AN EMPIRICAL STUDY OF CSS CODE SMELLS IN WEB FRAMEWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Tobias Bleisch

March 2018

COMMITTEE MEMBERSHIP

TITLE:	An Empirical Study of CSS Code Smells in Web Frameworks
AUTHOR:	Tobias Bleisch
DATE SUBMITTED:	March 2018
COMMITTEE CHAIR:	Motahareh Bahrami Zunjani, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Foaad Khosmood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

An Empirical Study of CSS Code Smells in Web Frameworks

Tobias Bleisch

Cascading Style Sheets (CSS) has become essential to front-end web development for the specification of style. But despite its simple syntax and the theoretical advantages attained through the separation of style from content and behavior, CSS authoring today is regarded as a complex task. As a result, developers are increasingly turning to CSS preprocessor languages and web frameworks to aid in development. However, previous studies show that even highly popular websites which are known to be developed with web frameworks contain CSS code smells such as duplicated rules and hard-coded values. Such code smells have the potential to cause adverse effects on websites and complicate maintenance. It is therefore important to investigate whether web frameworks may be encouraging the introduction of CSS code smells into websites.

In this thesis, we investigate the prevalence of CSS code smells in websites built with different web frameworks and attempt to recognize a pattern of CSS behavior in these frameworks. We collect a dataset of several hundred websites produced by each of 19 different frameworks, collect code smells and other metrics present in the CSS code of each website, train a classifier to predict which framework the website was built with, and perform various clustering tasks to gain insight into the correlations between code smells. Our results show that CSS code smells are highly prevalent in websites built with web frameworks, we achieve an accuracy of 39% in correctly classifying the frameworks based on CSS code smells and metrics, and we find interesting correlations between code smells.

ACKNOWLEDGMENTS

I give my utmost thanks to:

- Dr. Sara Bahrami, for teaching me what it means to do research and sticking it through with me to the end. Thanks for believing in me!
- Dr. Foaad Khosmood, for his excellent mentorship and advice. Our brainstorming sessions were a real pleasure!
- Dr. Axel Böttcher at the Munich University of Applied Sciences, for his patience and feedback. This thesis truly started with you, and I'm grateful for that!
- Dr. Gudrun Socher at the Munich University of Applied Sciences for facilitating my research stay at MUAS! I sincerely hope the partnership between Cal Poly and MUAS continues to flourish.
- Dr. Franz Kurfess, for his continued support. You've been a fantastic mentor to me throughout my time at Poly, thank you!
- Mr. Elbert Alias at Wappalyzer, for the extremely valuable dataset. You truly helped make this thesis what it is.
- Andrew Guenther, for providing this wonderful LaTeXtemplate. I almost used Word for this...
- Mike Ryu, for his feedback and words of encouragement. You should know that your opinion, like your friendship, is highly valuable to me, thank you!
- My Oma Liana, for her love and support. All the times I complained to you on my way to the lab helped more than you know.
- My parents Frank and Carolin, and my brother Jonas, for their love and support. My accomplishments are worth so much more when I can share them with you.

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER

1 Introduction 1

1.1 Overview 1

1.2 Chapter Outline 5

2 Background 7

2.1 Web Technologies 7

2.1.1 HTML and XML 7

2.1.2 The Document Object Model 8

2.1.3 Cascading Style Sheets 8

2.1.4 JavaScript 9

2.1.5 Uniform Resource Locators & Website Domains 9

2.1.6 AJAX - Static vs Dynamic Websites 10

2.1.7 Web Frameworks & Content Management Systems (CMS) . . 10

2.2 Software Engineering Principles 11

2.2.1 Code Smells 11

2.2.2 Coupling & Cohesion 11

2.2.3 Separation of Concerns 12

2.3 Machine Learning & Analysis Techniques 12

2.3.1 Supervised Learning - Classification 13

2.3.2 Unsupervised Learning - Clustering 13

2.3.3 Neural Network 14

2.3.4 Euclidean Space & Distance 14

2.3.5 K-Means Clustering 14

2.3.6 Hierarchical & Agglomerative Clustering 15

3 Cascading Style Sheets & Code Smells 16

3.1 Overview of CSS 16

3.1.1 CSS Rules . . . 16
3.1.2 CSS Selectors . . . 17
3.1.3 Pseudo-Classes & Pseudo-Elements . . . 18
3.1.4 Code Location . . . 19
3.1.5 Specificity . . . 19
3.1.6 Importance . . . 20
3.1.7 Inheritance . . . 21
3.1.8 Cascading . . . 21
3.1.9 @Rules . . . 22
3.2 CSS Code Smells . . . 23
3.2.1 Inlined Rules . . . 23
3.2.2 Embedded Rules . . . 23
3.2.3 Too Long Rules . . . 23
3.2.4 Empty Catch Rules . . . 24
3.2.5 Too Specific Selectors Type I (Too Much Cascading) . . . 24
3.2.6 Too Specific Selectors Type II (High Specificity Values) . . . 25
3.2.7 Universal Selectors . . . 25
3.2.8 Selectors with ID and at Least One Class or Element . . . 26
3.2.9 Selectors with Erroneous Adjoining Pattern . . . 26
3.2.10 Too General Selectors . . . 27
3.2.11 Properties with Hard-Coded Values . . . 27
3.2.12 Properties with Value Equal to None or Zero (Undoing Style) 27
3.3 CSS Metrics . . . 28
3.3.1 External Rules . . . 29
3.3.2 Total Defined CSS Rules . . . 29
3.3.3 Total Defined CSS Selectors . . . 29
3.3.4 Ignored CSS Selectors . . . 29
3.3.5 Undefined Classes . . . 29
3.3.6 Matched Selectors . . . 29
3.3.7 Unmatched Selectors . . . 30
3.3.8 Effective Selectors . . . 30
3.3.9 Ineffective Selectors . . . 30

3.3.10 Total Defined CSS Properties . . . 30
3.3.11 Unused Properties . . . 30
3.3.12 Ignored Properties . . . 31
3.3.13 Files with CSS Code . . . 31
3.3.14 Lines of Code (LOC) . . . 31
4 Separation of Concerns in Cascading Stylesheets . . . 32
4.1 Overview . . . 32
4.2 Motivation . . . 34
4.3 Separation of Concerns & Coupling . . . 34
4.4 Co-occurence Coupling . . . 35
4.5 Structural Coupling . . . 36
4.6 Syntactic Coupling . . . 38
4.7 Coupling - The Reality . . . 40
5 Related Work . . . 41
5.1 Web Crawling . . . 41
5.2 CSS Metrics . . . 42
5.3 CSS Code Smells & Defects . . . 43
5.4 CSS Refactoring & Tool Support . . . 43
5.5 Contributions of this Work . . . 45
6 Empirical Study . . . 47
6.1 Overview . . . 47
6.2 Data Collection . . . 48
6.2.1 Collecting Websites . . . 50
6.2.2 CSSNose . . . 50
6.2.3 Code Smell Collection . . . 51
6.3 Analysis Methods . . . 52
6.3.1 Predictive Model . . . 52
6.3.2 Clustering Model . . . 53
7 Results and Discussion . . . 55
7.1 Metrics . . . 55
7.2 RQ1 - Code Smell Prevalence in Web Frameworks . . . 57
7.3 RQ2 - Identifying Web Framework from CSS Code Smells . . . 59

7.4 RQ3 - Correlation Between Code Smells 67
8 Threats to Validity 76
9 Conclusion & Future Work 77
9.1 Conclusion 77
9.2 Future Work 78
BIBLIOGRAPHY 80
APPENDICES
A CSSNose Code Smell Name Mapping 89

LIST OF TABLES

6.1 The frameworks under consideration in this study as well as the language of development and the number of code smell samples collected for each. "CMS" stands for Content Management System. 49

7.1 The percentages of the websites for a specific framework that contain at least one instance of the specified code smell. 60

7.2 The mean, standard deviation, and z-score mean for each code smell/-metric across frameworks 1-9. 61

7.3 The mean, standard deviation, and z-score mean for each code smell/-metric across frameworks 10-19. 62

7.4 The full set of CSS code smells and metrics collected from websites using CSSNose along with their descriptions and the functional group they belong to (rule-based, selector-based, property-based, or file-based). 63

7.5 The accuracies of various classifiers tested during framework prediction. 65

7.6 The precision, recall, F1-score, and support on a neural network classifier. 66

7.7 The number of features and accuracies for each functional group of smells and metrics trained on a classifier with 10-fold cross validation. 67

7.8 Example feature sets of two websites along with the actual and predicted framework from the classifier. 68

LIST OF FIGURES

3.1 An example of (1) CSS linked via an external stylesheet using the HTML `link` tag, (2) CSS inlined in an HTML `p` tag using the `style` attribute, and (3) CSS embedded in the HTML using the HTML `style` tag. 20

6.1 The software architecture setup for the experiment. 48

7.1 The website clusters produced by K-Means. The units of the axes represent a linear combination of the feature set produced by Principle Component Analysis (PCA). 69

7.2 Clusters based on (a) code smell/metric and (b) framework. Each color/number represents a unique cluster. The units of the axes represent a linear combination of the feature set produced by Principle Component Analysis (PCA). 70

7.3 The dendrogram produced by agglomerative clustering of code smells and metrics. 74

7.4 The dendrogram produced by agglomerative clustering of frameworks. 75

Chapter 1

INTRODUCTION

1.1 Overview

Modern front-end web development consists of the specification of content, behavior, and style. These separate concerns are addressed respectively by the Hypertext Markup Language (HTML), JavaScript, and Cascading Style Sheets (CSS) and together along with XML/JSON comprise the Asynchronous JavaScript and XML (AJAX) web development paradigm. CSS in particular has a relatively simple syntax and was adopted by the World Wide Web Consortium (WC3) as a way to separate presentation from HTML [49]. This separation of concerns led to reduced effort in content-authoring [37], the ability for designers to work independently from developers, and, most importantly, code reuse on the file level.

However, despite the theoretical advantages that the separation of presentation from content provides, CSS is not easily understood or maintained [30] [37] [48] [55]. Features of CSS such as *inheritance*, *cascading*, and *selector specificity* all contribute to the challenging task of understanding how style properties are applied to the document object model (DOM) at runtime [48]. The result is that presentation authoring has become a complex and time-consuming task in which more time is often spent on coding decisions than on graphic design [37]. This is evidenced by the large body of books, articles, and blog posts presenting a variety of CSS frameworks, development methodologies, tips and tricks, best practices, etc [37]. CSS has also historically not received much attention from the research community [13] [20] [30] [32] [42] [48] [55].

The complexity of CSS development has led developers to pursue alternative tools [20] in the form of CSS preprocessor languages, CSS frameworks, and web frameworks,

many of which typically allow one to specify style abstractly and simply generate CSS code for interpretation by the browser. While these tools have their advantages and attempt to solve some identified problems in CSS, they often do so at the cost of quality in the actual generated CSS code. Some examples of this are style sheets with duplicated CSS rules, highly specific selectors, hard-coded property values, etc. Important to note, however, is that these and other code smells in CSS aren't simply bad for maintenance [32] (should the CSS still be authored by hand) but can also have other adverse affects on websites, including increased load on the network and server in transferring website files to the browser [48], increased browser rendering time of the page layout [20], and negatively impacted accessibility, device independence, ubiquity, and mobility [55].

The likely prevailing school of thought allowing for the introduction of these code smells is that the codebase being actively manipulated and maintained by the developer is of sole importance, such that the quality and maintainability of generated/transpiled code need not be considered. The result is that many front-end web frameworks and development tools such as preprocessors introduce unnecessary code smells into the generated CSS code being served to the browser, inviting the negative effects mentioned earlier. In one study, Gharachorlu found that 99.8% of the websites studied contain at least one type of CSS code smell [32]. Nguyen et al. found that, for six PHP-based web applications, 89-100% of a certain set of six code smells that violate various software engineering principles on the client side can be mapped back to code smells embedded in PHP string literals [50]. Mazinanian et al. found that an average of 66% of style declarations are repeated at least once in a CSS file [46]. Keller and Nussbaumer found that CSS generated by Adobe Fireworks CS4 and WuarkXPress 8 are much less abstract than those authored by hand [37].

Nevertheless, millions of websites written today are built using content management systems, web frameworks, and preprocessor languages. This is because, al-

though developers have the most control when designing a website "from scratch" - writing custom HTML, CSS, and JavaScript code paired with some server-side language such as PHP or JavaScript - they are willing to sacrifice control for the sake of speed, scalability, and ease of maintenance. These frameworks make certain design decisions on their behalf in order to build websites in a semi-automated way - often using scripts that are following a design template to generate the resulting site's files. Even though a web framework may not enforce style decisions on the developer, they may *encourage* the introduction of code smells through their development process. It's important, then, that developers have as much information as possible when making the decision to adopt a framework over a custom solution in order to combat any hidden consequences in the selection, such as the quality of the resulting CSS code. We attempt to provide some of this information to developers through descriptive statistics and, more importantly, pave the road for future investigations into the development practices of web frameworks.

Recognizing a pattern of CSS behavior applied by certain web frameworks would allow us to ask the question: which design decisions led to the pattern of CSS behavior being predictable? Following that are questions such as (1) based on this pattern, which frameworks have problems and to what extent, (2) are CSS code smells the result of generated CSS code or the encouragement of bad practice in CSS through the development process, and (3) which frameworks should we begin to fix, and in what ways? Before these questions can be addressed, however, we must first determine whether web frameworks appear to have an influence on the presence of code smells or not. While there have been efforts in this domain to improve the CSS language and its development process [20] [48] [54] [55] [57], there haven't been any efforts thus far to characterize the types and prevalence of smells introduced into websites by popular web frameworks. Perhaps the closest works to this thesis are those by Keller and Nussbaumer [37] and Nguyen et al. [50].

Establishing a correlation between individual CSS code smells and metrics could provide insight into which smells tend to follow each other and how the introduction of smells can be influenced by some practices in CSS. This would allow for the development of analysis tools that suggest possible problems in development practices and help framework developers and website developers fix code smell issues.

In order to gain insight into the types and prevalence of code smells generated by certain web frameworks, we pose the following research questions:

RQ1 How prevalent are code smells in websites built with different web frameworks?

RQ2 Can we recognize a pattern of CSS behavior in certain web frameworks? Can a set of CSS code smells and metrics collected from a website be used as a unique identifier for the web framework used to develop the website?

RQ3 Does there exist a correlation between CSS code smells and metrics collected from a website?

To answer these research questions, we collect a dataset of thousands of websites produced from 19 different web frameworks, utilize the CSS code smell detection tool CSSNose [32] to extract code smells and other CSS metrics, and train a classifier to predict the framework of a website based on its code smells, as well as perform a clustering analysis on the extracted code smells. Our results show that code smells are highly prevalent in websites produced with web frameworks, CSS code smells and metrics can be used to predict one of 19 frameworks with up to 39% accuracy, and a useful correlation does appear to exist between code smells.

In addition to the empirical study setup to answer these research questions, this thesis presents new terminology for three types of coupling that exist between CSS and HTML documents. These ideas of coupling can be considered the application of the Software Engineering principle Separation of Concerns to modern front-end web

development and they're intended to provide a mental model and terms for discussion to developers within the community.

1.2 Chapter Outline

The rest of this thesis is structured as follows:

- In Chapter 2, **Background**, we provide background information on web technologies, Software Engineering principles, and machine learning and analysis techniques.
- In Chapter 3, **Cascading Style Sheets & Code Smells**, we present an overview of the CSS language and introduce the code smells and metrics relevant to the empirical study.
- In Chapter 4, **Separation of Concerns in Cascading Stylesheets**, we expand the notion of Separation of Concerns in CSS by formally defining three types of coupling between HTML and CSS and identify the code smells and metrics which have tracked Separation of Concerns in CSS thus far.
- In Chapter 5, **Related Work**, we introduce work in the field relevant to web crawling, CSS metrics, CSS code smells and defects, and CSS refactoring and tool support.
- In Chapter 6, **Empirical Study**, we introduce the setup and approach of the study including the software architecture and tools, the data collection process, and the models used for analysis of the data.
- In Chapter 7, **Results & Discussion**, we present the results from the study and discuss their relevance in answering the posed research questions.

- In Chapter 8, **Threats to Validity**, we discuss issues which may affect the validity of the findings.
- In Chapter 9, **Conclusion & Future Work**, we summarize the findings and contributions of this work as well as outline possible avenues for future work.

Chapter 2

BACKGROUND

This chapter details some web technologies, software engineering principles, and machine learning and analysis techniques that are important for understanding the concepts presented in following chapters.

2.1 Web Technologies

Before introducing the specifics of code smells and metrics in Cascading Style Sheets, it is important that the reader has an understanding of the basics of the technologies used in modern front-end web development.

2.1.1 HTML and XML

The Hypertext Markup Language (HTML) and Extensible Markup Language (XML) are markup languages [1] that allow for the human- and machine-readable specification of structured content. HTML is considered to be the original language of the World Wide Web, providing a standard which allows web browsers to interpret and compose text, images, and other material into visual or audible web pages. It addresses the concern of content specification in web applications. HTML was developed by Tim Berners-Lee at the European Organization for Nuclear Research (CERN) in 1990 as an application of the Standard Generalized Markup Language (SGML) [5], a meta-language standard for producing generalized markup languages. XML is another application of SGML developed by the XML Working Group in 1996 to allow the use of more than just the fixed vocabulary of HTML on the web and enforce strict syntax

[1] a **markup language** is a syntax and grammar for annotating a document in a way that indicates its logical structure

adherence lacking in most web browsers [12].

2.1.2 The Document Object Model

The document object model (DOM) is a logical representation of the structure of documents that specifies how a document may be accessed and manipulated - essentially, a programming API for HTML and XML documents. Documents are broken down into 'objects' which are organized into a tree-like structure and given functions and identity. In the context of web development, the HTML file represents the specific instance of a document, the browser constructs the DOM after parsing the HTML tags (turning them into objects), and the JavaScript file contains functionality to locate and manipulate the structure and content of the DOM objects. Thus, the DOM identifies: (1) the interfaces and objects used to represent and manipulate a document, (2) the semantics of these interfaces and objects - including both behavior and attributes, and (3) the relationships and collaborations among these interfaces and objects [1].

2.1.3 Cascading Style Sheets

Cascading Style Sheets (CSS) is a style sheet language [2] primarily used for describing the presentation of HTML documents on the web. CSS was created by Lie in 1994 [41] and adopted as an official standard by the W3C in 1996 [49]. CSS allows developers to specify style in the form of *rule blocks* containing *style properties* which are applied to a certain subset of DOM elements determined by a *selector*.

[2] a **style sheet language** is a computer language used to specify the presentation of a structured document

2.1.4 JavaScript

JavaScript (JS) is a general-purpose programming language primarily used for specifying dynamic behavior in web applications through client-side interactions with the DOM. It was originally developed in 1995 by computer scientist Brendan Eich at Netscape and taken to the European Computer Manufacturers Association (ECMA) in 1996 for standardization in the form of ECMAScript [6]. The EMCAScript standard allows web browsers to implement JavaScript engines[3] for the local execution of JavaScript code. Thanks to JavaScript engines such as V8 and Rhino, JavaScript has applications outside of the web browser such as for client-side scripting, video-game development, and desktop/mobile application development.

2.1.5 Uniform Resource Locators & Website Domains

A Uniform Resource Locator (URL) is a string reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it (usually HTTP) [11]. A URL is a specific type of Uniform Resource Identifier (URI) which is simply any string reference to a resource. The host component of the URL specifies the type of service being requested and the host computer of the resource on the internet. Domain names are most often used as the identifier of that host to preclude the need to memorize IP addresses. For example, in the URL `http://www.tobiasbleisch.com`, HTTP is the protocol for accessing the internet resource, www is the type of service (web server), and `tobiasbleisch` and `com` are the second and top level domain names respectively which together can be called the domain of the website being accessed. Domain names are used in this study when communicating which websites are to be downloaded and analyzed.

[3] a **JavaScript engine** is a program or interpreter which executes JavaScript code

2.1.6 AJAX - Static vs Dynamic Websites

Asynchronous JavaScript and XML (AJAX) is a collection of technologies and techniques for the development of highly interactive websites. HTML is used for specifying the content, CSS for the presentation, JavaScript in combination with the DOM for dynamically altering the website content and layout, and the XMLHttpRequest (XHR) object in JavaScript for transferring data between a web browser and a web server to avoid full page reloads. The invention of these technologies allowed for a shift from the development of static websites to the development of dynamic web applications [47]. Traditional static websites are based on a multi-page interface paradigm consisting of multiple unique web pages, each having a unique URL and hypertext link. Dynamic web applications using AJAX are able to alter the state of the DOM in real-time through the execution of JavaScript code, and it's therefore possible to represent unique web pages without a corresponding hypertext link [47].

2.1.7 Web Frameworks & Content Management Systems (CMS)

A web framework is a software environment and/or collection of development tools which provide generic functionality and utilities for the development of web applications and services [24]. They typically provide a standard way to automate the redundant tasks involved in web development such as authentication, database communications, style templating, etc., and may include functionality for building up both server-side and front-end components. Content management systems are similar but typically cater to consumers without technical experience, further offering tools for management of digital content and website serving. In the rest of this thesis, we refer to both web frameworks and content management systems as simply "web frameworks". Web frameworks such as ASP.NET, React, and Django are built for use with various general-purpose programming languages. The frameworks under

consideration in this study were selected based primarily on popularity and language distribution [29]. See Table 6.1 for a complete list of the frameworks under consideration.

2.2 Software Engineering Principles

The following terms are important ideas in Software Engineering which are applied to front-end web development in both the empirical study and the presentation of ideas related to coupling between CSS and HTML.

2.2.1 Code Smells

Code smells are structures in software that indicate the violation of fundamental design principles and negatively impact design quality. Though these structures may produce functionally correct results in the software and are not themselves bugs, they often contribute to bugs later in development as the size or complexity of the software grows [53].

2.2.2 Coupling & Cohesion

Coupling and cohesion are two general principles of Software Engineering that are concerned respectively with the level of dependency and the level of semantic or logical consistency between two software modules [59]. Coupling is usually contrasted with cohesion - that is, low coupling often correlates with high cohesion and vice versa. Coupling is considered to be high when, given two software modules A and B, a change to module A necessitates a change to module B. Cohesion is considered to be high when module A and module B both have unique, well-defined responsibilities and simultaneously contribute to a higher-level semantic responsibility [53]. A

maintainable software project tends to demonstrate low coupling and high cohesion. Stevens et al. invented these notions of coupling and cohesion [59].

2.2.3 Separation of Concerns

Separation of concerns is a general principle of Software Engineering that seeks to control complexity in software projects by separating a program into distinct sections, such that each section addresses a separate concern [62]. In a broader sense, a program is simply the implementation of a solution to a complex problem. Thus, separation of concerns can be defined more intuitively as the decomposition of a complex problem into manageable "concerns", solving these concerns individually without detailed knowledge of the other parts, and then combining them into one result. The result of applying separation of concerns to a program is modular code that also demonstrates *low coupling* and *high cohesion*. The term was originally introduced by Dijkstra [25]. One very typical example that demonstrates the concepts of the Separation of Concerns is the Model-View-Controller (MVC) design pattern. A software application which has a need for a user interface, internal data representation in the form of data models, and logic for the presentation of that data is best organized in such a way that these three distinct concerns are separated.

2.3 Machine Learning & Analysis Techniques

The following concepts are important for understanding the analysis methods used in the empirical study.

2.3.1 Supervised Learning - Classification

Supervised learning is the process of approximating a mathematical function that maps some set of inputs (independent variables) to some set of outputs (dependent variables) in a dataset [39]. These outputs are called *labels* and are typically manually collected and associated with the set of inputs called *features*. Each set of features and its corresponding label is referred to as an *example*. Once a mapping between features and labels has been established through a process called *training*, the developed model can be used for *classification*, the task of deriving labels on new (unseen) data points. There are a variety of supervised learning algorithms which all differ in their approach to the function approximation and which may perform better or worse on a dataset depending on the properties of the dataset they are trying to exploit and the assumptions they are making about the dataset. Supervised learning can be seen as having two main uses: (1) building a model for decision-making which depends on the accurate derivation of a data point's label and (2) establishing correlations between variables of interest within the dataset.

2.3.2 Unsupervised Learning - Clustering

Unsupervised learning is the process of approximating a mathematical function which maps some set of inputs to some set of inferred outputs in a dataset [31]. These inferred outputs are usually the result of some discovery of hidden structure in the dataset which can be achieved through various methods. One such method is *clustering*, in which the discovery of hidden structure is based on a notion of similarity between data points. Clustering is the task of dividing each data point (commonly referred to as an *object*) in a dataset into two or more groupings called *clusters*. There are a variety of unsupervised learning algorithms which all differ in their approach to the function approximation and which may produce different results depending on the

properties of the dataset they are trying to exploit and the assumptions they are making about the dataset. There is typically no notion of performance in unsupervised algorithms due to the lack of knowledge about the outputs. Unsupervised learning is mostly used for exploratory data mining and discovering interesting relationships in a dataset.

2.3.3 Neural Network

An (artificial) neural network is a learning algorithm which leverages a unique structure in its function approximation that is inspired by the structure of neurons and synapses in biological systems [51]. The function to be approximated is implicitly stored in the weights of the network. There are applications of neural networks for both supervised and unsupervised tasks, though supervised learning is more common.

2.3.4 Euclidean Space & Distance

Euclidean space is a finite-dimension real vector space $\mathbb{R}^N$ described by the axioms of Euclidean geometry and usually represented by Cartesian coordinates [34]. In essence, it's a system which facilitates the notion of points and provides methods for expressing the relationships between them such as lines, angles, and distances. Euclidean distance is the straight-line distance between two points in this space.

2.3.5 K-Means Clustering

K-Means is a clustering algorithm which aims to partition n observations into k clusters by minimizing the within-cluster variance (sum of squares) using Euclidean distance [23]. There are a variety of metrics to determine which value of k will result in the "best" clustering, but ultimately it's dependent on the user to choose the number of clusters the algorithm should output.

2.3.6 Hierarchical & Agglomerative Clustering

Hierarchical clustering is a method of cluster analysis which attempts to build a hierarchy of clusters such that they're successively formed. Agglomerative clustering is a "bottom up" hierarchical clustering approach which initially considers each object to be its own cluster and then successively forms clusters by merging neighboring objects. A common algorithm used to perform agglomerative clustering is the single-linkage clustering algorithm using the Euclidean distance as a metric of object similarity.

Chapter 3

CASCADING STYLE SHEETS & CODE SMELLS

This chapter presents an overview of the CSS language and its features as well as describes the code smells and metrics collected and used in the empirical study.

3.1 Overview of CSS

This section presents an overview of the CSS language and its features.

3.1.1 CSS Rules

A rule is a block of CSS code which specifies the style to be applied to specific parts of the document object model (DOM). A rule is composed of a *selector* and a *declaration block* in which a list of *style declarations* is placed. The selector specifies the locations in the DOM to which the rule block will apply. A single style declaration is composed of a *style property* to be altered and the value that the property should hold. Examples of common style properties are `color`, `font`, `width`, `height`, etc. The collection of declaration blocks specified in a style sheet will determine the appearance and layout of a page once rendered by the browser. The following snippet shows the typical structure of a CSS rule:

```
selector {
    property : value;
    ...
}
```

3.1.2 CSS Selectors

A selector is a collection of identifiers which correspond to locations in the tree structure composing the DOM. Selectors which share styling can even be grouped together so that the style properties only need be specified once. There are five types of basic selectors [10]:

Type Selector - A type selector (referred to in this thesis as an *element* selector) targets the nodes in the DOM that correspond to HTML elements sharing the same name. For example, the `h1` selector will target all `<h1></h1>` tags.

Class Selector - A class selector targets DOM nodes associated with a `class` attribute in the HTML document. Class selectors are preceded with a dot '.' and mirror the name of the class value that should be selected. For example, the `.intro` selector will target all nodes corresponding to HTML elements containing the attribute `class="intro"`.

ID Selector - An id selector targets all DOM nodes associated with an `ID` attribute in the HTML document. ID selectors are preceded with a pound symbol '#' and mirror the name of the ID that should be selected. For example, the `#launch` selector will target all nodes corresponding to HTML elements containing the attribute `id="launch"`.

Attribute Selector An attribute selector targets all DOM nodes either (1) containing a certain attribute, (2) containing an attribute with a certain value, (3) and containing an attribute with a certain substring within a value. For example, `a[title]` will select link elements containing a `title` attribute, `a[href="https://example.org"]` will select link elements containing an `href` attribute with the value `https://example.org`, and `a[href*="example"]` will select link elements containing an `href` attribute with the value containing the substring `example`.

Universal Selector - The universal selector '`*`' will select all nodes in the DOM, or all remaining nodes if used in combination with other selectors that have narrowed down the searchable space of the DOM.

Combinators - Combinators are symbols placed between selectors that narrow the search space of the remaining nodes of the DOM. They allow for simple selectors to be joined into ***combined selectors*** (referred to in this thesis as *complex* selectors) that target more specific parts of the DOM. Mesbah and Mirshokraie define them best [48]:

- `A B` - Descendent Combinator - A space ' ' between selectors A and B targets all elements selected by B which are descendents of A on the DOM.
- `A > B` - Child Combinator - A '>' between selectors A and B targets all elements selected by B which are direct children of the elements selected by A on the DOM.
- `A ~ B` - General Sibling Combinator - A '~' between selectors A and B targets all elements selected by B which have an element selected by A as a sibling on the DOM.
- `A + B` - Adjacent Sibling Combinator - A '+' between selectors A and B targets all elements selected by B which are directly preceded by a sibling eleemnt selected by A.

3.1.3 Pseudo-Classes & Pseudo-Elements

Pseudo-classes are class selectors which correspond to a DOM element's attributes, relative position, or *state* in the browser rather than to a class attribute specified in the HTML [8]. For example, the pseudo-class `a:hover` applies style to a link only when user's mouse currently hovers over it. Pseudo-elements are element selectors which

correspond to a specific part of a DOM element's content rather than to the entire content of a DOM element [9]. For example, the pseudo-element `p::first-line` adds style to the first line of a selected paragraph's text. Pseduo-classes are specified with a keyword preceded by a single colon ':' while pseudo-elements are specified with a keyword preceded with two colons '::', and they are each appended to the element selector to which they are intended to apply.

3.1.4 Code Location

There are three main ways for developers to include CSS:

- **external stylesheets** linked to an HTML document via the `<link>` element or `@import` rule
- **inlined** in HTML elements using the `style` attribute
- **embedded** in HTML using the `<style>` element

See Figure 3.1 for a visual of the various ways to include CSS in HTML. External stylesheets allow style code to be applied to multiple document instances. Although there are some valid uses for inlined and embedded CSS code, they are generally considered bad practice because they apply CSS to a specific document instance, therefore inhibiting code reuse and complicating maintenance.

3.1.5 Specificity

Specificity is a CSS language mechanism that provides a relevance measure for style rules. It can help serve as a tie-breaker if two rules are meant to be applied to the same set of DOM nodes. Selectors that target more specific subsets of the DOM

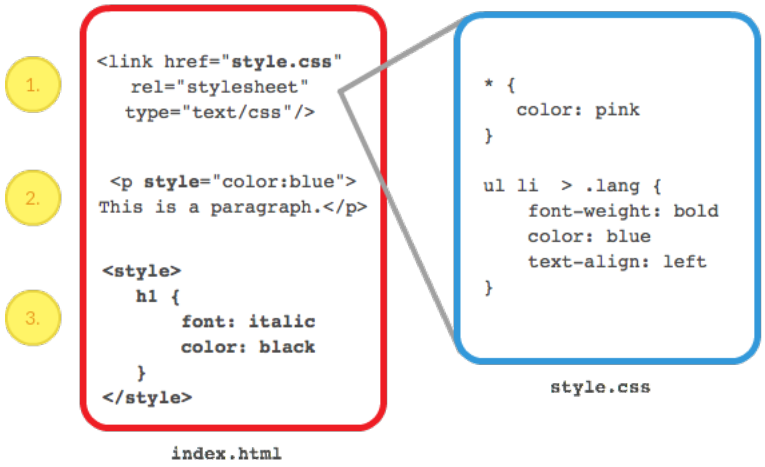

Figure 3.1: An example of (1) CSS linked via an external stylesheet using the HTML `link` tag, (2) CSS inlined in an HTML `p` tag using the `style` attribute, and (3) CSS embedded in the HTML using the HTML `style` tag.

are considered more relevant and therefore hold more weight over others in being applied. Specificity is measured as the concatenated counts of simple selectors contained in a complex selector in the following order: *inlined, ID, class/attribute/pseudo-class, element/psuedo-element*. For example, the complex selector `.intro [title] p` would have specificity `0,0,2,1` and the selector `#launch` would have specificity `0,1,0,0`. In this case, the latter has a higher specificity than the former.

3.1.6 Importance

Importance is a CSS language feature that marks certain style properties as more relevant than others, even those with rules of higher specificity. Specificity can be used as a tie-breaker among style properties claiming importance. Importance is used by specifying the term `!important` immediately following a value declaration but before the semicolon. For example, a property declaration `color:green !important;` contained in a rule with selector `h1` will take precedence over a property declaration `color:blue;` with selector `div h1.coolTitle`.

3.1.7 Inheritance

Inheritance is a CSS language feature that applies style properties to descendent nodes of a styled DOM node until overridden by another rule. The DOM is hierarchical in nature and inheritance takes advantage of this, the result being that the developer need not specify style for every node and all its children - only those which differ. Not all properties are inherited - only those which reasonably should be applied will be. For example, a style rule containing properties `border:solid; color:pink;` with selector `p` will also be applied to a `<span></span>` should it be present in the text of the paragraph selected by `p`. However, `border` would not be applied to `<em></em>` inside the same `<p></p>` because it's unreasonable to expect emphasized words in a paragraph to have a border [49].

3.1.8 Cascading

Cascading is a CSS language mechanism that determines which style rules are applied to the DOM. When a node has no style rules targeting it, it will simply inherit its style properties. When a node has a single rule targeting it, it will adopt that rule's style properties and inherit any leftover properties not specified in that rule. When a node has multiple rules targeting it, the characteristics of those rules are considered and resolved based on the importance (presence of the importance attribute on the style property), the origin of the CSS code, the specificity, and, lastly, the location of the CSS code. Important to note is that cascading only comes into effect when *more than one* of the *same* type of property is specified for a DOM element. This means that when multiple *different* properties are specified for a DOM element at varying levels of importance, origin, specificity, location, or through inheritance, such that there is no conflict, they will all be applied.

The cascade in CSS is specified in the following order:

- **Importance** - the presence of the important attribute on the style property
- **Origin** - the origin of the CSS code:
    1. author - style specified by the developer
    2. user - style specified by the user, given to the browser
    3. user-agent - style specified by the browser (defaults)
- **Specificity** - the specificity of the style property's selector
- **Location** - the *closeness* of the style code to the HTML elements. Rules at the same level of closeness to the HTML but which are specified later in the document are prioritized: inlined, embedded, then external.
- **Inheritance** - style properties specified for parent elements which inherit by default. Note that properties with the `inherit` value would not fall into this category.

3.1.9 @Rules

The "at" rules are those embedded or external rules which are preceded with an '@' symbol and provide instructions to the browser rather than directly specify styling [2]. The syntax of @Rules varies, but they can roughly grouped into three categories: (1) strictly standalone rules, (2) rules that can be standalone or nested in conditional group rules, and (3) conditional group rules which can contain non-@Rules and nestable @Rules. Some examples of @Rules are the `@media` rule which applies style based on a device's characteristics and environment (retrieved through media queries in the browser) and the `@import` rule which allows for the inclusion of an external stylesheet in the current HTML or CSS document.

3.2 CSS Code Smells

This section introduces CSS code smells collected in the empirical study, gives a brief explanation of why they're classified as smells, and provides short code snippets to demonstrate. Figure 7.4 lists and gives a brief summary of these code smells. The smells in this list are those collected by the tool CSSNose [32].

3.2.1 Inlined Rules

Inlined Rules are those contained in the `style` attributes of HTML elements. See Figure 3.1 for a visual example. Inlined style rules apply only to the HTML document in which they are included, which limits code reuse, and they can increase the complexity of CSS development by overriding styles defined in external stylesheets.

3.2.2 Embedded Rules

Embedded Rules are those contained between the `<style>` tags in an HTML document. See Figure 3.1 for a visual example. Embedded style rules apply only to the HTML document in which they are included, which limits code reuse, and they can increase the complexity of CSS development by overriding styles defined in external stylesheets.

3.2.3 Too Long Rules

Too Long Rules are those in which the number of style declarations is high enough that it starts to affect maintainability. Gharachorlu conducted a small-scale study on 20 websites and computed averages to conclude that rules containing more than five style property declarations should be considered too long.

```
body {
  font: 1em/150\% Helvetica, Arial, sans-serif;
  padding: 1em;
  margin: 0 auto;
  max-width: 33em;
  background-color: blue;
  color: red;
}
```

3.2.4 Empty Catch Rules

An Empty Catch Rule is simply a CSS rule which contains a selector but no properties. Empty rules add bloat to CSS code in terms of file size and maintenance without any style benefit.

```
.button {

}
```

3.2.5 Too Specific Selectors Type I (Too Much Cascading)

Too Much Cascading occurs when a complex selector has a high number of simple selector units and therefore targets a highly specific part of the DOM. In this way, it relies too heavily on the structure of the DOM and becomes inflexible to changes in DOM structure. Gharachorlu conducted a small-scale study on 20 websites and computed averages to conclude that rules containing more than four simple selectors is considered too specific.

```
body div > span .boat h3 {
```

```
    color: red;
}
```

3.2.6 Too Specific Selectors Type II (High Specificity Values)

Similar to Too Much Cascading, selectors with high specificity values indicate a high reliance on the structure of the DOM but from the point of view of the *types* of simple selectors used rather than simply the number of units. Gharachorlu analyzed the impact levels of different selector types on the specificity number `(a,b,c,d)` (ignoring inlined rules) and concluded that a complex selector is too specific if the number of id selectors is greater than one, number of class selectors is greater than two, or the number of element units is greater than three.

```
#listA #listB {
    border-style: dotted;
}

.example1 .example2 .example3 {
    font-size: large;
}

html body span div {
    color: pink;
}
```

3.2.7 Universal Selectors

Use of the universal selector '`*`' slows down rendering because it selects the entire document. It can also short-circuit inheritance because it has specificity of 0 whereas

inherited values have no inheritance [49].

```
* .desktop {
    display: block
}
```

3.2.8 Selectors with ID and at Least One Class or Element

Selectors containing an ID already reference a single unique HTML element, so adding a class or element to the selector adds unnecessarily to cognitive load and to rendering time. If HTML elements below a certain element needs to be selected on the DOM, classes alone or in conjunction with element selectors should be used, or an ID should be placed directly on the element to be targeted.

```
span #block {
    top: 50\%;
    margin-top: -5.5px;
}
```

3.2.9 Selectors with Erroneous Adjoining Pattern

Two selectors are considered to be erroneously adjoined when the white space between two simple selectors is removed. The result is that these selectors are not parsed correctly by the browser and constitute bloat in the code without any style properties being applied.

```
spandiv {
    margin-right: 0.3em;
}
```

3.2.10 Too General Selectors

Selectors are too general when they target too much of the DOM to be of any real use, often resulting in undoing style (essentially changing a style attribute back to its original or default value after its been changed once already). Gharachorlu proposed the following simple selectors to be too general when used as standalone selectors: `html, head, body, div, header, aside`.

```
html {
    border: none;
    padding: 0px;
}
```

3.2.11 Properties with Hard-Coded Values

Properties with Hard-Coded values are those which use constants ("magic numbers") to specify values. Property values are better specified relatively using CSS functions like `calc()` or units like `ems` or `percentages`.

```
body h4 {
    font-size: 16px;
    padding: 3px 20px;
}
```

3.2.12 Properties with Value Equal to None or Zero (Undoing Style)

Undoing Style as defined by Gharachorlu are properties with values equal to `0` or `none`. However, as mentioned by Punt et al., this definition isn't quite appropriate because it doesn't cover the truly broad set of cases that could lead to undoing style. Undoing

Style as defined by Punt et al. are properties applying to an HTML element that have been set or defaulted to some initial value, overridden any number of times to different values (as expected), and then subsequently changed back to the initial value [54]. This is harmful because more CSS is written to effectively reduce the styling and it works against the cascade because properties only need to be overridden when they should be adopting some new value, not reverting back to the original.

The definition used by CSSNose produces false positives because a `none` value for property displays is a standard way for web developers to hide an element and reset styles are used to remove inconsistencies in the presentation defaults used by browsers. It also produces false negatives because a style can also be undone by any valid value for a property, whereas CSSNose only looks at `0` or `none` values. The definition used by CSSNose was kept in the analysis, however, because it still serves its purpose in providing information as a feature in differentiating CSS behavior.

```
.chart {
    border: none;
    margin : 0;
    text-shadow : none;
}
```

3.3 CSS Metrics

This section introduces and describes the CSS metrics collected in the empirical study. Figure 7.4 lists and gives a summary of these metrics. The smells in this list are those collected by the tool CSSNose.

3.3.1 External Rules

External Rules is the number of rules contained in an external stylesheet and linked to from an HTML document. See Figure 3.1 for a visual example.

3.3.2 Total Defined CSS Rules

Total Defined CSS Rules is the number of all defined CSS Rules including inlined, embedded, and external. The inlined style properties extracted from a single DOM element counts as a single rule.

3.3.3 Total Defined CSS Selectors

Total Defined CSS Selectors is the number of all CSS selectors defined in all CSS external or embedded rules. Inlined CSS rules do not contain selectors and aren't counted here.

3.3.4 Ignored CSS Selectors

Ignored CSS Selectors is the number of selectors that were ignored when extracting code smells and metrics from CSS code due to errors in parsing the selector.

3.3.5 Undefined Classes

Undefined Classes are the number of classes used in selectors which do not successfully target a corresponding class included in a DOM node.

3.3.6 Matched Selectors

Matched Selectors is the number of selectors that target existing DOM nodes.

3.3.7 Unmatched Selectors

Unmatched Selectors is the number of selectors that do not target any DOM elements. Note that unmatched selectors are not considered code smells because it's perfectly reasonable for selectors to *become* matched in successive DOM state changes. It's simply the rules with selectors that *never* become matched that should be removed for being unnecessary.

3.3.8 Effective Selectors

Effective Selectors is the number of selectors that target existing DOM nodes and have style properties that are successfully applied.

3.3.9 Ineffective Selectors

Ineffective Selectors are those which do apply to a target DOM element but which are overridden by other rules in the cascade. Note that ineffective selectors are not considered code smells because it's perfectly reasonable for selectors to *become* effective in successive DOM state changes. It's simply the rules with selectors that *never* become effective that should be removed for being unnecessary.

3.3.10 Total Defined CSS Properties

Total Defined CSS properties is simply the number of properties defined in all CSS rules, whether external, embedded, or inlined.

3.3.11 Unused Properties

Unused properties are those which do not get applied to the targeted DOM elements specified by the selector of the rule in which they reside. Unused properties are a

natural result of the cascade as property values override each other.

3.3.12 Ignored Properties

Ignored Properties is the number of properties that were ignored when extracting code smells and metrics from CSS code due to errors in parsing the property.

3.3.13 Files with CSS Code

Files with CSS code is the total number of files containing CSS code, either from external style sheets or HTML containing embedded CSS code (but not inlined).

3.3.14 Lines of Code (LOC)

Lines of code are the number of lines contained in the CSS source code including inlined, embedded, and external.

Chapter 4

SEPARATION OF CONCERNS IN CASCADING STYLESHEETS

This chapter introduces three types of coupling between HTML and CSS and identifies the relevant code smells and metrics which have tracked Separation of Concerns in CSS thus far. The ideas presented in this chapter can be considered the result of applying Separation of Concerns as a software engineering principle to CSS.

4.1 Overview

CSS was originally designed to remove the responsibility of style authoring from HTML [41] [49]. Ideally, one or more separate style sheets in a web project are linked to from an HTML file to specify style. The original line of thinking was that both HTML and CSS documents could utilize the semantic, standardized HTML elements [1] (such as `<p>` for paragraph, `<ul>` for an unordered list, etc.) as an agreed-upon interface between content and style and thus have CSS documents be interchangeable with any other HTML documents. The major benefit from this approach is that individual style sheets can be reused to supply style for a variety of different HTML documents.

However, the reality is that content authoring is actually quite complex, to the point that the semantic HTML elements aren't sufficient enough to differentiate all content structures that require different styling [37]. This led to the introduction of more style-motivated HTML elements such as `<span>` and `<div>`. This practice, however, is not consistent with the original motivating principle of CSS: separation of content authoring from style authoring. Even further, the combination of client-

[1] **Standardized HTML elements** are those which are specified and maintained by the World Wide Web Consortium (w3c)

side language features and leniency practiced by browsers enables the violation of well-established software engineering principles [50]. For example, the `<style>` and `<script>` elements in HTML are capable of supporting embedded CSS and JavaScript code snippets respectively which violates Separation of Concerns. Another example is that CSS and HTML are not always strictly required to conform to the current language standards maintained by W3C, which sometimes results in HTML code with missing closing tags or CSS code with unmatched selectors [32].

The result is that many web developers can and do violate software engineering principles such as Separation of Concerns and it has very real consequences on maintainability [20] [32]. Critical to note, however, is that Separation of Concerns is much more than simply the mixing of CSS code with HTML code. Separation of Concerns as a principle seeks to control complexity in software projects by regulating the degree of coupling and cohesion between software components. Code mixing is only one facet of coupling, and so Separation of Concerns as it relates to CSS can be more rigorously applied to derive a broader scope of the relationship between CSS and HTML beyond the mixing of code. Keller and Nussbaumer's notion of abstractness [37] in CSS is what comes closest to defining this broader scope and what serves as an inspiration for the classifications of coupling between CSS and HTML presented in this chapter (see section 5.2 in Related Work). Although Keller and Nussbaumer's defined Average Scope and Universality metrics are used in the context of the abstractness of CSS and are also very important for that vein of analysis, these metrics can also be re-framed to be more consistent with Separation of Concerns. These metrics are detailed further in the following sections.

4.2 Motivation

Given that CSS was designed to separate style authoring from HTML and given that there are features and common practices in modern front-end web development which currently facilitate the violation of well-established Software Engineering principles such as Separation of Concerns, solutions concerning these language features or developer practices must be investigated. However, although there exist code smells and metrics which track various facets of Separation of Concerns, they aren't sufficient in providing a holistic view of the principal applied to CSS.

In order to promote the active awareness of Separation of Concerns as a Software Engineering principle and encourage discussions on the topic in the web development community, it's important to give developers the proper mental model and terminology to discuss Separation of Concerns as it applies to front-end web development and, more specifically, to CSS. We attempt to provide this mental model and terminology by classifying and formally defining three types of coupling between CSS and HTML that can be present in a web project. An excessive amount of any of these three types of coupling can be considered violations of Separation of Concerns. Although cohesion is also an important element of separation of concerns, expressing the violations purely in terms of coupling is much more convenient and leads to a more easily understood classification.

4.3 Separation of Concerns & Coupling

Separation of Concerns is a general principle of Software Engineering that seeks to control complexity in software projects by separating a program into distinct sections, such that each section addresses a separate concern. The result of applying Separation of Concerns to a program is modular code that also demonstrates *low coupling*

and *high cohesion*. Coupling and cohesion are two general principles of Software Engineering that are concerned respectively with the level of dependency and the level of semantic or logical consistency between two software modules. Coupling is usually contrasted with cohesion - that is, low coupling often correlates with high cohesion and vice versa. Coupling is considered to be high when, given two software modules A and B, a change to module A necessitates a change to module B. (See Chapter 2, Background, for more information).

4.4 Co-occurence Coupling

Co-occurence coupling can be defined as the dependency of two or more software components on each other due to the interchanging of code between components. Ideally, code used to define a particular software component is allocated its own file or section of a file and therefore not mixed with code that is used to define other software components. However, some software paradigms allow or even encourage the mixing of code from different software components, thus breaking separation of concerns.

In modern front-end web development, HTML allows co-occurence coupling with CSS in two forms: inlined and embedded CSS. *Inlined CSS* utilizes the `style` property of an HTML element to specify the style of that particular element (and, due to inheritance in CSS, of all descendent elements) within a particular HTML document. *Embedded CSS*, also called an *Internal Style Sheet*, utilizes the `<style>` HTML tag and CSS selectors to specify style of an arbitrary number of HTML elements within a particular HTML document. Otherwise, CSS is included in an *External Style Sheet* and linked to from the HTML file. See Figure 3.1 for a visual explanation of inlined, embedded, and external CSS rules. In a web project, high co-occurence coupling occurs when a majority of the style code for a website appears as inlined and embedded

CSS code in the HTML documents. Low co-occurence coupling occurs when nearly all of the style code is included in external style sheets and linked to from the HTML.

The above definition of co-occurence coupling is consistent with the earlier definition of coupling in that, given two HTML files A and B which share a block of inlined/embedded style code C, a change to the style code C necessitates a change to files A and B for as long as the two are intended to remain stylistically similar. Important to note is that the "concern" under consideration here when applying separation of concerns is content vs style rather than the concern of one page vs another page within a website. High co-occurence coupling can negatively impact maintainability in CSS because changes to the style code of one HTML documents requires changes to the style code of all HTML documents that are intended to remain stylistically similar.

Co-occurence coupling is currently kept account of through the amount of Inlined Rules, Emedded Rules, and External Rules included in a web project. These metrics are sufficient to measure co-occurence coupling especially when they're represented as percentages of the total defined number of rules because they show the distribution of style code co-occurring with HTML code and existing in external stylesheets.

4.5 Structural Coupling

Structural coupling can be defined as the dependency of two or more software components on each other due to a reliance on the structure or organization of other components. The knowledge of structure is necessary to a degree if software components are to interact with each other - one example being a reliance on the "structure" of a function via its return value. In this case, the caller is coupled with the callee via the return value: should the callee change the type of object it's returning, the caller must also change its use of that return value. For example, if a function `loadData()`

makes the call to a function `openFile()` under the assumption that it will return a `File` object, it can be considered to rely on the structure of the `openFile()` function. Should the `openFile()` function change its return type to instead be a file descriptor, the `loadData()` function much also be changed to accommodate the new return type. Although necessary to a degree, a software component ideally relies as little as possible on the structure of other components, abstracting when possible.

In modern front-end web development, HTML and CSS allow structural coupling in the form of complex selectors. Complex selectors can be used to select highly specific sections of the DOM that are not likely to appear in many other content document instances. In a web project, high structural coupling occurs when a majority of the selectors rely on the structure of the DOM through highly specific selectors. Low structural coupling occurs when a majority of the selectors rely less on structure by utilizing less specific selectors and class/ID selectors in cases when highly specific targeting is needed.

The above definition of structural coupling is consistent with the earlier definition of coupling in that, given an HTML document A and an external style sheet B where B provides style for A, a change to the structure of file A necessitates a change to file B. A web project demonstrates a high level of structural coupling when the CSS selector is highly complex, referring to a deeply nested document tree structure. High structural coupling can negatively impact the reusability of style sheets because a style sheet tied too closely to the structure of any one HTML document cannot be easily applied to other HTML documents. However, low structural coupling creates another set of problems in the form of an over-reliance on identifiers, which can be called syntactic coupling (described in the next section).

Structural coupling is currently kept account of through the Too Specific Selectors Type I, Too Specific Selectors Type II, and Too General Selectors code smells. The

former two describe selectors that contain either too many simple selectors or more than a specified amount of one of the four types of simple selectors. The latter describes selectors that are one of a set of element selectors with a very wide scope in the HTML document. (See Section 3.2 for descriptions of these code smells.) Although they consider the specificity or generality of individual selectors, they fail to consider coupling at the file-level. However, the Average Scope metric defined by Keller and Nussbaumer can be used to effectively track structural coupling on the file level because it measures the average number of DOM nodes that are targeted with the selectors in a stylesheet [37].

The full definition of Average Scope is defined below:

$$AverageScope = \frac{\sum_{i=1}^{m} \#\text{ of elements in scope of selector i}}{m \cdot n} \tag{4.1}$$

where $\mathbf{m} = \#$ of selectors

and $\mathbf{n} = \#$ of elements in the document tree

4.6 Syntactic Coupling

Syntactic coupling can be defined as the dependency of two or more software components on each other due to a reliance on the programmer-defined identifiers of other components. The knowledge of identifiers is certainly necessary if software components are to interact, as it's what forms the basis of the notion of a software interface or API. For example, the caller of a function must know the identifier of the callee in order to invoke it. However, if every atomic action in a code block (such as addition) were decomposed into a function and the corresponding identifier `add(x, y)` used, the code block would become too highly saturated with identifiers. In this way, syntactic coupling highly relates to the idea of the decomposition of code into

smaller, modular functions in that a higher degree of decomposition results in higher saturation of identifiers. Ideally, a software component balances the decomposition of code into function calls (and therefore the use of identifiers) of other components with explicit code blocks.

In modern front-end web development, CSS and HTML allow syntactic coupling in form of class and ID attributes/selectors. Class and ID attributes specified in HTML tags can be used in CSS as selectors to target specific DOM nodes bearing the corresponding class or ID identifier. In a web project, high syntactic coupling occurs when selectors are composed entirely of class or ID identifiers, targeting highly specific DOM nodes. Low syntactic coupling occurs when selectors rely only on DOM *structure* and are composed entirely of element selectors joined by combinators.

The above definition of syntactic coupling is consistent with the earlier definition of coupling in that, given an HTML document A and an external style sheet B where B provides style for A, change to the class or ID attribute identifier in A necessitates a change to the corresponding selector identifier in B. High syntactic coupling can negatively impact maintainability as identifiers must be appropriately synced and updated for each HTML document to which a style sheet applies.

Syntactic coupling doesn't currently appear to have many metrics that track it except for perhaps the code smell IDs in Selectors captured by CSS Lint which captures any selectors containing. The use of IDs in CSS in general is somewhat controversial though as good arguments are made within the community both for and against their use, and this code smell isn't enough to categorize syntactic coupling on the file level. However, the Universality metric defined by Keller and Nussbaumer can be used to effectively track syntactic coupling because it measures the extent to which a stylesheet relies on non-element selectors to achieve it's targeting of DOM nodes [37].

The full definition of Universality is defined below:

$$Universality = \frac{\#\text{ element selectors in SSel}}{\#\text{ selectors}} \tag{4.2}$$

where **SSel** is the set containing all osolated simple selectors and the last simple selector from each combined selector

4.7 Coupling - The Reality

In theory, one would do well to reduce the amount of co-occurence coupling as much as possible and simply balance the semantic and syntactic coupling. Ultimately, the concerns of content, style, and behavior specification are connected, and some degree of semantic and syntactic coupling will always be necessary. Keller and Nussbaumer put it best when they say, "Although content and presentation can be separated physically in different files, in a logical way they are closely connected" [37]. This line of reasoning applies to front-end web development as whole as well: although HTML, CSS, and JavaScript were designed to separate the concerns of web development, they must still interact with each other in order to achieve the end goal of creating a website.

The notions of Average Scope and Universality presented by Keller and Nussbaumer provide an excellent basis for thinking about and measuring structural and syntactic coupling respectively. As [37] mention, "the abstractness-factor can be considered as an indicator of the degree of the separation of content and presentation".

Chapter 5

RELATED WORK

Although CSS has historically not been given much attention in academia [13] [20] [30] [42] [48] [55], there are some works that explore CSS metrics, code smells, and refactoring opportunities. The works in these areas admittedly do not fall clean-cut into these categories, so we place them according to where we feel the priorities of the papers lie.

5.1 Web Crawling

Mesbah et al. develop a tool called Crawljax for crawling [1] AJAX web applications [47]. Traditional static websites assign a hypertext link and URL for each unique state of the user interface. However, the state of the user interface in AJAX applications is determined dynamically though changes to the DOM as a result of executing JavaScript code. Thus, Crawljax creates a state-flow graph to represent the various possible states of a web application, detects a list of clickable elements from the DOM of the current application state, recursively navigates the website to build the graph by executing the JavaScript code associated with those clickable elements, and, finally, generates a static HTML document of each state from a saved snapshot of the DOM. This website navigation is achieved through the use of Selenium WebDriver [26], a browser driver [2] used for automated web testing. A case study by the authors confirms that Crawljax performs well on six sample AJAX web applications. The primary tool used for data collection in this work, CSSNose, performs dynamic crawling using Crawljax.

[1] **web crawling** is the programmatic accessing of content and functionality on the web

[2] a **browser driver** is a program that sends commands to a web browser and retrieves results, allowing for the automation of tests previously requiring a user interacting with a browser UI

5.2 CSS Metrics

Software metrics are measures of the extent to which a particular piece of software possesses some property [27]. Metrics are important for quantifying the quality of software for maintenance purposes in software engineering. Adewumi et al. define and implement a tool to collect complexity metrics for CSS such as Rule Length and Number of Cohesive Rule Blocks and validate them against the Kaner Framework for Software Engineering Metrics [13] [14]. Mesbah and Mirshokraie build a tool, Cilla, which detects unmatched and ineffective selectors, overridden declaration properties, and undefined class values in CSS code, finding that an average of 60% unused CSS selectors in deployed applications [48]. Cilla's metrics are included in the tool CSSNose [32], which is used in this study.

Keller and Nussbaumer define an interesting metric for Abstractness based on the notions of Universality and Average Scope of selector statements [37]. Universality refers to the ratio of simple selectors that carry high semantic value, namely, element selectors, to those that do not, regardless of whether a simple selector is standalone or finishes a complex selector. Average Selector Scope refers to the average of the number of HTML elements that are included in the scope of a selector per rule block. They define Abstractness as the result of the floor function between Universality and Average Selector Scope. Keller and Nussbaumer then use the abstractness metric to highlight a difference in quality between human-authored and machine-generated CSS code. They find that for 90% of 100,000 HTML (mostly human-authored) documents, the abstractness of the corresponding CSS falls between 0.13 and 0.49 and that all machine-generated documents had an abstractness factor of less than 0.13. They furthermore find a very weak correlation between a defined CSS complexity metric and abstractness.

5.3 CSS Code Smells & Defects

Nguyen et al. detect and analyze embedded server-side code smells in dynamic web applications [50]. They establish a negative correlation between these code smells and code maintainability metrics. Gharachorlu defines and aggregates a set of 26 CSS code smells, integrating the tools *CSSNose* [48], *CSS Lint* [7], and *W3C CSS Validator* [61] into a single smell-detection framework [32]. The framework is used to dynamically crawl 500 random websites in order to inspect their CSS practices and analyze the prevalence, the extent, and the correlation between code smells and various CSS metrics. Gharachorlu's experimental setup and codebase served as an inspiration and basis for this work. Gharachorlu finds that 99.8% of the websites studied contain at least one type of CSS code smell [32]. In order to obtain an understanding of CSS behavior, the scope of this study is expanded to include 5,525 websites. Serdar Biçer and Diri collect some CSS metrics at the rule level in order to train multiple machine learning classifiers for defect prediction on CSS code [57]. Geneves et al. implement a static debugging tool that's able to identify defects in CSS code without being applied to a specific document instance.

5.4 CSS Refactoring & Tool Support

Quint and Vatton identify the major style issues web authors face and implement solutions to these issues in a hybrid IDE and web browser named Amaya [55]. They account for CSS code generation and syntax checking, retrieving elements affected by a style rule, retrieving rules applied to a given element, visual validation of style on various document instances, and the loading of remote web resources for use in the IDE. They additionally extend their support to documents that use (X)HTML, SVG, MatchML, generic XML, and documents that use several of these languages.

Bosch et al. develop a tool to detect and delete redundant CSS rules [20]. They do so by developing equivalence relations via tree logic through the static analysis of CSS selectors. The result is that *masked* rules, those that remain inactive due to the presence of equivalent rules which have preference, and *verbose* rules, those that may be active at times but don't apply any additional styling, are removed from the style sheet and the size of the file is thereby reduced. They demonstrate an average size reduction of single CSS files from well-known sites of 7.75% with a maximum of 17.83%, and show that some information about HTML documents that use multiple style sheets can increase size reduction on any one CSS file up to 30%.

Punt et al. investigate the *undoing style* code smell in CSS and develop a tool to detect and automatically refactor varying patterns of undoing style. Undoing style refers to style properties that are initially set to a value A (either explicitly by the author or implicitly via a language feature), then overridden to a value B, possibly multiple times, and then set back to the original value of A. The authors refer to this as the A?B*A pattern, and they check for this pattern by analyzing the cascade of the style applied to a single HTML element, observing when a property is overriden multiple times. They can then guarantee that the style semantics for this element are preserved on any proposed refactor that fulfills the following two preconditions: (1) the most specific part of the selector is an ID or class and (2) the elements that the reset rule (the last undoing rule) applies to are the subset of the elements the initial rule applies to. On a dataset of 41 well-known websites, they found the ratio of undoing style detections to the number of style rules to be 25% with 2060 occurrences of undoing style on average, an average of 11 opportunities for automatic refactoring per webapp with a high of 62, and only 38 out of 8789 semantic changes caused by refactoring.

Mazinanian et al. identify three different types of CSS duplication, namely, declarations with lexically identical values for given properties, declarations with seman-

tically equivalent values for given properties, and a set of individual-property declarations which are equivalent to a single shorthand-property declaration, and develop a static analysis tool to automatically refactor them [46]. Their approach consists of parsing and normalizing CSS rules, extracting refactoring opportunities through the use of association rule mining algorithms, ranking the resulting refatorings by their computed file reduction potential, and ensuring semantic preservation on refactorings by preserving the order dependencies of rules through the expression of a constraint satisfaction problem. An experimental run on 38 web applications finds that on average 66% of style property declarations are repeated at least once, there are on average 62 refactoring opportunities associated positively with file size reduction, and on average an 8% reduction in file size of examined CSS files can be achieved by applying refactorings.

Hague et al. present an abstraction of HTML5 applications based on monotonic tree-rewriting and proceed to study its "redundancy problem" in order to detect redundant CSS rules [33]. Unlike dynamic analysis techniques, their technique will only report rules which are definitely redundant, but as such will not report rules that *may* be redundant based on a particular DOM state. In a case study of 8 real and invented web pages, TreePed correctly identified all definitely redundant rules.

5.5 Contributions of this Work

None of these studies appear to consider frameworks as a perspective in their analyses of CSS code smells and defects. Furthermore, although [32] establishes a correlation between some CSS metrics and the number and types of smell instances, no other work has studied the correlation between code smells themselves and between smells and metrics. In this study, we investigate the prevalence of CSS code smells in websites built with various web frameworks, attempt to identify a pattern of CSS

pattern in web frameworks, and study the correlation between and among CSS code smells and metrics. This study has the potential to enhance the tools presented in this chapter with the feature to detect the framework a website was built with and either recommend solutions for CSS maintenance based on the observed correlations between CSS code smells and metrics or perform an automatic refactoring.

Chapter 6

EMPIRICAL STUDY

This chapter details the empirical study setup in order to answer the proposed research questions. The following research questions are under consideration:

RQ1 How prevalent are code smells in websites built with different web frameworks?

RQ2 Can we recognize a pattern of CSS behavior in certain web frameworks? Can a set of CSS code smells and metrics collected from a website be used as a unique identifier for the web framework used to develop the website?

RQ3 Does there exist a correlation between CSS code smells and metrics collected from a website?

In order to investigate the CSS behavior patterns of various frameworks and observe correlations between code smells and metrics, we collect a sample of websites from various frameworks, collect code smells and metrics from the CSS code of each website (see Table 7.4), train a model to predict the framework from a feature set of CSS code smells from a website, and perform various clustering tasks on the code smells and metrics.

6.1 Overview

The empirical study consists of two main phases: (1) data collection and (2) data analysis. Section 6.2 details the process of data collection and Section 6.3 discusses the analysis methods. A collection of scripts and various software tools were developed and integrated to setup a software pipeline for running the experiment. Figure 6.1

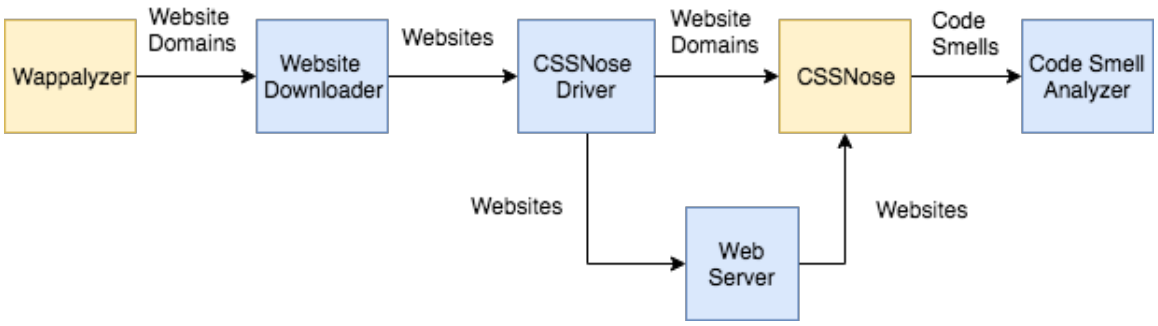

Figure 6.1: The software architecture setup for the experiment.

gives an overview of the software architecture. The pipeline can be explained as followed:

First, lists of website domains built with 19 different web frameworks are obtained from the web platform analysis tool Wappalyzer [16]. These lists of domains are fed to a Website Downloader script which downloads local copies of several hundreds of websites from each web framework. The downloaded websites are passed to a CSSNose Driver script, which for each website instructs a Web Server to serve the website on localhost and passes the domain of the website to an instance of CSSNose [32]. CSSNose dynamically crawls the website, aggregates CSS code from each new DOM state it encounters, and outputs a set of CSS code smells and metrics collected for that website in the form of a report. The code smell reports for all websites across all frameworks are then parsed and analyzed by a Code Smell Analyzer script which outputs descriptive statistics (RQ1), a classification model (RQ2), and several clustering models (RQ3).

6.2 Data Collection

This section describes the process of collecting websites for analysis along with the extraction of code smells and the tool used to accomplish this, CSSNose.

Table 6.1: The frameworks under consideration in this study as well as the language of development and the number of code smell samples collected for each. "CMS" stands for Content Management System.

#	Language	Framework	# Samples
1	.NET	ASP.NET	212
2	CMS	Drupal	331
3	CMS	Adobe Experience Manager	240
4	CMS	Wordpress	272
5	CMS	Joomla	285
6	HTML/CSS	Bootstrap	415
7	Java	Apache Wicket	210
8	Java	Google Web Toolkit	238
9	JavaScript	AngularJS	375
10	JavaScript	React	242
11	JavaScript	Meteor	753
12	Perl	Mojolicious	97
13	Perl	Dancer	507
14	PHP	CakePHP	203
15	PHP	Laravel	250
16	Python	Django	206
17	Ruby	Ruby on Rails	206
18	Scala	Play	257
19	Scala	Lift	226
			5,525

6.2.1 Collecting Websites

Local copies of hundreds of websites from each of the 19 frameworks were collected. Lists of website domains that were built using different frameworks were acquired from the web analytics platform Wappalyzer [16]. Frameworks were selected primarily based on popularity and language distribution [29]. See Table 6.1 for the full list of frameworks. Collecting local copies of websites was important to ensure reproducability of the experiment, as the live version of a website can change. A recursive crawl was restricted to a depth of 3, as that should be enough to provide a representative sample of the CSS used on the website. A random wait between 20-40 seconds was introduced between the download of each resource to show mercy to web servers and to attempt to avoid throttling from websites detecting high download frequency. Most files deemed unnecessary for crawling were rejected, including images, pdfs, archives, etc. `Robots.txt` was obeyed in these calls for the ethical collection of data. For further explanation of technical details and problems encountered during collection, see the repository released for this study.

6.2.2 CSSNose

CSS code smells and metrics are collected using the tool CSSNose implemented by Gharachorlu [32]. CSSNose is a code smell detector built in Java using Cilla [48], which is an open source tool for supporting style code maintenance [48]. Cilla utilizes a dynamic web crawler, Crawljax [47], to detect and navigate DOM state changes in a website - parsing and aggregating CSS rules along the way before analyzing them. CSSNose is reported to define and aggregate a set of 26 CSS code smells, integrating the analysis tools *CSS Lint* [7] and *W3C CSS Validator* [61] into a single smell-detection framework [32]. It was additionally reported to have been evaluated on 20 websites with 100% recall and a 99.6% precision rate, demonstrating high accuracy.

Although some of the smells described in [32] may be considered "controversial" in the web development community, as mentioned briefly by Mazinanian [43], they are still useful to this study because they still help to characterize CSS behavior, which is the ultimate goal.

CSSNose required some adjustment initially to get running, however. We invested some effort in bringing the tool to a state in which it could be used to collect the eight code smells proposed in the CSSNose paper as well as an additional four code smells and 14 descriptive statistics included in the resulting tool from Cilla. Most code smell collections were left as they were originally implemented and some bugs and errors were corrected. See Appendix A for a list of code smells that were renamed from CSSNose's original paper [32]. However, the code smells and metrics obtained from W3C Validator and CSS Lint were not used in the final study. Aside from initial refactors to get the tool running, important changes made include:

1. altering the "Lines of Code" (LOC) calculation
2. ensuring that URLs referring to a site's index.html are not counted twice
3. **adding a feature to collect inlined CSS (as previously only embedded rules were collected from HTML and counted as non-external CSS)**

6.2.3 Code Smell Collection

A script is setup to serve the local copies of websites and pass domains to CSSNose. Only the CSSNose output of successful runs are used for the analysis of data. Important to note is that even a successful run of CSSNose may encounter exceptions during website navigation as well as ignore CSS rules as a result of parsing errors. Ultimately, a successful run is considered to be any run for which a non-zero amount of code smells or metrics are collected. Another script is setup to then parse the CSS-

Nose output files from each framework and collect the values of the CSS code smells and metrics. See Table 7.4 for a list of the code smells and metrics for 2 collected websites. Table 7.8 shows example values for code smell and metrics. Any CSSNose output file with 0 lines of CSS code or an invalid address is counted as unsuccessful and ignored. The code smell and metric values are then aggregated and assigned a label based on the framework of the website they came from.

6.3 Analysis Methods

This section describes the analysis methods used in the empirical study for answering the proposed research questions. Statistical and machine learning models from scikit-learn [56] are used for aggregating and displaying descriptive statistics (RQ1), predicting framework from CSS code smells and metrics (RQ2), and code smell clustering (RQ3).

6.3.1 Predictive Model

In order to determine whether a set of CSS code smells and metrics collected from a website could be used as a unique identifier for certain web frameworks, a number of supervised learning techniques for fitting a predictive model are selected and evaluated. In the supervised learning paradigm, a mathematical model is constructed which can predict which class a particular object belongs to given multiple examples of objects belonging to various classes. The model is constructed by learning a mapping between objects and their classes based on the features that describe the properties of those objects. In this case, a single website is considered to be an object, the set of 26 code smell and metric values collected from the CSS code of that website are the features, and the framework used to develop the website is the class to be predicted. See Table 7.8 for an example of two real feature sets selected from the

training data along with their predicted and actual frameworks.

Features are normalized by transforming each feature's value to the *z-score* (see Section 7.1). The accuracies of following classifiers were all assessed on the training data:

- K-Nearest Neighbors (with labels)
- Linear SVM
- RBF SVM
- Decision Tree
- Random Forest
- Neural Network
- Adaboost
- Naive Bayes
- Quadratic Discriminant Analysis

6.3.2 Clustering Model

In order to determine the correlations among code smells and investigate the relationships between websites, K-Means and agglomerative clustering methods are used. K-Means is an unsupervised clustering technique that determines similarity by leveraging the vector space model to represent features of size n as vectors in n-dimensional space and then calculating the Euclidean distance between objects, outputting k clusters. Agglomerative clustering similarly leverages the Euclidean distance between

objects but instead generates successive clusters in a hierarchical fashion - beginning with each object as its own cluster and recursively joining neighboring clusters together.

Three approaches to clustering using K-Means are investigated. Firstly, each set of code smells from a website is treated as an individual object in order to investigate what relationships exist between websites developed by various frameworks (see Figure 7.1). For the next two approaches, the mean of each code smell is computed for each framework as feature sets for clustering (see Tables 7.2-7.3). The second clustering approach treats the collection of means of a single code smell across all frameworks as an individual object in order to investigate the relationship between various code smells (see Figure 7.2a). The third clustering approach treats the collection of means of all code smells for a single framework as an individual object in order to investigate the relationship CSS code smell behaviors exhibited by various web frameworks (see Figure 7.2b). This process is then repeated with agglomerative clustering for all but the first clustering task (by individual website) (see Figures 7.3 and 7.4).

Chapter 7

RESULTS AND DISCUSSION

In this chapter, we introduce some metrics used for aggregating data and proceed to address each research question and discuss the results of the empirical study. See Table 7.4 for a quick reference to code smell and metrics definitions.

7.1 Metrics

The following definitions of precision, recall, and F1-score typically apply in the context of *binary classification* - predicting whether an object either belongs or does not belong to a class.

- **True Positive (TP)** - an object correctly identified as belonging to a class.
- **False Positive (FP)** - an object incorrectly identified as belonging to a class (when it actually does not belong)
- **False Negative (FN)** - an object incorrectly identified as not belonging to a class (when it actually does belong)

However, as we consider a website to be an object and a framework to be a class (of which we have 19), we must calculate precision, recall, accuracy, and F1-score for each framework f_i which belongs to the set of frameworks F. In the following equations, w_{ii} is the set of websites predicted to belong to framework f_i which actually belong to f_i (true positive), w_{ij} is the set of websites predicted to belong to framework f_i which actually belongs to f_j (false positive), w_{jj} is the set of websites predicted to belong to framework f_j which actually belongs to f_j (true negative), and w_{ji} is the set of websites predicted to belong to framework f_j which actually belongs to f_i (false

negative), (where $i \neq j$ and |F| is the cardinality of F). Based on this explanation we can define precision, recall, accuracy, and F1-score of each framework $f_i \epsilon F$ where $1 \leq i \leq |F|$ as follows:

Precision is a measure of a classifier's correctness when claiming that an object belongs to a class.

$$Precision(f_i) = \frac{w_{ii}}{w_{ii} + w_{ij}} \tag{7.1}$$

Recall is a measure of a classifier's completeness in identifying objects which belong to a class.

$$Recall(f_i) = \frac{w_{ii}}{w_{ii} + w_{ji}} \tag{7.2}$$

F1-Score is a combined measure of classifier's precision and recall.

$$F1(f_i) = \frac{2 \cdot precision_i \cdot recall_i}{precision_i + recall_i} \tag{7.3}$$

Accuracy is a measure of a classifier's overall ability to correctly identify belonging and non-belonging objects.

$$Accuracy = \sum_{i=1}^{|F|} \frac{w_{ii} + w_{jj}}{w_{ii} + w_{jj} + w_{ij} + w_{ji}} \tag{7.4}$$

Support is simply the number of positive objects included in the test. For example, in a binary classification task containing 100 objects, if 25 objects are actually positive and 75 are actually negative, then the support is 25.

Population Standard Deviation is a measure of the amount of variation in a dataset. It's either zero, in which case all points carry the same value, or it's an unbounded positive value. [*$\{x_1, x_2, ..., x_N\}$ are the values of the population items, $\overline{x}$ is the mean value of these items, N is the size of the population.*]f

$$StandardDeviation = \sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}} \tag{7.5}$$

Standard (Z) Score is the signed, unbounded number of standard deviations that a single value is away from the mean of that value's population. Positive indicates that the value falls above the population mean while negative indicates it falls below the population mean. [μ *is the mean of the population,* σ *is the standard deviation of the population.*] $z_i = \frac{x_i - \mu}{\sigma}$

$$z_i = \frac{x_i - \mu}{\sigma} \tag{7.6}$$

7.2 RQ1 - Code Smell Prevalence in Web Frameworks

The results reveal that code smells are highly prevalent in websites built with web frameworks. Table 7.1 shows the percentages of websites built with a certain framework that contain at least one instance of each code smell or metric. These percentages are thus strictly a measure of the *prevalence* of each smell/metric and do not consider the extent or severity. To illustrate, the CSS code smell Embedded Rules for the framework Google Web Toolkit has a value of 39.5%. This means that 39.5% of the collected websites built with Google Web Toolkit contain at least one Embedded Style Rule in their HTML.

The high prevalence of CSS code smells is evidenced by the consistently high percentages for smells across all 19 frameworks in Table 7.1. Percentages seldom dip below 70%, and the majority of the values appear to be in the nineties. These results are consistent with Gharachorlu's finding that 99.8% of the websites studied contain at least one type of CSS code smell [32], also demonstrating high prevalence. One interesting observation from this table is that the code smell with the most variability is the **Embedded Rule** smell which has a low of 39.50% in Google Web Toolkit and a high of 90.07% in Wordpress. By contrast, the code smell with least amount of variability is the **Properties with Hard-Coded Values** smell with a low of 96.91% in Mojolicious and a high of 99.58% in Adobe Experience Manager.

One reason for this may be that it simply takes more effort to reduce the amount of Hard-Coded properties than it does to reduce the amount of Embedded Rules. Another reason may be that the web development community cares more avoiding Embedded Rules than Hard-Coded properties.

In addition to high prevalence, the results in Tables 7.2-7.3 show a high degree of variance in the code smell and metric values. A *mean* measurement on the table is the average value of the specified code smell/metric for all of the websites from the specified framework. A *standard deviation* measurement represents the variance of the specified smell/metric when compared to the mean value for that smell/metric from the specified framework. Important to note is that both the mean and the standard deviation are expressed in the units of the smell/metric that they represent, which can be found in Table 7.4. A consequence of this is that the mean and standard deviations of different smells/metrics can't be directly compared. To resolve this, the *z-score* measurement is included, which represents the z-score of the mean relative to the means of that smell/metric in other frameworks. Note that the "population" under consideration with the mean and standard deviation was the values of the code smell/metric in the set of websites for a specific framework, but the population under consideration for the z-score mean is the set of means for the code smell/metric across all frameworks. To illustrate, in Tables 7.2 and 7.3, the CSS code smell Embedded Rules for the framework Google Web Toolkit has a mean of 30.79, a standard deviation of 103.2, and a z-score mean of -1.63. This means that websites built with Google Web Toolkit have on average 30.79 Embedded Rules which varies on average by 103.2 rules, and this is 1.63 standard deviations below the average numbers of Embedded Rules found in other frameworks.

The high standard deviations could suggest that most frameworks don't highly restrict or control developers' specification of style in websites, such that much of the style in websites is still mostly authored by the developer. Still, it's possible that

frameworks in some way *encourage* certain CSS practices on the part of developers, or module creators, and therefore introduce the code smells that we see in Table 7.1. Despite the high variance, it's worth investigating whether it's possible to recognize a pattern of CSS behavior in frameworks based only on the code smells and metrics. A predictive model is needed to explore the more subtle patterns in the dataset not revealed to us in the descriptive statistics from Tables 7.1 and 7.2-7.3.

Another interesting observation from Tables 7.2-7.3 is that the framework **Google Web Toolkit** actually has nearly all negative values for its z-score means. This suggests that websites built with Google Web Toolkit tend to have better CSS practices compared to the other frameworks because the smell and metric values are on average lower. By contrast, the framework **Laravel** has nearly all positive values for its z-score means. This suggests that websites built with Laravel tend to have worse CSS practices compared to the other frameworks because the smell and metric values are on average higher.

7.3 RQ2 - Identifying Web Framework from CSS Code Smells

The results show that smells and metrics extracted from the CSS code of a website can indeed be used to identify the framework that was used to build it, up to an accuracy of 39%. A number of classifiers were tested on the training data to explore their compatibility with the dataset before selecting a standard neural network, which had the highest accuracy. Table 7.5 shows the classifiers along with their accuracies. Between 200-700 CSSNose outputs were collected for each of 19 frameworks, resulting in 5,525 total training examples with 26 features each. 10-fold cross validation was performed on the classifier to ensure that the accuracy was truly representative of the data and not dependent on any random initialization of the model or bias in any one test/train dataset split. In each fold, 67% of the data was used for training and 33%

Table 7.1: The percentages of the websites for a specific framework that contain at least one instance of the specified code smell.

Code Smell	adobe-experience-manager %	angularjs %	apache-wicket %	bootstrap %	cakephp %	dancer %	django %	drupal %	google-web-toolkit %	joomla %	laravel %	lift %	meteor %	microsoft-asp-net %	mojolicious %	play %	react %	ruby-on-rails %	wordpress %
Selectors with Invalid Syntax	90.42	85.6	82.86	95.18	79.8	96.45	79.13	88.52	90.76	90.18	87.2	88.5	90.56	75.0	80.41	88.33	86.78	85.44	89.71
Files with CSS code	100.0	99.2	98.1	99.76	99.51	99.41	100.0	100.0	99.58	100.0	100.0	100.0	100.0	99.53	100.0	100.0	99.59	100.0	99.26
Embedded Rules	74.58	85.6	63.33	77.83	61.58	64.1	65.53	73.72	39.5	80.0	64.4	79.65	58.32	66.51	51.55	75.1	82.23	54.85	90.07
Ignored	92.92	86.93	88.1	96.63	91.13	98.03	89.81	90.03	90.76	95.44	90.4	97.79	88.47	90.57	88.66	90.66	89.26	87.38	94.49
Selectors with ID and at least one class or element	78.75	55.47	79.05	85.54	78.82	91.12	76.7	87.31	85.71	93.33	62.0	86.28	53.34	72.64	77.32	85.21	79.75	63.11	85.29
Undefined Classes	97.5	92.8	91.9	97.11	93.6	97.24	94.17	99.09	96.22	97.89	94.4	95.13	95.81	91.04	88.66	95.72	92.56	94.17	97.06
Properties with Hard-Coded Values	99.58	97.33	97.62	99.04	98.52	99.21	99.03	99.4	98.74	99.3	98.4	99.12	99.34	97.64	96.91	98.44	97.93	98.06	98.16
Universal Selectors	95.0	92.8	88.1	97.35	92.12	97.83	89.81	91.84	93.28	96.14	90.8	97.79	92.27	89.15	90.72	92.61	90.5	88.35	95.59
Unused Properties	97.92	96.53	91.9	98.55	97.04	98.82	94.66	93.96	94.96	96.84	94.0	98.67	96.07	91.98	90.72	97.28	94.63	94.17	97.06
Effective	93.33	87.47	93.33	96.39	93.6	98.42	95.63	91.84	93.7	94.74	95.2	96.46	81.39	91.04	94.85	95.33	93.8	94.66	94.49
Too General Selectors	89.58	86.93	83.33	96.39	89.66	97.83	88.83	79.76	91.18	91.58	89.6	95.13	70.38	84.91	92.78	91.44	82.64	86.41	86.4
Too Specific Selectors Type I	91.25	82.4	84.29	94.94	86.21	73.37	82.52	88.82	88.24	94.04	85.2	90.27	82.96	79.72	85.57	85.21	83.88	78.64	91.54
Unmatched	97.92	96.53	90.48	98.31	97.04	98.82	94.66	93.66	94.96	96.84	94.0	98.67	95.67	91.51	90.72	97.28	94.63	94.17	97.06
Properties with Value Equal to None or Zero	99.17	96.53	96.19	99.28	99.01	99.21	97.57	99.7	97.48	97.89	98.0	99.12	98.82	97.17	96.91	98.44	97.93	97.09	98.9
Total Defined CSS Selectors	98.75	97.6	93.81	98.8	97.04	99.01	96.6	94.56	94.96	97.54	98.8	99.12	98.56	94.34	94.85	99.22	96.69	96.6	97.43
External Rules	92.08	69.87	88.1	96.63	91.63	97.44	88.35	89.73	92.02	96.49	92.8	97.35	87.68	88.21	94.85	94.16	85.54	94.17	91.91
Too Long Rules	95.83	74.4	93.33	98.31	94.58	98.82	93.2	96.68	95.8	97.19	95.2	98.67	94.5	94.81	89.69	95.72	94.21	91.75	97.43
Empty Catch Rules	92.92	69.87	87.14	95.66	89.16	97.04	87.38	92.15	92.44	94.04	90.8	94.25	92.53	80.66	89.69	89.88	90.08	89.32	93.01
Total Defined CSSProperties	98.75	97.6	93.81	98.8	97.04	99.01	96.6	94.56	94.96	97.54	98.4	99.12	98.43	94.34	94.85	98.83	96.28	96.12	97.43
Too Specific Selectors Type II	90.0	81.87	84.29	94.94	85.71	73.57	80.58	89.12	86.97	93.33	84.0	88.05	81.91	77.83	83.51	84.82	83.06	79.13	91.18
Ignored Properties	92.92	86.93	87.62	96.63	90.15	98.03	89.81	90.03	90.76	95.44	90.4	97.79	88.34	90.57	88.66	90.66	89.26	87.38	94.49
Lines of Code	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Total Defined CSS Rules	100.0	99.2	98.1	99.76	99.51	99.41	100.0	100.0	99.58	100.0	100.0	100.0	100.0	99.53	100.0	100.0	99.59	100.0	99.26
Inlined Rules	98.33	82.67	92.86	95.18	97.04	74.56	89.81	99.09	96.64	98.25	85.6	97.79	60.42	96.7	84.54	90.66	90.91	83.5	95.96
Ineffective	81.25	75.47	72.38	82.65	78.82	93.49	73.3	68.58	74.79	77.19	76.4	65.93	71.04	71.7	83.51	75.88	76.45	79.61	73.9
Matched	93.33	92.0	93.33	96.87	94.09	98.42	95.63	91.84	94.54	96.49	96.8	96.9	84.14	92.45	94.85	96.5	94.21	95.63	94.49

Table 7.2: The mean, standard deviation, and z-score mean for each code smell/metric across frameworks 1-9.

		Undefined Classes	Too General Selectors	Effective Selectors	Embedded Rules	Empty Catch Rules	External Rules	Files with CSS Code	Ignored Selectors	Ignored Properties	Ineffective Selectors	Inlined Rules	Lines of Code	Matched Selectors	Properties with Hard-Coded Values	Properties with Value Equal to None or Zero	Selectors with ID and at Least one Class or Element	Selectors with Invalid Syntax	Too Long Rules	Too Specific Selectors Type I	Too Specific Selectors Type II	Total Defined CSS Properties	Total Defined CSS Rules	Total Defined CSS Selectors	Universal Selectors	Unmatched Selectors	Unused Properties
adobe-experience-manager	mean	150.09	13.16	110.3	146.62	55.73	2110.7	8.67	255.02	464.26	24.58	139.35	29251.33	134.88	2034.25	776.29	214.94	524.63	199.52	898.07	758.26	7224.102	2396.67	3316.75	2027.72	2925.95	6267.97
	std-dev	118.09	12.45	140.81	681.68	64.94	2693.38	6.76	385.12	691.05	37.53	180.33	37082.18	174.98	2625.79	1066.05	536.02	839.27	330.74	1345.27	1201.43	9498.05	2770.3	3871.99	2864.44	3443.21	8509.45
	z-score	1.65	0.91	2.19	1.43	1.32	3.03	0.19	0.75	0.6	0.63	0.51	3.04	2.4	3.05	3.22	2.35	2.09	3.32	3.63	3.69	3.0	3.11	2.93	3.41	3.07	3.11
angularjs	mean	65.65	8.25	31.87	18.6	22.72	782.94	6.31	175.27	330.42	61.45	48.1	12647.08	93.31	661.83	237.51	80.08	479.62	58.0	306.96	240.75	3169.03	1015.65	1613.92	853.74	1345.34	2686.89
	std-dev	88.26	8.18	45.1	329.44	40.82	1172.61	7.23	317.26	623.3	111.15	123.08	14382.71	109.28	987.15	354.15	230.45	692.97	90.14	525.79	458.5	3783.32	1166.19	1747.66	1076.3	1453.66	3207.32
	z-score	-0.38	-1.24	-1.11	2.43	-0.68	-0.74	-0.72	-0.37	-0.36	3.9	-1.06	-0.22	0.72	-1.08	-1.06	-0.51	1.72	-0.97	0.04	-0.0	-0.49	-0.62	-0.14	0.05	-0.14	-0.38
apache-wicket	mean	72.9	9.22	56.1	38.28	46.55	1387.59	11.35	159.59	303.5	16.1	109.4	20749.99	72.2	1432.62	466.96	92.29	216.52	136.81	308.44	246.75	4439.57	1536.26	1892.24	874.57	1660.46	3887.23
	std-dev	93.12	15.49	66.96	120.41	62.97	1286.03	10.69	188.73	377.58	27.1	164.51	72413.46	81.27	1458.99	434.43	191.16	248.34	155.0	372.38	303.79	3882.93	1377.44	1573.55	763.13	1437.36	3599.14
	z-score	-0.21	-0.81	-0.09	-1.43	0.77	0.98	1.23	-0.58	-0.55	-0.12	-0.0	1.37	-0.14	1.24	0.76	-0.25	-0.42	1.42	0.04	0.04	0.6	0.78	0.36	0.1	0.5	0.79
bootstrap	mean	82.03	15.18	50.38	92.26	50.44	1670.01	11.74	421.01	812.92	17.66	99.68	18706.65	68.04	1374.52	492.27	115.68	471.24	111.43	457.02	364.34	5854.67	1861.94	2759.41	1336.36	2270.36	4789.27
	std-dev	80.55	9.35	50.46	257.43	63.3	1314.79	8.1	400.4	776.59	32.64	235.19	25410.14	67.32	1057.35	410.69	178.73	783.74	98.24	431.42	360.46	4552.57	1419.6	2038.11	1025.82	1677.78	3914.95
	z-score	0.01	1.79	-0.33	-0.01	1.0	1.78	1.38	3.02	3.1	0.02	-0.17	0.97	-0.31	1.07	0.96	0.24	1.65	0.65	0.95	0.88	1.82	1.66	1.92	1.43	1.74	1.67
cakephp	mean	50.1	10.43	61.79	67.45	23.18	788.97	9.39	164.68	304.58	12.57	109.81	9501.02	74.35	843.74	293.92	95.79	187.68	72.99	221.62	175.87	2877.34	966.23	1302.25	614.67	1063.21	2294.68
	std-dev	68.72	12.73	59.61	180.02	32.68	927.32	8.61	252.29	451.84	16.94	187.91	16129.53	68.0	887.09	308.9	210.59	447.24	84.51	465.8	374.24	3445.72	1013.0	1661.7	957.44	1421.92	3021.83
	z-score	-0.75	-0.28	0.15	-0.66	-0.66	-0.72	0.47	-0.51	-0.54	-0.43	0.0	-0.84	-0.05	-0.53	-0.02	-0.18	-0.65	-0.51	-0.48	-0.47	-0.74	-0.75	-0.7	-0.64	-0.71	-0.77
dancer	mean	59.85	13.48	123.04	87.53	15.85	812.95	7.21	99.42	190.82	15.07	172.89	10055.21	138.11	869.61	381.02	74.74	143.58	87.59	169.77	131.79	2788.41	1073.37	1216.53	511.57	979.01	2203.76
	std-dev	60.99	7.47	127.81	211.59	22.64	833.1	4.84	180.36	340.56	18.06	450.6	9021.55	141.73	826.63	320.88	110.67	260.79	75.67	271.54	225.67	2656.38	1041.51	1248.23	691.0	1069.83	2290.9
	z-score	-0.52	1.04	2.73	-0.13	-1.1	-0.65	-0.37	-1.41	-1.36	-0.21	1.09	-0.73	2.53	-0.45	0.08	-0.63	-1.01	-0.07	-0.8	-0.78	-0.82	-0.46	-0.85	-0.94	-0.88	-0.85
django	mean	63.6	14.24	53.8	73.14	32.1	938.79	7.35	192.64	354.61	12.02	69.88	9270.67	65.82	890.2	287.11	108.7	207.86	68.14	235.01	187.8	3288.15	1081.81	1522.52	716.57	1264.07	2659.11
	std-dev	96.1	41.33	51.15	313.64	66.65	1616.37	7.45	330.97	603.49	17.31	193.64	11147.89	60.76	1514.72	413.41	263.53	489.58	104.11	446.01	357.64	4833.12	1694.35	2317.63	1231.21	2011.18	4240.9
	z-score	-0.43	1.38	-0.19	-0.51	-0.11	-0.3	-0.32	-0.13	-0.19	-0.48	-0.69	-0.89	-0.4	-0.39	-0.67	0.09	-0.49	-0.66	-0.4	-0.38	-0.39	-0.44	-0.3	-0.35	-0.3	-0.41
drupal	mean	191.06	10.15	59.32	98.77	34.12	1146.11	8.47	187.29	339.57	11.98	113.17	12437.31	71.29	1118.53	448.46	196.05	225.87	99.27	425.51	341.52	4051.59	1358.05	1818.98	970.8	1560.4	3462.0
	std-dev	149.37	9.06	69.59	217.42	47.3	1245.84	6.46	304.67	594.37	17.36	171.26	11416.9	81.53	1202.35	437.35	351.12	355.93	110.27	716.18	602.81	4548.25	1275.57	2046.22	1357.86	1776.39	4086.63
	z-score	2.63	-0.41	0.05	0.17	0.01	0.29	0.11	-0.2	-0.29	-0.48	0.06	-0.26	-0.17	0.3	0.61	1.95	-0.34	0.28	0.76	0.71	0.27	0.3	0.23	0.41	0.3	0.37
google-web-toolkit	mean	96.44	6.45	60.59	37.09	21.82	462.39	4.33	78.54	158.08	4.58	197.43	8107.08	65.17	648.84	277.78	48.31	101.47	71.07	85.36	60.4	1793.25	696.92	686.57	330.52	542.86	1456.66
	std-dev	64.42	5.02	47.34	103.2	27.31	572.04	3.06	240.84	488.05	10.45	478.51	9827.15	49.56	802.51	423.93	70.89	508.23	84.09	157.09	134.08	3081.29	777.15	1015.74	533.05	801.21	2800.43
	z-score	0.36	-2.02	0.1	-1.46	-0.74	-1.65	-1.49	-1.7	-1.59	-1.13	1.52	-1.11	-0.42	-1.12	-0.74	-1.19	-1.36	-0.57	-1.31	-1.29	-1.68	-1.48	-1.81	-1.46	-1.77	-1.58

Table 7.3: The mean, standard deviation, and z-score mean for each code smell/metric across frameworks 10-19.

		Undefined Classes	Too General Selectors	Effective Selectors	Embedded Rules	Empty Catch Rules	External Rules	Files with CSS Code	Ignored Selectors	Ignored Properties	Ineffective Selectors	Inlined Rules	Lines of Code	Matched Selectors	Properties with Hard-Coded Values	Properties with Value Equal to None or Zero	Selectors with ID and at Least one Class or Element	Selectors with Invalid Syntax	Too Long Rules	Too Specific Selectors Type I	Too Specific Selectors Type II	Total Defined CSS Properties	Total Defined CSS Rules	Total Defined CSS Selectors	Universal Selectors	Unmatched Selectors	Unused Properties
joomla	mean	80.6	11.34	47.18	116.33	85.15	1198.23	15.55	251.84	501.59	17.16	256.9	12261.21	64.34	1385.49	517.63	167.71	226.19	112.81	329.47	246.05	4328.74	1571.46	2605.29	910.64	1089.11	3637.31
	std-dev	97.85	9.74	49.24	333.93	134.64	1046.83	12.1	234.52	468.06	27.85	381.14	10155.51	64.54	1145.7	420.8	225.67	260.68	101.09	356.82	265.24	3478.33	1230.28	1692.09	884.67	1465.86	3051.42
	z-score	-0.02	0.12	-0.46	0.63	3.11	0.44	2.86	0.69	0.87	-0.02	2.55	-0.3	-0.46	1.1	1.16	1.35	-0.34	0.69	0.17	0.03	0.51	0.88	0.57	0.21	0.56	0.55
laravel	mean	63.44	12.04	45.1	105.02	28.01	1103.04	8.24	265.54	506.11	19.68	62.8	14100.55	64.78	1008.41	351.53	54.52	317.23	90.22	306.48	246.77	4013.98	1270.86	1868.52	907.53	1538.2	3237.41
	std-dev	66.17	11.21	45.39	381.02	47.74	1708.1	9.77	412.88	849.71	49.01	123.85	15685.04	70.92	1719.36	640.52	154.55	754.03	188.07	595.1	497.9	6494.34	1774.11	2586.5	1411.14	2178.12	5563.37
	z-score	-0.43	0.42	-0.55	0.33	-0.36	0.17	0.03	0.88	0.9	0.2	-0.81	0.06	-0.44	-0.04	-0.16	-1.06	0.4	0.01	0.03	0.04	0.24	0.07	0.32	0.2	0.25	0.16
lift	mean	31.17	12.04	38.45	94.46	16.53	888.38	6.48	209.68	396.35	10.75	55.89	19944.14	49.2	744.7	261.04	55.15	141.1	53.12	181.01	134.51	3056.28	1038.73	1501.15	667.01	1242.27	2428.73
	std-dev	51.25	7.58	43.16	200.16	27.5	917.94	5.18	198.95	375.89	13.64	130.8	98819.12	48.16	879.18	298.44	147.19	287.28	69.9	278.51	219.75	3068.52	939.78	1393.46	790.3	1201.99	2575.45
	z-score	-1.21	0.42	-0.83	0.05	-1.06	-0.44	-0.66	0.11	0.11	-0.59	-0.93	1.21	-1.07	-0.83	-0.88	-1.04	-1.03	-1.12	-0.73	-0.76	-0.59	-0.56	-0.34	-0.49	-0.35	-0.63
meteor	mean	23.88	7.64	13.96	72.86	18.97	909.86	4.0	177.63	348.86	17.12	13.43	15357.67	31.09	600.8	223.48	46.24	418.22	56.6	250.53	211.71	2912.18	996.15	1408.83	691.3	1200.11	2439.52
	std-dev	45.21	8.48	24.31	243.22	31.14	1176.45	4.18	266.29	546.15	50.87	55.18	16864.33	60.09	725.82	289.96	179.26	1022.71	72.99	396.06	349.84	3546.68	1189.7	1659.3	869.34	1425.14	3983.82
	z-score	-1.38	-1.5	-1.86	-0.52	-0.91	-0.38	-1.62	-0.33	-0.23	-0.03	-1.66	0.31	-1.8	-1.26	-1.18	-1.23	1.22	-1.01	-0.31	-0.21	-0.71	-0.67	-0.51	-0.42	-0.43	-0.62
microsoft-aspnet	mean	50.01	9.26	50.8	72.96	29.59	815.67	8.32	141.97	275.32	10.37	125.34	9048.23	61.17	986.75	309.11	111.45	128.11	97.58	195.25	151.58	3436.12	1013.97	1259.16	553.94	1056.01	2899.8
	std-dev	70.32	10.85	61.85	265.13	101.73	1028.03	7.93	178.79	352.89	19.12	212.97	11095.29	72.38	1117.78	327.59	291.43	199.2	132.05	269.91	216.73	7173.59	1137.71	1476.57	685.93	1297.76	6989.23
	z-score	-0.76	-0.79	-0.31	-0.52	-0.27	-0.64	0.06	-0.82	-0.75	-0.62	0.27	-0.93	-0.58	-0.1	-0.5	0.15	-1.14	0.23	-0.64	-0.64	-0.26	-0.62	-0.78	-0.81	-0.73	-0.17
mojolicious	mean	53.62	11.88	73.25	140.81	28.79	921.07	9.1	187.21	348.23	14.98	161.92	11322.35	88.23	1059.55	389.06	68.96	197.1	79.56	212.1	171.02	3327.0	1223.8	1538.48	717.79	1263.05	2693.95
	std-dev	71.16	11.86	90.63	367.92	41.85	1122.32	9.63	235.06	435.94	19.99	416.68	12325.08	103.8	1506.06	759.21	118.87	319.91	106.85	320.35	283.78	3730.21	1517.75	1725.24	942.6	1447.33	3153.65
	z-score	-0.67	0.35	0.63	1.27	-0.31	-0.35	0.36	-0.2	-0.23	-0.21	0.9	-0.48	0.51	0.12	0.14	-0.75	-0.58	-0.32	-0.54	-0.5	-0.35	-0.06	-0.27	-0.34	-0.31	-0.38
play	mean	80.71	14.13	62.93	62.96	47.98	1050.15	7.75	275.5	518.27	22.46	61.85	13459.89	85.39	884.18	307.79	90.53	315.06	71.61	319.35	255.02	4183.82	1174.96	1850.56	934.16	1489.68	3126.85
	std-dev	111.65	11.51	62.75	177.1	65.12	1015.32	6.61	284.98	563.54	27.61	121.08	14152.95	84.58	936.86	302.86	175.88	559.89	78.22	371.82	310.93	4708.3	1095.71	1663.43	940.12	1379.58	3153.94
	z-score	-0.02	1.33	0.2	-0.78	0.85	0.02	-0.16	1.02	0.99	0.45	-0.83	-0.06	0.4	-0.41	-0.51	-0.29	0.38	-0.56	0.11	0.1	0.38	-0.19	0.29	0.28	0.15	0.05
react	mean	112.91	11.32	50.45	82.46	31.21	889.7	7.86	157.72	280.66	10.58	131.09	9820.15	61.03	962.72	355.02	153.21	208.57	84.88	242.86	194.0	3175.45	1103.25	1390.44	673.54	1171.69	2599.33
	std-dev	111.22	9.9	53.78	176.25	55.48	1240.91	6.09	254.15	480.04	18.0	214.7	11031.98	64.79	1266.23	371.97	177.61	518.47	102.05	443.12	399.39	3754.56	1307.32	1746.05	966.85	1508.55	3318.38
	z-score	0.75	0.1	-0.33	-0.27	-0.17	-0.44	-0.12	-0.61	-0.72	-0.6	0.37	-0.78	-0.59	-0.17	-0.13	1.04	-0.48	-0.15	-0.35	-0.34	-0.48	-0.38	-0.54	-0.47	-0.49	-0.47
ruby-on-rails	mean	69.01	10.35	57.17	47.3	26.4	993.56	5.92	201.66	378.63	16.78	48.4	11461.33	73.95	931.63	320.35	92.41	254.74	72.99	260.8	211.45	3488.95	1089.26	1547.33	782.19	1271.72	2799.74
	std-dev	84.48	10.16	66.57	107.81	46.92	1556.37	5.27	328.53	607.83	23.73	155.6	12490.4	83.28	1581.85	488.44	224.58	497.15	116.4	442.22	384.65	4949.16	1589.15	2223.49	1278.66	1870.1	4291.7
	z-score	-0.3	-0.32	-0.04	-1.19	-0.46	-0.14	-0.87	-0.0	-0.01	-0.06	-1.06	-0.46	-0.07	-0.27	-0.41	-0.25	-0.11	-0.51	-0.25	-0.21	-0.21	-0.42	-0.26	-0.16	-0.29	-0.27
wordpress	mean	151.03	9.97	59.26	136.72	30.25	951.46	7.26	230.58	416.54	14.85	105.78	14307.46	74.11	949.43	359.02	113.92	329.41	85.55	316.12	256.47	3613.57	1193.95	1630.57	838.25	1325.87	2919.86
	std-dev	128.59	8.52	76.43	382.53	39.48	1122.27	5.58	341.23	634.27	35.87	296.36	16513.1	91.92	1024.94	400.43	220.55	666.13	83.68	456.51	376.39	3630.61	1234.29	1826.81	1083.59	1485.65	3008.27
	z-score	1.67	-0.48	0.04	1.17	-0.23	-0.26	-0.35	0.4	0.26	-0.23	-0.07	0.1	-0.06	-0.21	-0.09	0.21	0.5	-0.13	0.09	0.11	-0.11	-0.14	-0.11	0.0	-0.18	-0.15

Table 7.4: The full set of CSS code smells and metrics collected from websites using CSSNose along with their descriptions and the functional group they belong to (rule-based, selector-based, property-based, or file-based).

#	Type	Group	Name	Description
CS1	Code Smell	Rule-Based	Inlined Rules	Number of rules inlined in the HTML with the `style` attribute.
CS2	Code Smell	Rule-Based	Embedded Rules	Number of rules embedded in the HTML with the `style` tag.
CS3	Code Smell	Rule-Based	Too Long Rules	Number of rules containing more than five style property declarations.
CS4	Code Smell	Rule-Based	Empty Catch Rules	Number of rules containing no style property declarations.
CS5	Code Smell	Selector-Based	Too Specific Selectors Type I	Number of rules containing more than four simple selectors.
CS6	Code Smell	Selector-Based	Too Specific Selectors Type II	Number of rules containing more than one ID selector, more than two class selectors, or more than three element selectors.
CS7	Code Smell	Selector-Based	Universal Selectors	Number of selectors containing an instance of the universal selector `*`.
CS8	Code Smell	Selector-Based	Selectors with ID and at Least One Class or Element	Number of selectors containing an ID along with at least one class or element selector.
CS9	Code Smell	Selector-Based	Selectors with Erroneous Adjoining Pattern	Number of selectors without white space between classes or IDs.
CS10	Code Smell	Selector-Based	Too General Selectors	Number of selectors targeting a large percentage of the DOM: `html, head, body, div, header, aside`.
CS11	Code Smell	Property-Based	Properties with Hard-Coded Values	Number of style property declarations containing constants.
CS12	Code Smell	Property-Based	Properties with Value Equal to None or Zero	Number of style property declarations with the value `0` or `none`.
M1	Metric	Rule-Based	External Rules	Number of rules contained in external style sheets.
M2	Metric	Rule-Based	Total Defined CSS Rules	The total number of rules.
M3	Metric	Selector-Based	Total Defined CSS Selectors	The total number of selectors.
M4	Metric	Selector-Based	Ignored CSS Selectors	Number of selectors ignored during smell collection.
M5	Metric	Selector-Based	Undefined Classes	Number of rules containing a class selector which does not target any existing DOM nodes.
M6	Metric	Selector-Based	Matched Selectors	Number of selectors which target existing DOM nodes.
M7	Metric	Selector-Based	Unmatched Selectors	Number of selectors which do not target any existing DOM nodes.
M8	Metric	Selector-Based	Effective Selectors	Number of selectors which target existing DOM nodes and have style properties that are successfully applied.
M9	Metric	Selector-Based	Ineffective Selectors	Number of selectors which target existing DOM nodes but have no properties applied due to being overridden by other rules in the cascade.
M10	Metric	Property-Based	Total Defined CSS Properties	The total number of style property declarations.
M11	Metric	Property-Based	Unused Properties	Number of properties that are not applied to the DOM.
M12	Metric	Property-Based	Ignored Properties	Number of ignored style property declarations.
M13	Metric	File-Based	Files with CSS Code	The total number of files containing CSS Code, either from external style sheets or HTML.
M14	Metric	File-Based	Lines of Code	The total number of lines of style code collected.

was used for testing. Furthermore, the per-class precision, recall, and f1 scores were calculated to determine the reliability of the model. See Table 7.6 for the results and Section 7.1 for an explanation of these metrics. The average precision, recall, and f1-score values were 0.34, 0.38, 0.32 respectively, showing good reliability.

An accuracy of 39% in prediction is significant given that the random chance of guessing a website's framework is 1 / 19 = 5.26%. This suggests that there exists a pattern of CSS behavior in the data not easily recognized by humans through descriptive statistics. Important to note is that not much effort was invested in raising the accuracy beyond what could be achieved with the default parameters of the machine learning library used for training. Presumably with more training data and model parameter tuning, an even higher accuracy could be achieved. Now that a correlation has been established between web frameworks and a pattern of CSS behavior derived from CSS code smells and metrics, we can begin to question why this behavior is predictable at all. Is it the result of design decisions made by frameworks? The result of some process of development they impose on framework users? Or perhaps it has less to do with any action on the part of the framework and more to do with the choices of developers? For example, perhaps lightweight frameworks tend to attract developers who are more likely to write compact CSS code with little duplication, whereas more feature-complete frameworks attract developers who are more likely to include duplicated CSS rules. These ideas can be explored in further studies.

Code Smell Groups - In order to capture which sets of features were most important for prediction, the code smells and metrics were separated into various groups and each used to train a neural network classifier. See Table 7.7 for the groups and their accuracies and Table 7.4 to reference smell/metric belongs to which group. 10-fold cross validation was also used on the classifiers trained on these groupings to ensure a representative accuracy. One division simply grouped all code smells (12

Table 7.5: The accuracies of various classifiers tested during framework prediction.

Classifier	% Accuracy
Naive Bayes	18.92
Quadratic Discriminant Analysis	27.10
Decision Tree	28.55
Adaboost	31.30
Random Forest	32.70
Radial Basis Function SVM	32.70
Linear SVM	36.84
K-Nearest Neighbors (with labels)	37.60
Neural Network	39.08

features) and grouped all metrics (14 features), which achieved accuracies of 30.82% and 33.48% respectively. This demonstrates that metrics provide more information for prediction but not much more than code smells, (most likely due to the fact that the feature count is higher for metrics). A second division created functional groups of file-based (2 features), rule-based (6 features), selector-based (13 features), and property-based (5 features) smells and metrics, achieving accuracies of 21.95%, 24.53%, 32.11%, and 22.12% respectively. Code smells and metrics dealing exclusively with the CSS selectors appear to provide the most information in prediction. The result of 32.11% for selector-based smells is somewhat close to the full feature set accuracy of 39% and indicates that the composition of a selector when specifying style is an important indicator of CSS behavior. It further indicates that websites built with differing frameworks tend to write their selectors in fairly different ways. Also important to note is that the file-based features, of which there are only 2, are approximately as successful as feature sets of size 5 and 6, meaning file-based features are also quite informative.

Table 7.6: The precision, recall, F1-score, and support on a neural network classifier.

Framework	Precision	Recall	F1-score	Support
a.e.m.	0.35	0.25	0.29	76
angularjs	0.41	0.17	0.24	129
apache-wicket	0.35	0.11	0.16	56
bootstrap	0.24	0.46	0.31	131
cakephp	0.25	0.01	0.02	79
dancer	0.42	0.71	0.53	154
django	0.15	0.03	0.06	59
drupal	0.32	0.52	0.39	116
g.w.t.	0.42	0.56	0.48	75
joomla	0.34	0.49	0.40	80
laravel	0.33	0.03	0.05	75
lift	0.56	0.51	0.53	73
meteor	0.43	0.84	0.57	282
m.a.n.	0.38	0.08	0.13	77
mojolicious	0.00	0.00	0.00	35
play	0.38	0.23	0.28	93
react	0.35	0.28	0.31	69
ruby-on-rails	0.00	0.00	0.00	70
wordpress	0.33	0.07	0.12	98
Average / Total	0.34	0.38	0.32	1827

Table 7.7: The number of features and accuracies for each functional group of smells and metrics trained on a classifier with 10-fold cross validation.

Division	Smell Grouping	# Features	Accuracy
1	Smell-Based	12	31.26%
	Metric-Based	14	35.51%
2	Property-Based	6	21.23%
	Rule-Based	13	23.94%
	Selector-Based	5	34.30%
	File-Based	2	23.26%

7.4 RQ3 - Correlation Between Code Smells

The results from three different clustering tasks using K-Means and two clustering tasks using Agglomerative clustering show that there are indeed correlations between code smells and metrics, that there are interesting relationships between frameworks, and they support the finding in RQ1 of high smell and metric value variance across websites. Clustering as a task groups objects based on a notion of similarity between the objects. In this case, the notion of similarity is the Euclidean distance between feature sets of the objects as computed by the clustering algorithm K-Means. Looking at the resulting object clusters, we can attempt to derive interesting observations that inform us of correlations or relationships between objects.

The first K-Means clustering task treats websites as individual objects. Specifically, the full set of smells/metrics for each website form a feature set (an example of which can be seen in Table 7.8). See Figure 7.1 for a plot of the website clusters. A majority of the websites bunch together without many distinct divisions between clusters. The frameworks used to build the websites also appear to be fairly well-distributed over the clusters produced by K-Means, with the exception that Adobe Experience Manager primarily composes the fringe data points identified as cluster

Table 7.8: Example feature sets of two websites along with the actual and predicted framework from the classifier.

Smell / Metric	jide.com	escuela20.com
UndefinedClasses	37	4
Too General Selectors	8	3
Effective	13	174
EmbeddedRules	5	0
EmptyCatchRules	3	0
ExternalRules	61	823
FileswithCSScode	5	5
Ignored	7	89
IgnoredProperties	14	107
Ineffective	0	15
InlinedRules	0	8
LOC(CSS)	953	5010
Matched	13	189
Properties with Hard-Coded Values	50	673
Properties with Value Equal to None or Zero	22	546
Selectors with ID and at least one classor element	0	259
Selectors with Invalid Syntax	2	19
TooLongRules	6	266
TooSpecificSelectorsTypeI	7	43
TooSpecificSelectorsTypeII	5	19
TotalDefinedCSSProperties	218	3267
TotalDefinedCSSrules	66	831
TotalDefinedCSSselectors	111	836
UniversalSelectors	8	158
Unmatched	91	558
UnusedProperties	139	2330
Predicted	ASP.NET	ASP.NET
Actual	Play	ASP.NET

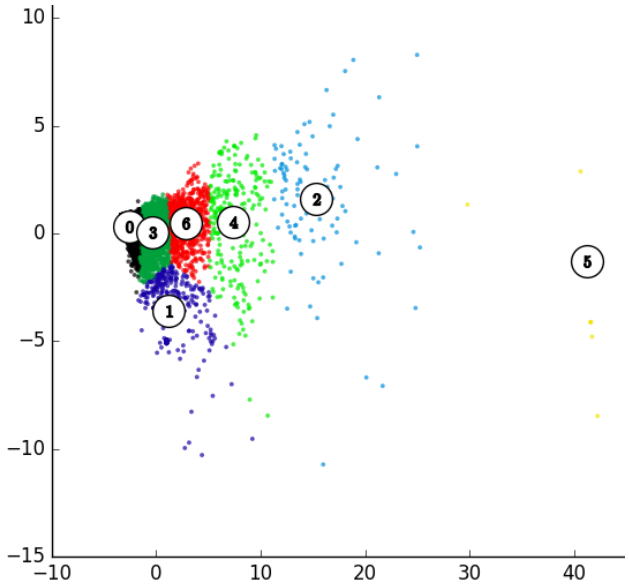

Figure 7.1: The website clusters produced by K-Means. The units of the axes represent a linear combination of the feature set produced by Principle Component Analysis (PCA).

two on the plot. This result is consistent with the high code smell and metric values found in Tables 7.2-7.3, supporting the idea that frameworks do not highly restrict developers' specification of style in websites.

The second K-Means clustering task treats frameworks as individual objects. Specifically, the mean values for all smells/metrics of a specific framework form a single feature set (see Plot 7.2b). Clustering by framework reveals that there are indeed some relationships in CSS behavior between various frameworks, though it's not immediately obvious what factors may be affecting this. CakePHP, Mojolicious, ASP.net, and React all appear to form a fairly distinct cluster (1, blue), as do Lift, Laravel, Play, Ruby on Rails, Wordpress, and Django (2, orange). Joomla, Drupal, and Apache-Wicket also form a distinct cluster (2, yellow). Bootstrap (2, yellow), Dancer (1, blue), and Google Web Toolkit (1, blue) all appear on the fringes of the these clusters which indicates that they are unique somehow in their CSS behavior. There don't appear to be any strong similarities between frameworks based on the

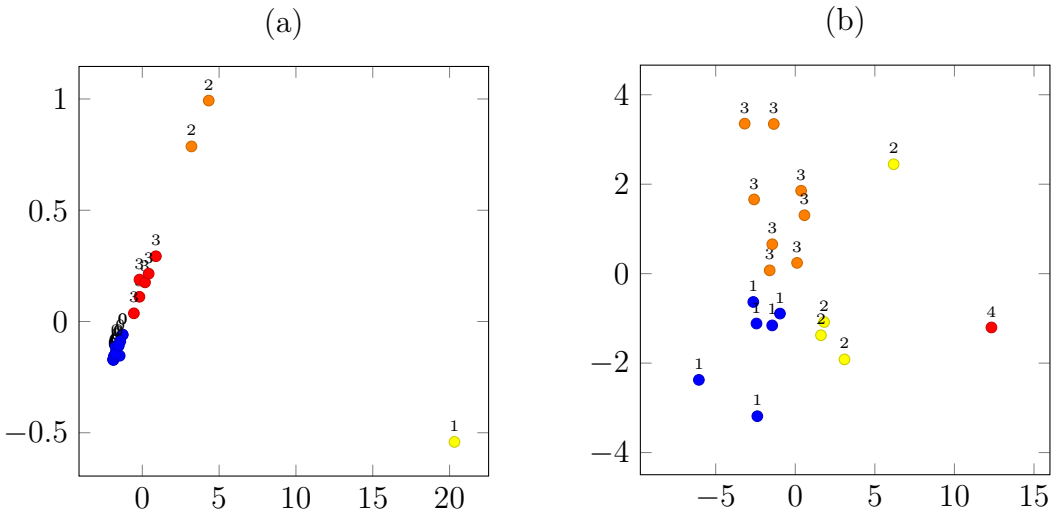

Figure 7.2: Clusters based on (a) code smell/metric and (b) framework. Each color/number represents a unique cluster. The units of the axes represent a linear combination of the feature set produced by Principle Component Analysis (PCA).

language of development, though two JavaScript frameworks, AngularJS and Meteor (3, orange), appear very close to each other and somewhat removed from the other frameworks in their cluster. One highly interesting observation is that Google Web Toolkit, which is found in RQ1 to have on average lower code smell/metric values, appears close to the framework Dancer. Revisiting the z-scores on Tables 7.2-7.3, we observe that Dancer also has lower than average smell/metric values. Similarly, Meteor and AngularJS have a high amount of negative z-scores and appear next to each other. Ultimately, clustering by framework provides good insight into which frameworks have similar patterns of CSS behavior and can be a good starting point for investigating and comparing development features that lead to certain CSS practices.

The third K-Means clustering task treats code smells and metrics as individual objects. Specifically, the mean values for a specific smell/metric for all frameworks form a single feature set (see Plot 7.2a). There appear to be two fairly distinct clusters that are each a mix of various smells and metrics. Specifically, Total Defined CSS Selectors, Properties with Hard-Coded Values, Total Defined CSS Rules, External

Rules, and Universal Selectors form a cluster together (2, red), Total Defined CSS Properties and Unused properties are found together (orange), Lines of Code is found on its own far removed from all over clusters (yellow), and the remaining smells/-metrics are found together (blue). The high concentration of smells and metrics in the blue and red clusters indicate that those smells and metrics are highly correlated which each other, meaning that the presence and/or extent of those smells strongly affect each other.

Interesting to note is that Total Defined CSS Selectors isn't as closely correlated with a majority of the selector-based smells and metrics as we might have expected. Lines of Code appears on its own far removed from the other smells and metrics, indicating that it doesn't have a very strong relationship with them which is surprising. Total Defined CSS Properties is found together with Unused Properties, which we might expect to see, but the fact that they're rather far away from other property-based smells and metrics is also surprising. Ultimately, clustering by smells and metrics is useful in exploring the correlations and relationships between code smells and metrics and can be highly useful in directing developer maintenance efforts as well as recommendation of development practices. For example, because we find that Total Defined CSS Selectors does not have a very strong correlation with selector-based smells and metrics, we know that reducing the number of CSS Selectors may not necessarily be an effective way to combat high selector-based smell values.

Lastly, the code smells/metrics and frameworks are clustered with an Agglomerative clustering method. Figures 7.3 and 7.4 show the resulting dendrograms for smells/metrics for frameworks respectively. Dendrograms are useful in showing a more comparative view of the features while the clustering plots shown earlier show a more relative view of the features. Each leaf node represents one of the 26 code smells or metrics and each non-leaf node represents the joining of two sub-groupings as more clusters are successively discovered. The values on these non-leaf nodes repre-

sent the distance in Euclidean space between the left and right groupings, indicating their similarity. The higher up on the tree a non-leaf node is, the farther away and therefore less similar two subgroups are to each other.

For example, in Figure 7.3, Total Defined CSS Properties and Unused Properties are found to be correlated and are therefore grouped (clustered) together at a distance of 63.6. They are then together grouped with the set of all other features (except for Lines of Code) at a distance of 430 as the next cluster is discovered at that step in the clustering process. The smaller the distance measure, the more similar or correlated the two features or two sets of features are deemed to be. Important to note is that because K-Means and agglomerative clustering use the same metric for similarity and simply differ in their approach to the actual notion of a cluster, the dendrogram diagrams are consistent with the cluster plots shown earlier in terms of correlations between features.

For the first agglomerative clustering task, code smells/metrics are treated as individual objects. Figure 7.3 shows the resulting dendrogram. The dendrogram reiterates for us that Lines of Code appears to have little influence over any of the code smells and metrics as it appears in an outer group at a very high distance from the others. Some relationships indicated by the dendrogram make sense, such as Total Defined Properties being highly correlated with the number of Unused Properties. However one interesting relationship discovered through the dendrogram is the very high correlation observed between Dangerous Selectors (Too General Selectors) and Files with CSS Code. One reason for this correlation might be that the more CSS files a developer tends to add to and use in a web project, the more specific (and therefore less general) they have to make their selectors so that styles rules apply as intended. Another interesting relationship is the one between Selectors with ID and at Least One Class or Element and Undefined Classes because it suggests that developers who tend to abuse IDs by combining them with other simple selectors also typically abuse

classes by providing style to classes that don't exist in the HTML. Ultimately the relationships between code smells in metrics as displayed in the dendrogram are useful in forming hypotheses about how CSS is developed and the difficulties developers might encounter.

For the second agglomerative clustering task, frameworks are treated as individual objects. Figure 7.4 shows the resulting dendrogram. The dendrogram reiterates for us that there doesn't seem to be much similarity in CSS behavior between frameworks based on language of implementation nor based on whether the framework is a content management system or not. It additionally provides an excellent starting point for hypotheses concerning this similarity. For example, we observe that Adobe Experience Managers appears quite separated from the other frameworks and that CakePHP and Microsoft ASP.net appear very close to one another, so we can begin investigating the feature sets and developer communities of these frameworks in order to gain insight into what may make them so similar or different to one another in the CSS behavior found in the resulting websites.

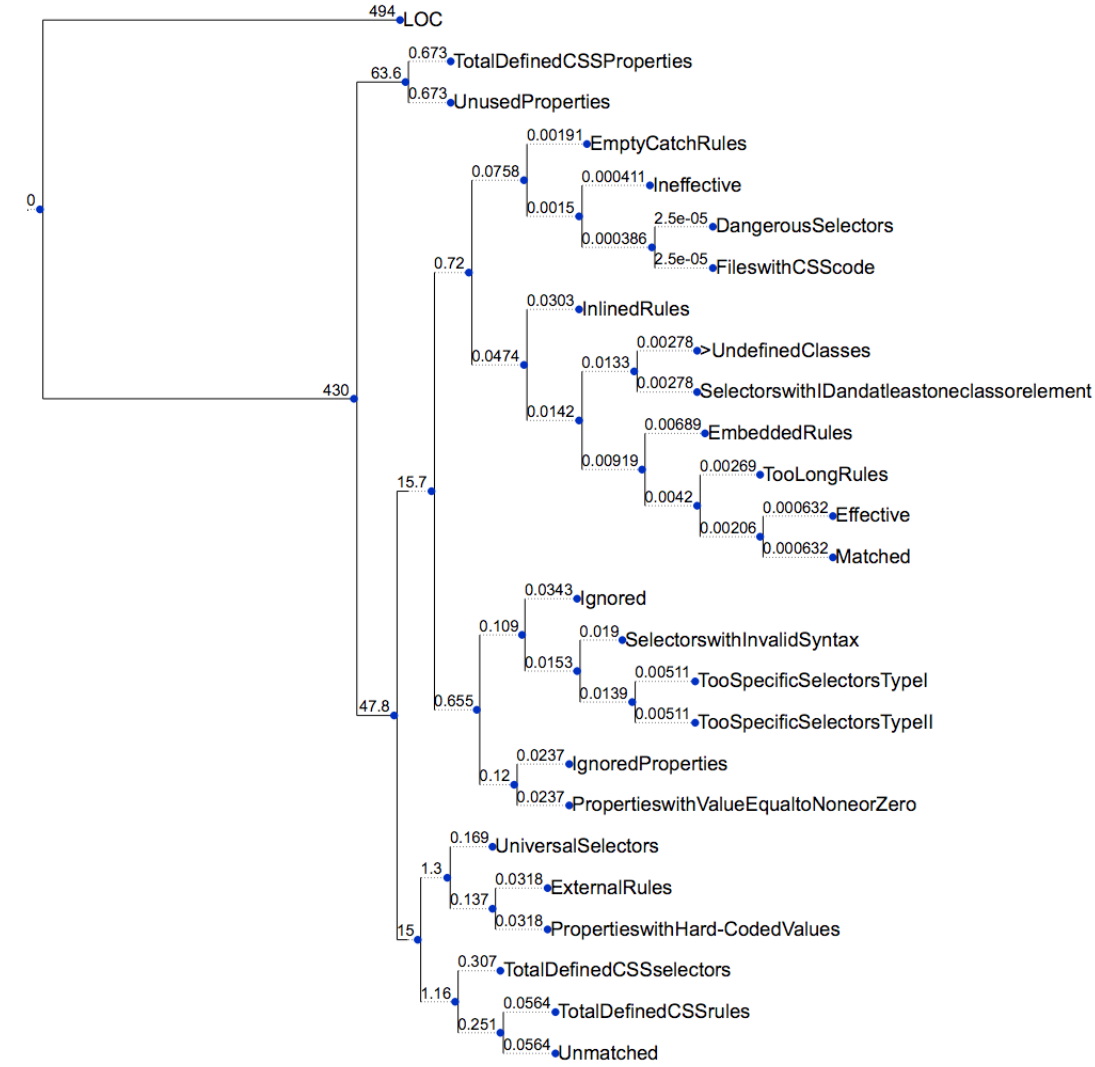

Figure 7.3: The dendrogram produced by agglomerative clustering of code smells and metrics.

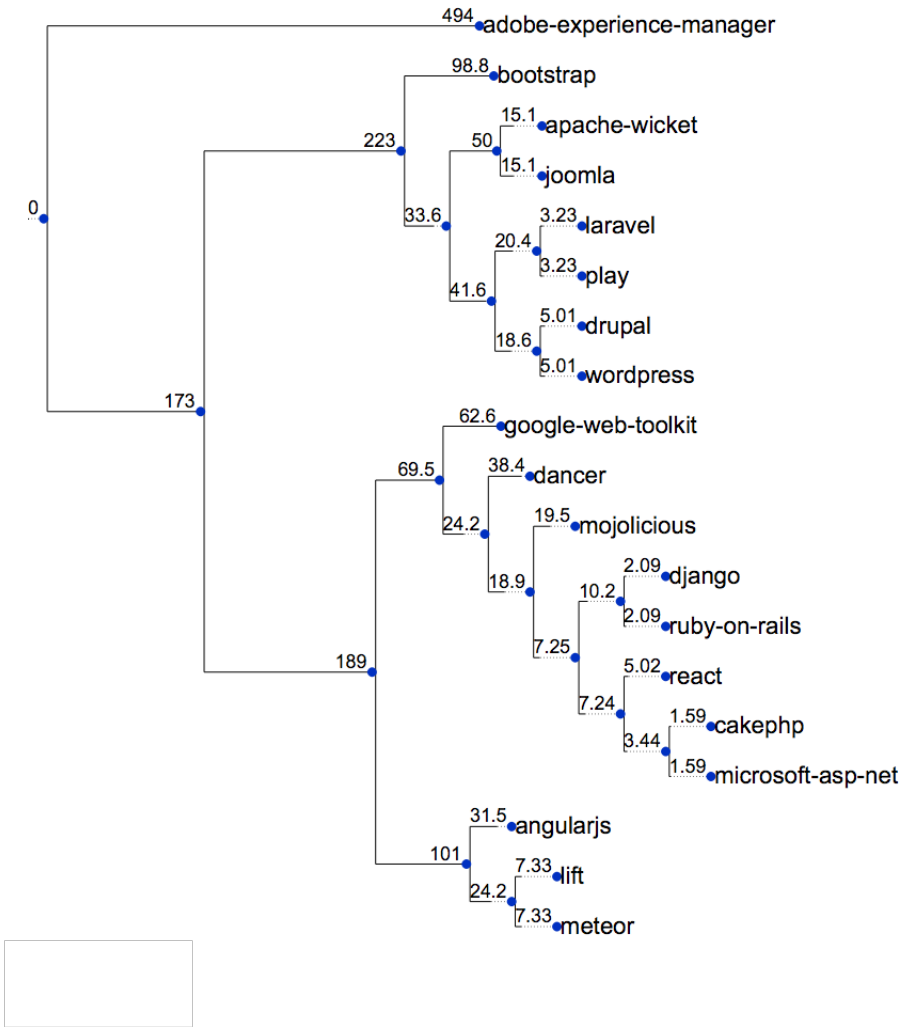

Figure 7.4: The dendrogram produced by agglomerative clustering of frameworks.

Chapter 8

THREATS TO VALIDITY

The main threat to validity facing this work is the accuracy of the tool used for the collection of the CSS code smells and metrics, CSSNose. CSSNose and the tool upon which it's based, Cilla, have both been manually verified in previous works and shown to have a high level of accuracy, but CSSNose required some adaption to be used in the empirical study. It's possible that the code smells and metrics collected aren't entirely accurate for all websites. Other threats include the analysis of sites with a restrictive attitude towards automatic web crawlers as defined by the *robots.txt*, resulting in the retrieval of a limited amount of pages that may not give a truly representative sample of a website.

Chapter 9

CONCLUSION & FUTURE WORK

9.1 Conclusion

In this thesis, we attempt to recognize a pattern of CSS behavior in web frameworks. We collect a dataset of several hundred websites produced by each of 19 different frameworks, collect code smells and other metrics from the CSS code of each website using the tool CSSNose, train a classifier to predict which framework the website was built with, and perform various clustering tasks to gain insight into the correlations between code smells. Our results show that CSS code smells are highly prevalent in websites built with web frameworks, we achieve an accuracy of 39% in correctly classifying the frameworks based on CSS code smells and metrics, and we find interesting correlations between code smells and metrics. In addition to these results, the contributions of this paper include descriptive statistics of the CSS practices of web frameworks, a feature added to the tool CSSNose to analyze inlined CSS code, the dataset of CSS metrics and code smells from 5,525 websites built with 19 different web frameworks, and three ideas of coupling that provide a holistic view of Separation of Concerns between CSS and HTML and provide web developers with a mental model and terminology. Most importantly, the findings in this work will allow for an investigation of the characteristics and design decisions in web frameworks that allow for the introduction of CSS code smells. All software artifacts and datasets are publicly available at `https://github.com/bleischt/`.

9.2 Future Work

This thesis constitutes a first step in the attempt to better understand CSS behavior in websites built with web frameworks and can be expanded upon in several ways. First and foremost, improving the strength and scope of the analysis can be achieved by adding more CSS code smells and metrics to the analysis, such as the smells and syntax errors captured by W3C Validator [61] and CSS Lint [7], duplicated CSS code captured by Mazinanian's tool [43], Keller and Nussbaumer's notion of abstractness [37], and Adewumi et al.'s complexity metrics [13] [14]. The scope could be further expanded by accounting for CSS code embedded in JavaScript and server-side code (perhaps using techniques explored by Nguyen et al. in [50]) and by extending the code smell, metrics, and syntax error analysis to HTML and JavaScript in order to analyze the practices of frameworks more holistically.

Secondly, because this work has established a correlation between CSS code smells and web frameworks, further studies can investigate the characteristics and design decisions of web frameworks that account for the introduction of these code smells in websites. In this regard, it would be interesting to analyze the differences in practices between various frameworks, the developer communities surrounding these frameworks, and the typical applications of these frameworks (the types of websites typically built with certain frameworks). The clustering plots and dendrograms may serve as a good starting point for hypothesizing about the various characteristics of frameworks that may affect CSS development with an ultimate goal of understanding and improving the development processes of frameworks that are considered problematic.

Lastly, the correlations discovered between various CSS code smells and metrics can serve as the basis for further investigation into CSS development and maintenance techniques. The clustering plots and dendrograms may serve as a good starting point for hypothesizing about the interactions between certain code smells and metrics and

what factors of development may cause a developer to introduce code smells into CSS code, inhibit their maintenance efforts, or inhibit their understanding of the language features. Future work in this direction combined with a predictor for the framework a website was built with could contribute to the creation of a robust tool for CSS maintenance which can provide CSS code smell refactorings and code recommendations to the developer based on the observed difficulties with the framework being used in creation of the website.

BIBLIOGRAPHY

[1] What is the document object model? URL https://www.w3.org/TR/WD-DOM/introduction.html. Accessed: 2017-04-04.

[2] At rules. https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule. Accessed: 2017-05-02.

[3] Salt lab - software analysis and testing - cssnose. http://salt.ece.ubc.ca/software/cssnose.

[4] Cal Poly Github. http://www.github.com/CalPoly.

[5] Introduction to html 4. https://www.w3.org/TR/REC-html40/intro/intro.html#h-2.2. Accessed: 2017-05-02.

[6] A short history of javascript. https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. Accessed: 2017-05-09.

[7] Css lint - automated linting of cascading stylesheets. https://github.com/CSSLint/csslint. Accessed: 2017-04-09.

[8] Pseudo classes. https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes, . Accessed: 2017-05-02.

[9] Pseudo elements. https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements, . Accessed: 2017-05-02.

[10] Css selectors - css. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors. Accessed: 2018-02-28.

[11] Uniform resource locators (url). https://www.w3.org/Addressing/URL/url-spec.txt. Accessed: 2018-02-26.

[12] Extensible markup language (xml) 1.0 (fifth edition). https://www.w3.org/TR/REC-xml/. Accessed: 2017-05-02.

[13] Adewole Adewumi, Sanjay Misra, and Nicholas Ikhu-Omoregbe. Complexity metrics for cascading style sheets. In Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, Ana Maria A. C. Rocha, David Taniar, and Bernady O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2012: 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part IV*, pages 248–257, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-31128-4. doi: 10.1007/978-3-642-31128-4_18. URL https://doi.org/10.1007/978-3-642-31128-4_18.

[14] Adewole Adewumi, Onyeka Emebo, Sanjay Misra, and Luis Fernandez. Tool support for cascading style sheets' complexity metrics. In Rolly Intan, Chi-Hung Chi, Henry N. Palit, and Leo W. Santoso, editors, *Intelligence in the Era of Big Data: 4th International Conference on Soft Computing, Intelligent Systems, and Information Technology, ICSIIT 2015, Bali, Indonesia, March 11-14, 2015. Proceedings*, pages 551–560, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46742-8. doi: 10.1007/978-3-662-46742-8_50. URL https://doi.org/10.1007/978-3-662-46742-8_50.

[15] Amaia Aizpurua, Myriam Arrue, Markel Vigo, and Julio Abascal. Exploring Automatic CSS Accessibility Evaluation. *LNCS*, 5648:16–29, 2009.

[16] Elbert Alias. Wappalyzer, feb 2018. URL https://www.wappalyzer.com.

[17] Greg J Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint Cascading Style Sheets for the Web. 1999.

[18] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Synthesizing Web Element Locators (T). In *2015 30th IEEE/ACM International Conference on*

Automated Software Engineering (ASE), pages 331–341. IEEE, nov 2015. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.23. URL http://ieeexplore.ieee.org/document/7372022/.

[19] Mehmet Serdar Biçer and Banu Diri. Predicting Defect Prone Modules in Web Applications. pages 577–591. Springer, Cham, 2015. doi: 10.1007/978-3-319-24770-0_49. URL http://link.springer.com/10.1007/978-3-319-24770-0{_}49.

[20] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Reasoning with style. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 2227–2233. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://dl.acm.org/citation.cfm?id=2832415.2832558.

[21] Scott Buckley, Anthony Sloane, and Matthew Roberts. Specifying CSS layout with reference attribute grammars. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity - SPLASH Companion 2016*, pages 29–30, New York, New York, USA, 2016. ACM Press. ISBN 9781450344371. doi: 10.1145/2984043.2989216. URL http://dl.acm.org/citation.cfm?doid=2984043.2989216.

[22] Alan Charpentier, Jean-Remy Falleri, and Laurent Reveillere. Automated Extraction of Mixins in Cascading Style Sheets. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 56–66. IEEE, oct 2016. ISBN 978-1-5090-3806-0. doi: 10.1109/ICSME.2016.15. URL http://ieeexplore.ieee.org/document/7816454/.

[23] C. Darken and J. Moody. Fast adaptive k-means clustering: some empirical

results. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 233–238 vol.2, June 1990. doi: 10.1109/IJCNN.1990.137720.

[24] María del Pilar Salas-Zárate, Giner Alor-Hernández, Rafael Valencia-García, Lisbeth Rodríguez-Mazahua, Alejandro Rodríguez-González, and José Luis López Cuadrado. Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, 102(Supplement C):1 – 19, 2015. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2014.12.004. URL `http://www.sciencedirect.com/science/article/pii/S0167642314005735`.

[25] Edsger W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982. ISBN 978-1-4612-5695-3. doi: 10.1007/978-1-4612-5695-3_12. URL `http://dx.doi.org/10.1007/978-1-4612-5695-3_12`.

[26] Selenium Web Driver. Selenium hq - browser automation, feb 2018. URL `https://hotframeworks.com/`.

[27] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., Boca Raton, FL, USA, 3rd edition, 2014. ISBN 1439838224, 9781439838228.

[28] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[29] Hot Frameworks. Hot frameworks, feb 2018. URL `https://hotframeworks.com/`.

[30] Pierre Geneves, Nabil Layaida, and Vincent Quint. On the analysis of cascading style sheets. In *Proceedings of the 21st international conference on World Wide Web - WWW '12*, page 809, New York, New York, USA, 2012. ACM Press.

ISBN 9781450312295. doi: 10.1145/2187836.2187946. URL http://dl.acm.org/citation.cfm?doid=2187836.2187946.

[31] R. Gentleman and V. J. Carey. *Unsupervised Machine Learning*, pages 137–157. Springer New York, New York, NY, 2008. ISBN 978-0-387-77240-0. doi: 10.1007/978-0-387-77240-0_10. URL https://doi.org/10.1007/978-0-387-77240-0_10.

[32] Golnaz Gharachorlu. Code smells in cascading style sheets: an empirical study and a predictive model. Master's thesis, University of British Columbia, 2014. URL https://open.library.ubc.ca/cIRcle/collections/24/items/1.0167067.

[33] Matthew Hague, Anthony W. Lin, and C.-H. Luke Ong. Detecting redundant css rules in html5 applications: A tree rewriting approach. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 1–19, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814288. URL http://doi.acm.org/10.1145/2814270.2814288.

[34] George A. Jennings. *Modern Geometry with Applications*. Universitext. Springer, 1994.

[35] Gorjan Jovanovski and Vadim Zaytsev Raincode Brussels. Critical CSS Rules - Decreasing time to first render by inlining CSS rules for over-the-fold elements.

[36] Bernat Kallo. Extending the cascading style sheets (css) language with programming constructs. Master's thesis. URL http://urn.fi/URN:NBN:fi:aalto-201509184442.

[37] M. Keller and M. Nussbaumer. Css code quality: A metric for abstractness; or why humans beat machines in css coding. In *2010 Seventh International Con-*

ference on the Quality of Information and Communications Technology, pages 116–121, Sept 2010. doi: 10.1109/QUATIC.2010.25.

[38] Matthias Keller and Martin Nussbaumer. Cascading Style Sheets: A Novel Approach Towards Productive Styling with Today's Standards.

[39] Sotiris Kotsiantis. Supervised machine learning: A review of classification techniques. 31, 10 2007.

[40] Hsiang-Sheng Liang, Kuan-Hung Kuo, Po-Wei Lee, Yu-Chien Chan, Yu-Chin Lin, and Mike Y. Chen. SeeSS. In *Proceedings of the 26th annual ACM symposium on User interface software and technology - UIST '13*, pages 353–356, New York, New York, USA, 2013. ACM Press. ISBN 9781450322683. doi: 10.1145/2501988.2502006. URL `http://dl.acm.org/citation.cfm?doid=2501988.2502006`.

[41] Håkon Wium Lie. PhD Thesis: Cascading Style Sheets, 2005. URL `https://people.opera.com/howcome/2006/phd/`.

[42] P.M. Marden and E.V. Munson. Today's style sheet standards: the great vision blinded. *Computer*, 32(11):123–125, 1999. ISSN 00189162. doi: 10.1109/2.803645. URL `http://ieeexplore.ieee.org/document/803645/`.

[43] Davood Mazinanian. Eliminating code duplication in cascading style sheets. September 2017. URL `https://spectrum.library.concordia.ca/982965/`.

[44] Davood Mazinanian and Nikolaos Tsantalis. Migrating cascading style sheets to preprocessors by introducing mixins. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 672–683, New York, New York, USA, 2016. ACM Press. ISBN 9781450338455. doi: 10.1145/2970276.2970348. URL `http://dl.acm.org/citation.cfm?doid=2970276.2970348`.

[45] Davood Mazinanian and Nikolaos Tsantalis. An Empirical Study on the Use of CSS Preprocessors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 168–178. IEEE, mar 2016. ISBN 978-1-5090-1855-0. doi: 10.1109/SANER.2016.18. URL `http://ieeexplore.ieee.org/document/7476640/`.

[46] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 496–506, New York, New York, USA, 2014. ACM Press. ISBN 9781450330565. doi: 10.1145/2635868.2635879. URL `http://dl.acm.org/citation.cfm?doid=2635868.2635879`.

[47] A. Mesbah, E. Bozdag, and A. v. Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134, July 2008. doi: 10.1109/ICWE.2008.24.

[48] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE, jun 2012. ISBN 978-1-4673-1066-6. doi: 10.1109/ICSE.2012.6227174. URL `http://ieeexplore.ieee.org/document/6227174/`.

[49] Eric A. Meyer. *CSS: The Definitive Guide*. O'Reilly, 3 edition, 2006. ISBN 978-0596527334.

[50] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 282, New York, New York,

USA, 2012. ACM Press. ISBN 9781450312042. doi: 10.1145/2351676.2351724. URL http://dl.acm.org/citation.cfm?doid=2351676.2351724.

[51] Kshirsagar A. P. and Rathod M. N. Article: Artificial neural network. *IJCA Proceedings on National Conference on Recent Trends in Computing*, NCRTC (2):12–16, May 2012. Full text available.

[52] Anirudh Prabhu. Introduction to Preprocessors. In *Beginning CSS Preprocessors*, pages 1–12. Apress, Berkeley, CA, 2015. doi: 10.1007/978-1-4842-1347-6_1. URL http://link.springer.com/10.1007/978-1-4842-1347-6{_}1.

[53] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010. ISBN 0073375977, 9780073375977.

[54] Leonard Punt, Sjoerd Visscher, and Vadim Zaytsev. The A?B*A Pattern: Undoing Style in CSS and Refactoring Opportunities It Presents. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 67–77. IEEE, oct 2016. ISBN 978-1-5090-3806-0. doi: 10.1109/ICSME.2016.73. URL http://ieeexplore.ieee.org/document/7816455/.

[55] Vincent Quint and Irne Vatton. Editing with style. In *Proceedings of the 2007 ACM Symposium on Document Engineering*, DocEng '07, pages 151–160, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-776-6. doi: 10.1145/1284420.1284460. URL http://doi.acm.org/10.1145/1284420.1284460.

[56] Scikit-Learn. Scikit-learn - machine learning in python, feb 2018. URL http://scikit-learn.org/stable/.

[57] M. Serdar Biçer and Banu Diri. Defect prediction for Cascading Style Sheets. *Applied Soft Computing*, 49:1078–1084, 2016. ISSN 15684946. doi: 10.1016/j.asoc.2016.05.038.

[58] Manuel Serrano and Manuel. HSS. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '10*, page 109, New York, New York, USA, 2010. ACM Press. ISBN 9781450301329. doi: 10.1145/1836089.1836104. URL `http://portal.acm.org/citation.cfm?doid=1836089.1836104`.

[59] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. ISSN 0018-8670. doi: 10.1147/sj.132.0115.

[60] Harry Thornburg. Introduction to bayesian statistics, mar 2001. URL `http://ccrma.stanford.edu/~jos/bayes/bayes.html`.

[61] W3C. Css markup validation service, jan 2013. URL `https://jigsaw.w3.org/css-validator/`.

[62] Bart De Win, Frank Piessens, Wouter Joosen, Bart De, Win Frank, Piessens Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering, 2002.

A CSSNose Code Smell Name Mapping

The following code smells were defined by Gharachorlu in [32] but renamed by Gharachorlu in the CSSNose tool output:

1. **Non-External Rule** —> Embedded Rules
2. **Too Long Rules** —> Too Long Rules
3. **Too Much Cascading** —> Too Specific Selectors Type I
4. **High Specificity Values** —> Too Specific Selectors Type II
5. **Selectors with Erroneous Adjoining Pattern** —> Selectors with Invalid Syntax
6. **Too General Selectors** —> Dangerous Selectors
7. **Properties with Hard-Coded Values** —> Properties with Hard-Coded Values
8. **Undoing Style** —> Properties with Value Equal to None or Zero