Law Smells

Defining and Detecting Problematic Patterns in Legal Drafting

Corinna Coupette[1,2] · **Dirk Hartung[2,3]** · **Janis Beckedorf[2,4]** ·
Maximilian Böther[5] · **Daniel Martin Katz[2,3,6]**

Abstract

Building on the computer science concept of *code smells*, we initiate the study of *law smells*, i.e., patterns in legal texts that pose threats to the comprehensibility and maintainability of the law. With five intuitive law smells as running examples—namely, duplicated phrase, long element, large reference tree, ambiguous syntax, and natural language obsession—, we develop a comprehensive law smell taxonomy. This taxonomy classifies law smells by when they can be detected, which aspects of law they relate to, and how they can be discovered. We introduce text-based and graph-based methods to identify instances of law smells, confirming their utility in practice using the United States Code as a test case. Our work demonstrates how ideas from software engineering can be leveraged to assess and improve the quality of *legal* code, thus drawing attention to an understudied area in the intersection of law and computer science and highlighting the potential of computational legal drafting.

Keywords Refactoring · Software engineering · Law · Natural language processing · Network analysis

✉ Corinna Coupette
coupette@mpi-inf.mpg.de

✉ Dirk Hartung
dirk.hartung@law-school.de

1 Max Planck Institute for Informatics, Saarbrücken, Germany

2 Center for Legal Technology and Data Science, Bucerius Law School, Hamburg, Germany

3 CodeX - The Stanford Center for Legal Informatics, Stanford Law School, Stanford, CA, USA

4 Ruprecht-Karls-Universität Heidelberg, Heidelberg, Germany

5 Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

6 Illinois Tech - Chicago Kent College of Law, Chicago, IL, USA

1 Introduction

In modern societies, law is one of the main tools to regulate human activities. These activities are constantly changing, and law co-evolves with them. In the past decades, human activities have become increasingly differentiated and intertwined, e.g., in developments described as *globalization* or *digitization*. Consequently, legal rules, too, have grown more complex, and statutes and regulations have increased in volume, interconnectivity, and hierarchical structure (Katz et al. 2020; Coupette et al. 2021a).

A similar trend can be observed in software engineering, albeit on a much shorter time scale. The global codebase has grown exponentially in recent years (Yu et al. 2014), with GitHub as the largest source code host alone accounting for 250 million repositories between 2008 and 2020.[1] Over the course of its growth, this codebase has become increasingly interconnected when viewed through the lens of network analysis (Lima et al. 2014). Naturally, software engineers have turned to technology to keep track of this development and manage code interdependencies. While the challenges for law and software engineering and the constraints within which these challenges must be addressed are not identical, both domains share three important characteristics: *Materially*, their subject matters, legal rules on the one hand and code fragments on the other hand, contain commands intended to control (human or machine) behavior in order to achieve specific outcomes. *Procedurally*, output creation in both law and software engineering is distributed in time and space, and thus, both domains are subject to the challenges of dynamic multi-agent systems. *Methodologically*, lawyers and software engineers alike use abstractions to accommodate problems that are not fully known in the present.

The similarities between the law and software engineering domains suggest possibilities for knowledge transfer between them. In this paper, we explore one such possibility for the software engineering subfield of *refactoring*. Introduced into academic discourse by Opdyke and Johnson (1990) and popularized by Becker et al. (1999) (Second Edition: Fowler 2018), refactoring "describes a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."[2] In software engineering, refactoring is indispensable for ensuring software quality and maintainability, and it is also subject to vivid academic discourse, inspiring detailed analyses of large code bases and even dedicated conferences.[3] This has resulted in an actionable understanding of how various variables— e.g., programming language, team size, project size, or commit size (Ray et al. 2017), repetition and duplicated code (Lopes et al. 2017), or component size and open source model (Stamelos et al. 2002)— impact code quality.

In this paper, we demonstrate how concepts from refactoring can be used in law, focusing on the example of *code smells*. At a high level, a code smell is a characteristic of (a part of) the source code that may indicate a deeper problem in its

---

[1] According to https://octoverse.github.com, 60 million repositories were created in 2020.

[2] Cf. https://refactoring.com/.

[3] Cf. http://www.msrconf.org/.

design and implementation, highlighting a need for refactoring (Tufano et al. 2015). We port this idea to the legal domain, introducing the concept of *law smells*. Law smells constitute the first step towards both (semi-)automatically *detecting* problematic parts of existing codifications and (semi-)automatically *improving* codification quality. They also pave the path towards reproducible, *quantitative* quality measures for collections of legal rules, which allow us to assess the overall *law climate* and facilitate automatic code review and quality assurance.

The remainder of this paper is structured as follows. In Sect. 2, we present related literature from law and computer science, along with interdisciplinary work in the intersection of both fields. We develop the concept of *law smells*, guided by five illustrative examples, in Sect. 3, and describe methods to detect law smells in Sect. 4. In Sect. 5, we demonstrate the utility of our methods for detecting five example smells, deliberately showcasing a variety of ways to present law smell detection results to end users. We discuss the limitations of our approach along with opportunities for future work in Sect. 6, and conclude in Sect. 7.

2 Related work

Given its interdisciplinary nature, our work relates to prior research in law, in computer science, and in the intersection of both fields. In the legal domain, the question of what constitutes high-quality law is the realm of legislative theory (Noll 1973), sometimes also referred to as *legisprudence* (Wintgens 1999). Within this wider field, our work is most closely related to research on legislative drafting, which, inter alia, develops suggestions to mitigate law smells. Therefore, our work should be read in light of the debate on drafting manuals and other guidelines for the legislative process (cf. van Lochem and Westerman 2010; Xanthaki 2010). While there exist handbooks that cover a wide range of drafting techniques and best practices (cf. Xanthaki 2014), our work helps detect where these can be applied to improve existing legal texts. Our improvements do not focus on criteria such as coherence, precision (cf. Karpen 2008), or "plain" language (cf. Butt 2013), which are subject to an adjacent debate (cf. Mousmouti 2012; Xanthaki 2011). Moreover, there is extensive doctrinal debate on the use of digital technology in the legislative process (cf. Dorsey 2014; Sartor 2008), for which law smells in legislative documents are an example.

In computer science, this paper draws inspiration from the work by Fowler (2018) as laid out in Sect. 1. As the survey by Sharma and Spinellis (2018) demonstrates, *code smells* or *software smells* are subject to extensive research, including in-depth studies on their limitations, interactions, and relations to other concepts such as software design ideas (cf. Yamashita 2013; Speicher 2020). Since the transfer of code smells to the legal domain is in an early stage, no meta-analysis currently exists for law smells, and we are only beginning to understand their prevalence and impact.

While the use of some concepts from mathematics or computer science, such as symbolic or deontic logic (cf. Allen 1957; Allen and Engholm 1980), or domain-specific (markup) languages (cf. Allen and Saxon 1995), in legal drafting has been debated for several decades (cf. Saxon 1982; Ziegler 1989), few publications in the

intersection of law and computer science have used ideas from software engineering (e.g., debugging Fungwacharakorn et al. (2021)) in the legal domain. Among these, the work by Li et al. (2015) is most similar to ours, as the authors adapt four code quality metrics to legal texts in order to quantitatively assess the quality of the United States Code. The metrics they select differ from ours, however, and the results are not embedded in an overarching conceptual framework. Our work shares the goal of improving legislative texts using insights from computer science with papers applying computer science techniques to improve statutory texts directly (e.g., measuring and improving readability; cf. Alschner et al. 2020; Curtotti et al. 2015). Also related to our topic is the study by Marcus (2012), which introduces fifteen types of legal programming errors in telecommunication law and discusses ways to resolve them. While starting from a similar point of departure, our work is distinct in that it provides operational tools to detect the deficiencies of and advocates for solutions independent from specific programming paradigms. Finally, the tools presented in this paper could be used as a building block for legal drafting software, whose potential (including, e.g., the automatic detection of conflicts between individual provisions by Li et al. (2013), or in-domain style checking by Sugisaki (2016)) has been explored in prior work (cf. Moens 2006; Hafner and Lauritsen 2007) or combined with existing approaches to detect undesired patterns in statutory texts Markovich and Hamp (2015). As such, our ideas relate to the grander vision of a computerized legal information (retrieval) system (Bing 1988; Liebwald 2015), which has been discussed for several decades, yet will probably never be fully developed (Bing 2010). Our concept of law smells is much more narrow but directly applicable to existing corpora of legal texts. Hence, unlike a computerized legal information (retrieval) system, its implementation does not require large infrastructure investments.

3 Theory: What are law smells?

In this section, we make the notion of a *law smell* more precise (3.1), illustrate it via examples (3.2), and develop a taxonomy of law smells that prove particularly problematic in practice (3.3). For conciseness, we refer to the government parties involved in creating legal rules as *lawmakers* (e.g., parliaments or agencies), to the parties affected by such legal rules as *lawtakers* (e.g., law firms, companies, and citizens), to people with a legal education as *lawyers* (e.g., judges, attorneys, and legal scholars), and to the labeled, often nested units of legal texts as *elements of law* (e.g., Titles, Chapters, Sections, and Subsections of the United States Code).

Table 1 Categories used in a law smell profile

#	Category	Question
1	Description	What is it?
2	Problem	Why is it problematic?
3	Detection	How do we detect it?
4	Mitigation	If we decide that it needs mitigation, how can we mitigate it?
5	Example	For illustration, how does it manifest in practice?

3.1 Definition

A *code smell*, as understood by Fowler (2018), is "a surface indication that usually corresponds to a deeper problem in the system."[4] Making only small changes to this statement, we arrive at our working definition of a *law smell*:

Definition 1 (Law Smell) A *law smell* is a surface indication that usually corresponds to a deeper problem in a legal system.

To clarify what this means, we highlight the most important building blocks of our definition.[5] First, a law smell is a *surface indication*: It can be easily identified, perhaps algorithmically, from the words and the structure of a legal text— even with very limited semantic knowledge. Second, the problem indicated by a law smell concerns *a legal system*, which—regardless of how exactly we define it[6] — goes beyond the body of rules in which the problem is identified (for example, a smell discovered in the text of a regulation can indicate a problem in the statute-making process). Third, the indicated problem is a *deeper problem*, that is, an aspect of the design of a legal system that affects its usability for at least one of its stakeholders— e.g., by making it harder to maintain (for lawmakers) or to navigate (for lawtakers). Finally, a law smell only *usually* indicates a problem, i.e., upon closer inspection, it can turn out to be harmless or even necessary, given the legal rules that need to be expressed. From a practical perspective, law smells can tell us where we should look to improve the legal system (as lawmakers) or where we should pay extra caution (as lawtakers).

So far, we have defined law smells in terms of their *form* and their *consequences*. To capture their *content*, we propose to characterize each law smell systematically,

---

[4] Cf. https://martinfowler.com/bliki/CodeSmell.html.

[5] Some of the following clarifications are adapted to the legal domain from clarifications found for the code smell definition at https://martinfowler.com/bliki/CodeSmell.html.

[6] Defining what a legal system is, or when it exists, is a classic question in legal theory, see, e.g., Hart (1961); Raz (1970/1980); Luhmann (1987).

using the five categories laid out in Table 1. We refer to the resulting overview as a *law smell profile*. For a given law smell, a law smell profile sketches (1) what the law smell is and (2) why it is problematic. It summarizes (3) how we might detect the smell and (4) how we might mitigate it, and gives (5) at least one illustrative example.

3.2 Examples

We now introduce some particularly intuitive example law smells, which will accompany us throughout the paper. To maximize comprehensibility for readers with computer science or legal backgrounds, we choose law smells that can be derived both via adaptation of the code smells known from the software engineering literature (e.g. Fowler 2018) and by appealing to the intuition and experience of lawyers. As the primary objective is to build intuition, our discussion is deliberately brief, and we detail the structured profiles of our example law smells in the Supplementary Information. We provide a broader view of the law smell landscape in Sect. 3.3, elaborate on the critical question of *law smell detection* in Sect. 4, and assess the practical prevalence of our example smells in Sect. 5.

3.2.1 Duplicated phrase

Duplicated phrase is the legal equivalent of the software engineering code smell *duplicated code*. Lawyers smell it when they get the feeling that a text is verbose and repetitive. More formally, a duplicated phrase is a phrase above a specified *length* that has more than a specified number of *occurrences* in a legal text. Here, we construe "phrase" broadly and allow phrases to have parameters. That is, a *phrase* is a nonempty sequence of terms, where a term is either a token (i.e., a sequence of non-whitespace characters roughly corresponding to a word) or a placeholder for an argument that is itself a phrase (e.g., "not later than {period} after {date}").

Duplicated phrases are problematic because they increase the risk of inconsistencies introduced by incomplete edits, and because they communicate legal content inefficiently. We can detect duplicated phrases using scalable variants of *n*-gram search, possibly after some preprocessing, where, for intuitive results, the length and occurrence thresholds should be adaptive and inversely correlated— i.e., the longer a phrase, the fewer occurrences should be required for it to qualify as duplicated (see Sect. 4.3.1 for details). To eliminate duplicated phrases, we can introduce and rigorously reuse named variables and definitions.

3.2.2 Long element

Long element is a legal adaptation of the software engineering code smell *long function*. Lawyers smell it when they get lost in the text they are reading or have trouble discerning those parts of the text that are relevant to them. More formally, a long element is an element of law containing text that is long as assessed by some absolute or relative measure (e.g., its number of tokens or its quantile in a token length

distribution). Long elements are problematic because they may indicate a lack of structure or abstraction in the legal text, making the law both harder to read and more complicated to maintain. We can detect long elements using domain-specific adaptations of outlier detection methods (see Sect. 4.3.2 for details). To eliminate long elements, we can move some of their text to other (preexisting or newly created) elements or rewrite the text itself, e.g., by applying mitigation strategies for other law smells.

3.2.3 Ambiguous syntax

Ambiguous syntax is a legal adaptation of the software engineering code smell *mysterious name*, with pinches of *repeated switches* and *speculative generality*. Lawyers smell it, inter alia, when they litigate over the meaning of commas or argue about whether an *or* is inclusive or exclusive. More formally, ambiguous syntax is the use of logical operators (e.g., *and*, *or*, and *no(t)*), control flow operators (e.g., *if*, *else*, and *while*), or punctuation (e.g., commas and semicolons) in a way that leaves room for interpretation. Ambiguous syntax is problematic because it creates legal uncertainty, which is often removed through costly lawsuits and sometimes leads lawmakers to adopt *mathematically redundant syntax* like *and/or*.[7] We can detect instances of potentially ambiguous or mathematically redundant syntax using pattern matching with regular expressions (see Sect. 4.3.3 for details). To eliminate truly ambiguous instances, logical operators can be used with their mathematical meaning, *xor* can be introduced as a shorthand for the exclusive *or* (reserving *or* for the cases with inclusive meaning), and brackets can be added as syntax (e.g., to clarify operator binding).

3.2.4 Large reference tree

Large reference tree is a legal adaptation of the software engineering code smell *message chain*. Lawyers smell it when they find themselves following many references to understand the full meaning of the text they were originally interested in. More formally, a *reference tree* rooted at an element of law r is a tuple $T_r = (V_r, E_r)$, where V_r is the set of elements of law reachable from r by following references (including r), and E_r is a minimal set of edges (references), which is generally not unique, such that each element of V_r can be reached from r. To illustrate this definition, a sample reference tree is depicted in Fig 1. A *large reference tree* is a reference tree whose edge set exceeds a given size x, which represents the largest acceptable number of references that we must follow to fully understand the content of a root element. The simplest case of a large reference tree is a *long reference chain*. Large reference trees are problematic because they increase the cognitive load

---

[7] For an example of costly syntactic ambiguity that received plenty of media coverage in 2018, see O'Connor v. Oakhurst Dairy – 851 F.3d 69 (1st Cir. 2017), where the opinion of the First Circuit starts with "For want of a comma, we have this case". In defense of the mathematically redundant *and/or*, see Robbins (2017).
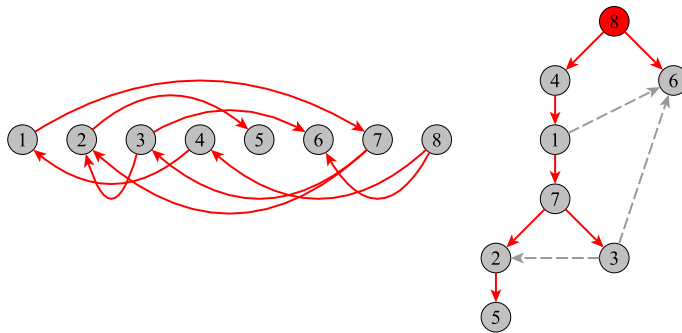
Fig. 1 Toy elements of law connected by reference edges (left) and a reference tree rooted at element 8 (right), where edges not in E_8 are drawn as gray dashed arrows

involved in navigating legal texts, and they raise the risk of unintended normative side effects when elements deep in the tree hierarchy are changed. We can detect large reference trees using domain-specific adaptations of graph traversal algorithms (see Sect. 4.3.4 for details). To eliminate large reference trees, we can restructure the texts and references contained in its elements, and we can make all references as specific as possible.

3.2.5 Natural language obsession

Natural language obsession is a legal adaptation of the software engineering code smell *primitive obsession*. Lawyers smell it, inter alia, when they are interested in the law related to a specific named entity (e.g., a committee of the United States Senate), or when they try to automate their clients' compliance. More formally, natural language obsession is the representation of typed data as natural language text, often without a standardized format. It is problematic because typed data is notoriously hard to extract or maintain when represented using (inconsistently formatted) natural language. We can detect natural language obsession using Named Entity Recognition methods (including methods as simple as pattern matching with regular expressions), potentially augmented or validated by techniques detecting duplicated phrases (see Sect. 4.3.5 for details). To eliminate natural language obsession, we can create a data layer that is separate from the text (representation) layer, use strong types for named entities, and associate type checking, highlighting, and data analysis rules with the introduced types.

3.3 Taxonomy

We now abstract and expand the ideas introduced above into a taxonomy of law smells. To this end, we first derive the necessary distinctions and then give more details on ten additional smells that we position within the resulting taxonomy. This taxonomy is not meant to be exhaustive. Rather, our goal is to provide a starting point for conceptual and empirical validation, discussion, and further refinement.

Table 2 Overview of distinctions used in the law smell taxonomy

Distinction	Categories	Question
Stability	Static, dynamic	Can it be detected by looking at the text alone?
Modality	Data, structure, grammar, relevance	What aspect of the law does it relate to?
Locality	Local, non-local	Can it be detected by looking at a single element of law?

3.3.1 Distinctions

To develop our taxonomy, we observe some commonalities and differences between the exemplary law smells introduced in Sect. 3.2.

First, all of the law smells from Sect. 3.2 can be sniffed from the text only (in programming language parlance: they can be detected at *compile time*; in legal theory parlance: they concern *law in books*), but there might also be law smells that can only be detected when the law is used (i.e., at *runtime*, concerning *law in action*). Thus, we distinguish *static* and *dynamic* law smells.

Second, we find that the *duplicated phrase* and *natural language obsession* smells are strongly related to the content of legal rules, i.e., the *data* of the law, while the *ambiguous syntax* smell is related to how this content is expressed, i.e., the *grammar* of the law, and the *long element* and *large reference tree* smells relate to how the legal content is organized, i.e., the *structure* of the law. Consequently, we differentiate between *data-related*, *grammar-related*, and *structure-related* law smells. Note that since grammar-related law smells are generally static, there is no need for this category in the dynamic setting. Instead, we require an additional dynamic category, called *relevance-related*, to gather smells related to elements of law with limited overall utility (including, but not limited to, *legal cleaning*, i.e., identifying and removing elements of law that are no longer relevant).

Third, while *long element* (when measured in absolute terms, on which we focus here), *ambiguous syntax*, and *natural language obsession* can often be detected by looking at a single element of the law, *duplicated phrase* and *large reference tree* (as well as *long element* when measured in relative terms) mostly require looking at multiple elements of law together. Therefore, we further distinguish *local* (i.e., locally detectable) and *non-local* (i.e., locally undetectable) law smells.

Table 2 summarizes the distinctions just introduced, and Figure 2 gives an overview of the resulting taxonomy. Figure 2 also contains ten new law smells in addition to those already classified above, thus painting a more complete picture of the law smell landscape we envisage.
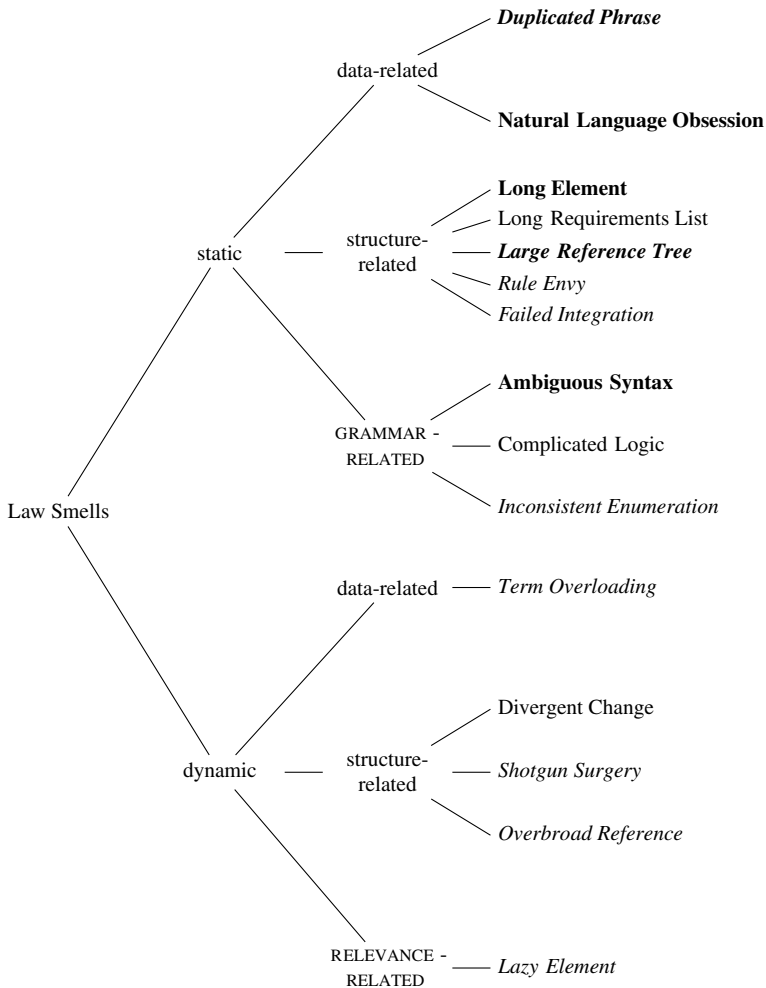
Fig. 2 Depiction of the law smell taxonomy derived in the main text. Smells from Subsection 3.2 are printed in bold, and primarily non-local smells are typeset in italics. The names of subcategories that differ across static and dynamic smells are emphasized using small caps

3.3.2 Details

We now elaborate on the new law smells from Fig. 2. Of these smells, five are static and five are dynamic, and we discuss them in the order in which they occur in Fig. 2.

Long requirements list, a legal adaptation of the software engineering code smell *long parameter list*, is a static law smell describing the situation in which a stated legal consequence only applies given a large number of preconditions. As preconditions might have their own preconditions to be resolved recursively, computer scientists may think of the setting as a search tree whose initial branches need to be expanded until we have either a negative answer on one branch or

positive answers on all branches. The efficient lawyer's task is to pick, using their legal intuition and the facts of an individual case, an expansion order that minimizes the number of search nodes to be expanded. This can only work if they can retain at least the initial preconditions in short-term memory, and therefore, the cognitive load imposed by these preconditions should not exceed the average lawyer's—if not the average lawtaker's— short-term retention capacity. Although the precise measurement of short-term memory is still debated in cognitive psychology, for the purposes of flagging long requirements lists, we can still use a conservative maximum number of requirements x as a heuristic. To shorten long requirements lists, lawmakers can introduce new concepts bundling requirements that cooccur frequently.

Rule envy, a legal adaptation of the software engineering code smell *feature envy*, describes a setting in which a legal rule is more closely related to rules from other parts of the law than to the rules with which it is grouped (e.g., the rules in its own section). Typical warning signs that an element of law is affected by rule envy are its extensive (implicit or explicit) references to rules from far-away parts of the law, paired with an absence of (implicit or explicit) references to rules in its immediate neighborhood. Rule envy is problematic because it goes against the rationale that underlies the structuring of law into separate elements: improving its comprehensibility and maintainability by grouping related content together. *In theory*, rule envy can be mitigated by placing the envious element with the elements it envies. However, *in practice*, many elements of law rely on legal rules from different places, and consequently, it may not always be clear where an element naturally belongs. Furthermore, some prominent drafting techniques such as *index sections* deliberately create rule envy, and when used with caution, the gain in navigational efficiency associated with these techniques may outweigh the drawbacks of the resulting law smell.

Failed integration can be conceived as the group-level opposite of *rule envy*: it refers to a sequence of (mostly lower-level) elements of law that are isolated from the environment in which they are placed— although their position within the structure of the law suggests otherwise. As the train of thought spun in the environment of an affected sequence is interrupted by failed integration, it makes navigating the law unnecessarily difficult. This smell frequently results, e.g., from hasty or unenthusiastic implementation of supranational legal rules into national legal systems (e.g., when the German legislator needs to implement law mandated by the European Union), especially when supranational and national law differ in their legal concepts. As a result, affected sequences often differ from their environment in drafting style and desired interpretation of core terms, which increases, inter alia, the risk of *inconsistent enumeration* and *term overloading* (described below). Failed integration can be mitigated by either moving the affected sequence completely out of its environment, i.e., making it into an independent unit of law, or rephrasing its elements for better coherence with its environment.

Complicated logic arises when legal rules contain nested conditionals, switches, or loops, often with elaborate heads including negations. It sometimes cooccurs with *long requirements list* and frequently results in *ambiguous syntax*. In legal texts just as in code, humans have trouble parsing complicated logic because it puts significant

strain on their mental cache, impairing the usability of the law for both lawmakers and lawtakers. Lawmakers can follow programmers' mitigation principles to eliminate complicated logic, e.g., they can decompose or consolidate conditionals where possible. In extreme cases, e.g., when lawmakers communicate a table in writing (example: 7 U.S.C. §8732, which details loan rates per unit of crop per crop type and year), the best choice is to use an entirely different structure to present the content (in the example: a table).

Inconsistent enumeration refers to the unstandardized usage of enumeration labels (e.g., large Latin alphabets, small Latin alphabets, Arabic numbers, large Roman numerals, small Roman numerals, etc.) across nesting levels in legal texts (or, more mathematically: the lack of a bijection between enumeration labels and nesting levels). As programming languages use the same syntax at all levels to indicate nesting (e.g., lists are nested simply by using list syntax inside list syntax) or enumeration (e.g., switch-case expressions in some languages), this law smell has no close counterpart in the software engineering code smell literature. Its repercussions, however, could be compared to those of overly lenient XML schema definitions: The affected text becomes hard to parse, be it for humans or machines, because the meaning of an encountered nesting element can only be resolved within a larger context. Lawmakers can mitigate inconsistent enumeration by (legally) mandating a one-to-one mapping between enumeration labels and nesting levels, which is then used consistently across legal documents and can, consequently, be relied upon by all stakeholders.

Term overloading, a legal mixture of the software engineering code smells *mysterious name* and *mutable data*, occurs when the same term is used to denote different concepts in different places, either explicitly (*explicit namespacing*, e.g., using *scoping language* such as "For the purposes of this section, …") or implicitly (*implicit namespacing*, e.g., through jurisprudence). It is classified as a dynamic smell because divergent meanings of the same term often result from the interpretation of legal rules in practice, e.g., by courts. Term overloading increases the cost of navigating the law for lawtakers, as the meaning of a term becomes context-dependent, and the context-applicable meaning becomes accessible only through (often non-local) further reading (explicit namespacing) or background knowledge (implicit namespacing). Additionally, it raises maintenance costs for lawmakers, especially when tasked with applying changes to all instances of a concept labeled by an overloaded term. Term overloading can be mitigated by introducing separate labels for each concept referenced by the overloaded term, and by using strategies similar to those combating *natural language obsession*.

Divergent change, a legal adaptation of the eponymous software engineering code smell, describes a situation in which the same element of law needs to be changed in different ways for different reasons. For example, an element holding a sequence of generic legal rules is affected by divergent change if in different application contexts, different specific exceptions need to be made to rules contained in different sequence parts, and the lawmaker's original strategy is to place the exceptions with the generic provisions. Divergent change is a dynamic smell because it can be detected only during maintenance. It indicates that the chosen level of abstraction is inadequate for the affected legal rules, thus pointing to a problem in the structure of the law that may also impair its usability (e.g., the affected element of law may lack

conceptual clarity or be hard to navigate). Divergent change can be counteracted by restructuring the code to enforce a clear separation of concerns between application contexts, such that the rules that are truly shared between contexts are separated from the context-specific details.

Shotgun surgery, like the eponymous software engineering code smell, is the opposite of (the legal adaptation of) divergent change. It captures the setting in which one high-level change to the law necessitates many small changes to legal rules in many different parts of the law, and as such, it often indicates *natural language obsession*. Just as divergent change, shotgun surgery is a dynamic smell because it can only be detected during maintenance. Shotgun surgery is not only slow but also hard for lawmakers to perform consistently (and painful for lawtakers to follow). As a result, conducting shotgun surgery may lead to inconsistencies in the law. This makes future amendments to the law even harder, if not impossible, to perform correctly, and produces unwarranted legal uncertainty (and litigation costs associated with removing that uncertainty) for lawtakers confronted with the resulting inconsistencies. Where it is due to natural language obsession, shotgun surgery can be counteracted by the strategies used to neutralize that smell. Otherwise, close inspection of those parts of the law that are altered by shotgun surgery might yield characteristics that could be abstracted into new concepts or generalized rules for smell mitigation.

An *overbroad reference* is a reference to a structural element of law that is more general than its use case requires. It is a legal adaptation of the software engineering code smell *speculative generality*, which we classify as dynamic because the use case requirements of a legal rule often become clear only as it is used in practice. This smell typically manifests in formulations such as "Subject to the provisions of X [related to Y]", where X is a large structural element of the law, e.g., a Title, a Part, or a Chapter of the United States Code, and Y (if present) is some roughly sketched topic. Overbroad references shift the cost of identifying the *relevant* referenced material from lawmakers to lawtakers, decreasing efficiency and legal certainty and increasing the probability of errors. Lawmakers can eliminate overbroad references by identifying the applicable legal rules themselves and referencing these rules as precisely as possible. Their need for using overbroad references to hedge against future changes could be diminished by using computational tools to track the dependencies among legal rules.

Lazy element, a legal adaptation of the eponymous software engineering code smell, refers to elements of law that are hardly relevant in practice. It is a dynamic smell because both laziness and non-laziness can develop over time: For example, a legal rule can be relied upon by a lot of other rules when it is first promulgated, but newer rules can gradually supersede it, or it can be completely isolated at the start but then form the basis of many lawsuits or contracts. The most extreme case of lazy elements is *dead law*: legal rules that are simply not needed (anymore)— e.g., because their preconditions imply that a person needs to have witnessed a certain event that happened x years ago, and x exceeds maximum observed human lifespan. In the absence of systematic compression or cleaning strategies for law (sunset clauses are still a fringe phenomenon), lazy elements cause bloating, and bloated law is frustrating to read and maintain. Lawmakers can eliminate lazy elements by

integrating them with their more industrious peers or, if the elements turn out to be dead, removing them entirely (through *cleanup acts* such as the *Rechtsbereinigungsgesetze* in Germany).

4 Methods: How do we detect law smells?

In this section, we introduce methods to detect the law smells presented in Sect. 3. After clarifying our assumptions regarding the input we are working with (4.1), we assemble our law smell detection toolkit (4.2), and then provide more detail on the tools to detect our example law smells (4.3). There is currently no single method to reliably detect all or even a majority of the law smells set forth in our taxonomy. Developing a general approach is an important direction for future work, but it will require much more work than what can be presented in a single paper. Therefore, we start by providing a set of techniques suitable for individual law smells, based on common characteristics of legal texts (language and structure). Many of the techniques we build on are simple by themselves, but intricacies arise when adapting them to tackle the *law smell detection problem*. For clarity, we define this problem as follows.

Definition 2 (Law Smell Detection Problem) Given a set of legal documents with associated metadata, identify instances of law smells in elements of the set.

Here, *set* is used in the mathematical sense, and the terms *legal document* and *associated metadata* are deliberately open. For example, the legal documents could be different versions of all Titles of the United States Code, and metadata could specify versioning relationships between them. Note that we do not require the documents to be *amendable*, but some dynamic law smells can only occur in amendable documents.

4.1 Assumptions

Which methods are available for detecting law smells depends heavily on how the law (used in the following to denote the legal document set of interest) is stored and accessed. If the law is accessed in printed form, manual detection is the only method that is immediately applicable, and there are generally no guarantees that manual detection is performed consistently or documented comprehensively. If the law is stored in a version-controlled database with rich semantic annotations and a plethora of systematic indices, automatic detection of law smells can be performed consistently and documented comprehensively by the database administrator using mostly adaptations of standard software engineering and database management tools. But while the latter state certainly seems desirable, most use cases for law smell detection lie between the sketched extremes: Lawmakers and lawtakers access the law in a semi-structured format, e.g., HTML or XML, with hardly any access to systematic indices or semantic annotations. Therefore, in the following, we focus on methods
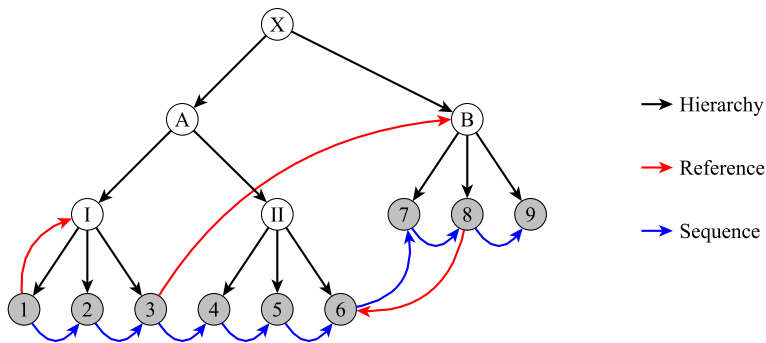
Fig. 3 Structure types typically found in legal documents, illustrated using a toy document X. X consists of nine sequence-level elements (gray) containing text (e.g., sections), which are connected via sequence edges (blue). These are nested inside two higher-level structures (A and B, e.g., parts), the first of which holds two additional mid-level structures (I and II, e.g., chapters), and all elements are connected by a tree of hierarchy edges (black). Finally, the text in some elements references other elements (not necessarily on the same level), and these references are captured by reference edges (red). Note that the sequence-level elements can have further substructures (e.g., subsections), which can also reference or be referenced by other elements

to detect law smells automatically or semi-automatically, given access to the law in a semi-structured format. We also assume that all legal documents provided in this format contain *text*, which can be segmented into *tokens*.

We make additional assumptions regarding the information that can be extracted from the semi-structured format (e.g., via preprocessing). These assumptions strike a balance between generality (to ensure that they hold in a large number of jurisdictions and for a large number of document types) and specificity (to ensure that they provide useful constraints for solving the law smell detection problem). They are encapsulated in the legal document model developed by Coupette et al. (2021a), which builds on ideas introduced by Katz et al. (2020). In brief, this model assumes that legal documents, in addition to their text, contain at least three types of *structure*: *hierarchy* (elements of law are nested), *sequence* (most text is placed in elements that are uniquely labeled and sequentially ordered, and the labels of *sequence-level elements* are used as the primary index for text retrieval), and *reference* (text in one element can contain cross-references or citations to other elements). For example, the United States Code is structured into *Titles* as the top level of a *hierarchy* containing substructures such as *Parts*, *Subparts*, *Chapters*, *Subchapters*, and *Sections*, its *sequence* is indicated by section labels, and *reference* occurs to elements on all levels of the hierarchy. Similar observations apply to many collections of rules, be it from other branches of government (e.g., the Code of Federal Regulations), in other countries (e.g., Germany), or on other regulatory levels (e.g., European Union law). Note that all assumed structure types can be captured by graphs (ideally, directed multigraphs with one edge type per structure type), and that the tree induced by the hierarchy edges need not be regular or balanced, as illustrated in Fig. 3. This facilitates the development of our law smell detection toolkit, which we describe next.

Table 3 Classification of law smells by primary detection method. Dynamic smells are typeset in italics; their detection may involve consulting external data

Law smell	Relevant abstract task(s)
(a) Text-based detection	
Duplicated phrase	Pattern detection (Unknown Patterns)
Natural language obsession	Pattern detection (Known/Unknown Patterns)
Long element	Segmentation, Counting
Long requirements list	Pattern detection (Known Patterns), Counting
Ambiguous syntax	Pattern detection (Known Patterns)
Complicated logic	Pattern detection (Known Patterns)
Term overloading	Pattern detection (Known/Unknown Patterns)
Divergent change	Diffing
Shotgun surgery	Diffing

Law smell	Relevant edge type(s)
(b) Graph-based detection	
Large reference tree	Hierarchy, Reference
Rule envy	Sequence, Reference
Failed integration	Hierarchy, Sequence, Reference
Inconsistent enumeration	Hierarchy
Overbroad reference	Hierarchy, Reference
Lazy element	Reference

4.2 Tools

Since legal documents contain *text* with structure that can be represented as a *graph*, our law smell detection tools naturally fall into two categories: *text-based* and *graph-based*. Text-based tools comprise law-specific adaptations of methods from natural language processing and sequence mining (where the text is understood as a sequence of tokens or characters), while graph-based tools encompass law-specific adaptations of graph algorithms and methods from legal network analysis. Table 3 groups the law smells from our taxonomy (cf. Fig. 2) according to these categories, placing each smell where the emphasis of our detection strategies lies. For each law smell in the *text* category, we list the abstract task(s) associated with its detection, and for each law smell in the *graph* category, we list the detection-relevant edge types (hierarchy, sequence, or reference).

Most of the tools we propose are deliberately designed to be language-agnostic. This applies naturally to the methods based on graphs (although *constructing* these graphs in the first place may require language-specific methods, depending on the structure of the raw data), and it also extends to many text-based methods. Among the text-based approaches, identifying *duplicated phrase*, *long element*, and *natural language obsession* works largely independently of the document language, whereas detecting *complicated logic* and *ambiguous syntax* requires some language-specific adaptation, just like uncovering their *code smells* equivalents. Our current

approaches here require language-specific Regular Expressions (detailed below), and therefore, are currently limited to English-language documents. The resulting expressions are quite simple, however, such that following our rationales and English-language examples, scholars working with other languages should be able to devise equivalent expressions for these languages with little additional effort.

Among the tasks relevant for text-based detection, *segmentation*, *counting*, and *diffing* are relatively simple *conceptually*, but the devil lies in the details. Here, the primary challenges are choosing adequate inputs and making sensible judgment calls when translating raw outputs into law smell detection results. For the *detection of known patterns*, our Swiss army knives are Regular Expressions (Regex), as they allow us to incorporate legal background knowledge easily and their results are completely interpretable. These expressions benefit from domain expertise and careful revisions, but they can be created and managed by individual researchers or small research groups, especially in the contexts for which we deploy them. While they could theoretically be learned in a semi-automated process, appropriately labeled legislative datasets are currently hardly available, and some human input beyond labeling would likely remain required, as fully automated tools still struggle to parse legal documents with very long sentences correctly. For some of the grammar-related smells (e.g., *ambiguous syntax* or *long requirements list*), detection could further rely on parse trees or other text representations containing syntactic information, provided they are deemed sufficiently reliable. Additionally, Named Entity Recognition (NER) tools could be used whenever the known patterns of interest are named entities according to their definition.

For the *detection of unknown patterns*, what methods we can choose from depends on how we define a *pattern*: Is a pattern a sequence of tokens that is repeated in some sense *frequently*— or is it something different (e.g., a characteristic sequence of part-of-speech tags)? If the former, is frequency assessed using some fixed (absolute or relative) threshold ($\rightarrow$ frequent pattern mining), is it determined relative to our expectations under a certain null model ($\rightarrow$ statistical pattern mining), or is it based on whether replacing the pattern with a new symbol aids compression ($\rightarrow$ information-theoretic approaches)? Should we allow a pattern to contain gaps or mismatches ($\rightarrow$ no traditional *n*-grams), to contain or overlap with other patterns ($\rightarrow$ no greedy assignment approaches), or even to have parameters (e.g., P(term, reference) = "{term} has the meaning given such term in {reference}")? Our answers to these questions should be guided by the law smell of interest (e.g., we might require exact matches for *duplicated phrase*) and the characteristics of the documents in which we would like to detect it (e.g., the importance of algorithm scalability increases with the length of the document at hand). As corpora of appropriately annotated documents become available (which they currently unfortunately are not), some (parts of) answers, such as thresholds for the detection of law smells like *duplicated phrase* and *long element*, could also be determined using supervised machine learning techniques.

When performing graph-based law smell detection, the biggest challenge lies in finding an appropriate resolution level at which to represent the edges of interest. This is a common problem in legal network analysis (cf. Coupette et al. 2021a; Katz et al. 2020), and its solutions are highly context-dependent. Once the edge resolution

level has been determined, most graph-based law smell detection methods can be built from relatively simple primitives, such as walks along edges and set operations on node neighborhoods. In specific cases, prepackaged graph algorithms might also be available (e.g., given an intelligently constructed input graph, a graph clustering algorithm could be used to identify instances of *failed integration*). Here, we must take extra care to ensure that all assumptions made by the prepackaged algorithm (including those that are only *implicit*) still hold in the scenario of interest.

4.3 Examples

To show the previous deliberations in action, we now elaborate on our methods to detect the example law smells from Sect. 3.2. For concreteness, we focus on sets of legal documents that fulfill the assumptions specified in Sect. 4.1 and contain *codified statutes* or *codified regulations* (e.g., the United States Code, the Code of Federal Regulations, or consolidated versions of German *root laws* [*Stammgesetze*] and *root regulations* [*Stammverordnungen*]). To demonstrate that our methods work well in practice, we use them to detect law smells in the United States Code in Sect. 5.

4.3.1 Duplicated phrase

Recall from Sect. 3.2.1 that a *duplicated phrase* is a phrase above a specified length (*maximum phrase length*) that has more than a specified number of occurrences in a legal text (*maximum occurrence frequency*), where a *phrase* is a nonempty sequence of terms, each of which is either a token or a placeholder for an argument that is itself a phrase. We classify this smell as static, data-related, and non-local (cf. Figure 2), with text-based detectability based on discovering unknown patterns (cf. Sect. 4.2).

With this definition, the number of duplicated phrases in a legal document depends heavily on how we treat the parameters *maximum phrase length* and *maximum occurrence frequency*. Furthermore, if we choose the parameters to be constant, duplicated phrases become practically *downward closed*, i.e., any minimum-length subsequence of a duplicated phrase will also be a duplicated phrase. For naïve parametrizations, we thus face problems similar to those encountered in frequent pattern mining (see Aggarwal and Han (2014) for an overview): (1) If we choose the thresholds too high, we will miss duplicated phrases that we would like to include, and if we choose them too low, we will catch phrases that we would like to exclude, (2) there will be lots of redundancy in the result set, and (3) reporting all our results will overwhelm the users of our method.

Thus, drawing inspiration from pattern set mining (see, e.g., Vreeken et al. 2011), rather than identifying *all* duplicated phrases, our goal becomes to identify a *set* of duplicated phrases whose refactoring we expect to yield the biggest text quality improvements. To reach this goal, we use the DUPEX algorithm introduced by Coupette et al. (2021b). DUPEX leverages information theory to identify duplicated phrases in legal documents, selecting sequences of tokens as duplicates based on the compression we achieve when replacing them (which results in adaptive length and

occurrence frequency thresholds). As input to Dupex, we feed a preprocessed version of the text we are interested in, with common named entity patterns replaced by correspondingly labeled placeholders. Hence, some of the duplicated phrases we discover are naturally parametrized.

4.3.2 Long element

Recall from Sect. 3.2.2 that a *long element* is an element containing legal text that is long as assessed by some absolute or relative measure. We classify this smell as static, structure-related, and—emphasizing absolute measurement—local (cf. Figure 2), with text-based detectability based on solving segmentation and counting tasks (cf. Sect. 4.2).

When detecting long elements, we need to answer three questions:

1. Which *type of element* are we interested in (e.g., do we want to identify long Titles, Chapters, or Sections in the United States Code)?
2. How do we define the *length* of a single element?
3. Given the length of an element (and potentially that of other elements), how do we decide whether it is *long*?

The answer to the first question depends on the context in which the method is deployed, e.g., the highest level at which a lawmaker using the method can amend legal rules. Therefore, we propose a generic long element detection method that works for all element types, provided that for each element of a given type, we have access to the text nested inside it (i.e., its own text and the text contained in its hierarchical descendants).

To answer the second question, we define the length of an element to be the number of tokens contained in its text. Notable alternatives to this definition of length are the number of non-whitespace characters, the overall number of characters, or (unless we are dealing with leaves in the element hierarchy) the number of children or descendants of an element.

In answering the third question, we propose two complementary views. First, since legal texts are typically accessed at their sequence level (e.g., the section level in the United States Code), for sequence-level element types, an *absolute* length measure based on readability seems appropriate, and we suggest to flag all elements of these types as *long* if they exceed the length of a typical page (i.e., 500 tokens). Second, for elements of all types, we can additionally adopt a *relative* length measure based on the length distribution of all elements of the same type that share a common ancestor of a specified other type in the element hierarchy (cf. Figure 3— e.g., all Sections in the same Title of the United States Code). Here, our approach is to flag an element as *long* if it is longer than a user-chosen number of tokens. This number can be chosen as an absolute threshold based on semantic considerations (e.g., by converting a number of pages to a number of tokens), as a relative threshold based on the complementary cumulative distribution function (CCDF) of the relevant token length distribution, which reveals what percentage of elements (y-axis)

are longer than a certain number of tokens (x-axis), or as a function (e.g., the maximum) of both.

4.3.3 Ambiguous syntax

Recall from Sect. 3.2.3 that *ambiguous syntax* is the use of logical operators, control flow operators, or punctuation in a way that leaves room for interpretation. We classify this smell as static, grammar-related, and local (cf. Figure 2), with text-based detectability based on discovering known patterns (cf. Sect. 4.2).

Since the known patterns we are looking for are characterized by syntactic ambiguity, syntactic parsers can be helpful only in the sense that parse tree ambiguity *suggests* ambiguous syntax. Instead, we therefore resort to Regex. These Regex are designed to capture formulations that are likely to create syntactic ambiguity, using operators, punctuation, and wild cards as building blocks.

Our Regex allow us to flag problematic sequences of tokens— e.g., sequences featuring *and* and *or* in close proximity, or contextualized usages of mathematically redundant syntax like *and/or*—, which can then be judged quickly by human experts.

4.3.4 Large reference tree

Recall from Sect. 3.2.4 that a *large reference tree* is a reference tree whose edge set exceeds a given size x, where a reference tree rooted at an element of law r is a tuple composed of the elements V_r reachable from r by following references (including r), and a minimal set of reference edges E_r such that each element of V_r can be reached from r. We classify this smell as static, structure-related, and non-local (cf. Figure 2), with graph-based detectability based on hierarchy edges and reference edges (cf. Sect. 4.2).

To detect large reference trees, we use the representation of legal documents as directed multigraphs from Coupette et al. (2021a), resolving all hierarchy edges and reference edges at the lowest possible level. From the resulting directed multigraphs, we remove those nodes (and their incident edges) that contain more than a specified number of tokens. This effectively excludes reference trees that are unlikely to be constructed in practice (e.g., when a Section references a Chapter of the United States Code, lawtakers are unlikely to read through the entire Chapter to understand the referencing Section). We replace each reference edge to a non-leaf node with edges to all of its descendants that directly wrap text, and construct trees rooted at all elements of law which contain references. This allows us to assess the minimum number of references a lawtaker must follow in order to fully resolve the content of the provision at its root (i.e., $n - 1$ references if the tree has n nodes), provided they do not encounter any semantic stopping criteria along the way.

A *long reference chain* is any path from the root of a reference tree to a leaf at depth ℓ, where $\ell > x$ for a user-specified maximum acceptable chain length x. x might lie around 3 if chosen in accordance with common (but not uncontested) user interface design heuristics, or around 6 if we trust users of *law* to be twice as patient as regular users. The length of reference chains depends on the reference

resolution strategy (e.g., if it is more like breadth-first search or depth-first search), regarding which we deliberately refrain from assumptions. We can, however, learn about the *best* possible *worst* case by analyzing reference chains in *shortest path trees*.

4.3.5 Natural language obsession

Recall from Sect. 3.2.5 that *natural language obsession* is the representation of typed data as natural language text. We classify this smell as static, data-related, and local (cf. Figure 2), with text-based detectability based on discovering known or unknown patterns (cf. Sect. 4.2).

Observe that, due to how we define a phrase in Sect. 4.3.1, natural language obsession leads to duplicated phrases. However, this only holds, at least in part, because we preprocess the input text for duplicated phrase detection to replace named entities with known patterns. Therefore, the detection of natural language obsession and the detection of duplicated phrases can be viewed as mutually enhancing: We can preprocess the input text for duplicated phrase detection using tools to detect natural language obsession via *known* patterns, and use the results from duplicated phrase detection to detect instances of natural language obsession involving *unknown* patterns. Here, the classification of patterns as known versus unknown can refer to not only to us as the method designers but also to the users of our methods, who might want to integrate their background knowledge into the law smell detection process.

It is worth noting that, although NER tools seem to be made for the task of detecting instances of natural language obsession using known patterns, they usually require preprocessing steps that do not work reliably on legal texts (e.g., sentence splitting or part-of-speech tagging), or their decisions are intransparent (e.g., when they are based on pretrained neural networks). Thus, Regex incorporating domain expertise, along with lists of known legal entities, are again our tool of choice for the discovery of known patterns. By running duplicated phrase detection and natural language obsession detection in alternation, and treating past false negatives and false positives as future tests, we can then reveal unknown patterns, and iteratively improve or complete our Regex and named entity lists.

5 Practice: What law smells exist in the wild?

In this section, we demonstrate the utility of the methods laid out in Sect. 4 to detect the law smells described in Sect. 3. We focus on identifying the example smells from Sect. 3.2 with the methods described in Sect. 4.3, using the United States Code at the end of each year between 1998 and 2019 (inclusive) as an example. Our input data is taken from Coupette et al. (2021a), who transform the XML files provided by the Office of the Law Revision Counsel of the United States House of Representatives for the annual versions of each Title of the United States Code into files following the data model sketched in Sect. 4.1 (for details on the data preprocessing, see the Supplementary Material of Coupette et al. 2021a).

Table 4 Examples of duplicated phrases identified in multiple Titles of the United States Code in 2019 (top) and in the 2019 version of *Title 6—Domestic Security* only (bottom), with options and alternatives written in Regex syntax. For the general phrases, we additionally report the absolute and relative maximum occurrence frequency, abs_{max} and rel_{max}, of each listed phrase across all Titles, along with the Title T in which it occurs. For the specific phrases, we instead report the minimum and maximum occurrence frequency of all options or alternatives, c_{min} and c_{max}

General	abs_{max} (T), rel_{max} (T)
for [the] purposes of this (chapter|subchapter|section|subsection|paragraph)	4 280 (26), 2.90 (26)
[except] as (defined|provided) (by|in) {reference}	706 (42), 0.55 (11)
there (are|is) authorized to be appropriated	1 009 (42), 0.55 (34)
as may be necessary to carry out the	133 (16), 0.10 (2)
the term {term} means	4 438 (42), 2.38 (26)

Specific	c_{min}, c_{max}
[[director] of the] bureau of citizenship and immigration services	18, 66
natural disasters , acts of terrorism , (and|or) other man - made disasters	11, 52
the committee on homeland security (and governmental affairs of the senate|of the house of representatives)	30, 31
cyber threat indicators (and|or) defensive measures	20, 26
information within the scope of the information sharing environment , including homeland security information , terrorism information , and weapons of mass destruction information	26

5.1 Duplicated phrase

As sketched in Sect. 4.3.1, to detect duplicated phrases, we leverage the DUPEX algorithm introduced by Coupette et al. (2021b). We run this algorithm on each Title in each year separately, using the same data preprocessing and parametrization as the authors (i.e., we replace selected named entities identified by Regex with placeholders and set the maximum number of failures until the algorithm stops to 10 000). To facilitate the analysis of our results, we draw on the postprocessing suggested in the DUPEX paper (i.e., we hierarchically cluster the term vectors of long and frequent duplicated phrases by their cosine similarity with Ward linkage and use the resulting phrase ordering for our manual inspection).

Table 4 gives examples of duplicated phrases discovered in the United States Code in 2019, focusing on duplicates occurring across multiple Titles (top) as well as duplicates specific to Title 6 (bottom). The general duplicated phrases we identify often contain language related to scoping and definitions (e.g., "the term {term} means") or boilerplate conferring varying degrees of authority or discretion (e.g., "as may be necessary to carry out the"), whereas Title-specific duplicated phrases frequently contain named entities not replaced during preprocessing (e.g., "[[director] of the] bureau of citizenship and immigration services") or topic-related boilerplate (e.g., "natural disasters, acts of terrorism, (and\|or) other man-made disasters").

While some of the variation we observe is entirely grammatical (e.g., "there (are\|is) authorized to be appropriated") or semantically inconsequential (e.g., "for
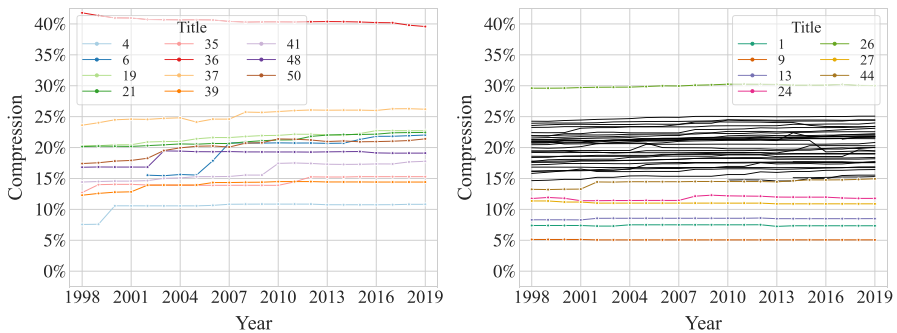
Fig. 4 Evolution of the compression achieved by the DUPEX algorithm for different Titles of the United States Code (1998–2019), in percent of the original encoded bit length. The left panel shows only Titles with a range of compression values of at least 2%. The right panel shows the remaining Titles, colorizing those that constantly have a comparatively high compression (more than 25%) or a comparatively low compression (less than 15%)

[the] purposes of this…"), other variation could be semantically meaningful (e.g., "and|or" variation, as seen in "cyber threat indicators (and|or) defensive measures"), raising the question whether all nuances in meaning are indeed intended. Moreover, many of the legibility and maintainability challenges created by duplicated phrases could be easily mitigated through straightforward refactorings. For example, the last duplicated phrase from Table 4 could be eliminated in three steps:

1. Introduce *ISE* as an abbreviation for *Information Sharing Environment*.
2. Define the term *ISE information* as *information within the scope of the information sharing environment, including homeland security information, terrorism information, and weapons of mass destruction information*.
3. Use the newly introduced term to replace all occurrences of its definition.

DUPEX extracts duplicated phrases by *exploiting* redundancies to compress the input text. Hence, we can use the compression it achieves to *measure* the redundancy in this text. This allows us to track the development of redundancy—i.e., an upper
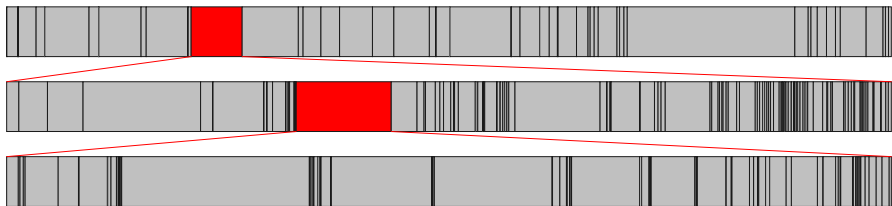
Fig. 5 Zoomable icicle plot enabling visual element length assessment at different hierarchical levels of the 2019 United States Code: Titles (top), Chapters (middle), and Sections (bottom). Here, we zoom in on Chapters from Title 15 (Commerce and Trade) and Sections from 15 U.S.C. Ch. 14A (Aid to Small Business)

Table 5 The ten longest Sections of the United States Code in 2019 (measured in tokens), along with the *verbosity category* of their content

Section	Heading	Length (K)	Category
42 U.S.C. § 1395ww	Payments to hospitals for inpatient hospital services	50.3	Inline math
16 U.S.C. § 1274	Component rivers and adjacent lands	35.4	Listing
42 U.S.C. § 1396a	State plans for medical assistance	34.6	Substructure
42 U.S.C. § 1395w-4	Payment for physicians' services	34.5	Inline math
42 U.S.C. § 1395l	Payment of benefits	29.0	Inline math
15 U.S.C. § 636	Additional powers	28.1	Substructure
42 U.S.C. § 1395x	Definitions	27.7	Listing
42 U.S.C. § 1395m	Special payment rules for particular items and services	25.9	Inline math
42 U.S.C. § 1396b	Payment to States	24.9	Substructure
8 U.S.C. § 1182	Inadmissible aliens	23.6	Listing

bound on the *deduplication potential*— in the Titles of the United States Code over time. As shown in the right panel of Fig. 4, the deduplication potential of most Titles ranges between 15% and 25%, with *Title 9—Arbitration* (low deduplication potential) and *Title 26—Internal Revenue Code* (high deduplication potential) as notable exceptions. However, the left panel of Fig. 4 highlights that the highest deduplication potential lies in *Title 36—Patriotic and National Observances*, although this is one of the few Titles whose deduplication potential has *decreased* over time. In the left panel, we also see that the Title with the by far largest *range* of deduplication potentials is *Title 6—Domestic Security*, whose deduplication potential rises dramatically from 2005 to 2007. This increase follows the addition of Chapters 2 and 3 (in 2006) and of Chapters 4 and 5 (in 2007), which demonstrates how duplicated phrase detection can help pinpoint potentially problematic legislative acts.

5.2 Long element

As discussed in Sect. 4.3.2, many judgment calls need to be made to define what constitutes long elements in legal documents. Eager to avoid making these calls blindly, we propose zoomable icicle plots to interactively explore the element length distribution of hierarchically structured legal documents. Figure 5 depicts a static version of such a graphic, showing the Titles of the United States Code in 2019 at the top, then zooming in on *Title 15—Commerce and Trade* and its Chapters, and finally focusing on *Chapter 14A—Aid to Small Business* and its Sections. As all horizontal space on layer x (the *higher* layer) is reclaimed on layer $x + 1$ by the substructures of the selected element, the element length distribution remains clearly discernible even on lower layers. Thus, zoomable icicle plots allow us to intuitively identify long elements as candidates for refactoring, and to set length thresholds based on the lengths of these elements.

For a quantitative assessment, we focus on *Sections* in the United States Code, whose length we measure in *tokens*, and use a length *ranking*, rather than a length
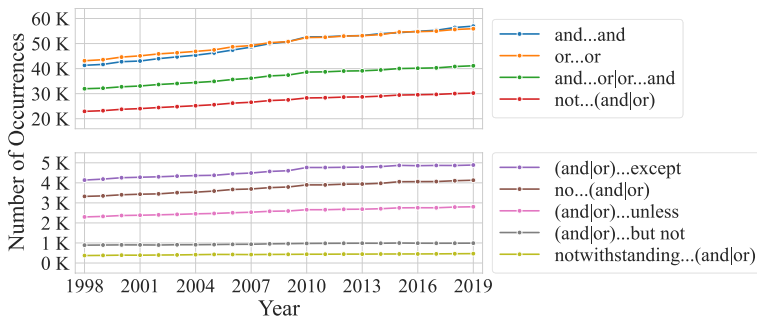
Fig. 6 Number of occurrences of ambiguous syntax candidates identified by problematic patterns in the United States Code (1998–2019)

threshold, to explore our results. Consequently, we list the ten longest Sections, along with their headings and lengths, in Table 5, and observe that the longest Section, 42 U.S.C. §1395ww, occupies roughly one hundred pages in standard print. We further label the long Sections in Table 5 with their *verbosity category* (i.e., the primary reason for their lengthiness, as apparent from manual inspection): *inline math* (i.e., verbal representations of computations and their parametrizations), *listing* (i.e., verbal representations of tables, indices, or other enumerations), or *substructure* (i.e., sections with elaborate substructure containing related rules).

Identifying the verbosity category of a long element can help us find applicable refactorings. For example, *inline math* could be outsourced to authoritative code that both *documents* and *implements* the mandated computation, *listings* could be outsourced to authoritative appendices and reformatted to better bring out their intrinsic structure, and elements with *substructure* could be reshaped into a sequence of shorter elements. Therefore, deriving a full, potentially more fine-grained taxonomy of verbosity categories constitutes an attractive avenue for further research.

5.3 Ambiguous syntax

As described in Sect. 4.3.3, we flag candidate instances of ambiguous syntax using Regex, leaving the final verdict regarding their ambiguity to a human reader (at least for the time being). Here, we focus on three triples of *problematic patterns* that revolve around the operators *and* and *or*. Using "…" as a placeholder to denote *at most fifty characters*, the first triple of patterns concerns multiple instances of *and* or *or* occurring together (*and…and*, *or…or*, *and…or|or…and*), the second triple involves the use of these operators with negation (*no…(and|or)*, *not…(and|or)*, *notwithstanding…(and|or)*), and the third triple combines them with oppositional conjunctions (*(and|or)…but not*, *(and|or)…except*, *(and|or)…unless*). When tracking the absolute and relative prevalence of these patterns in the United States Code over time, as depicted in Figs. 6 and 7, we observe a substantial and growing number of ambiguous syntax candidates, whose frequency remains relatively constant over time— despite personnel changes and technological advances.
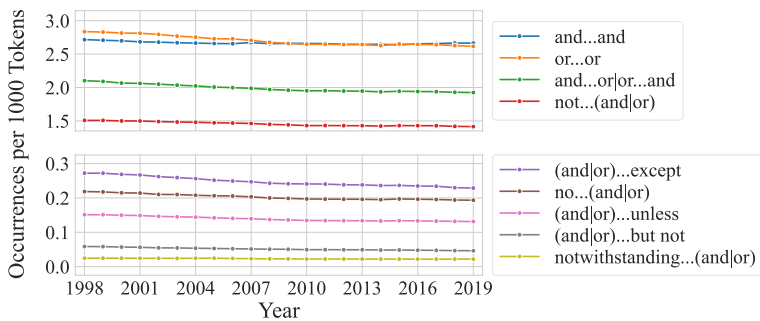
Fig. 7 Number of occurrences per 1 000 tokens of ambiguous syntax candidates identified by problematic patterns in the United States Code (1998–2019)

We suspect that the set of ambiguous syntax candidates we generate using the Regex representing our problematic patterns is over-inclusive: For example, when we find *and* and *or* in close proximity, the operator binding is often clarified by the context. To estimate the degree of over-inclusiveness in our candidate set, we randomly sample 100 candidates flagged by the *and…or* pattern. In 38 sampled cases, the operator binding is unclear from a grammatical point of view, i.e., deciphering the meaning of the affected phrase requires nontrivial background knowledge. This demonstrates that instances of ambiguous syntax *can* be identified in a computer-aided process, although more work needs to be done to exclude false positives (e.g., by refining our Regex or filtering flagged candidates).

To give a concrete example, one of our randomly sampled candidates is the second sentence of *12 U.S.C. §5538(a)(1)*—Mortgage loans; rulemaking procedures; enforcement (emphasis added):

> Such rulemaking shall relate to unfair **or** deceptive acts **or** practices regarding mortgage loans, which may include unfair **or** deceptive acts **or** practices involving loan modification **and** foreclosure rescue services.

The desired operator binding is *probably*:

> Such rulemaking shall relate to (unfair **or** deceptive) (acts **or** practices) regarding mortgage loans, which may include (unfair **or** deceptive) (acts **or** practices) involving (loan modification **and** foreclosure rescue services).

Here, we performed the simplest possible refactoring for instances of ambiguous syntax: adding brackets. Without brackets, alternative readings of our sample sentence are possible— and the sentence is also much harder to read.

5.4 Large reference tree

As delineated in Sect. 4.3.4, we detect large reference trees in legal documents on the basis of their directed multigraph representation as introduced by Coupette et al. (2021a), excluding overly broad references by removing all nodes that contain more than a specified number of tokens. For our exploratory purposes, we
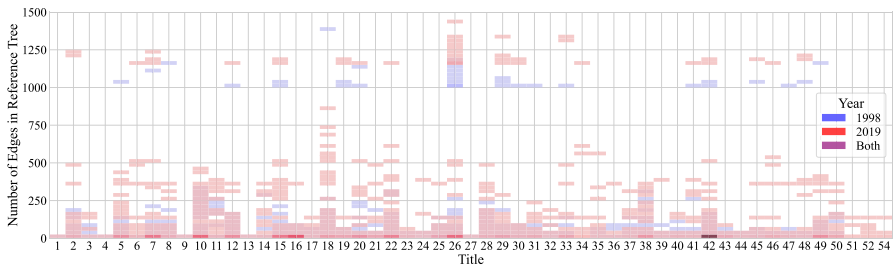
Fig. 8 Distribution of reference tree sizes across Sections of the United States Code in 1998 and in 2019. The two-dimensional histograms depict the density of Sections across Titles (x-axis) and reference tree sizes (y-axis)
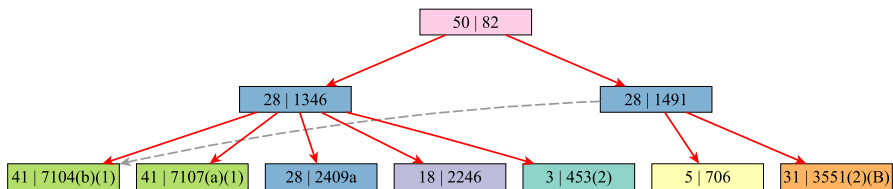

Fig. 9 The first two layers of the reference tree of 50 U.S.C. § 82 in 2019. Nodes are labeled with "{Title} | {Section[Subsection]}", and node colors correspond to Titles

set this number to 1 000 tokens, i.e., roughly the amount of content that can be consumed on a large monitor without scrolling, but other choices may be reasonable depending on the context. We compute reference trees for all Sections of the United States Code from 1998 to 2019. In Fig. 8, we depict the distribution of reference tree sizes (i.e., their number of edges) across the different Titles in 1998 and 2019 as two-dimensional density histograms, where we include edges leading to nodes that have already been visited (i.e., edges closing cycles in the underlying undirected graph), following the intuition that lawtakers might not keep track of this information. These histograms not only show that Sections with relatively large reference trees exist in practice; when overlaid as in Fig. 8, they also reveal a shift towards larger reference trees in most Titles from 1998 to 2019, especially in the tail of the size distribution. This extends prior findings that the law is growing increasingly interconnected (Katz et al. 2020; Coupette et al. 2021a): Even if we exclude overly broad references, thus modeling that they are unlikely to be resolved completely, understanding a Section of the United States Code in its context appears to become more laborious over time.

The reference tree of a Section r in the United States Code can be measured in many dimensions: Apart from its *size* (i.e., its number of edges $|E_r|$), we can assess, e.g., its path lengths—including, most notably, its *depth* (i.e., the maximum length of a path from r to any other node in its reference tree)—, its *weight* (i.e., its total volume of text), and its *diversity* (e.g., number of distinct Sections or Titles from which it contains nodes). As *large* reference trees are hard to depict statically, and *deep* or *heavy* reference trees are often also large, for concreteness, we show a particularly *diverse* reference tree in Fig. 9. Here, we resolve the first two layers of (United States
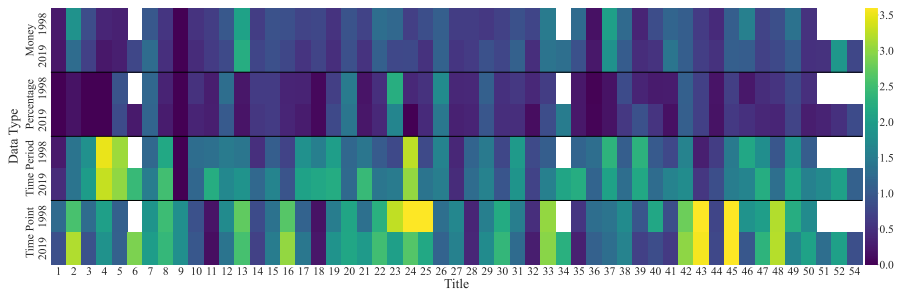
Fig. 10 Relative prevalence (instances per 1 000 tokens) of natural language obsession concerning data of the types *money*, *percentage*, *time period*, and *time point*, for each Title in the United States Code in 1998 and 2019. The maximum of the colormap is computed using the ninety-ninth percentile (rather than the most extreme value); white fields signal that the corresponding Title has no tokens in the given year (Titles 6 and 34 had been repealed and not repurposed in 1998, and Titles 51, 52, and 53 did not even exist)

Code-internal) references from *50 U.S.C. § 82—Procurement of ships and material during war*, which takes us through seven different Titles in total (Title 50 included).

In the reference tree of 50 U.S.C. § 82, all references from the root to the first layer are found in *Subsection (d)—Compensation for commandeered material*. This Subsection—a single sentence of more than 130 tokens—not only exhibits a mixture of particularly unsavory law smells, but it also addresses parties seeking compensation from the United States (i.e., the legislator), sending them to two Sections in Title 28 for details on what they can demand. To understand these details, potential claimants need to visit yet another five Titles on the second layer of the reference tree (in the worst case). Our statistics show that this example is by no means exceptional, which raises questions about the organization of the United States Code into Titles and the resulting user experience. As refactoring reference trees is nearly impossible without dedicated computational tools (due to the many interdependencies we need to account for), the conceptualization and implementation of such tools is an important field for future work.

5.5 Natural language obsession

As explained in Sect. 4.3.5, natural language obsession is closely related to duplicated phrases. First, we may use Regex as part of a preprocessing step to extract and replace instances of natural language obsession following *known* patterns, which enables us to discover *parametrized* duplicated phrases (as demonstrated in Sect. 5.1). Second, we can leverage the results of duplicated phrase detection to identify instances of natural language obsession following *unknown* patterns.

Starting with the Regex-based approach, we extract text corresponding to data of the types *money* (e.g., *$1,000*), *percentage* (e.g., *50 percent*), *time period* (e.g., *30 days*), and *time point* (e.g., *January 1*) for each Title of the United States Code from 1998 to 2019. Above all, our statistics confirm that *data as text* is ubiquitous in the United States Code: In 2019, for example, we identify roughly 19 K amounts of money, 17 K percentages, 35 K time periods, and 44 K time
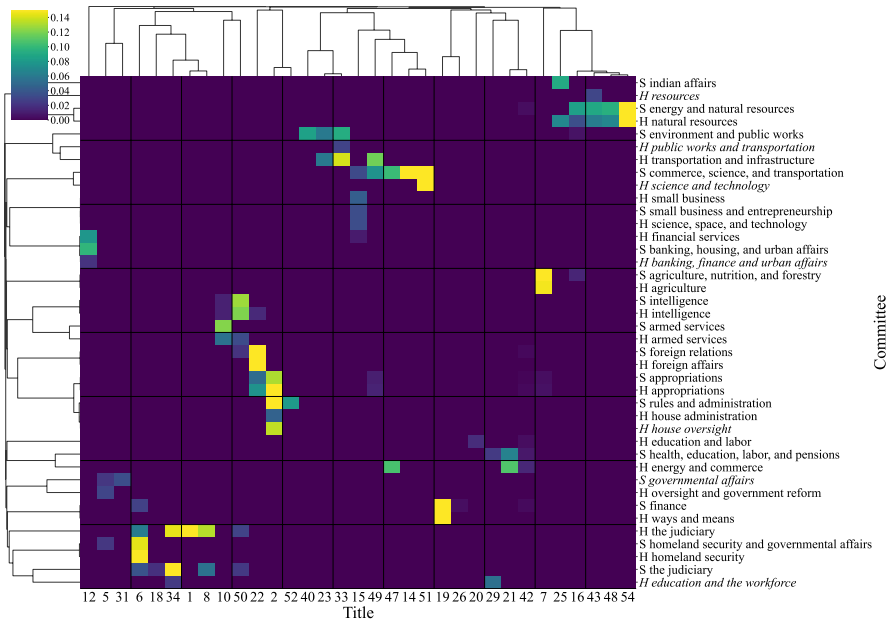
Fig. 11 Committee profiles for the Titles of the United States Code in 2019. We depict the number of times per 1 000 tokens that a Committee is mentioned with a name of the shape "Committee on {topic} of the {parent body}" in the duplicated phrases we extract for a Title, where the maximum of the color-map is again computed using the ninety-ninth percentile. The data are hierarchically clustered using correlation as a distance metric and average linkage. We omit Committees not contained with a name of the above-stated shape in a duplicated phrase for any Title, and Titles in which we do not identify any Committee in a duplicated phrase. Committee names are abbreviated by the first letter of their parent body (*Senate* or *House of Representatives*) and their topic, and Committees that do not exist anymore are typeset in *italics*

points. However, these statistics come with three caveats: First, our Regex are deliberately both simple and conservative, i.e., all numbers should be treated as lower bounds. Second, the categories *time period* and *time point* group several time-related Regex, and the distinction between the classes is not always clearcut. Third, when comparing annual counts across years, with or without normalization by annual Title size, we can only *observe* changes but not *explain* them.

Figure 10 depicts how many instances of the data types described above we detect per 1 000 tokens in each Title of the United States Code in 1998 and 2019. With due caution, we make the following observations. First, some Titles have distinct *data profiles*, e.g., *Title 24—Hospitals and Asylums* is unique in its comparatively high reliance on time periods *and* time points. Similarly, individual data types are particularly prevalent in a small set of Titles, e.g., in 2019, *Title 13—Census*, *Title 37—Pay and Allowances of the Uniformed Services*, and *Title 52—Voting and Elections* have the highest frequency of money mentions. Finally, the relative prevalence of certain data types increases between 1998 and 2019 for some Titles, while it decreases for others. For example, *Title 2—The Congress*

states relatively more on time points in 2019 than in 1998, whereas *Title 25—Indians* does the opposite.

Further illustrating the interplay between natural language obsession and duplicated phrases, we now turn to the instances of the former revealed by detecting the latter. We focus on two related types of named entities: Committees of the United States Senate and Committees of the United States House of Representatives. Figure 11 shows *Committee profiles*, i.e., frequency estimates of Committee mentions per 1 000 tokens, for those Titles in the 2019 United States Code in which we find at least one Committee mention of the shape "Committee on {topic} of the {parent body}" as part of a duplicated phrase. The statistics underlying Fig. 11 are again lower bounds on the actual frequencies, as we only count Committee occurrences when they are specified in the format stated above and part of at least one duplicated phrase. Hierarchically clustering the data using correlation distance and average linkage, we observe that in most cases, Senate and House Committees with similar topics are grouped together. The clustering also brings out similarities between Titles. For example, *Title 16—Conservation*, *Title 43—Public Lands*, *Title 48—Territories and Insular Possessions*, and *Title 54—National Park Service and Related Programs* form a group (on the right) that has very low "Committee correlation distance", chiefly due to their frequent mentions of the *Committee on Energy and Natural Resources of the Senate* and the *Committee on Natural Resources of the House of Representatives*.

We find that almost all Standing Committees of the 116th Congress (active in 2019) are part of at least one duplicated phrase, the most notable exceptions being the *Budget Committees* of the Senate and the House of Representatives (which are mentioned only in *Title 2—The Congress*, often together, and mostly in abbreviated forms). Furthermore, not all of the "Committees" listed in Fig. 11 existed in the 116th Congress. For example, the *Committee on Resources of the House of Representatives*, which we find in the duplicated phrases of Title 43, has been the *Committee on Natural Resources* since 2007 (returning to the name it had been carrying from 1993 to 1995). *Committee fossils* like this are widespread, especially among the House Committees, and we typeset them in *Italics* in Fig. 11. They allow us to conclude with a textbook example highlighting the perils of natural language obsession: In the 2019 version of Title 5, both *Committee on Governmental Affairs of the Senate* and *Committee on Homeland Security and Governmental Affairs of the Senate* are among our duplicated phrases, each with ten mentions.[8] All of these mentions reference the same entity, i.e., what was known under the name *Committee on Government Operations* from 1952 to 1977, reorganized as the *Committee on Governmental Affairs* in 1977, and renamed as the *Committee on Homeland Security and Governmental Affairs* in 2005. Upon closer inspection, we find additional mentions of the *Committee on Governmental Affairs of the Senate*, in several name variants, also in other Titles of the 2019 United States Code (most notably in Title 31, cf. Fig. 11). As these are references to an entity that was renamed roughly fifteen

---

[8] Note that these duplicated phrase counts do not necessarily correspond to the raw counts of the relevant phrase in the document of interest, as the DUPEX algorithm computes a *cover* of the input token sequence, i.e., enforces that patterns are *non-overlapping*.

years ago, this state is dissatisfying. While one might argue that the problem could be (and commonly *is*) addressed in practice by the *Committee Name History* provided by Congress[9] and proper interpretation, this solution is neither efficient nor user-friendly. Here, computational approaches provide a viable alternative: Once a Committee is represented as an *entity*, rather than a mere string, changing its name in all Titles of the United States Code becomes as simple as "refactor...rename".

6 Discussion

In Sect. 5, we have shown that the law smells described in Sect. 3 can be detected by the methods developed in Sect. 4—at least in the United States Code from 1998 to 2019. While we have focused on our five example law smells, i.e., *duplicated phrases*, *long elements*, *large reference trees*, *ambiguous syntax*, and *natural language obsession*, it would be interesting to explore the prevalence of other law smells from our taxonomy in this particular corpus. Moreover, our concepts and methods could immediately be applied to other corpora of statutes or regulations across countries, time periods, and regulatory levels, such as the United States Code of Federal Regulations or the collection of European Union Regulations and Directives. Leveraging law smells to monitor other collections of legal documents, e.g., judicial decisions or contracts, is less straightforward but also deserves further study.

Beyond its practical extensions, our work also leaves room for theoretical and methodological improvements. While our law smell taxonomy attempts to be comprehensive, it is most likely neither exhaustive nor ideally structured, and other smells and distinctions might prove useful in the future. Likewise, our methodological toolkit is functional but by no means perfect. For example, our Regex could easily integrate more background knowledge and cover more edge to reduce the number of false law smell negatives. Furthermore, we currently provide no interactive interfaces to investigate law smell detection results.

Finally, our work calls for extensions in three directions. First, we focus on law smell *detection*, but our ultimate goal is law smell *deodorization*. To this end, we need both a better understanding of the landscape of *legal refactorings* and better tools to perform those refactorings identified as applicable— in the best case, *legal integrated development environments* combined with *legal version control*. Second, in many jurisdictions, there is currently no way for the legislator to change the language or structure of the law creating interpretive uncertainty concerning its content— i.e., refactoring in the classical software engineering sense is hardly possible and potentially undesirable, and researchers could help legislators change this situation. Third, we have deliberately skirted the questions *how* and *to which extent* law can be made computable, which we consider an important research direction that could also build on some of our results. Quests to answer this question might eventually lead us to adapt principles and ideas from object-oriented programming to the legal domain, which would yield many new law smells and corresponding refactorings.

---

[9] https://www.congress.gov/help/committee-name-history.

7 Conclusion

We have initiated the systematic study of *law smells*, i.e., patterns in legal texts that might impair the comprehensibility and maintainability of the law and indicate the need for *legal refactoring*. Guided by five illustrative examples, we have developed a comprehensive law smell taxonomy, introduced a methodological toolkit for law smell detection, and confirmed the utility of this toolkit in practice by applying it to 22 years of codified statutory legislation in the United States Code (1998–2019). Our work demonstrates how ideas from software engineering can be ported to the legal domain, and it creates numerous opportunities for further research on *defining*, *detecting*, and *deodorizing* law smells in *all types of legal texts*. Thus, we hope to have contributed a building block for the road towards truly computational legal drafting.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Aggarwal CC, Han J (2014) Frequent pattern mining. Springer, Berlin. https://doi.org/10.1007/978-3-319-07821-2

Allen LE (1957) Symbolic logic: a razor-edged rool for drafting and interpreting legal documents. Yale Law J 66(6):833–879. https://doi.org/10.2307/794073

Allen LE, Engholm CR (1980) The need for clear structure in plain language legal drafting. Univ Mich J Law Ref 13:455–513

Allen LE, Saxon CS (1995) Better language, better thought, better communication: the a-hohfeld language for legal analysis. In: Proceedings of the international conference on artificial intelligence and law (ICAIL), pp 219–228

Alschner W, D'Alimonte D, Giuga GC, Gadbois S (2020) Plain language assessment of statutes. In: Proceedings of the conference on legal knowledge and information systems (JURIX), pp 207–210. https://doi.org/10.3233/FAIA200865

Becker P, Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Bing J (1988) Computerized legal information services: an introduction. Nordic J Int Law 57(4):393–404. https://doi.org/10.1163/157181088x00416

Bing J (2010) Let there be LITE: a brief history of legal information retrieval. European J Law Technol 1(1)

Butt P (2013) Modern legal drafting: a guide to using clearer language. Cambridge University Press, Cambridge

Coupette C, Beckedorf J, Hartung D, Bommarito M, Katz DM (2021) Measuring law over time: a network analytical framework with an application to statutes and regulations in the United States and Germany. Front Phys 9:269:1-269:31. https://doi.org/10.3389/fphy.2021.658463

Coupette C, Singh J, Spamann H (2021b) Simplify your law: using information theory to deduplicate legal documents. In: Proceedings of the IEEE International Conference on Data Mining Workshops (ICDMW 2021), pp 631–638

Curtotti M, McCreath E, Bruce T, Frug S, Weibel W, Ceynowa N (2015) Machine learning for readability of legislative sentences. In: Proceedings of the international conference on artificial intelligence and law (ICAIL), pp 53–62. https://doi.org/10.1145/2746090.2746095

Dorsey TA (2014) The impact of information technology on drafting offices. Int J Legisl Draft Law Reform 2(1):86–102

Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Fungwacharakorn W, Tsushima K, Satoh K (2021) Resolving counterintuitive consequences in law using legal debugging. Artif Intell Law 29(4):541–557. https://doi.org/10.1007/s10506-021-09283-7

Hafner CD, Lauritsen M (2007) Extending the power of automated legal drafting technology. In: Proceedings of the conference on legal knowledge and information systems (JURIX), pp 59–68. https://doi.org/10.5555/1565610.1565621

Hart HLA (1961) The concept of law, oxford University Press. oxford university press, Oxford

Karpen U (2008) Instructions for law drafting. European J Law Reform 10:163–181

Katz DM, Coupette C, Beckedorf J, Hartung D (2020) Complex societies and the growth of the law. Sci Rep 10(1):1–14. https://doi.org/10.1038/s41598-020-73623-x

Li T, Balke T, De Vos M, Padget J, Satoh K (2013) A model-based approach to the automatic revision of secondary legislation. In: Proceedings of the international conference on artificial intelligence and law (ICAIL), pp 202–206. https://doi.org/10.1145/2514601.2514627

Li WPL, Azar P, Larochelle D, Hill P, Lo AW (2015) Law is code: a software engineering approach to analyzing the United States code. J Bus Technol Law 10(2):297–374

Liebwald D (2015) On transparent law, good legislation and accessibility to legal information: towards an integrated legal information system. Artifi Intell Law 23(3):301–314. https://doi.org/10.1007/s10506-015-9172-z

Lima A, Rossi L, Musolesi M (2014) Coding together at scale: Github as a collaborative social network. In: Proceedings of the international conference on weblogs and social media (ICWSM), pp 295–304

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: a map of code duplicates on GitHub. Proceedings of the ACM on programming languages 1(OOPSLA):1–28. https://doi.org/10.1145/3133908

Luhmann N (1987) The unity of the legal system. In: Autopoietic Law – A New Approach to Law and Society, pp 12–35. https://doi.org/10.1515/9783110876451.12

Marcus JS (2012) Structured legislation: toward the synthesis of better law and regulation of electronic communications. Legisprudence 6(1):1–33. https://doi.org/10.5235/175214612800902534

Markovich R, Hamp G (2015) Elliptical lists in legislative texts. In: Proceedings of the international conference on artificial intelligence and law (ICAIL), pp 192–195. https://doi.org/10.1145/2746090.2746112

Moens MF (2006) Improving access to legal information: how drafting systems help. In: Information technology and lawyers, pp 119–136, https://doi.org/10.1007/1-4020-4146-2_5

Mousmouti M (2012) Operationalising quality of legislation through the effectiveness test. Legisprudence 6(2):191–205. https://doi.org/10.5235/175214612803596686

Noll P (1973) Gesetzgebungslehre. Rowohlt, Hamburg

Opdyke WF, Johnson RE (1990) Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In: Proceedings of the symposium on object-oriented programming emphasizing practical applications (SOOPPA)

Ray B, Posnett D, Devanbu P, Filkov V (2017) A large-scale study of programming languages and code quality in GitHub. Commun ACM 60(10):91–100

Raz J (1970/1980) The concept of a legal system

Robbins IP (2017) And/or and the proper use of legal language. Maryland Law Rev 77(2):311–337

Sartor G (2008) Open management of legislative documents. In: Stefanou C, Xanthaki H (eds) Drafting legislation. Ashgate publishing, Farnham, pp 259–285

Saxon CS (1982) Computer-aided drafting of legal documents. Am Bar Found Res J 7(3):685–754

Sharma T, Spinellis D (2018) A survey on software smells. J Sys Softw 138:158–173. https://doi.org/10.1016/j.jss.2017.12.034

Speicher D (2020) Did JHotDraw respect the law of good style? - a deep dive into the nature of false positives of bad code smells. Art Sci Eng Program 4:14:1–14:81, https://doi.org/10.22152/programming-journal.org/2020/4/14

Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. Inf Syst J 12(1):43–60. https://doi.org/10.1046/j.1365-2575.2002.00117.x

Sugisaki K (2016) Towards data-driven style checking: an example for law texts. In: Proceedings of the conference on legal knowledge and information systems (JURIX), pp 93–100. https://doi.org/10.3233/978-1-61499-726-9-93

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: Proceedings of the international conference on software engineering (ICSE), pp 403–414. https://doi.org/10.1109/icse.2015.59

van Lochem P, Westerman P (2010) Rules on rulemaking introduction. Legisprudence 4(2):107–109. https://doi.org/10.1080/17521467.2010.11424704

Vreeken J, Van Leeuwen M, Siebes A (2011) Krimp: mining itemsets that compress. Data Min Knowl Discov 23(1):169–214. https://doi.org/10.1007/s10618-010-0202-x

Wintgens LJ (1999) To follow a rule as a legislation - some observations from a legisprudential perspective. Rechtstheorie 30(1):11–46

Xanthaki H (2010) Drafting manuals and quality in legislation: positive contribution towards certainty in the law or impediment to the necessity for dynamism of rules? Legisprudence 4(2):111–128. https://doi.org/10.1080/17521467.2010.11424705

Xanthaki H (2011) Quality of legislation: an achievable universal concept or a utopian pursuit? In: Quality of legislation - principles and instruments: proceedings of the congress of the international association of legislation (IAL), pp 75–85

Xanthaki H (2014) Drafting legislation: art and technology of rules for regulation. Bloomsbury Publishing, London

Yamashita A (2013) How good are code smells for evaluating software maintainability? Results from a comparative case study. In: Proceedings of the international conference on software maintenance (ICSM), pp 566–571. https://doi.org/10.1109/icsm.2013.97

Yu Y, Yin G, Wang H, Wang T (2014) Exploring the patterns of social behavior in GitHub. In: Proceedings of the international workshop on crowd-based software development methods and technologies (CrowdSoft), pp 31–36. https://doi.org/10.1145/2666539.2666571

Ziegler P (1989) The status of normalized drafting: the need for theory building and empirical verification. Osgoode Hall Law J 27(2):337–358