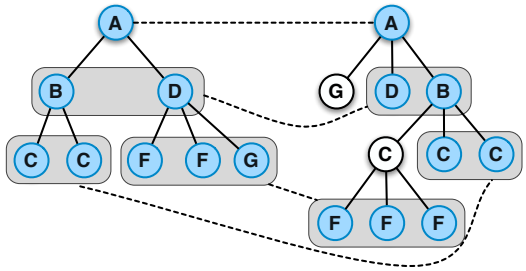
Unordered Tree Matching

$w_r = 2.0; w_n = 1.0$



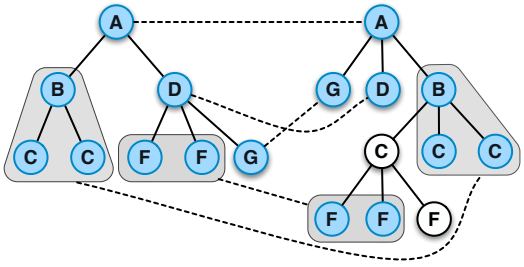
Flexible Tree Matching

$w_r = 0.9; w_n = 1.0; w_a = 1.0; w_s = 0.1$



Flexible Tree Matching

$w_r = 0.9; w_n = 1.0; w_a = 0.1; w_s = 1.0$



Flexible Tree Matching

$w_r = 1.0; w_n = 1.0; w_a = 0.5; w_s = 0.5$

Figure 5.6: Examples of ordered, unordered, and flexible tree matchings.

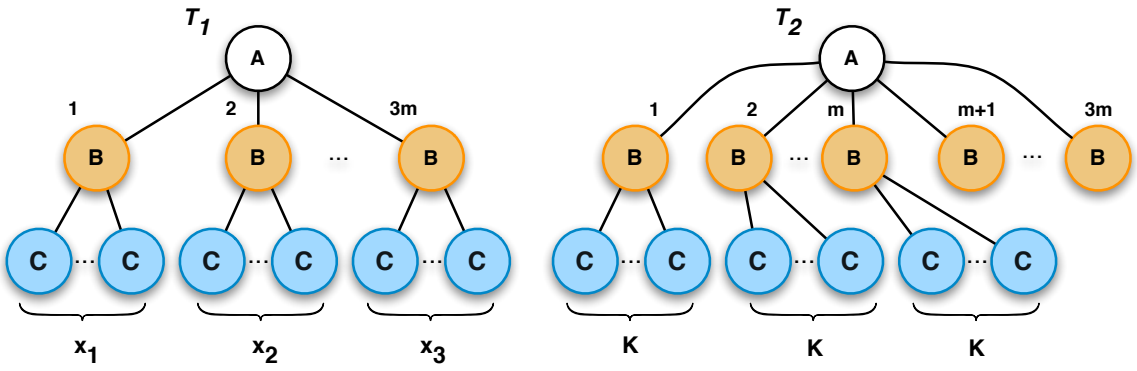


Figure 5.7: The tree construction for the reduction from 3-PARTITION.

we address the decision problem "Does there exist a zero-cost flexible mapping between the trees?"

This problem is in $\mathcal{NP}$, since a proposed zero-cost mapping can be verified in polynomial time. To show that the problem is $\mathcal{NP}$-hard, we formulate a polynomial-time reduction from 3-PARTITION [68].

The 3-PARTITION problem is to decide whether a given multiset of integers can be partitioned into triples that all have the same sum. More formally, each instance is a finite set S of $3m$ integers, where each integer $x_i \in S$ satisfies $K/4 < x_i < K/2$ for some bound $K \in \mathbf{Z}^+$ and $\sum x_i = 3K$. The question is to determine whether or not S can be partitioned into m disjoint sets $U_1, \ldots, U_m$, each with three elements, so that $\sum_{u \in U_i} u = K$ for every $i \in \{1, 2, \ldots, m\}$. This problem is $\mathcal{NP}$-complete even when K is polynomial in m [68].

Given a 3-PARTITION instance, we construct trees T_1 and T_2 as in Figure 5.7. Each tree consists of three levels, and the nodes are labeled based on their level. T_1 represents the set S, and contains $3m$ subtrees on the second level, with subtree i possessing x_i leaf nodes. T_2 contains m subtrees on the second level with K children each, and $2m$ more one-node subtrees. Since K is polynomial in m, this construction takes time polynomial in the size of the 3-PARTITION instance.

Under what circumstances can a zero-cost matching exist between these two trees? Since $w_n = 1$, such a matching must map every node in T_1 to some node in T_2, and vice versa. Similarly, since $w_r = 1$, the matching cannot identify nodes in

different levels of the trees. Most importantly, since $w_s = 1$, a zero-cost matching must preserve the $3m$ sibling groups of leaf nodes in T_1: each of the m leaf node sibling groups in T_2 must be matched to exactly three leaf node sibling groups in T_1. Thus, a zero-cost matching exists only if the corresponding 3-PARTITION instance has a solution. Conversely, a solution to the 3-PARTITION instance indicates which leaf node sibling groups in T_1 should be matched to a common leaf node sibling group in T_2, naturally inducing a zero-cost matching.

This reduction implies that no polynomial-time (or even pseudopolynomial-time) algorithm can exist for flexible tree matching. For this reason, we propose a stochastic optimization algorithm for approximating the optimal matching, based on bounding the edge costs.

5.8 Bounding Edge Costs

While the cost model described in Section 5.5 balances labeling and structural constraints, it cannot be used to search for an optimal mapping $M^\star$ directly. Although $c_r([m, n])$ can be evaluated for an edge by inspecting m and n, $c_a(\cdot)$ and $c_s(\cdot)$ require information about the other edges in the mapping.

While we cannot precisely evaluate $c_a(\cdot)$ and $c_s(\cdot)$ *a priori*, we can compute bounds for them on a per-edge basis. Moreover, each time we accept an edge $[m, n]$ into M, we can remove all the other edges incident on m and n from G. Each time we prune an edge in this way, the bounds for other nearby edges may be improved. Therefore, we employ a Monte Carlo algorithm to approximate $M^\star$, stochastically fixing an edge in G, pruning away the other edges incident on its nodes, and updating the bounds on those that remain.

To bound the ancestry cost of an edge $[m, n] \in G$, we consider each child of m and n and answer two questions. First, is it *impossible* for this node to induce an ancestry violation? Second, is it *unavoidable* that this node will induce an ancestry violation? The answer to the first question informs the upper bound for $c_a(\cdot)$; the answer to the second informs the lower.

A node $m' \in C(m)$ can induce an ancestry violation if there is some edge between it and a node in $T_2 \setminus (C(n) \cup \{\otimes_2\})$. Conversely, m' is not *guaranteed* to induce an ancestry violation if some edge exists between it and a node in $C(n) \cup \{\otimes_2\}$. Accordingly, we define indicator functions

$$\mathbf{1}_a^{\mathcal{U}}(m', n) = \begin{cases} 1 & \text{if } \exists [m', n'] \in G \text{ s.t. } n' \notin C(n) \cup \{\otimes_2\} \\ 0 & \text{else} \end{cases},$$

$$\mathbf{1}_a^{\mathcal{L}}(m', n) = \begin{cases} 1 & \text{if } \nexists [m', n'] \in G \text{ s.t. } n' \in C(n) \cup \{\otimes_2\} \\ 0 & \text{else} \end{cases}.$$

Then, the upper and lower bounds for $c_a([m, n]; M)$ are

$$\mathcal{U}_a([m,n]) = w_a \left(\sum_{m' \in C(m)} \mathbf{1}_a^{\mathcal{U}}\left(m', n\right) + \sum_{n' \in C(n)} \mathbf{1}_a^{\mathcal{U}}\left(n', m\right) \right),$$

and

$$\mathcal{L}_a([m,n]) = w_a \left(\sum_{m' \in C(m)} \mathbf{1}_a^{\mathcal{L}}\left(m', n\right) + \sum_{n' \in C(n)} \mathbf{1}_a^{\mathcal{L}}\left(n', m\right) \right).$$

Figure 5.8 illustrates the computation of these bounds. Pruning edges from G causes the upper bound for $c_a([m, n]; M)$ to decrease, and the lower bound to increase.

Similarly, we can bound $c_s([m, n]; M)$ by bounding the number of divergent siblings, invariant siblings, and distinct families: $|D(\cdot)|$, $|I(\cdot)|$, and $|F(\cdot)|$. Let $\bar{S}(m) = S(m) \setminus \{m\}$ and consider a node $m' \in \bar{S}(m)$. It is possible that m' is in $D_M(m)$ as long as some edge exists between it and a node in $T_2 \setminus (\bar{S}(n) \cup \{\otimes_2\})$. Conversely, m' cannot be guaranteed to be in $D_M(m)$ as long as some edge exists

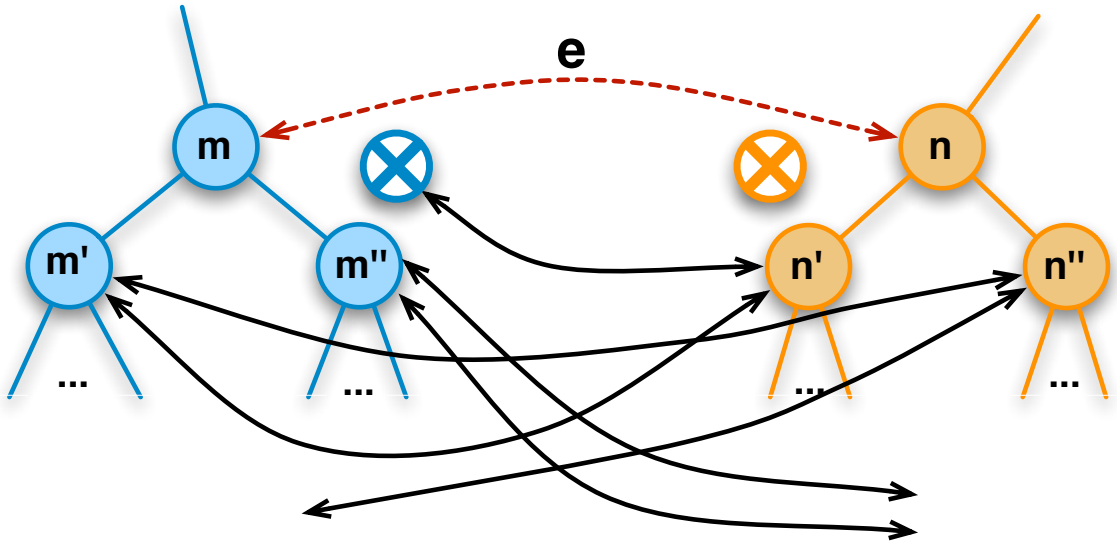


Figure 5.8: To bound $c_a([m,n];M)$, observe that neither m' nor n' can induce an ancestry violation. Conversely, m'' is guaranteed to violate ancestry. No guarantee can be made for n''. Therefore, the lower bound for c_a is w_a, and the upper bound is $2w_a$.

between it and a node in $\bar{S}(n) \cup \{\otimes_2\}$. Then, we have

$$\mathbf{1}_D^{\mathcal{U}}(m',n) = \begin{cases} 1 & \text{if } \exists [m',n'] \in G \text{ s.t. } n' \notin \bar{S}(n) \cup \{\otimes_2\} \\ 0 & \text{else} \end{cases},$$
$$\mathcal{U}_D(m,n) = \sum_{m' \in \bar{S}(m)} \mathbf{1}_D^{\mathcal{U}}(m',n),$$

and

$$\mathbf{1}_D^{\mathcal{L}}(m',n) = \begin{cases} 1 & \text{if } \nexists [m',n'] \in G \text{ s.t. } n' \in \bar{S}(n) \cup \{\otimes_2\} \\ 0 & \text{else} \end{cases},$$
$$\mathcal{L}_D(m,n) = \sum_{m' \in \bar{S}(m)} \mathbf{1}_D^{\mathcal{L}}(m',n).$$

The bounds for $|I_M(m)|$ are similarly given by

$$\mathbf{1}_I^{\mathcal{U}}(m', n) = \begin{cases} 1 & \text{if } \exists [m', n'] \in G \text{ s.t. } n' \in \bar{S}(n) \\ 0 & \text{else} \end{cases},$$
$$\mathcal{U}_I(m, n) = 1 + \sum_{m' \in \bar{S}(m)} \mathbf{1}_I^{\mathcal{U}}(m', n),$$

and

$$\mathbf{1}_I^{\mathcal{L}}(m', n) = \begin{cases} 1 & \text{if } \forall [m', n'] \in G,\ n' \in \bar{S}(n) \\ 0 & \text{else} \end{cases},$$
$$\mathcal{L}_I(m, n) = 1 + \sum_{m' \in \bar{S}(m)} \mathbf{1}_I^{\mathcal{L}}(m', n).$$

For all nonzero sibling costs, the lower bound for $|F_M(m)|$ is 2 and the upper bound is $\mathcal{L}_D(m, n) + 1$. All remaining quantities are defined symmetrically. Then, upper and lower bounds for $c_s([m, n]; M)$ are given by

$$\mathcal{U}_s([m, n]) = \frac{w_s}{2} \left(\frac{\mathcal{U}_D(m, n)}{\mathcal{L}_I(m, n)} + \frac{\mathcal{U}_D(n, m)}{\mathcal{L}_I(n, m)} \right)$$

and

$$\mathcal{L}_s([m, n]) = \\ w_s \left(\frac{\mathcal{L}_D(m, n)}{\mathcal{U}_I(m, n) \left(\mathcal{L}_D(m, n) + 1 \right)} + \frac{\mathcal{L}_D(n, m)}{\mathcal{U}_I(n, m) \left(\mathcal{L}_D(n, m) + 1 \right)} \right).$$

Figure 5.9 illustrates the computations of the bounds for the sibling cost term.

With bounds for the ancestry and sibling terms in place, upper and lower bounds for the total edge cost are $c_{\mathcal{U}}(e) = c_r(e) + \mathcal{U}_a(e) + \mathcal{U}_s(e)$ and $c_{\mathcal{L}}(e) = c_r(e) + \mathcal{L}_a(e) + \mathcal{L}_s(e)$.

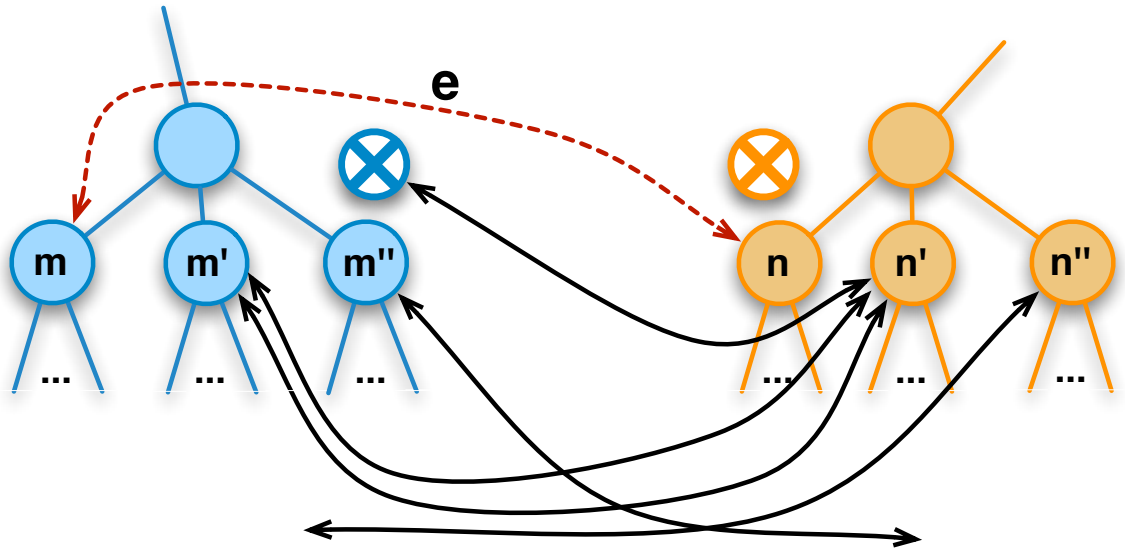


Figure 5.9: To bound $c_s([m,n];M)$, observe that m' is guaranteed to be in $I_M(m)$, and m'' is guaranteed to be in $D_M(m)$. No guarantees can be made for n' and n''. Therefore, the lower bound for c_s is $w_s/4$, and the upper bound is $3w_s/4$.

5.9 Approximating the Optimal Mapping

To approximate the optimal mapping M^*, we use the Metropolis algorithm [146]. We represent each matching as an ordered list of edges M, and define a Boltzmann-like objective function

$$f(M) = \exp\left[-\beta\, c(M)\right],$$

where β is a constant. At each iteration of the algorithm, a new mapping $\hat{M}$ is proposed, and becomes the new reference mapping with probability

$$\alpha(\hat{M}|M) = \min\left(1, \frac{f(\hat{M})}{f(M)}\right).$$

The algorithm runs for N iterations, and the mapping with the lowest cost is returned.

To initialize M, the bipartite graph G is constructed and the edge bounds initialized. Then, the edges in G are traversed in order of increasing bound. Each edge is considered for assignment to M with some fixed probability γ, until an edge is chosen. If the candidate edge can be fixed and at least one complete matching still exists, it is appended to M, the other edges incident on its terminal nodes are pruned, and the bounds for the remaining edges in G are tightened.

To propose $\hat{M}$, we choose a random index $j \in [1, |M|]$. Then, we re-initialize G, and fix the first j edges in M. To produce the rest of the matching, we repeat the iterative edge selection process described above. In our implementation, we take $\gamma = .7$ and $N = 100$; β is chosen on a per-domain basis.

5.10 Learning Cost Models

While flexible tree matching can be used with any cost model comprising weights w_r, w_a, w_s, and w_n, it is often desirable to *learn* a model that will produce mappings with domain-dependent characteristics. In particular, given a set of trees and exemplar matchings defined between them, we can use the generalized perceptron algorithm to learn weights under which the example matchings are minimal [40].

First, we reformulate the cost of a mapping $c(M)$ in terms of a weight vector $\mathbf{w} = \langle w_r, w_a, w_s, w_n \rangle$. For each edge, we compute the difference between the real-valued labels of its terminal nodes and the exact ancestry and sibling costs, and concatenate these values along with a Boolean no-match indicator into a feature vector $\mathbf{f}_e$. The edge cost can then be computed as $c(e) = \mathbf{w}^T \mathbf{f}_e$. Given a mapping M, the algorithm assembles an aggregate feature vector $\mathbf{F}_M = \frac{1}{|T_1|+|T_2|} \sum_{e \in M} \mathbf{f}_e$ to calculate the mapping cost $c(M) = \mathbf{w}^T \mathbf{F}_M$.

In each training iteration, the perceptron randomly selects a pair of trees and an associated mapping M from the training set. Next, it computes a new mapping $\hat{M} \approx \operatorname{argmin}_M \mathbf{w}_i^T \mathbf{F}_M$ using the current cost model $\mathbf{w}_i$. For the first iteration, $\mathbf{w}_0 = 0$. Based on the resultant mapping, a new aggregate feature vector $\mathbf{F}_{\hat{M}}$ is calculated, and the cost model is updated by $\mathbf{w}_{i+1} = \mathbf{w}_i + \alpha_i \left(\mathbf{F}_{\hat{M}} - \mathbf{F}_M \right)$, where $\alpha_i = 1/\sqrt{i+1}$ is the learning rate.

Figure 5.10: A current limitation of the content transfer algorithm illustrating the challenges of HTML/CSS. The target page's CSS prevents the bounding beige box from expanding. This causes the text to overflow (synthesized page). Also, the target page expects all headers to be images. This causes the "About Me" header to disappear (synthesized page). An improved content transfer algorithm could likely address both of these issues.

While the perceptron algorithm is only *guaranteed* to converge if the training set is linearly separable, in practice it produces good results for many diverse data sets [40]. Since the weights may oscillate during the final stages of the learning, the final cost model is produced by averaging over the last few iterations.

5.11 Web Content Retargeting

Bricolage uses flexible tree matching to compute a minimum-cost mapping between visual page hierarchies which balances structural and semantic constraints. To learn a cost model that will produce human-like mappings, Bricolage trains the parameters of the algorithm, $\mathbf{w}$, on the set of human mappings collected during the online study. The feature vector $\mathbf{f}_e$ is computed based on the ancestry and sibling relationships in the visual hierarchy, and a subset of the per-element features computed and stored by Webzeitgeist.

Once a cost model is trained, it is fed to the matching algorithm, which uses it to predict mappings between any two pages. Bricolage then uses these computed

mappings to automatically transfer the content from one page into the style and layout of another. In its segmented page representation, page content (text, images, links, form fields) lives on the leaf nodes of the page tree. Before transferring content, the inner HTML of each node in the source page is preprocessed to inline CSS styles and convert embedded URLs to absolute paths. Then, content is moved between mapped nodes by replacing the inner HTML of the target node with the inner HTML of the source node.

Content matched to a no-match node can be handled in one of two ways. In the simplest case, unmatched source nodes are ignored. However, if important content in the source page is not mapped, it may be more desirable to insert the unmatched node into the target page parallel to its mapped siblings, or beneath its lowest mapped ancestor.

This approach works well for many pages. Occasionally, the complexity and diversity of modern Web technologies pose practical challenges to resynthesizing coherent HTML. Many pages specify style rules and expect certain markup patterns, which may cause the new content to be rendered incorrectly (Figure 5.10). Furthermore, images and plugin objects (*e.g.,* Flash, Silverlight) have no CSS style information that can be borrowed; when replaced, the new content will not exhibit the same visual appearance and may seem out of place. Lastly, embedded scripts are often tightly coupled with the original page's markup and break when naïvely transferred. Consequently, the current implementation ignores them, preventing dynamic behavior from being borrowed. A more robust content transfer algorithm is required to address these issues and remains future work.

5.12 Results

We demonstrate the efficacy of Bricolage in two ways. First, we show several practical examples of Bricolage in action. Second, we evaluate the machine learning components by performing a hold-out cross-validation experiment on the gathered human mappings.

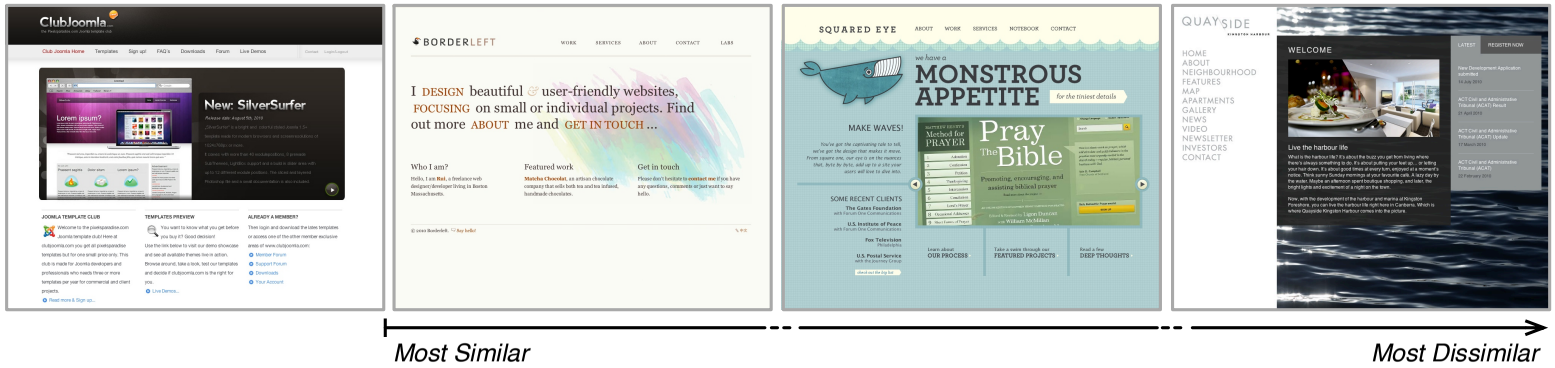


Figure 5.11: Bricolage can be used to induce a distance metric on the space of Web designs. By mapping the leftmost page onto each of the pages in the corpus and examining the mapping cost, we can automatically differentiate between pages with similar and dissimilar designs.

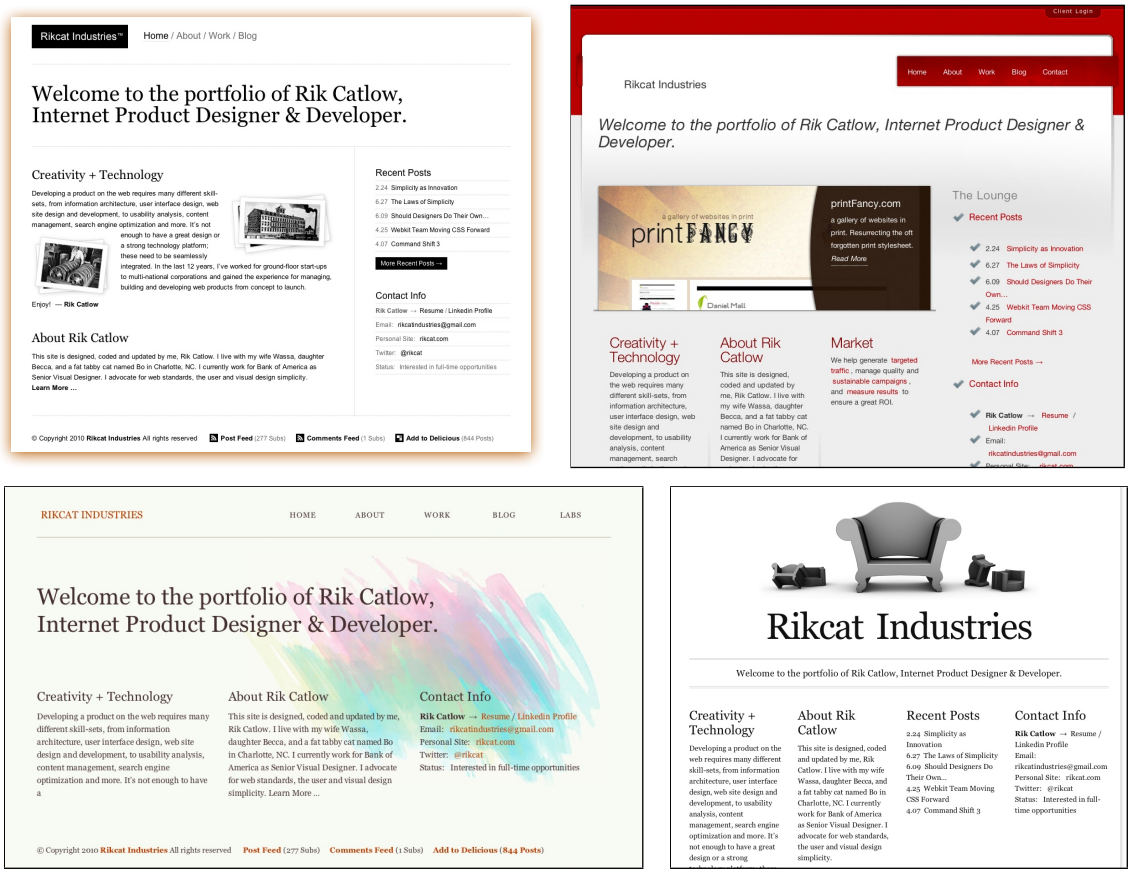


Figure 5.12: Bricolage used to rapidly prototype many alternatives. Top-left: the original Web page. Rest: the page automatically retargeted to three other layouts and styles.

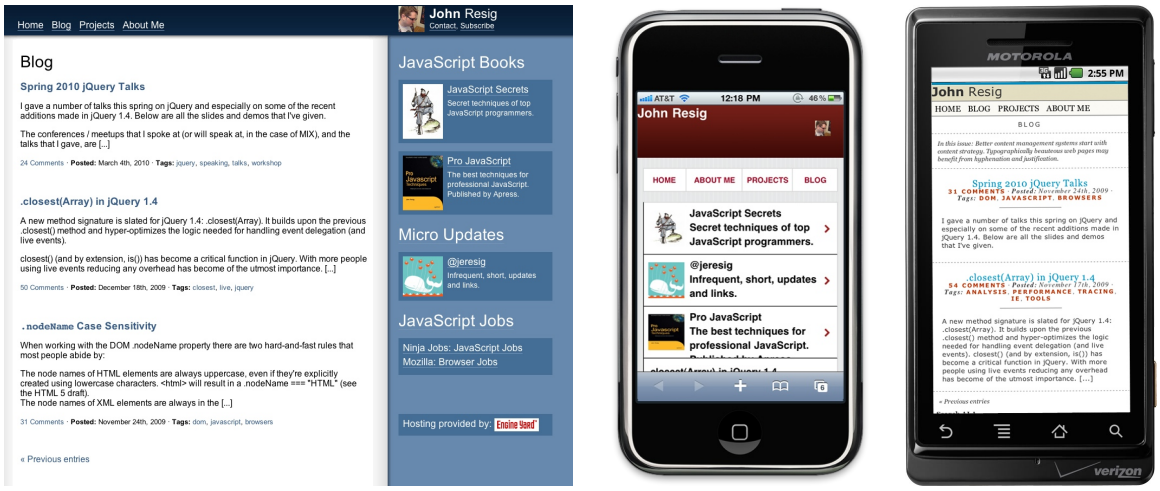


Figure 5.13: Bricolage can retarget Web pages designed for the desktop to mobile devices. Left: the original Web page. Right: the page automatically retargeted to two different mobile layouts.

5.12.1 Examples

Figure 5.12 demonstrates the algorithm in a rapid prototyping scenario, in which an existing page is transformed into several potential replacement designs. Creating multiple alternatives facilitates comparison, team discussion, and design space exploration [53, 78, 174]. Figure 5.13 demonstrates that Bricolage can be used to retarget content across form factors, showing a full-size Web page automatically mapped into two different mobile layouts.

Figure 5.11 illustrates an ancillary benefit of Bricolage's cost model. Since Bricolage searches for the optimal mapping between pages, the returned cost can be interpreted as an approximate distance metric on the space of page designs. Although the theoretical properties of this metric are not strong (it satisfies neither the triangle inequality nor the identity of indiscernibles), in practice it may provide a useful mechanism for automatically differentiating between pages with similar and dissimilar designs.

5.12.2 Machine Learning Results

To test the effectiveness of Bricolage's machine learning components, we ran a hold-out test. We used the 44 collected mappings outside the focus set as training data, and the mappings in the focus set as test data. The perceptron was run for 400 iterations, and the weight vector averaged over the last 20. The learned cost model was used to predict mappings for each of the 8 focus pairs. Table 5.1 shows the comparison between the learned and reference mappings using three different metrics: average similarity, nearest neighbor similarity, and percentage of edges that appear in at least one mapping.

The online mapping experiment found a 78% inter-mapping consistency between the participants. This might be considered a gold standard against which page mapping algorithms are measured. Currently, Bricolage achieves a 69% consistency. By this measure, there is room for improvement. However, Bricolage's mappings overlap an average of 78% with their nearest human neighbor, and 88% of the edges generated by Bricolage appear in some human mapping.

This structured prediction approach was motivated by the hypothesis that ancestry and sibling relationships are crucial to predicting human mappings. We tested this hypothesis by training three additional cost models containing different feature subsets: visual terms only, visual and ancestry terms, and visual and sibling terms. Considering only local features yields an average nearest neighbor match of 53%; mapping with local and sibling features yields 67%; mapping with local and ancestry features yields 75%. Accounting for all of these features yields 78%, a result that dominates that of any subset. In short, flexibly preserving structure is crucial to producing good mappings.

5.13 Implementation

Bricolage's page segmentation, mapping, and machine learning libraries are implemented in C++ using the Qt framework, and use Qt's WebKit API in order to

interface directly with a browser engine. Once a cost model has been trained, Bricolage produces mappings between pages in about 1.04 seconds on a 2.55 Ghz Intel Core i7, averaging roughly 0.02 seconds per node.

The corpus pages are archived using the Mozilla Archive File Format and hosted on a server running Apache. For efficiency, page segmentations and associated DOM node features are computed and cached for each page when it is added to the corpus. Each feature has its own dynamic plug-in library, allowing the set of features to be extended with minimal overhead, and mixed and matched at runtime. The Bricolage Collector is written in HTML, Javascript, and CSS. Mapping results are sent to a centralized Ruby on Rails server and stored in a SQLite database.

5.14 Discussion and Future Work

This chapter introduced the Bricolage algorithm for automatically transferring design and content between Web pages. Given the rapid rate at which web technology changes, automatic retargeting offers a scalable solution for keeping existing pages up-to-date. There are many diverse future use cases for Bricolage, including making pages responsive, transforming HTML for screen reader accessibility, and creating low bandwidth versions of designs.

Bricolage's major algorithmic insight was a tunable algorithm for flexible tree matching for capturing the structural relationships between elements, and using an optimization approach to balance local and global concerns. This algorithm may be useful for matching in many domains in which hierarchy is suggestive rather than definitive. The current Bricolage implementation is HTML specific; however, in principle, the retargeting algorithm can be applied to any document with hierarchical structure such as slide presentation and vector graphics files. With richer vision techniques [182], the Bricolage approach might extend to documents and interfaces without accessible structure.

Metric	*Cost Model*	*%*
Average Similarity	c_v c_a and c_s	68.7
	c_v and c_a	65.8
	c_v and c_s	59.1
	c_v alone	44.6
Nearest Neighbor	c_v c_a and c_s	77.7
	c_v and c_a	75.1
	c_v and c_s	67.0
	c_v alone	53.2
Edge Frequency	c_v c_a and c_s	87.9
	c_v and c_a	84.5
	c_v and c_s	81.4
	c_v alone	64.4

Table 5.1: Results of the hold-out cross-validation experiment. Bricolage performs substantially worse without both the ancestry and sibling terms in the cost model.

Chapter 6

Structural Semantics

Researchers have long envisioned a Semantic Web, where unstructured Web content is replaced by documents with rich semantic annotations. Unfortunately, this vision has been hampered by the difficulty of acquiring semantic metadata for Web pages. This chapter introduces a method for automatically "semantifying" structural page elements: using machine learning to train classifiers that can be applied in a post-hoc fashion. We focus on one popular class of semantic identifiers: those concerned with the *structure* — or information architecture — of a page. To determine the set of structural semantics to learn and to collect training data for the learning, we gather a large corpus of labeled page elements from a set of online workers. We discuss the results from this collection and demonstrate that our classifiers learn structural semantics in a general way.

6.1 Introduction

Although Web search engines today offer more than just ranked results (*e.g.,* price comparisons for products, weather forecasts for particular city), most queries in the long tail still involve collecting, organizing and understanding information from multiple pages, which can be difficult and time-consuming [15]. One reason search engines find it difficult to directly answer queries is that Web content is largely *unstructured* [94]. Although Web formats provide rich presentation semantics for

displaying Web data, they typically offer little support for other kinds of automated processing. This desire for flexible reuse of Web information has engendered a vision of a Semantic Web, where documents are annotated in a way that allows machines to "understand" Web content and respond to complex human requests based on their meaning [14].

While information extraction has traditionally targeted textual information, some recent attempts to semantify the Web have focused on page *structure* rather than *content*. In HTML 5, the World Wide Web Consortium added semantic tags (*e.g.* <ARTICLE>, <NAV>, <FIGURE>, <SUMMARY>, etc.) to help developers describe the information architecture of pages [154, 84]. These structural semantics are a small step on the road to a semantic "web of data" [160], aiding applications like search [50], retargeting [111], remixing [31], and user interface enhancement [178].

Relying on Web designers to annotate pages with semantic markup, however, is problematic. Designers, many of whom are primarily concerned with how their Web content is displayed rather than how easily it can be reused, lack strong incentives to invest time and effort augmenting pages with tags that do not produce presentational benefits. As semantic specifications evolve, pages must be continually re-engineered even if their content remains unchanged. Furthermore, there is no universal consensus about the appropriate range and specificity of semantic terms to use. An alternative strategy is to allow end-users to add personal semantics to page data on a case-by-case basis [87, 94], but these manual techniques are difficult to scale to the whole Web.

This thesis presents a different tactic for adding structural semantics to Web pages: learning classifiers for page elements from data. With accurate semantic classifiers, pages could be semantified automatically, in a post-hoc fashion, decoupled from the design and authoring process [186]. To this end, we present a classification method based on support vector machines [42], trained on a large collection of human-labeled page elements and employing a feature space comprised of visual, structural, and render-time page properties.

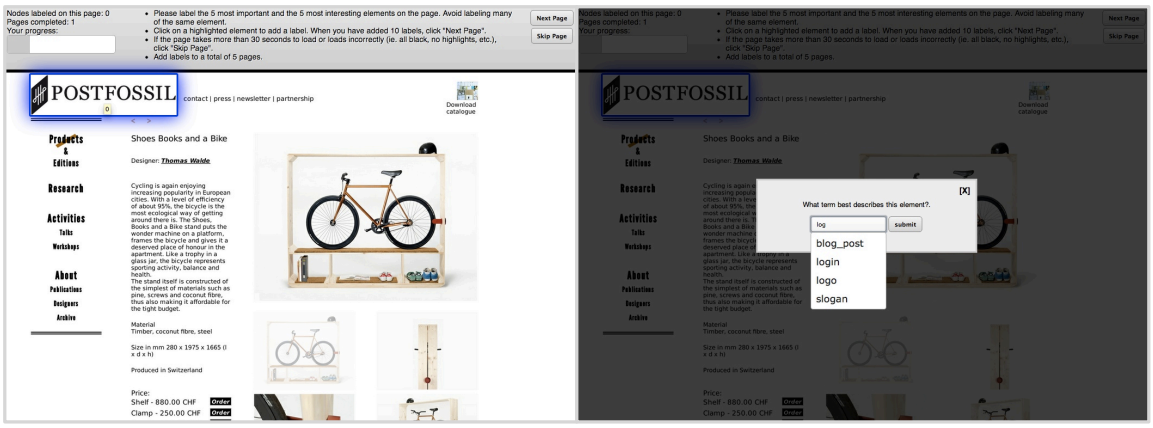


Figure 6.1: The interface used in the label collection study. Page elements are highlighted in blue upon mouseover (left). After clicking on the highlighted element, users enter semantic labels into a textbox (right).

Although some approaches to adding post-hoc semantics are domain-specific and/or make assumptions about the layout of Web documents, we aim to use a general set of semantic terms to describe structural elements across a wide range of pages. As a result, these terms can be applied to any HTML page that can be loaded and displayed in a browser.

To identify the set of structural semantics to learn, we take a crowdsourced approach. When the W3C selected the set of semantic tags to add to HTML 5, they focused on how content *producers* view the information architecture of pages [131]. We, instead, turn our attention to content *consumers* and they way they describe structural semantics. We recruited 400 participants on Amazon's Mechanical Turk [9], collecting more than 21,000 semantic labels over a corpus of over 1400 Web pages. We use these labels to determine the set of classifiers and provide training data for the learning.

The chapter describes the online label collection study and its results, and demonstrates that SVM-based classifiers can produce prediction accuracies as high as 94.7%.

6.2 Crowdsourced Label Collection

To drive the development of semantic classifiers, we collected a set of labeled page elements in an online study. We recruited 400 US-based workers from Amazon's Mechanical Turk to apply more than 21,000 labels across nearly 1500 Web pages. Every participant applied semantic labels to at least ten elements on each of five pages. The pages used in the study were drawn from the Webzeitgeist design repository [110], which provides visual segmentations and page features for more than 100,000 Web pages.

The label collection process comprised two phases: a focused phase, and a broad phase. In the focused phase, we hand-selected fifty pages from ten popular site genres that were adapted from [54]: e-commerce, news, community, informational, corporate, small company, blog, personal, Web service, and Web resource. A hundred participants each labeled ten of these pages, producing 6351 labels and ensuring that many page elements were labeled by more than one person. In the broad phase, 300 users each labeled five pages chosen randomly from the corpus, producing 15,644 labels.

6.2.1 Procedure

First, the Mechanical Turk interface redirected participants to a tutorial on the labeling interface. The instructions directed users to apply semantic labels to the five most *important* and the five most *interesting* elements on the page. Participants were also instructed to avoid labeling many elements of the same type, to encourage diversity in the data set.

Given our focus on structural semantics, workers were told to choose labels that described the element's *role* in the information architecture of the the page rather than its *content*. For instance, a picture of a silverware set on a shopping page should be labeled PRODUCT_IMAGE instead of SILVERWARE. Workers were also instructed to chose the most specific applicable label, eschewing generalities such as TEXT. To proceed to the labeling task, users were shown a few basic examples of appropriate labels, and required to correctly apply one label to a sample element.

The labeling interface presents workers with a screenshot of a Web page (Figure 6.1). When a participant hovers the mouse over part of the page, the corresponding element in the page's visual segmentation is highlighted. Clicking on an element allows the user to enter a text label for it, which can be edited later by clicking on the element again. When typing a label, users are prompted with a drop-down list of autocompleted suggestions — sourced from a small pilot labeling study — which they may use or ignore. Workers apply at least ten elements to each page before moving on to the next; after five pages have been labeled, the interface provides an identifier to the worker to verify the task's completion.

6.2.2 Results

Participants produced 21,995 labels across 16,753 distinct elements in 1490 Web pages. There were 2657 distinct labels in total, 716 of which occurred more than once, and 629 of which were applied by more than one user. Each participant used 23.6 distinct labels on average ($\min = 3$, $\max = 76$, $\sigma = 9.7$). Excluding labels from the autocomplete list, participants generated an average of nine original label names ($\min = 0$, $\max = 60$, $\sigma = 10$).

In addition to general characteristics of the resulting dataset, the following sections provide two statistical analyses for better understanding the labels that participants produced. First, we examined label *co-occurrence*, to determine which labels different workers commonly assign to the same page elements. Second, we examined the *spatial distribution* of labels to determine where certain kinds of page elements commonly appear on a page.

Characteristics of Dataset

The collected labels cover a wide range of concepts, with tags as general as IMAGE and as specific as COPYRIGHT. Workers tagged some elements common to most Web pages, such as NAVIGATION, and others that are highly domain specific, such as PRODUCT_IMAGE. The ten most common labels were NAVIGATION_ELEMENT, NAVIGATION_BAR, LOGO, SEARCH, SOCIAL_MEDIA, ADVERTISEMENT, ARTICLE_TITLE,

Figure 6.2: A tag cloud of the 110 most common semantic labels, sized to show relative frequency. The tags highlighted in red have direct analogues in HTML 5.

MAIN_CONTENT, BLOG_POST, and AND CONTACT_LINK, with frequencies ranging from 1772 to 436. The mean label frequency was 8.3 ($\min = 1$, $\max = 1772$, $\sigma = 65.8$).

Figure 6.2 shows the labels' relative frequencies in a tag cloud. Labels which have direct analogues to any one of the 106 tags in HTML 5 are highlighted in red. The 17 HTML tags to which these labels correspond include <A>, <ADDRESS>, <ARTICLE>, <BLOCKQUOTE>, <BODY>, <CAPTION>, <FIGCAPTION>, <FOOTER>, <FORM>, <H1-H6>, <HEADER>, <HGROUP>, <IMG>, <INPUT>, <NAV>, <TIME>, and <VIDEO>. At a high level, the relatively small overlap between our crowdsourced labels and the set of available HTML tags illustrates the difficulties of developing a semantic ontology that is sufficiently expressive and complete.

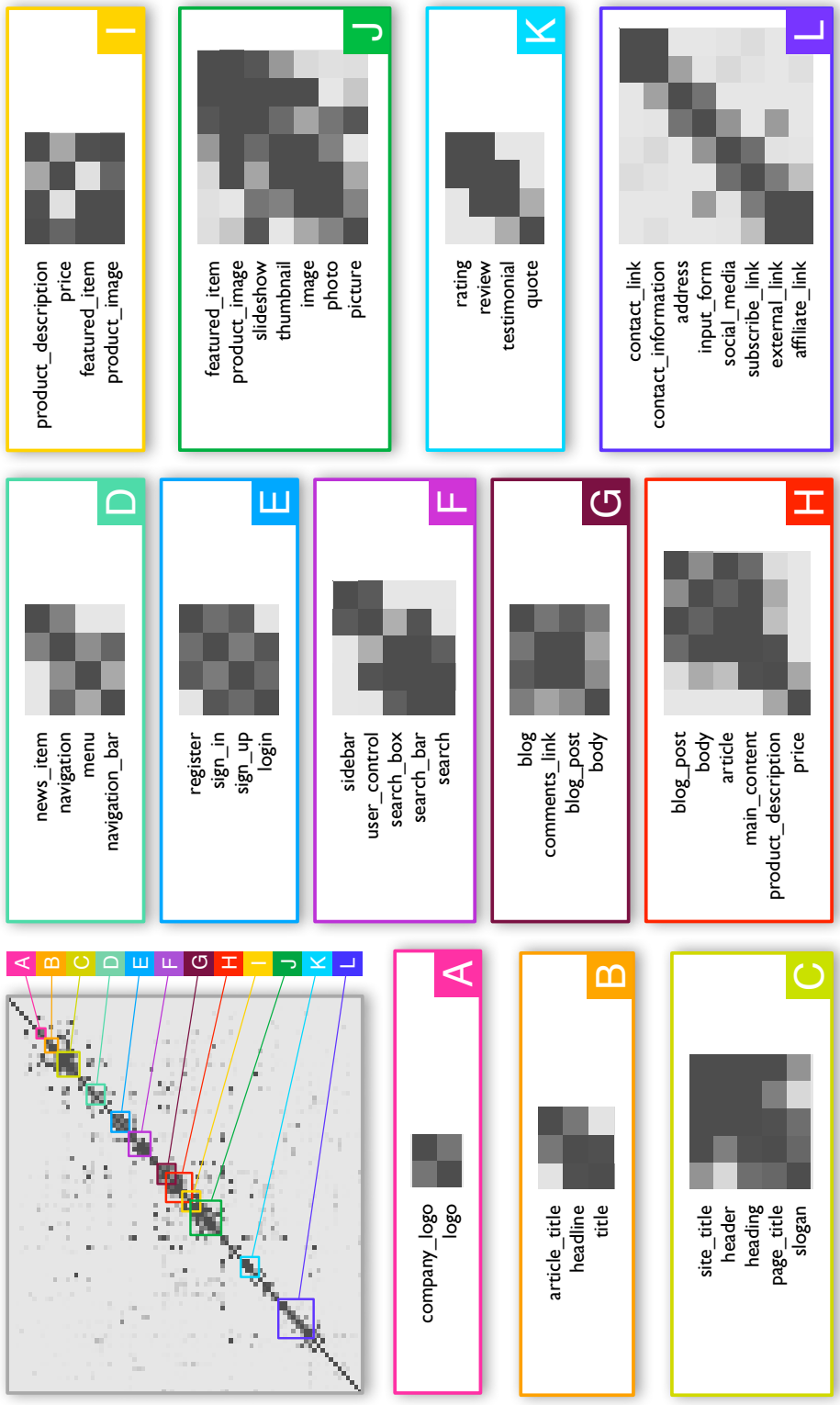


Figure 6.3: The label co-occurrence matrix, seriated via ARSA [26]. Overlapping sections of the matrix are highlighted and magnified to show labels that frequently occur together.

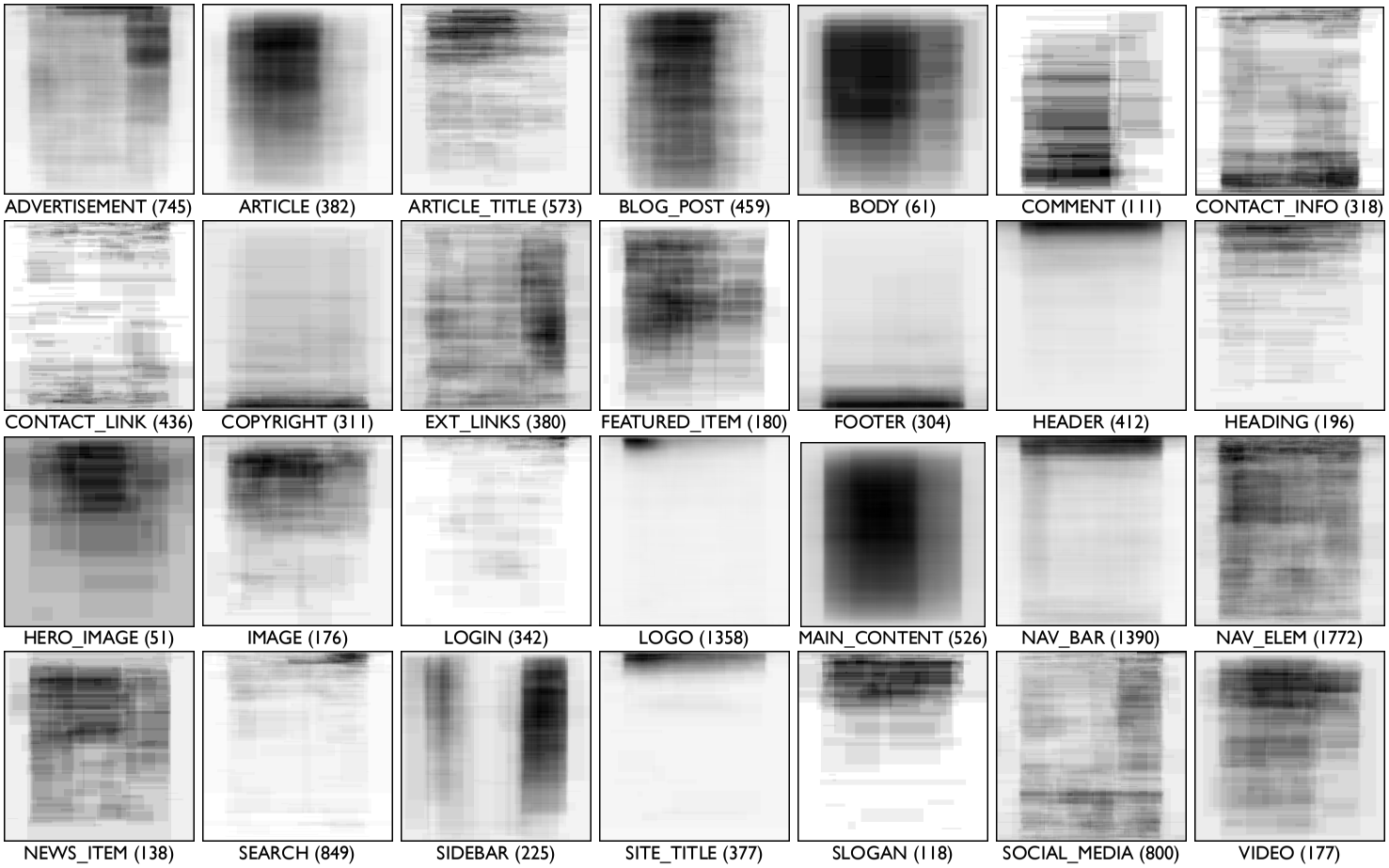


Figure 6.4: Spatial probability distributions for 28 labels, along with the number of elements used to construct each distribution. Many elements exhibit strong spatial correlations.

Label Co-occurrence

The dataset revealed that not all people will assign the same semantic label to a given page element. In addition, some workers may use different descriptors to label the same concept.

To more thoroughly understand how labels relate to one another, we created a co-occurrence matrix for the 85 most-frequent labels, each of which was used twenty or more times. We form an 85×85 symmetric matrix, where the value at (i, j) is the number of times that tag i and tag j were used to label the same page element, normalized by the total number of uses of i and j. Then, the matrix is reordered using Anti-Robinson seriation to form clusters of co-occurring labels along the diagonal [26].

Figure 6.3 shows the resulting matrix, with portions of the diagonal magnified to show co-occurring labels. The cell opacities represent the degree of co-occurrence between the corresponding labels: darker cells indicate more co-occurrences while lighter cells indicate fewer. A number of clusters with labels like RATING and REVIEW (panel **E**) simply point out elements that are closely related. Some show workers using different words to describe the same concept, like COMPANY_LOGO and LOGO (panel **A**). Other groupings reflect a lack of a clear consensus on the role of elements such as FEATURED_ITEM and PRODUCT_IMAGE (panel **I**). Labels like SITE_TITLE and HEADER (panel **C**) describe the same general structure with varying levels of specificity.

Overall the distinct clusters illustrate where users agreed upon and were consistent with their their semantic vocabulary. The heavy concentration of high-opacity cells along the diagonal indicates strong clusters of co-occurrence.

Spatial Distributions

Another useful way to gain insight about the labels participants produced is to examine the spatial distributions of their corresponding page elements. For a given label, we identify the set of page elements to which the label was assigned, and obtain the bounding rectangle for each one from the page's DOM tree. We rescale

these rectangles to the range $[0, 1] \times [0, 1]$ to make the coordinates comparable between pages, and rasterize them into a floating-point accumulation buffer. Normalizing the resultant image so that its pixel values sum to one approximates the two-dimensional spatial probability distribution of the tag. The value of any given point in the image is the probability of the label appearing in that position on a page.

Figure 6.4 shows spatial distributions for 28 popular labels. While some distributions useful but unsurprising (HEADER tags appear almost universally at the top of pages), others give more insight into the structure of Web pages. Note, for instance the strong concentration of LOGIN and SEARCH elements in the upper right corner of pages, the bimodal distribution of ADVERTISEMENT elements between sidebar and header, and the high frequency of EXTERNAL_LINKS along and increasing toward the middle of the right sidebar. Taken together, the strong spatial correlations that many of the collected tags exhibit provide a visual justification for learning classifiers for structural semantics.

6.3 Learning Structural Semantic Classifiers

To evaluate the feasibility of learning structural semantics from data, we trained binary SVM classifiers for the study's 40 most frequent labels. To determine the prediction accuracy of the classifiers, we ran a hold-out test on labeled pages. Finally, we used the learned classifiers to identify and rank semantic elements in a large dataset of pages.

6.3.1 Training

For each distinct label, we constructed a *training* set and a *test* set of page elements. The training set consisted of 80% of the page elements to which the

label had been applied (the positive examples), and twice that number of randomly selected page elements to which other labels were applied (the negative elements). The test set consisted of the remaining 20% of positively labeled page elements, and twice that number again of randomly selected negative elements.

To drive the learning, each page element was associated with a 1,679-dimensional feature vector provided by the Webzeitgeist repository. These features were drawn from three categories: render-time HTML and CSS properties computed by the DOM ($N = 691$), GIST descriptors computed on elements' rendered images (four scales and five orientations per scale on a 4×4 grid; $N = 960$) [139], and simple structural and computer vision properties provided by Webzeitgeist ($N = 28$).

We trained three regularized support vector classification SVMs for each label: one with DOM features, one with GIST features, and a third using all the features together. We used LIBSVM to perform the training [32], with radial basis kernels and $\gamma = \frac{1}{1679}$. Once a classifier is trained, it can be applied to a page element in under 1μs.

6.3.2 Prediction Accuracy Results

The prediction training and test accuracies for each classifier and data model are shown in the inset table, where the first column represents the number of positive examples in the training set for the corresponding label. Test accuracies ranged from 54.9% for COMMENT to 94.7% for ENTIRE_PAGE. This variation can be attribute to a number of reasons. First, some elements are structurally more consistent and/or prominent than others, for example FOOTER elements generally occupy a significant space at the bottom of a page, while DATE can be a variable-width text node that occurs anywhere on the page. While elements such as LOGO are clearly defined, others such as FEATURED_ITEM may exhibit more variation in the types of elements they refer to. Number of examples and structural dependencies may also affect prediction accuracy.

The average test accuracy for the DOM, GIST, and ALL models were 74.6%, 71.7%, and 76.6% respectively. The combined model equaled or outperformed the DOM- and GIST-alone models for all but seven of the forty labels; examining the training accuracy for those nine shows that this discrepancy is mostly attributable to overfitting. While these results are far from perfect, all of the classifiers do better than random, and most substantially so.

6.3.3 Identifying Structural Elements

To show the learned classifiers in action, we applied twelve of them across a database of 500k page elements spanning 3000 pages. We proceeded to rank the results in order of decreasing probabilities, which were obtained via the method described in [187]. A few representative results for each classifier are shown in Figure 6.5; page elements that appeared to be mis-classified are marked with a red border.

These examples offer some insight into the performance of the classifiers. Most of the highly-ranked elements are classified correctly, despite their diverse contexts and compositions. Given that these classifiers are trained only on visual and structural data, their expressive power provides support for the notion that structural semantics can be learned without requiring more complex content-based semantics (see, for instance, SLOGAN). Many of the errant classifications are subtle, and might plausibly confuse a human worker: see for instance ARTICLE_TITLE, which classifies several titles that are not, strictly speaking, associated with articles; and NAVIGATION_BAR, which identifies page elements filled with links directing users to *other* sites.

6.4 Incentives For Semantics

This chapter demonstrates how post-hoc structural semantic classifiers can bootstrap the semantic Web. These classifiers allow application developers to build technology operating under the assumption that most Web pages have semantics;

		DOM		GIST		ALL	
Label	#	Train	Test	Train	Test	Train	**Test**
ENTIRE_PAGE	74	91.9	**94.7**	86.9	75.4	94.6	**94.7**
SEARCH	551	88.7	88.2	82.8	84.5	91.8	**91.5**
FOOTER	186	83.0	78.0	74.6	75.9	90.0	**89.4**
IMAGE	169	84.0	81.0	79.7	79.4	86.4	**88.9**
SIDEBAR	133	86.0	84.8	82.7	84.8	86.7	**87.9**
COPYRIGHT	206	86.1	82.1	75.7	76.3	88.4	**87.8**
NAVIGATION_BAR	901	80.7	83.4	74.6	72.7	86.5	**87.4**
LOGO	770	80.7	84.0	77.9	77.6	87.0	**87.3**
ARTICLE_TITLE	373	84.2	82.8	80.3	82.4	86.7	**87.1**
MAIN_CONTENT	350	82.8	82.8	78.9	80.1	83.0	**83.1**
PRODUCT_IMAGE	65	79.5	79.2	77.4	75.0	85.1	**81.3**
THUMBNAIL	83	84.7	76.2	79.5	73.0	86.7	**81.0**
HEADING	134	79.4	**87.3**	74.9	67.6	77.9	80.4
ARTICLE	237	75.2	72.9	81.4	79.7	85.9	**80.2**
LOGIN	222	80.8	73.9	77.8	74.5	84.5	**78.8**
ADVERTISEMENT	487	79.2	76.2	75.2	72.4	85.6	**77.9**
NAV_ELEMENT	1138	75.9	75.4	72.6	73.5	78.6	**77.8**
VIDEO	107	74.8	71.6	81.9	72.8	81.9	**77.8**
BLOG_POST	259	75.5	72.3	73.6	73.8	81.9	**77.4**
HEADER	265	78.5	**77.3**	69.2	67.2	80.8	**77.3**
CONTACT_LINK	278	76.7	75.2	74.8	71.0	80.7	**76.2**
SOCIAL_MEDIA	514	75.0	72.7	76.5	70.1	82.2	**75.0**
SITE_TITLE	272	75.9	74.0	76.1	70.6	81.3	**75.0**
DATE	91	79.1	69.6	78.0	**73.9**	79.5	**73.9**
IMAGE_GALLERY	94	76.6	**76.8**	75.2	71.0	82.3	73.9
RECOMM_LINKS	137	69.8	66.7	68.9	66.7	76.4	**73.5**
CONTACT_INFO	183	77.2	67.4	69.9	68.1	79.6	**73.2**
LANG_SELECT	70	77.1	68.6	75.2	70.6	83.3	**72.5**
PROD_DESC	232	77.2	71.8	74.7	68.4	77.0	**72.4**
SLOGAN	79	69.6	63.3	70.5	**70.0**	76.4	**70.0**
AUTHOR	94	72.7	62.3	67.4	68.1	70.6	**69.6**
SUBSCRIBE_LINK	158	68.1	67.5	69.6	67.5	71.3	**68.4**
FEATURED_ITEM	107	71.3	**71.6**	73.5	58.0	76.9	66.7
COMMENTS_LINK	93	80.6	**71.0**	67.7	66.7	70.3	66.7
AFFILIATE_LINK	78	66.7	**66.7**	66.7	**66.7**	66.7	**66.7**
EXTERNAL_LINKS	270	67.2	**66.2**	68.9	**66.2**	73.8	**66.2**
SIGN_UP	134	74.9	65.7	69.2	**66.7**	73.9	65.7
NEWS_ITEM	121	68.9	**65.6**	71.1	64.4	72.2	**65.6**
DOWNLOAD_LINK	77	73.6	63.2	66.7	**66.7**	74.5	64.9
COMMENT	70	77.6	**76.5**	75.7	56.9	78.6	54.9

Table 6.1: The prediction training and test errors for each of our learned classifiers using the DOM, GIST, and ALL feature models.

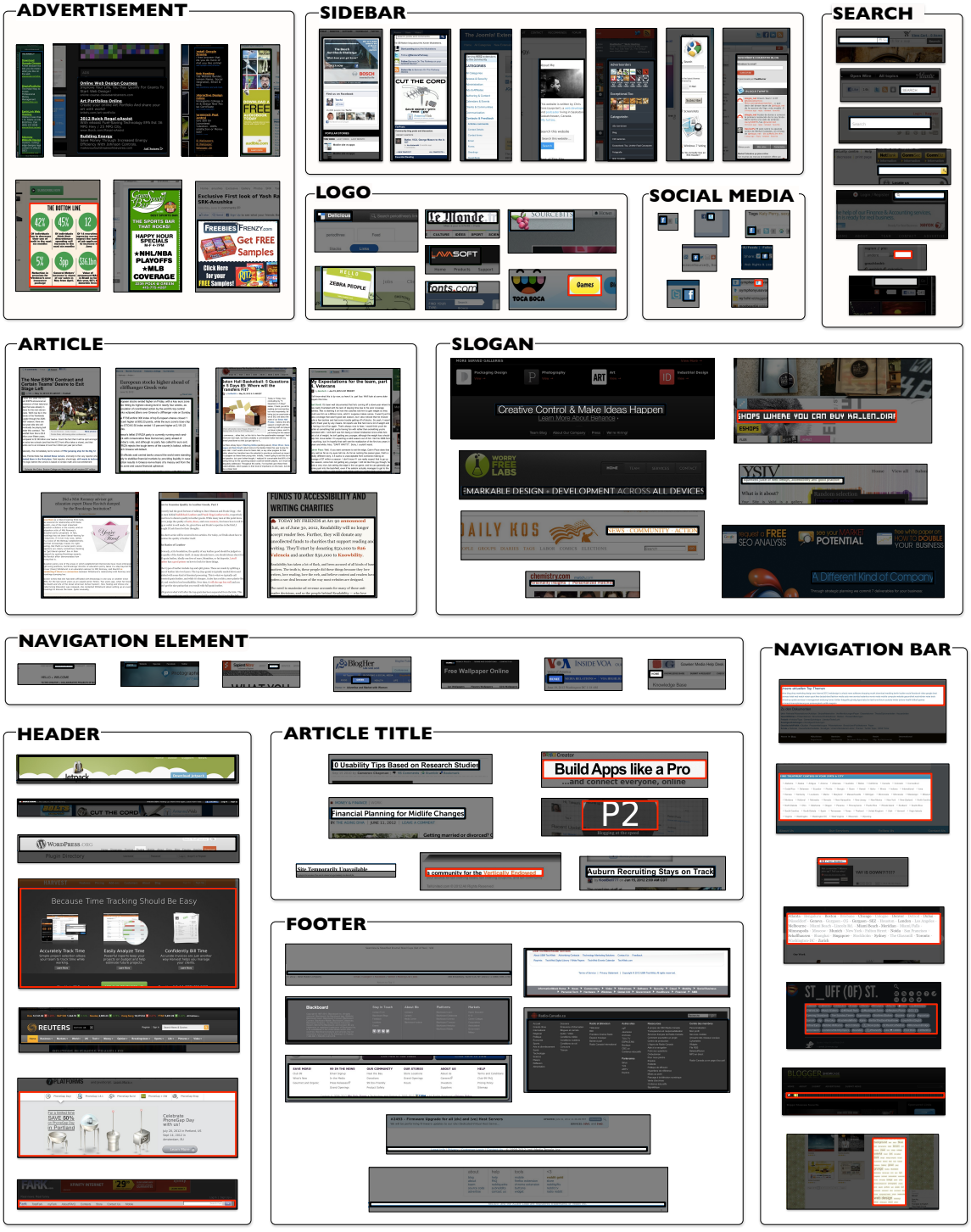


Figure 6.5: The seven highest-ranked results in our database of 500k pages for each of twelve classifiers learned by our method. Page elements were ranked by probability estimate, and a maximum of one node per page is displayed. Elements which were classified incorrectly are highlighted in red.

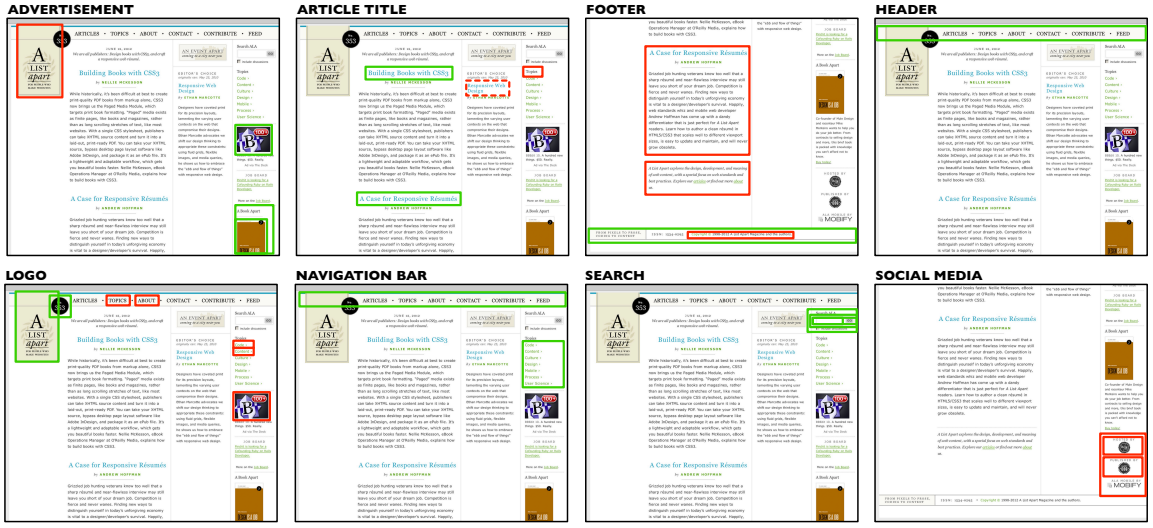


Figure 6.6: Eight learned classifiers used to identify structural semantic elements on a page with 67 DOM elements. Correct classifications are shown in green, false positives in solid red, and false negatives in dashed red.

as more semantic tools become available, more content producers will have incentives for adding semantics to their markup. On the flip side, designers themselves can use the classifiers to bootstrap semantics on the pages that they create or have already created. Others systems incentivize people to adopt semantics in alternative ways.

6.4.1 Personal Content Management

Some systems provide content consumers with incentives for manually adding personal semantics to pages they visit often [87, 94, 51, 50]. These systems usually leverage pattern matching and existing page semantics to semi-automate the labeling process, and afford users new ways of interacting with the annotated data in the future [52]. For example, by right-clicking on a "person" record, an end-user can choose to email the person, and have a compose window pop-up with the right email address already filled in. Moreover, these personal semantic records can be shared so that each individual can benefit from work done by others in the community.

6.4.2 Design Reuse

Structural semantic concepts facilitate design reuse in CSS frameworks such as Twitter's Bootstrap [22] and Zurb's Foundation [60]. These frameworks target developers who can write code, but do not want to spend time mucking with CSS files to get the design details right. To build a nice looking Web page with a responsive layout, developers simply have to write HTML using the Bootstrap or Foundation specified classes. Most of these classes correspond to structural semantic concepts (*e.g.,* "navbar", "breadcrumbs", "pagination") and have styles and behaviors predefined for them in stylesheets and javascript files provided by the framework. By doing a lot of the design work for free, these frameworks incentivize content producers to use structural semantic annotations in their markup.

Similarly, Benson *et al.* propose design reuse as the catalyst for getting content producers to adopt more semantic class names [13]. Instead of the fixed set of class names supported by CSS frameworks, they propose that CSS schemas will emerge in a "grassroots fashion" led by Web designers who want to their code to be reused by the masses.

6.5 Discussion and Future Work

This chapter introduced a technique for adding post-hoc structural semantics to the Web, demonstrating that a relatively simple machine learning technique trained on a corpus of human annotations and design-based features can identify semantic elements in pages. Many interesting avenues for future work remain.

6.5.1 Better Classifiers

First, it is important to note that our classifiers cannot necessarily be used to enable one-click annotation of pages in their current form. Pages in our training set averaged 1380 DOM nodes per page; with this many elements, even a 99.9% per-classifier accuracy rate from a classifiers trained on a perfectly labeled set of nodes

would yield several misclassified nodes on every page. These results would be acceptable for some real-world applications that can tolerate false positives (such as Web search), however the success rate would be inadequate for those requiring near-perfect accuracy (Figure 6.6).

Several possibilities for improving the learning come to mind. Using our classifiers to bootstrap an online learning process is one obvious approach, likely to significantly reduce overfitting and greatly simplify the acquisition of additional training data. Adding more sophisticated structural and computer vision features might is another possibility: estimates of foreground area, for instance, might prove useful in recognizing logos, while structural features like "number of links to external domains" could improve the classification of navigation bars.

Another promising approach is to turn to machine learning methods that make better use of page structure. Currently, the classification algorithm assumes that labels are independent between elements, a largely faulty assumption. Structured SVMs could be used to predict labels for the entire page as a whole [179]. Deep learning techniques, like those based on recursive neural networks — might allow the development of a more structurally-sensitive feature space. These methods would enable easier classification of elements whose semantic function is highly dependent on its relation to other elements in the page hierarchy [166].

6.5.2 Structural Semantic Applications

Structural semantic classifiers have the potential to increase the utility and accuracy of many of the design mining applications presented in this dissertation. Many of the queries shown in Chapter 4 involved structural semantic terms (*e.g.,* "find *headers* that take up more than 20 percent of the page's area"). Currently, Webzeitgeist can only use HTML5 tags and microformatting when they are available to perform structural semantic searches: automatic classifiers could be used to apply these queries to the data set as a whole. Similarly, when Bricolage executes its page mapping algorithm (Chapter 5), it implicitly assigns structural semantic labels to page regions during matching. Explicit structural semantics would make Bricolage

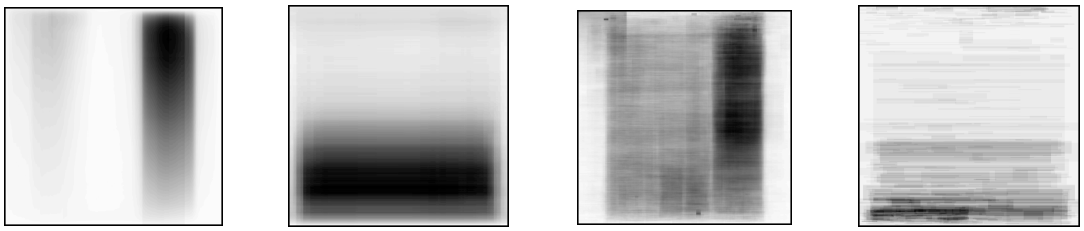


Figure 6.7: Spatial probability distributions for elements labeled “sidebar” (left) and “twitter” (right) in desktop and mobile settings.

faster by reducing the matching algorithm’s search space, and increase the overall mapping accuracy.

In fact, structural semantics are ofen the cornerstone of design reuse and remixing on the Web: template-based Web authoring tools like Google’s Blogger [20] (Figure 6.8) and CSS frameworks like Twitter Bootstrap [22] all expose structural semantic concepts as end-user knobs. In the future, structural semantic classifiers could be used to automatically generate page designs for different layouts (Figure 6.7) or personalize how people view information on their devices [21].

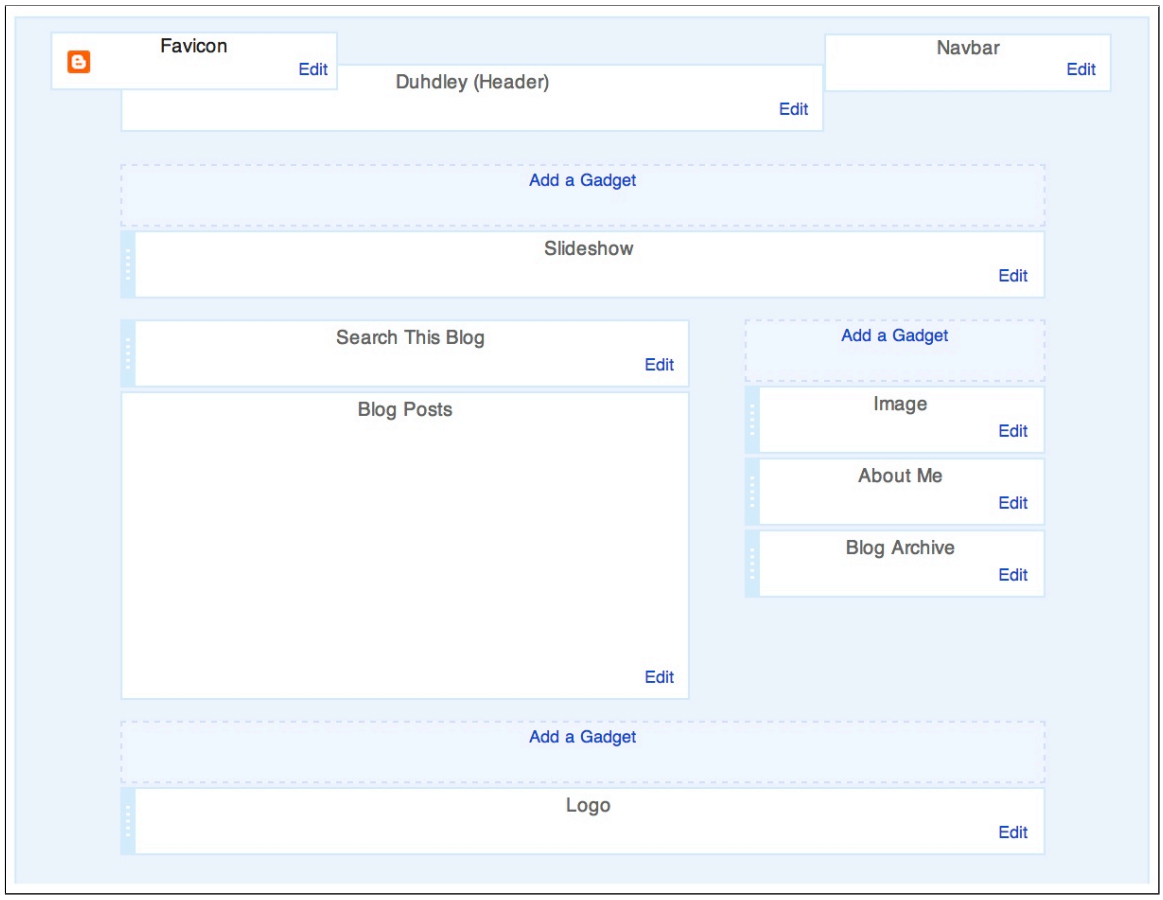


Figure 6.8: Google's Blogger allows users to add and drag structural semantic components to configure page layout [20].

Chapter 7

Conclusion

This thesis demonstrates for the first time the value of large-scale mining of design data, and offers a new class of data-driven problem-solving techniques to the design community. In particular, it makes the following set of contributions:

- Design mining for the Web: principles for indexing, analyzing, and adapting Web design data
- The Webzeitgeist platform for design mining, which comprises a repository of over 100,000 Web pages and 100 million design elements
- The Bento page segmentation algorithm for canonicalizing the DOM and generating structured, visual representations of pages
- A Design Query Language (DQL) for building design mining applications
- A design-based search engine that allows users to search Web designs at multiple scales
- The Bricolage algorithm for automatically transferring design and content between Web pages
- An tunable algorithm for flexible tree matching which can be used to computing mappings in domains where hierarchy is suggestive rather than definitive

- A set of classifiers for predicting the structural role of Web page elements from design data

7.1 The Future of Design Mining

While this thesis showcases several concrete design interactions, we imagine that the applications that eventually arise from design mining will greatly outstrip our power to predict them. There are a number of avenues for future work.

7.1.1 Scaling Up

For design mining applications to be useful to designers, they must function at a scale commensurate with the size of the Web. Although Webzeitgeist facilitates design analysis at scales an order of magnitude larger than prior work, recent breakthroughs in unsupervised learning have come from training models with billions of parameters on datasets with millions of examples [114, 47, 19]. In a domain as diverse as the Web, it is not hard to imagine that such models might be approprate: after all, in the 100,000 page repository crawled by Webzeitgeist, only 68 pages with horizontal layouts were found, accounting for slightly less than 0.07% of the repository. How many more pages must we crawl to compute robust statistics over horizontal page designs? How much data do we need to infer latent design patterns in the space of Web designs [19], or to understand analytically what makes eBay look like eBay and Google look like Google [47]?

7.1.2 Leveraging Structure

One way that design is distinct from other data-rich domains such as text or natural images is that the representations designers employ are typically annotated with explicit *structure*. On the Web, this structure comes in the form of DOM trees; in geometric modeling, for instance, this structure is found in scene graphs used to define relationships between components [10]. As we saw in Chapter 5, taking

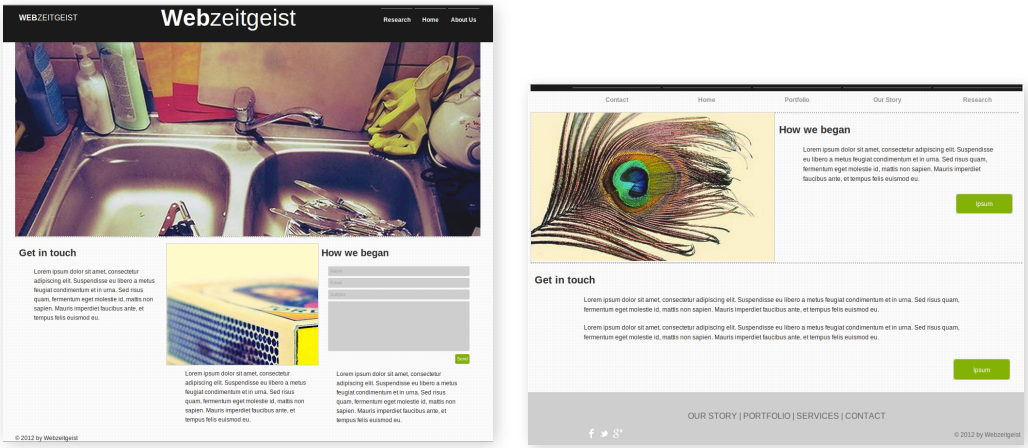


Figure 7.1: Web design tasks formulated as probabilistic inference over an induced grammar of page designs. Left: the most likely page with a logo, three navigation elements, a hero-image, and a three-column layout. Right: the most likely page with three navigation elements, a hero-image, and a two-column layout.

structure into account can drastically improve the performance of machine learning applications and mathematical models. We hypothesize that taking full advantage of structure is the key to enabling many new and useful design interactions.

Duplicating data in a graph database [136] may make it easier to formulate complex queries that express hierarchical constraints (*e.g.,* "find all the nodes whose children are all <IMAGE> elements"). Taking into account the interdependencies between structural semantic concepts (*e.g.,* navigation elements are usually children of navigation bars) via structured SVMs [179], may yield more robust semantic classifiers. Using recursive neural networks [165] to compute fixed-dimensional, structurally-sensitive representations [166] over design elements may allow more efficient design queries.

7.1.3 Design as Inference

Design patterns codify best practices into a collection of formal rules for designers, setting out principles of composition, describing useful idioms, and summarizing

common aesthetic sensibilities. While such guidelines can be invaluable to designers, they are also difficult to operationalize, and must be painstakingly formulated and compiled by experts [7, 67, 23, 54].

A more attractive proposition is to learn patterns directly from examples, and encapsulate them in a representation that can be accessed algorithmically [88]. In a recent paper [175], we cast this problem as grammar induction, bringing techniques from natural language processing and structured concept learning to design. Given a corpus of hand-labeled Web designs, we induce a probabilistic formal grammar over these exemplars. Once learned, this grammar gives a design pattern in a human-readable form that can be used to synthesize novel designs and verify extant constructions.

Inducing Web design patterns from a corpus of extant pages enables a new class of design interactions which exploit the rich mathematical structure of generative probabilistic models [112]. In particular, we have demonstrated how common design tasks can be formulated as probabilistic inference problems and solved via stochastic optimization. For instance, by defining a smooth function that scores page designs based on how closely they conform to a given specification, component-based Web design can be cast as a MAP estimation problem, and solved with Markov chain Monte Carlo methods (Figure 7.1).

Component-based Web design is not the only interaction technique that can be profitably cast as probabilistic inference. In fact, it seems likely that many design applications can be enabled with similar machinery. Developing tools for sketch-based Web design, adding autocomplete capabilities to direct manipulation editors, and improving existing designs via stochastic optimization are just a few promising directions.

7.1.4 Expanding to New Domains

Webzeitgeist currently harvests all the design components that contribute to the visual appearance of a Web page, but a page's design is not just *visual*: it also includes a set of interactions and behaviors that are triggered by user inputs such as

key presses and mouse actions. In the future, we could mine design *interactions* — in addition to visual attributes — during a crawl. In order to detect interaction on pages, changes in the DOM need to be registered when user inputs are performed; Mesbah *et al.*'s crawler for AJAX-based Web applications could provide a starting point for this kind of interaction mining [128].

Furthermore, as more and more creative work is done digitally and shared in the cloud, we hope that the lessons we learn and the techniques we develop to answer questions in Web design can eventually be transferred to other domains. The Web has many more datasets to offer beyond Web pages themselves, and rich repositories of 3D models [3], interior decor [90], and fashion [145] already exist. Layering crowdsourced annotations or using techniques from computer vision and computer graphics can provide the same types of rich metadata and structure in these domains that are naturally present in Web design [35, 56, 12, 16].

7.1.5 Design and Creativity Science

Fundamentally, what Webzeitgeist affords us is the ability to empirically study design at scale. In the long run, we hope that drawing analogy to the way we process and understand information on the Web will allow us to develop a similarly deep grasp design, and to answer questions like: What makes designs usable [121]? What makes them beautiful [191, 121, 151]? What aspects of design are durable or fashionable [5, 176]? Which parts are driven by evolution, or constructed by culture [150]? How closely tied are the different aspects of design to our cognitive, perceptual, and motor abilities [66, 58]? How can Google use design mining to update their rankings of site quality and trustworthiness [122, 142]?

Similarly, while work in cognitive science has studied the impact of examples on creativity [73, 164, 127, 109], few experiments have considered cognitive priming in light of the modern Web or using modern computational tools [74]. Given the scale and diversity of examples available on the Web, are psychological effects like conformity and fixation as relevant today as paradox of choice? How can we help users transfer design concepts from one creative domain to another [89]? Can we

build and validate computational models of human creativity that help us understand and enhance design?

7.2 Design Mining on the Web

For more information about the material described in this thesis — including links to source code and software — please visit `http://hci.stanford.edu/research/webzeitgeist`.

Bibliography

[1] *23 Examples of Flat Web Design*. Web Design Ledger. 2013. URL: http://webdesignledger.com/inspiration/23-examples-of-flat-web-design.

[2] *33 Examples of the Flat Web Design Trend*. Vandelay Design. 2013. URL: http://vandelaydesign.com/blog/galleries/flat-web-design/.

[3] *3D Warehouse*. Google, Inc. 2013. URL: http://sketchup.google.com/3dwarehouse/.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. "The Lorel query language for semistructured data". In: *International journal on digital libraries* 1.1 (1997), pp. 68–88.

[5] E. Adar, M. Dontcheva, J. Fogarty, and D. S. Weld. "Zoetrope: interacting with the ephemeral web". In: *Proc. UIST*. 2008, pp. 239–248.

[6] E. Adar, J. Teevan, and S. T. Dumais. "Resonance on the web: web dynamics and revisitation patterns". In: *Proc. CHI*. 2009, pp. 1381–1390.

[7] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[8] J. Alpert and N. Hajaj. *We knew the web was big...* 2008. URL: http://goo.gl/RtmG.

[9] *Amazon Mechanical Turk*. Amazon.com, Inc. 2010. URL: http://www.mturk.com/.

[10] E. Angel and D. Shreiner. *Interactive Computer Graphics*. 6th ed. Addison-Wesley, 2011.

[11] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. “Constraint cascading style sheets for the Web”. In: *Proc. UIST*. 1999, pp. 73–82.

[12] S. Bell, P. Upchurch, N. Snavely, and K. Bala. “OpenSurfaces: a richly annotated catalog of surface appearance”. In: *Proc. SIGGRAPH* (2013).

[13] E. O. Benson and D. R. Karger. “Cascading tree sheets and recombinant HTML: better encapsulation and retargeting of web content”. In: *Proc. WWW*. 2013, pp. 107–118.

[14] T. Berners-Lee, J. Hendler, and O. Lassila. “The semantic web”. In: *Scientific American* (May 2001), pp. 35–43.

[15] M. S. Bernstein, J. Teevan, S. Dumais, D. Liebling, and E. Horvitz. “Direct answers for search queries in the long tail”. In: *Proc. CHI*. 2012, pp. 237–246.

[16] F. Berthouzoz, A. Garg, D. M. Kaufman, E. Grinspun, and M. Agrawala. “Parsing sewing patterns into 3D garments”. In: *Proc. SIGGRAPH* (2013).

[17] J. P. Bigham, R. S. Kaminsky, and J. Nichols. “Mining web interactions to automatically create mash-ups”. In: *Proc. UIST*. 2009, pp. 203–212.

[18] P. Bille. “A survey on tree edit distance and related problems”. In: *Theoretical Computer Science* 337 (2005), pp. 217–239.

[19] D. M. Blei, A. Y. Ng, and M. I. Jordan. “Latent dirichlet allocation”. In: *The Journal of Machine Learning Research* (2003).

[20] *Blogger*. Google, Inc. 2013. URL: `http://www.blogger.com/`.

[21] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. “Automation and customization of rendered web pages”. In: *Proc. UIST*. 2005.

[22] *Bootstrap*. Twitter Inc. 2013. URL: `http://getbootstrap.com/`.

[23] J. Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, 2001.

[24] A. Borning, R. K.-H. Lin, and K. Marriott. “Constraint-based document layout for the Web”. In: *Multimedia Systems* 8.3 (2000), pp. 177–189.

[25] S. Brin and L. Page. “The anatomy of a large-scale hypertextual Web search engine”. In: *Computer networks and ISDN systems* 30.1 (1998), pp. 107–117.

[26] M. J. Brusco and S. Stahl. “An algorithm for extracting maximum cardinality subsets with perfect dominance or anti-Robinson structures”. In: *British journal of mathematical and statistical psychology* 60.2 (2007), pp. 377–393.

[27] V. Bush. “As we may think”. In: *The Atlantic Magazine* (1945), pp. 101–108.

[28] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. *VIPS: a Vision-based Page Segmentation Algorithm*. Tech. rep. MSR-TR-2003-79. Microsoft, 2003.

[29] *Cascading style sheets, level 2*. CSS Working Group. May 1998. URL: `http://www.w3.org/TR/2008/REC-CSS2-20080411/`.

[30] D. Chakrabarti, R. Kumar, and K. Punera. “A graph-theoretic approach to Webpage segmentation”. In: *Proc. WWW*. 2008, pp. 377–386.

[31] B. Chan, L. Wu, J. Talbot, M. Cammarano, and P. Hanrahan. “Vispedia: interactive visual exploration of Wikipedia data via search-based integration”. In: *Proc. InfoVis* 14 (6 2008), pp. 1213–1220.

[32] C.-C. Chang and C.-J. Lin. “LIBSVM: A library for support vector machines”. In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011), 27:1–27:27.

[33] K. S.-P. Chang and B. A. Myers. “WebCrystal: understanding and reusing examples in web authoring”. In: *Proc. CHI*. 2012, pp. 3205–3214.

[34] S. Chaudhuri, E. Kalogerakis, S. Giguere, and T. Funkhouser. “AttribIt: Content Creation with Semantic Attributes”. In: *Proc. UIST*. 2013.

[35] S. Chaudhuri, E. Kalogerakis, L. Guibas, and V. Koltun. “Probabilistic Reasoning for Assembly-Based 3D Modeling”. In: *Proc. SIGGRAPH*. 2011.

[36] S. S. Chawathe and H. Garcia-Molina. “Meaningful Change Detection in Structured Data”. In: *Proc. SIGMOD*. ACM, 1997, pp. 26–37.

[37] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. “The TSIMMIS Project: Integration of Heterogenous Information Sources”. In: *Proc. IPSJ*. 1994.

[38] G. Chechik, V. Sharma, U. Shalit, and S. Bengio. “An online algorithm for large scale image similarity learning”. In: *Proc. NIPS*. 2009.

[39] M. T. Chi, P. J. Feltovich, and R. Glaser. “Categorization and representation of physics problems by experts and novices”. In: *Cognitive science* 5.2 (1981), pp. 121–152.

[40] M. Collins. “Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms”. In: *Proc. EMNLP*. 2002.

[41] R. W. Cooley. “Web usage mining: discovery and application of interesting patterns from web data”. PhD thesis. University of Minnesota, 2000.

[42] C. Cortes and V. Vapnik. “Support-Vector Networks”. In: *Machine Learning*. 1995, pp. 273–297.

[43] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. Tech. rep. RFC 4627. IETF, July 2006.

[44] W. B. Croft, D. Metzler, and T. Strohman. *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010.

[45] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. “Indexing by latent semantic analysis”. In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407.

[46] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. “An optimal decomposition algorithm for tree edit distance”. In: *Transactions on Algorithms* 6.1 (2009), 2:1–2:19.

[47] C. Doersch, S. Singh, A. Gupta, J. Sivic, and A. A. Efros. “What makes Paris look like Paris?” In: *Proc. SIGGRAPH* (2012).

[48] *DOM4*. W3C Working Group. Dec. 2012. URL: `http://www.w3.org/TR/dom/`.

[49] W. Dong, C. Moses, and K. Li. “Efficient k-nearest neighbor graph construction for generic similarity measures”. In: *Proc. World wide web*. 2011, pp. 577–586.

[50] M. Dontcheva, S. M. Drucker, D. Salesin, and M. F. Cohen. “Relations, cards, and search templates: user-guided web data integration and layout”. In: *Proc. UIST*. 2007, pp. 61–70.

[51] M. Dontcheva, S. M. Drucker, G. Wade, D. Salesin, and M. F. Cohen. “Summarizing personal web browsing sessions”. In: *Proc. UIST*. 2006.

[52] M. Dontcheva, S. Lin, S. Drucker, and D Salesin. “Experiences with Content Extraction from the Web”. In: *Proc. CHI Workshop on Semantic Web User Interaction*. 2008.

[53] S. P. Dow, A. Glassco, J. Kass, M. Schwarz, D. L. Schwartz, and S. R. Klemmer. “Parallel prototyping leads to better design results, more divergence, and increased self-efficacy”. In: *TOCHI* 17.4 (2010), 18:1–18:24.

[54] D. K. van Duyne, J. A. Landay, and J. I. Hong. *Design of Sites*. 2nd ed. Addison-Wesley, 2009.

[55] O. Etzioni. “The World-Wide Web: quagmire or gold mine?” In: *Communications of the ACM* 39.11 (1996), pp. 65–68.

[56] M. Fisher, D. Ritchie, M. Savva, T. Funkhouser, and P. Hanrahan. “Example-based synthesis of 3D object arrangements”. In: *Proc. SIGGRAPH Asia*. 2012.

[57] M. Fitzgerald. *CopyStyler: Web Design by Example*. Tech. rep. MIT, 2008.

[58] D. R. Flatla, K. Reinecke, C. Gutwin, and K. Z. Gajos. “SPRWeb: preserving subjective responses to website colour schemes through automatic recolouring”. In: *Proc. CHI*. 2013.

[59] D. Florescu, A. Y. Levy, and A. O. Mendelzon. “Database techniques for the World-Wide Web: A survey”. In: *SIGMOD record* 27.3 (1998), pp. 59–74.

[60] *Foundation*. Zurb. 2013. URL: `http://foundation.zurb.com/`.

[61] M. R. Frank and J. D. Foley. “Model-based user interface design by example and by interview”. In: *Proc. UIST*. 1993, pp. 129–137.

[62] K. Z. Gajos and D. S. Weld. “Preference elicitation for interface optimization”. In: *Proc. UIST*. 2005, pp. 173–182.

[63] K. Z. Gajos and D. S. Weld. “SUPPLE: automatically generating user interfaces”. In: *Proc. IUI*. 2004, pp. 93–100.

[64] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. “Automatically generating personalized user interfaces with Supple”. In: *Artificial Intelligence* 174.12 (2010), pp. 910–950.

[65] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld. “Automatically generating user interfaces adapted to users’ motor and vision capabilities”. In: *Proc. UIST*. 2007, pp. 231–240.

[66] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld. “Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces”. In: *Proc. CHI*. 2008, pp. 1257–1266.

[67] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[68] M. R. Garey and D. S. Johnson. “Complexity Results for Multiprocessor Scheduling under Resource Constraints”. In: *SIAM Journal on Computing* 4.4 (1975), pp. 397–411.

[69] D. Gentner, S. Brem, R. W. Ferguson, A. B. Markman, B. B. Levidow, P. Wolff, and K. D. Forbus. “Analogical reasoning and conceptual change: A case study of Johannes Kepler”. In: *Journal of the Learning Sciences* 6.1 (1997), pp. 3–40.

[70] D. Gentner, K. Holyoak, and B. Kokinov. *The Analogical Mind: Perspectives From Cognitive Science*. MIT Press, 2001.

[71] L. Getoor and C. P. Diehl. “Link mining: a survey”. In: *SIGKDD Explorations Newsletter* 7.2 (2005), pp. 3–12.

[72] D. Gibson, K. Punera, and A. Tomkins. “The volume and evolution of web page templates”. In: *Proc. WWW special interest tracks and posters*. 2005, pp. 830–839.

[73] M. L. Gick and K. J. Holyoak. “Schema induction and analogical transfer”. In: *Cognitive psychology* 15.1 (1983), pp. 1–38.

[74] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. “Church: A language for generative models”. In: *Proc. UAI*. 2008, pp. 220–229.

[75] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[76] B. Hartmann, S. Doorley, and S. R. Klemmer. “Hacking, mashing, gluing: Understanding opportunistic design”. In: *Pervasive Computing, IEEE* 7.3 (2008), pp. 46–54.

[77] B. Hartmann, L. Wu, K. Collins, and S. R. Klemmer. “Programming by a sample: rapidly creating Web applications with d.mix”. In: *Proc. UIST*. 2007, pp. 241–250.

[78] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. “Design as exploration: creating interface alternatives through parallel authoring and runtime tuning”. In: *Proc. UIST*. 2008, pp. 91–100.

[79] Y. Hashimoto and T. Igarashi. “Retrieving Web Page Layouts using Sketches to Support Example-based Web Design”. In: *Proc. SBIM*. 2005.

[80] G. Hatano and K. Inagaki. “Two courses of expertise”. In: *Child development and education in Japan*. Ed. by H. Stevenson, H. Azuma, and K. Hakuta. New York: Freeman, 1986, pp. 262–272.

[81] J. Hays and A. A. Efros. “Scene completion using millions of photographs”. In: *Proc. SIGGRAPH*. 2007.

[82] *Heritrix*. Internet Archive. 2013. URL: `https://webarchive.jira.com/wiki/display/Heritrix/`.

[83] S. R. Herring, C.-C. Chang, J. Krantzler, and B. P. Bailey. “Getting inspired!: understanding how and why examples are used in creative design practice”. In: *Proc. CHI*. 2009.

[84] I. Hickson. *HTML5: Edition for Web Authors*. URL: http://dev.w3.org/html5/spec-author-view/.

[85] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. “WebBase: a repository of Web pages”. In: *Computer Networks* 33.1–6 (2000), pp. 277–293.

[86] D. Hofstadter. “Variations on a theme as the crux of creativity”. In: *Metamagical Themas*. Bantam Books, 1986.

[87] A. Hogue and D. Karger. “Thresher: automating the unwrapping of semantic content from the World Wide Web”. In: *Proc. WWW*. 2005, pp. 86–95.

[88] V. Hollink, M. van Someren, and V. de Boer. “Capturing the needs of amateur web designers by means of examples”. In: *Proc. ABIS*. 2008.

[89] K. J. Holyoak. “The pragmatics of analogical transfer”. In: *The psychology of learning and motivation* 19 (1985), pp. 59–87.

[90] *Houzz*. Houzz, Inc. 2013. URL: http://www.houzz.com/.

[91] *HTML 4.01 specification*. W3C Working Group. Dec. 1999. URL: http://www.w3.org/TR/html401/.

[92] *HTML5*. W3C. 2013. URL: http://www.w3.org/TR/html5/.

[93] N. Hurst, W. Li, and K. Marriott. “Review of automatic document formatting”. In: *Proc. DocEng*. 2009.

[94] D. Huynh, S. Mazzocchi, and D. Karger. “Piggy bank: experience the semantic web inside your Web browser”. In: *Proc. ISWC*. 2005, pp. 413–430.

[95] P. Indyk and R. Motwani. “Approximate nearest neighbors: towards removing the curse of dimensionality”. In: *Proc. STOC*. 1998.

[96] M. Y. Ivory and M. A. Hearst. “Statistical profiles of highly-rated Web sites”. In: *Proc. CHI*. 2002, pp. 367–374.

[97] C. Jacobs, W. Li, E. Schrier, D. Bargeron, and D. Salesin. “Adaptive grid-based document layout”. In: *Proc. SIGGRAPH*. 2003, pp. 838–847.

[98] S. Johnson. *Interface culture: how new technology transforms the way we create and communicate*. Basic Books, 1997.

[99] K. S. Jones. “A statistical interpretation of term specificity and its application in retrieval”. In: *Journal of Documentation* 28 (1972), pp. 11–21.

[100] J. Kang, J. Yang, and J. Choi. “Repetition-based Web page segmentation by detecting tag patterns for small-screen devices”. In: *IEEE Transactions on Consumer Electronics* 56 (2 2010), pp. 980–986.

[101] Y. Kim, J. Park, T. Kim, and J. Choi. “Web Information Extraction by HTML Tree Edit Distance Matching”. In: *International Conference on Convergence Information Technology*. 2007, pp. 2455–2460.

[102] J. M. Kleinberg. “Authoritative sources in a hyperlinked environment”. In: *JACM* 46.5 (1999), pp. 604–632.

[103] S. R. Klemmer, M. Thomsen, E. Phelps-Goodman, R. Lee, and J. A. Landay. “Where do web sites come from?: capturing and interacting with design history”. In: *Proc. CHI*. 2002.

[104] R. Kohavi, R. M. Henne, and D. Sommerfield. “Practical guide to controlled experiments on the web: listen to your customers not to the HiPPO”. In: *Proc. KDD*. 2007.

[105] J. L. Kolodner. “An introduction to case-based reasoning”. In: *Artif. Intell. Rev.* 6.1 (1992), pp. 3–34.

[106] J. L. Kolodner and L. M. Wills. “Case-Based Creative Design”. In: *AAAI Spring Symposium on AI and Creativity*. 1993, pp. 50–57.

[107] R. Kosala. “Web mining research: a survey”. In: *SIGKDD Explorations* 2 (2000).

[108] A. Kovashka, D. Parikh, and K. Grauman. “Whittlesearch: Image search with relative attribute feedback”. In: *Proc. CVPR*. 2012, pp. 2973–2980.

[109] C. Kulkarni, S. P. Dow, and S. R. Klemmer. “Early and Repeated Exposure to Examples Improves Creative Work”. In: *Cognitive science* (2012).

[110] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. “Webzeitgeist: design mining the web”. In: *Proc. CHI*. 2013.

[111] R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. “Bricolage: example-based retargeting for Web design”. In: *Proc. CHI*. ACM, 2011.

[112] R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. “Data-driven Web design”. In: *Proc. ICML*. 2012.

[113] R. Kumar, J. O. Talton, S. Ahmad, T. Roughgarden, and S. R. Klemmer. “Flexible tree matching”. In: *Proc. IJCAI*. 2011.

[114] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. “Building high-level features using large scale unsupervised learning”. In: *Proc. ICML*. 2012.

[115] B. Lee, S. Srivastava, R. Kumar, R. Brafman, and S. R. Klemmer. “Designing with interactive example galleries”. In: *Proc. CHI*. 2010.

[116] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. “IRLbot: scaling to 6 billion pages and beyond”. In: *Proc. WWW*. ACM, 2008, pp. 427–436.

[117] M. Leordeanu and M. Hebert. “A spectral technique for correspondence problems using pairwise constraints”. In: *Proc. ICCV*. 2005, pp. 1482–1489.

[118] L. Lessig. *Remix: Making Art and Commerce Thrive in the Hybrid Economy*. Penguin Press, 2008.

[119] M. Lim, R. Kumar, A. Satyanarayan, C. Torres, J. O. Talton, and S. R. Klemmer. *Learning Structural Semantics for the Web*. Tech. rep. CSTR 2012-03. Stanford University, 2012.

[120] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. “End-user programming of mashups with vegemite”. In: *Proc. IUI*. 2009, pp. 97–106.

[121] G. Lindgaard, C. Dudek, D. Sen, L. Sumegi, and P. Noonan. “An exploration of relations between visual appeal, trustworthiness and perceived usability of homepages”. In: *TOCHI* 18.1 (2011).

[122] G. Lindgaard, G. Fernandes, C. Dudek, and J. Brown. “Attention web designers: You have 50 milliseconds to make a good first impression!” In: *Behaviour & information technology* 25.2 (2006).

[123] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer-Verlag, 2011.

[124] B. Liu and K. Chen-Chuan-Chang. “Editorial: special issue on web content mining”. In: *SIGKDD Explorations Newsletter* 6.2 (2004), pp. 1–4.

[125] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. “Multi-probe LSH: efficient indexing for high-dimensional similarity search”. In: *Proc. VLDB*. 2007, pp. 950–961.

[126] *Main CSS Style Sheet*. Time Magazine. 2013. URL: `http://img.timeinc.net/time/assets/css/main.min.css`.

[127] R. L. Marsh, J. D. Landau, and J. L. Hicks. “How examples may (and may not) constrain creativity”. In: *Memory & Cognition* 24.5 (1996), pp. 669–680.

[128] A. Mesbah and A. van Deursen. “Invariant-based automatic testing of AJAX user interfaces”. In: *Proc. SIGSOFT*. 2009.

[129] J. Meskens, J. Vermeulen, K. Luyten, and K. Coninx. “Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me”. In: *Proc. AVI*. ACM. 2008, pp. 233–240.

[130] *MongoDB*. 10gen, Inc. 2012. URL: `http://www.mongodb.org`.

[131] L. Monroe. *How HTML5 element names were decided*. Apr. 2011. URL: `http://www.leemunroe.com/html5-element-names`.

[132] B. A. Myers. “User interface software tools”. In: *TOCHI* 2.1 (1995), pp. 64–103.

[133] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. “Garnet: Comprehensive support for graphical, highly interactive user interfaces”. In: *IEEE Computer* 23.11 (1990), pp. 71–85.

[134] B. A. Myers, B. V. Zanden, and R. B. Dannenberg. “Creating graphical interactive application objects by demonstration”. In: *Proc. SIGGRAPH*. 1989, pp. 95–104.

[135] *MySQL*. Oracle Corporation. 2012. URL: `http://www.mysql.com`.

[136] *Neo4j*. Neo Technology. 2012. URL: `http://neo4j.org/`.

[137] S. Nestorov, S. Abiteboul, and R. Motwani. “Inferring structure in semistructured data”. In: *SIGMOD record* 26.4 (1997), pp. 39–43.

[138] M. W. Newman and J. A. Landay. In: *Proc. DIS*. 2000, pp. 263–274.

[139] A. Oliva and A. Torralba. “Modeling the shape of the scene: a holistic representation of the spatial envelope”. In: *International Journal of Computer Vision* 42.3 (2001), pp. 145–175.

[140] D. R. Olsen Jr. “A programming language basis for user interface”. In: *Proc. CHI*. 1989, pp. 171–176.

[141] S. B. F. Paletz, K. H. Kim, C. D. Schunn, I. Tollinger, and A. Vera. “Reuse and Recycle: The Development of Adaptive Expertise, Routine Expertise, and Novelty in a Large Research Team”. In: *Applied Cognitive Psychology* 27.4 (2013), pp. 415–428.

[142] *Panda*. Google, Inc. 2013. URL: `http://googlewebmastercentral.blogspot.com/2011/05/more-guidance-on-building-high-quality.html`.

[143] *Pipes*. Yahoo! Inc. 2013. URL: `http://pipes.yahoo.com`.

[144] B. Pollak and W. Gatterbauer. “Creating permanent test collections of Web pages for information extraction research”. In: *Proc. SOFSEM*. 2. 2007, pp. 103–115.

[145] *Polyvore*. 2013. URL: `http://www.polyvore.com/`.

[146] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. 3rd. New York, NY, USA: Cambridge University Press, 2007.

[147] A. R. Puerta. “A model-based interface development environment”. In: *IEEE Software* 14.4 (1997), pp. 40–47.

[148] M. O. Rabin. *Fingerprinting by random polynomials*. Tech. rep. Center for Research in Computing Technology, Harvard University, 1981.

[149] S. Raghavan and H. Garcia-Molina. “Crawling the Hidden Web”. In: *Proc. VLDB*. 2001.

[150] K. Reinecke and A. Bernstein. “Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces”. In: *TOCHI* 18.2 (2011).

[151] K. Reinecke, T. Yeh, L. Miratrix, R. Mardiko, Y. Zhao, J. Liu, and K. Z. Gajos. “Predicting users’ first impressions of website aesthetics with a quantification of perceived visual complexity and colorfulness”. In: *Proc. CHI*. 2013, pp. 2049–2058.

[152] L. Richardson and S. Ruby. *Restful Web services*. 1st ed. O’Reilly, 2007.

[153] D. Ritchie, A. Kejriwal, and S. R. Klemmer. “d.tour: style-based exploration of design example galleries”. In: *Proc. UIST*. 2011.

[154] L. Rosenfeld and P. Morville. *Information architecture for the world wide web*. 2nd ed. O’Reilly & Associates, Inc., 2002.

[155] R. C. Schank. *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge University Press, 1983.

[156] D. A. Schön. *The design studio: an exploration of its traditions and potentials*. Architecture and the Higher Learning. Riba Publishing, 1985.

[157] D. A. Schön. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, 1983, p. 390.

[158] E. Schrier, M. Dontcheva, C. Jacobs, G. Wade, and D. Salesin. “Adaptive layout for dynamically aggregated documents”. In: *Proc. IUI*. 2008, pp. 99–108.

[159] D. L. Schwartz, J. D. Bransford, and D. Sears. “Efficiency and innovation in transfer”. In: *Transfer of learning from a modern multidisciplinary perspective* (2005), pp. 1–51.

[160] N. Shadbolt, T. Berners-Lee, and W. Hall. “The semantic web revisited”. In: *IEEE Intelligent Systems* 21.3 (2006), pp. 96–101.

[161] D. Shasha, J. T.-L. Wang, K. Zhang, and F. Y. Shih. “Exact and approximate algorithms for unordered tree matching”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.4 (1994), pp. 668–678.

[162] D. Shea. *CSS Zen Garden*. 2013. URL: http://www.csszengarden.com/.

[163] D. A. Smith. “Efficient inference for trees and alignments: modeling monolingual and bilingual syntax with hard and soft constraints and latent variables”. PhD thesis. Johns Hopkins University, 2010.

[164] S. M. Smith, T. B. Ward, and J. S. Schumacher. “Constraining effects of examples in a creative generation task”. In: *Memory & Cognition* 21.6 (1993), pp. 837–845.

[165] R. Socher, C. C. Lin, A. Y. Ng, and C. D. Manning. “Parsing Natural Scenes and Natural Language with Recursive Neural Networks”. In: *Proc. ICML*. 2011.

[166] R. Socher, C. Manning, and A. Ng. “Learning Continuous Phrase Representations and Syntactic Parsing with Recursive Neural Networks”. In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. 2010.

[167] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. “Web usage mining: discovery and applications of usage patterns from Web data”. In: *SIGKDD Explorations Newsletter* 1.2 (2000), pp. 12–23.

[168] *Star schema — Wikipedia, the free encyclopedia*. [Online; accessed 19-September-2012]. Wikipedia. 2012. URL: http://en.wikipedia.org/wiki/Star_schema.

[169] X. Su and T. M. Khoshgoftaar. "A survey of collaborative filtering techniques". In: *Advances in artificial intelligence* (2009).

[170] P. Sukaviriya, J. D. Foley, and T. Griffith. "A second generation user interface design environment: The model and the runtime architecture". In: *Proc. INTERCHI*. 1993, pp. 375–382.

[171] F. Sun, D. Song, and L. Liao. "DOM based content extraction via text density". In: *Proc. SIGIR*. Beijing, China: ACM, 2011, pp. 245–254.

[172] P. Szekely, P. Luo, and R. Neches. "Beyond interface builders: model-based interface tools". In: *Proc. INTERCHI*. 1993, pp. 383–390.

[173] K.-C. Tai. "The Tree-to-Tree Correction Problem". In: *Journal of the ACM* 26.3 (1979), pp. 422–433.

[174] J. O. Talton, D. Gibson, L. Yang, P. Hanrahan, and V. Koltun. "Exploratory Modeling with Collaborative Design Spaces". In: *Proc. SIGGRAPH Asia*. 2009.

[175] J. O. Talton, L. Yang, R. Kumar, M. Lim, N. D. Goodman, and R. Měch. "Learning design patterns with Bayesian grammar induction". In: *Proc. UIST*. 2012.

[176] J. Teevan, S. T. Dumais, D. J. Liebling, and R. L. Hughes. "Changing how people view changes on the web". In: *Proc. UIST*. 2009.

[177] *The WebKit open source project*. Apple Inc. 2012. URL: http://www.webkit.org/.

[178] M. Toomim, S. M. Drucker, M. Dontcheva, A. Rahimi, B. Thomson, and J. A. Landay. "Attaching UI enhancements to websites with end users". In: *Proc. CHI*. 2009, pp. 1859–1868.

[179] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. “Large margin methods for structured and interdependent output variables”. In: *Journal of Machine Learning Research* 6 (2005), pp. 1453–1484.

[180] G. Vossen and S. Hagemann. *Unleashing Web 2.0: from concepts to creativity*. Elsevier Science, 2010.

[181] W3C Working Group. *Cascading style sheets snapshot 2010*. May 2011. URL: http://www.w3.org/TR/CSS/.

[182] M. Wattenberg and D. Fisher. “Analyzing perceptual organization in information graphics”. In: *Information Visualization* 3.2 (2004), pp. 123–133.

[183] *Wayback Machine*. Internet Archive. 2013. URL: http://archive.org/web/web.php.

[184] *Web Authoring Statistics*. Google Inc. 2005. URL: https://developers.google.com/webmasters/state-of-the-web/.

[185] J. Wong and J. I. Hong. “Making mashups with marmite: towards end-user programming for the web”. In: *Proc. CHI*. 2007.

[186] F. Wu and D. S. Weld. “Autonomously semantifying wikipedia”. In: *Proc. CIKM*. 2007, pp. 41–50.

[187] T. fan Wu, C.-J. Lin, and R. C. Weng. “Probability estimates for multi-class classification by pairwise coupling”. In: *Journal of Machine Learning Research* 5 (2003), pp. 975–1005.

[188] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst. “Faceted metadata for image search and browsing”. In: *Proc. CHI*. 2003.

[189] L. Yi, B. Liu, and X. Li. “Eliminating noisy information in Web pages for data mining”. In: *Proc. SIGKDD*. 2003, pp. 296–305.

[190] K. Zhang, R. Statman, and D. Shasha. “On the editing distance between unordered labeled trees”. In: *Information Processing Letters* 42.3 (1992), pp. 133–139.

[191] X. S. Zheng, I. Chakraborty, J. J.-W. Lin, and R. Rauschenberger. “Correlating low-level image statistics with users - rapid aesthetic and affective judgments of web pages”. In: *Proc. CHI*. 2009.