Towards Automated Identification of Data Constraints in Software Documentation

Anonymous Author(s)

ABSTRACT

Data constraints are important business rules that specify the values allowed or required for the data used in a software system. These constraints are typically described in textual software artifacts (*e.g.,* requirements and design documents, or user manuals). Previous research on data constraints in software focused on studying their implementation in the code for identifying inconsistencies or to support their traceability.

In this paper, we add to this body of knowledge by studying 548 data constraints described in the documentation of nine systems. We uncovered and documented 15 linguistic discourse patterns that stakeholders use to describe data constraints in natural language. We conducted a second comprehensive study, where we explore the use of the discourse patterns we discovered, together with linguistic elements, the operands of the data constraints and their types, as features for automatically classifying sentence fragments as data constraint descriptions. The best combination of features and learner achieves 70.87% precision and 59.73% recall (64.76% F1).

Our findings are an important step towards the automated identification and extraction of data constraints from natural language text, which in turn is essential for automating their traceability to code and for test generation.

KEYWORDS

business rule, data constraint, empirical study, text classification

1 INTRODUCTION

Software systems perform operations on data that represent aspects of the application domain or parts of computing components. Such data are modeled during domain and requirements analysis or added during the design phase. Functional requirements and domain business rules define constraints on such data, often described in natural language. *Data constraints* specify what values are allowed or required for the given data [23, 74, 75]. For example, the parts of the sentences highlighted in Figure 1 show four data constraints described in natural language.

Keeping track of data constraints during software evolution is essential for generating test cases, checking how they are enforced in the source code, or making code changes when the constraints change. Given the importance of data constraints, recent research [23] studied how they are implemented in code and how to trace them to their implementations [24, 78]. These works focused on manually extracted constraints [24] or on constraints extracted from databases [78].

In this paper, we fill in the knowledge gap in the field and investigate *how to automatically identify data constraints* in natural language documents. Even in relatively small systems, the number of data constraints can be in the hundreds or thousands, scattered through many types of documents such as requirement and manuals, so having the ability to automatically identify data constraints is important, for example, for automating test generation and their traceability, among others tasks.

While the problem of automatically identifying data constraints in text has not been studied before, somewhat similar text classification problems have been solved via machine learning and natural language processing techniques, using text properties (*e.g.,* n-grams or part of speech) as training features for machine-learning classifiers [1, 4, 5, 9, 14, 35, 37, 60, 66, 79]. At the same time, previous research on classifying elements of bug reports [9] revealed that more complex linguistic features (*e.g.,* discourse patterns) are more effective for such a task.

With that in mind, we conducted an empirical study on 548 data constraints extracted from the documentation of nine systems and found that they are described using a set of 15 discourse patterns, which we documented in a catalog (Section 3). For example, the four data constraints shown in Figure 1 are described using four different discourse patterns (*i.e.,* `NP_COMP_NP`, `NP_IN_VALUESET`, `NP_IF_BINARY_VALUE`, `SET_NP_TO_VALUE` - described in Section 3). These discourse patterns have not been studied or documented in any previous research.

Previous work [24] also found that the data constraint types and operands are essential in tracing the constraints to their implementations in the code. For example, the data constraints from the examples 1 and 3 in Figure 1 all have two operands each, while the constraint from example 2 has seven operands. The four data constraints are of different types each: `value-comparison`, `binary-value`, `categorical-value`, and `concrete-value` (described in Section 2). We posit that these elements of the data constraints are important for determining if a sentence fragment describes a data constraint.

We studied the use of discourse patterns, operands, and data constraint types as novel features for the classification of text fragments into data constraints, in addition to the textual features of the constraint descriptions (Section 4). We classified 548 explicit data constraint descriptions, extracted from the documentation of nine open source systems, using five different machine learning techniques (*i.e.,* logistic regression, linear SVM, nonlinear SVM, Naive Bayes, and decision trees), and all combinations of the above-mentioned features. We found that using the three novel features together with the textual features leads to substantial improvement in classification, compared to only using textual features. More so, the best combination of features and learner achieves 70.87% precision and 59.73% recall (64.76% F1). Our in-depth analysis of the false negative results helped us formulate guidelines for collecting more balanced training data that would likely lead to even better classification performance.

The main contributions of the paper are:

- A catalog of 15 discourse patterns commonly used to describe data constraints in natural language.

ICSE 2024, April 2024, Lisbon, Portugal
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/#.##/####.####

Example1 (apache-ant-1.10.6)
The target file name is identical to the source file name.
Constraint type: `value-comparison`
Operands: target file name, source file name
Discourse pattern: `NP_COMP_NP`

Example2 (from log4j-2.13.3)
The set of built-in levels includes TRACE, DEBUG, INFO, WARN, ERROR, FATAL.
Constraint type: `categorical-value`
Operands: Built-in levels,TRACE, DEBUG, INFO, WARN, ERROR, and FATAL
Discourse pattern: `NP_IN_VALUESET`

Example3 (from swarm_v2) - two constraints in one sentence
Log frequency, if checked, will set the frequency axis to log mode.
Constraint1 type: `binary-value`
Operands1: log frequency, checked
Discourse pattern1: `NP_IF_BINARY_VALUE`

Constraint2 type: `concrete-value`
Operands2: the frequency axis, log mode
Discourse pattern2: `SET_NP_TO_VALUE`

Figure 1: Examples of data constraints (highlighted).

Table 1: Systems, sentences, and constraints.

	Labeled			Rep-pack		Paper	
System	**+sent (const)**	**- sent**	**ign.**	**exp.**	**nexp.**	**+sent (const)**	**-sent (frag)**
Joda-Time	51 (71)	742	672	52	16	36 (52)	742 (2,261)
Apache Ant	272 (336)	4,236	723	159	156	137 (159)	4,236 (12,435)
HTTPComp.	65 (79)	895	49	46	30	40 (46)	895 (3,143)
jEdit	162 (193)	1,539	575	67	110	65 (67)	1,539 (4,684)
ArgoUML	173 (277)	3,376	278	48	225	47 (48)	3,376 (10,640)
Swarm	68 (92)	206	·	58	31	45 (58)	206 (650)
Checkstyle	51 (112)	445	·	22	27	18 (22)	445 (1,122)
JabRef	120 (151)	375	·	39	41	36 (39)	375 (927)
Log4j	117 (320)	422	·	57	259	51 (57)	422 (977)
Jpos	28 (33)	311	·	18	5	discarded < 20 explicit constraints	
Skywalking	24 (84)	·	·	3	·		
Rhino	251 (359)	·	·	321	38		
Guava	26 (27)	1,612	227	13	4		
ShardingSphere	19 (60)	·	·	·	·		
MyBatis	69 (144)	500	·	18	69		
Total	**1,502 (2,338)**	**14,659**	**2,884**	**921**	**1,011**	**475 (548)**	**12,236 (36,839)**

- A new data set of 1,932 labeled data constraints, extracted from the documentation of 15 software systems, which complements data from existing research [24], essential for future research.
- An empirical study that identifies the best features and learners to use for the automatic classification of data constraints described in natural language.
- A replication package including all the data and the complete results of the empirical studies [2].

For simplicity, in the remainder of the paper, we will use the word *constraint* to refer to *data constraints*.

2 BACKGROUND

In this section, we introduce important terminology and definitions regarding data constraints, with a set of examples. Figure 1 shows four highlighted examples of data constraints. These were identified in three sentences selected from the documentation of software systems in our data set. The examples are annotated with the *type* and *operands* of each constraint (defined below), as well as their *discourse pattern* (defined in Section 3).

The *operands* of a data constraint are the domain data on which the constraint is defined. Data constraint descriptions contain at least one *data element* (often an attribute or property of some domain data), one or more *values* that the data element can or cannot have, and the *relationship* between the data element and the values. *Value* can be explicit or implicit (*i.e.,* expressed as the value of another data element). The *relationships* are synonymous to equality or relational operators, such as *greater than*, *less than*, *equal to* or *not equal*. The *data elements* and the *values* in a constraint constitute the *operands* of the constraint, while the *relationship* between the operands is referred to as the *operator*. For example, in the constraint `target file name is identical to the source file name` from Figure 1 (example 1), the *data element* is `target file name`, its *value* is given by the data element `source file name`, and their relationship is `is identical` (*i.e.,* the *equal to* operator). Likewise, for the constraint `set the frequency axis to log mode` in Figure 1 (example 3), the *data element* is `the frequency axis`, its *value* is `log mode`, and their relationship is `set to` (*i.e.,* the *equal to* operator).

Florez *et al.* [23] classified data constraints into four *types* (`binary-value`, `concrete-value`, `value-comparison`, and `categorical-value`). These constraint types are relevant for our work and we reuse the definitions introduced by Florez *et al.* For `value-comparison` constraints, the value of a data element X is constrained by the value of another data element Y (or constant C). The relationship between the data elements translates to equality or relational operators used to determine if X is greater than, less than, equal to or not equal to Y (or C). Constraints of this type always have two operands: the two data elements, X and Y (or C).

For `binary-value` constraints, a data element X has one of only two possible, mutually-exclusive values BV (*a.k.a.* binary values–*e.g.,* true/false, on/off, null/not-null). These are equivalent to `value-comparison` constraints if the operator is equality and there are only two possible values. Constraints of this type always have two operands: the data element X and the binary value BV. Note that this type of constraints was named `dual-value-comparison` by Florez *et al.* [23], but we consider that the `binary-value` name better reflects the type, and we adopt this name in this work.

For `categorical-value` constraints, the value of a categorical data element X is constrained to a finite set of values VS. Constraints of this type always have more than least two operands: the data element X and the set of values VS.

Constraints of `concrete-value` type explicitly dictate a concrete value V the data element X should have. These constraints always have two operands: the data element X and the concrete value V.

3 HOW ARE DATA CONSTRAINTS DESCRIBED IN NATURAL LANGUAGE?

We performed a study to understand how data constraints are described in natural language. Specifically, we aimed to identify discourse patterns (if any) that are used in describing the constraints to answer the following research question.

RQ: How are data constraints described in natural language?

3.1 Data Collection

Table1 summarizes the data we collected (and discarded along the way), including what we include in the replication package.

We started with a set of 299 constraints from 15 systems, which were collected in previous research [24].

We then labeled additional sentences from the documentation systems. Each sentence was labeled with one of the following labels: *ignore*, *negative*, *positive*. Note that the sentences of the 299 constraints from previous work did not have any of these labels, as they were used for different purpose (*i.e.,* tracing to code).

Ignore sentences are those that have content-lists (*i.e.,* bullet lists or enumerated lists), log/outputs, regular expressions, source code, tables, and XML-related information. We labeled 2,884 sentences as *ignore* and they were not used further in our study (Table 1 column *Labeled/ign.*).

Each sentence that contained at least one constraint was labeled as *positive*. The constraint description was highlighted as shown in the examples from Figure 1, using the MITRE Annotation Toolkit (*a.k.a.* MAT) version 3.3 [48]. For each constraint, the starting character and ending character of its description in the sentence was recorded. We labeled 1,502 sentences as *positive*, with 2,338 constraints (Table 1 column *Labeled/+ sent*).

We labeled 14,659 sentences as *negative*, which did not contain any data constraints (Table 1 column *Labeled/- sent*).

Upon inspection, we found that 188 constraints were in sentences that should have been ignored: 70 in content-list, 17 in log/console output, 36 in regex, 5 in source code, 59 in table, 1 in xml-related. We also excluded these sentences and constraints from the study.

The focus of our work is on *explicit* data constraint descriptions in natural language. Hence, we removed 359 constraint descriptions from the rhino-1.6R5 system as they were described in pseudo code, which we do not consider it as natural language (Table 1 column *Labeled/- sent* row *Rhino*). An *explicit* constraint description is a contiguous fragment of a sentence that includes all relevant data elements, values, and their relationships.

Conversely, constraint descriptions that are *not explicit* are those that span multiple sentences or refer to the data elements or values via indefinite pronouns (*e.g.,* it, them, *etc.*). Automatically classifying such description would require complex and notoriously imprecise NLP techniques such as automated reference resolution. Consequently, we removed the constraints that are not explicit: 250 constraints that span multiple sentences and 475 constraints that had potentially ambiguous references via indefinite pronouns. Furthermore, we removed 170 constraints that refer to GUI elements. We consider these a separate category of data constraints, as their relationship often involve spatial or visual references, such as *to the right of*, *on top of*, and *painted*. Hence, GUI-related data constraints are often implicitly *not explicit*, as the constraint applies to a property of the described GUI element (*e.g.,* the color or the coordinates of the GUI element).

We further removed the constraints from the systems where we had left fewer than 20 constraints. We also removed the *negative* sentences from these systems.

All these removed constraints can still be used in future research and we make them available in our replication package. The replication package includes 1,932 labeled constraints from 14 systems.

Finally, for this study, we were left with 475 *positive* sentences, containing 548 *explicit* constraints in 9 systems. These 548 constraints were also labeled with their *constraint type* and *operands*. We also labeled 12,236 *negative* sentences, which were used in the

Final label ***ignore*** | Total 5,860				
Coder lbls.	Ignore	Neg	N.-Expl.	Expl.
Explicit	8	0	0	0
Non-Expl.	5	0	0	
Negative	202	28		
Ignore	5,617			

Final label ***negative*** | Total=11,109				
Coder lbls.	Ignore	Neg	N.-Expl.	Expl.
Explicit	10	375	6	14
Non-Expl.	11	136	2	
Negative	419	10,136		
Ignore	0			

Final label ***not explicit*** | Total=221				
Coder lbls.	Ignore	Neg	N.-Expl.	Expl.
Explicit	13	22	16	1
Non-Expl.	34	95	38	
Negative	0	2		
Ignore	0			

Final label ***explicit*** | Total=247				
Coder lbls.	Ignore	Neg	N.-Expl.	Expl.
Explicit	21	136	13	72
Non-Expl.	0	2	0	
Negative	0	3		
Ignore	0			

Figure 2: Agreements and disagreements vs. final label

classification study from Section 4, where we also describe how they were divided into 36,839 fragments.

3.1.1 Constraint labeling protocol. We employed 10 CS graduate and undergraduate students, and four of the co-authors to label sentences and constraints. For all files that had not been examined for constraints by previous research [24], we used the following process for labeling.

Using the aforementioned MAT software [48], a labeler read through the entire document and marked each sentences with one label (*ignore*, *positive*, or *negative*). First, if the sentence was deemed to be ignored (based on the definition from above), it was labeled as *ignore*. Then, if at least one data constraint was described in the sentence, the labeler would label it and copy the operands into the label, as well as assign one of the four types defined in Section 2. In addition, for each constraint, the labeler would determine if the constraint met the criteria for being an *explicit* data constraint, or if not, provide a reason for why not (as described above). Finally, if the sentence contained no constraints, it was labeled as *negative*.

For this entire text labeling process, the labeler was either one of the authors, or two trained CS students, separately labeling each document. In both cases another author reviewed all the resulting constraint labels (*explicit* or *non-explicit*), as well as all disagreements whether by label type or reason for giving that label. For those documents labeled by two coders, Figure 2 aggregates four tables showing agreements and disagreements between the two labelers for *ignore* sentences, *negative* sentences, *explicit* constraints, and *non-explicit* constraints, as well as the finally agreed upon label (top row of each table). Agreements between both coders and the reviewer are highlighted in green, while agreement between only one coder and the reviewer are in yellow. The non-zero white cells are the cases where the final label is different from both coders' labels. During the review process, the authors (one author per document, but two authors split the work) kept note of the most common mistakes that labelers had. Particularly, notes were taken for a portion of the 375 times where a *negative* sentence was labeled as a *positive* one with an *explicit* constraint by one coder and correctly as a *negative* by the other. In 53 such cases this was because the sentence mentioned an attribute and potentially described that it could be set, but did not constrain it; for example: *"Statements can either be read in from a text file using the src attribute or from between the enclosing SQL tags."* In 30 cases, a data element was related to a value, but only as an example, or as a command to the user; for example: *"Label the current version of the VSS project $/source/aProject with the label Release1 using the username me and the password mypassword."* Other reasons for the disagreement were noted less than 10 times each. Over all resulting labels, the coders disagreed 533 times where one coder labeled the sentence as *negative* (*i.e.,* no constraints in it) while the other identified *explicit* constraints in the same sentence. In contrast, *non-explicit* vs. *negative* (233) and *non-explicit* vs. *explicit* (35) disagreements, happened only 268 times overall.

The disagreements illustrate the challenge of this task and indicate that *classifying data constraint descriptions is challenging even for humans*, hence serving as further motivation for our work on automating this task.

For those files and sections corresponding to the 299 constraints from previous work [24], we added more labels. All constraints were reviewed by an author and re-categorized as *explict*, *non-explicit*, and the sentences labeled as *ignore* or *negative*, as needed. Additionally, there were 3 systems for which the prior research did not seek to exhaustively extract all constraints in those documents, and we labeled them as well.

3.2 Discourse Pattern Coding Protocol

We analyzed the 548 *explicit* constraints from 9 systems to see if any discourse patterns are used in their descriptions. Our assumption is that the constraints are described in limited ways and we can identify discourse patterns on how they are written. These patterns define the parts of speech of the constraints and operands in the constraints.

In order to extract these patterns, we performed a qualitative discourse analysis [56, 68] of the 548 labeled contraints, based on open coding [47]. Two authors of the paper performed the *discourse pattern* coding task. First, the two authors randomly selected 100 constraints and analyzed them together. The joint analysis resulted in the identification of an initial set of patterns. A code was defined for each pattern and used for labeling. In the following steps the two coders each analyzed the remaining constraints, separately. For each new constraint, if they considered that it was described using one of the already identified patterns, they labeled it with the pattern's code. Otherwise, they defined a new pattern, which they discussed with the other coder. Once they both agreed on the new pattern, its code was defined and the previously labeled constraints were re-analyzed to see if they match the new pattern and need re-labeling. This analysis was done through consensus of the two coders. The coders then meet to discuss and resolve any discrepancies, refining the codes by combining, splitting or modifying them. This process is repeated for an additional hundred constraints at a time until all 548 constraints have been labeled.

In the end, this coding process resulted in the creation of a set of 15 *codes*, which we denominate *discourse patterns* (DPs), described in the following subsection. From here on, we refer to them as *patterns* or DPs.

3.3 Results - Discourse Patterns

The first and last columns in Table 3 show the codes and the frequency of the 15 patterns in our dataset. We list the descriptions and rules for the three most commonly used patterns in Table 2. The details of the remaining 12 patterns are available in the replication package [2].

Table 2: Three most commonly used discourse patterns.

Pattern code: `NP_BE_BINARY_VALUE` **Description**: A data element is assigned a binary value (e.g., *true/false, on/off, out/inout...*) by using an auxiliary verb. **Rule**: `[data] [be] [binary-value]` `[data]` ∈ {`NP`} `[be]` ∈ {*be, allow, remain, show*} `[binary-value]` ∈ {`JJ, VVN, $binary-word`} `[$binary-word]` ∈ {*true/false, on/off, out/inout...*} **Example**:[Background mode] [is] [on] (from jedit-5.6)
Pattern code: `NP_BE_VALUE` **Description**: A data element is assigned a concrete value by using an auxiliary verb. **Rule**: `[data] [be] [value]` `[data]` ∈ {`NP`} `[be]` ∈ {*be, allow, remain, show*} `[value]` ∈ {`NP, CD`} **Example**:[Default maxwait] [is] [3 minutes] (from apache-ant-1.10.6)
Pattern code: `NP_CD_TO_CD` **Description**: A data element is assigned a range of numeric values. The lower bound and upper bound are connected by a preposition/conjunction pair (*e.g., from. . . to, between. . . and*). **Rule**: `[data] [num][to][num] | [data] [from] [num][to][num]` `[data]` ∈ {`NP`} `[from]` ∈ {*from, between*} `[num]` ∈ {`CD`} `[to]` ∈ {*to, and*} **Example1**: [South latitude extent] [-90] [to] [90] (from swarm-2.8.11) **Example2**: [hour] [from] [00] [to] [23] (from joda_time-2.10.3)

The most common discourse pattern is `NP_BE_BINARY_VALUE`, which represents a binary value has a dependency on the modal term "*be*", "*allow*", "*remain*" or "*show*" with a data element. For example, "*manual scale power is enabled*" has the constraint *manual scale power = enabled* with operands *manual scale power* and *enabled.*

The second most common discourse pattern is `NP_BE_VALUE`. Similar to `NP_BE_BINARY_VALUE`, this pattern represents that a value has a dependency on the modal term "*be*", "*allow*", "*remain*" or "*show*" with a data element; in this pattern, the value is a concrete value. For example, "*default port is TCP 80*" has the constraint *default port = TCP 80* with operands *default port* and *TCP 80.*

The third most frequent pattern is `NP_CD_TO_CD`, which represents a range of numeric values has a dependency on a preposition/conjunction pair (*e.g.,* "*from. . . to*" and "*between. . . and*") with a data element. For example, "*a valid port number between 0 and 65,535*" has the constraint *port number >= 0 and port number <= 65535* with operands *port number*, *0*, and *65535.*

Discourse pattern rules and operands. The rules of the discourse patterns indicate which parts of the data constraints are the *operands* . For the rules shown in Table 2, the terms matching `[data]`, `[value]` or `[num]` are the operands. Specifically, for the `NP_BE_BINARY_VALUE` pattern example, *manual scale power* matches `[data]` in the rule and *enabled* matches `[binary-value]` in the rule;

Table 3: Distribution of DPs and constraint types.

Pattern code	conv	vcom	binv	catv	Total
NP_BE_BINARY_VALUE	·	·	236	·	236
NP_BE_VALUE	38	14	·	·	52
NP_CD_TO_CD	·	48	·	·	48
NP_SET_TO_BINARY_VALUE	·	·	38	·	38
SET_NP_TO_BINARY_VALUE	·	·	30	·	30
NP_COMP_NP	·	26	·	·	26
CD_NP	3	22	·	·	25
NP_IN_VALUESET	·	·	·	17	17
NP_IF_BINARY_VALUE	·	·	16	·	16
SET_NP_TO_VALUE	13	2	·	·	15
NP_SET_TO_VALUE	7	7	·	·	14
NP_OF_VALUE	9	3	·	·	12
NP_EXIST	·	·	10	·	10
VALUE_FOR_NP	7	·	·	·	7
NP_CD	2	·	·	·	2

hence, they are the operands. Likewise, for the NP_CD_TO_CD example, *port number* matches [data] in the rule, and both *0* and *65,535* match [num] in the rule, making them the three operands.

Correlation between discourse patterns and constraint types. Table 3 shows the distribution of discourse patterns over the constraint types. In columns 2-5, we refer to concrete-value as *conv*, value-comparision as *vcom*, binary-value as *binv*, categorical-value as *catv*. We observed that 10 out of 15 discourse patterns correlate to only one constraint type. For example, all the 236 constraints described with the NP_BE_BINARY_VALUE discourse pattern are of binary-value type. The rest five discourse patterns (CD_NP, NP_BE_VALUE, NP_OF_VALUE, NP_SET_TO_VALUE, and SET_NP_TO_VALUE) correlate to two constraint types: concrete-value and value-comparison. For example, of the 52 constraints described with the NP_BE_VALUE discourse pattern, 38 are of concrete-value type and 14 are of value-comparison type. This strong correlation indicates that we can use the discourse pattern of a constraint to automatically infer its type, which we actually do in the classification study (Sec. 4).

3.4 Threats to Validity

Construct Validity. Subjectivity in identifying constraints was mitigated by following a well-defined protocol to label ignore, positive, and negative cases, which included specific rules to categorize the relevant sentences and fragments. The labeling process was performed in iterations, and included coding reviews and discussion of disagreements to learn about the system documents and refine the rules. Subjectivity in deriving the discourse patterns from identified constrains was mitigated by following a open coding procedure with discussion of ambiguous cases, refinement of the pattern catalog and coded data, and assessment of the coding process.

External Validity. The identified discourse patterns may not generalize to explicit constraints documented for other software systems beyond our pool of systems. Analysis of additional documentation for the systems we used in our study may also reveal additional constraints and discourse patterns. That said, the systems we used are of different domains and kinds (*e.g.,* desktop apps and libraries), and the constraints came from different documents (user manuals, requirement and design specifications); this strengthens the generalization of our patterns. Additionally, discovering new patterns from more data in the future would not invalidate the ones we documented in this dataset. Like any pattern catalog, this one is meant to evolve over time.

4 DATA CONSTRAINTS CLASSIFICATION

We conducted a study on using machine learning approaches for automatically classifying the natural description of data constraints. In other words, given a sentence and a fragment from that sentence, we want to determine whether that fragment is a data constraint description or not. The goal of the study is to determine which features and learners are best suited for this task. As such, we aim to answer the following research question.

RQ: What are the best features, configurations, and learners for classifying data constraint descriptions?

4.1 Data

We used the 475 *positive* sentences from 9 systems that contain 548 *explicit* data constraint descriptions, collected in the study described in Section 3. For each constraint we have the start character and end character in the sentence, since they are contiguous sequences of words in a sentence. In addition, each constraint is labeled with its *operands*, *constraint type*, and the *discourse pattern*. We used the 12,236 sentences labeled *negative* from the 9 systems. The *negative* sentences did not have any fragments labeled.

4.1.1 *Labeling negative sentences.*

Given the large number of negative sentences, we could not label them manually, so we developed a tool (named S2F) to do that. S2F breaks a given sentence into fragments that correspond to any of the 15 discourse patterns we identified, extracts the operands that match the discourse pattern, and assigns a potential constraint type to each fragment.

Discourse pattern matching. We use spaCy Matcher [28] to identify the 15 discourse patterns, applying a set of linguistic features: part-of-speech tags, dependencies and entities. Previous research used spaCy Matcher modules for various natural language processing (NLP) tasks. For example, Zhang et al. [80] used it to extract dependency patterns of syntax-semantics constraints from source and target languages, while Monszpart et al. [50] used it to extract medical terminology from natural language corpora.

To split a sentence into fragments that match the discourse patterns, we adopt a layered analysis approach [62], which involves defining each pattern with two levels: one for token matching and another for dependency matching between the tokens.

(1) To perform tokens matching, we create patterns of tokens by defining their attributes such as part-of-speech tags or entity labels in the rules. Each token in the pattern is assigned a unique ID. For example, we can specify the part-of-speech tag of NOUN for NP, the lemma of be for BE, and part-of-speech tags of JJ, VVN, as well as the lemmas of a set of words observed in the constraints for BINARY_VALUE in the NP_BE_BINARY_VALUE pattern. In this case, we assign ID "data" to the token NP, and "value" to the token BINARY_VALUE to allow referencing these two specific tokens for further processing.

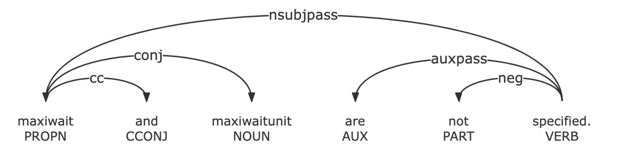

Figure 3: Examples of utilizing "conj" to extract fragments

(2) To perform dependency matching, we define specific relationships between tokens that correspond to the syntactic dependencies in the text. An example of `NP_BE_BINARY_VALUE` includes two specific dependency relationships: the "*nsubjpass*" relationship between the verb and its subject in passive voice sentences, which represents the relationship between `BINARY_VALUE` and `NP`, and the "*auxpass*" relationship linking a passive verb to its auxiliary verb, which represents the relationship between `NP_BE` and `BINARY_VALUE`. We extract fragments by selecting contiguous words between tokens with the maximum length.

We utilize four distinct patterns (*i.e.,* `NP_OF_VALUE`, `VALUE_FOR_NP`, `CD_NP`, and `NP_CD`) to extract complex nouns and noun phrases from constraints using spaCy. These patterns serve as potential noun phrases for facilitating boundary detection of noun phrases. Our approach first identifies the above patterns in the text, then merges the tokens from matched patterns into a single token, and finally identifies the other patterns in the text. It provides a greater degree of flexibility and expressiveness when representing tokens involving noun phrases in patterns.

We observe that the "*conj*" (conjunction) dependency, which implies that two words have interchangeable usage in a sentence, can be utilized to identify tokens that possess the same syntactic meaning as specific tokens (*e.g.,* `data` and `value`) in the pattern. By substituting specific tokens with such tokens that have similar syntactic meaning, additional similar fragments can be extracted as illustrated in the example in Figure 3. Our approach first extracts the fragment "*maxiwait and maxiwaitunit are not specified*" with the tokens `maxiwait` and `specified`. Then, by considering linking the words `maxwait` and `maxwaitunit` via the "*conj*" relationship, S2F is able to extract another fragment: "*maxwaitunit are not specified*" with the specific tokens `maxwaitunit` and `specified`.

Operands extraction. We extract the *operands* from the matched tokens that have an specific ID name containing the keywords "`data`" and "`value`".

Constraint type detection. Based on the correlation between discourse patterns and constraint types reported in Section 3, 10 out of 15 discourse patterns correspond to only one constraint type, which means we can unambiguously match the pattern to a constraint type. The other 5 discourse patterns (*i.e.,* `CD_NP`, `NP_BE_VALUE`, `NP_OF_VALUE`, `NP_SET_TO_VALUE`, and `SET_NP_TO_VALUE`) correspond to two constraint types: `value-comparision` and `concrete-value`. For these patterns, we implemented a set of five heuristic rules to infer the constraint type for a fragment.

- **Heuristic1**: If any word from the `$word-list` occurs in the sentence, then the `value-comparison` type will be assigned to the fragment.
  `$word-list`: *{at least, up to, lower, larger, more, negative, positive, multi, last, first...}*
- **Heuristic2**: If the conjunctions *if* or *when* are present in the sentence within five tokens preceding the start token of the fragment, then the `value-comparison` type will be assigned to the fragment.
- **Heuristic3**: If the negation word *not* is present in the fragment, then the `value-comparison` type will be assigned to the fragment.
- **Heuristic4**: If the fragment fails to meet the criteria in heuristics 1, 2 and 3, the fragment type will be assigned as `concrete-value`.

We obtained the `$word-list` in **Heuristic1** by analyzing the constraints with the five discourse patterns. We categorized the words that correspond to the definition of the `value-comparison` type, and observed that the constraint sentences contain words that indicate a comparison relationship. For example, "*If three or more channels are open, it will prompt the user for the desired channel*" contains the comparison word *more* which indicates a comparison relations between *the number of channels* and *3*. In another example, "*Make at least 1 pick in the clipboard to enable the Use Clipboard Picks option*", the comparison word *at least* implies a comparison between *the number of picks* and *1*. Hence, we add the comparison words *more* and *at least* into `$word-list` of **Heuristic1**. When inferring the constraint type for the five discourse patterns mentioned above, if the sentence contains any word from the `$word-list`, S2F will assign its constraint type as `value-comparison`.

Another type of heuristics we used is to detect the conjunctions *if* or *when* in the constraint sentence (**Heuristic2**) or to detect the negation word *not* in the constraint itself (**Heuristic3**). For example, in the sentences "*If the data source is not an RSAM data source, the RSAM viewer will indicate there is no RSAM data for the channel*" and "*This method can be used when the ExecutorType is ExecutorType.BATCH*", the conjunctions *if* and *when* indicate a conditional situation, which implies a comparison relationship of *==* or *!=* (if the negation word *not* is also present). We observed that these conjunctions normally occur in the close proximity to the first token of the constraint. We therefore set a search scope of five tokens preceding the beginning token of the constraint. When inferring the constraint type for the five patterns mentioned above, if the conjunction *if* or *when* is present in the constraint sentence within five tokens preceding the start token of the constraint, or if the negation word *not* is present in the constraint fragment, S2F will assign its constraint type to `value-comparison`.

If the constraint fails to satisfy any of these three heuristics, S2F will assign the constraint type to `concrete-value`.

4.2 Machine Learners

While many techniques have been proposed for text classification, in this study we apply five commonly used classifiers for constraints classification: Logistic Regression [45], Naive Bayes Classifier [41] , linear Support Vector Machine [29], non-linear Support Vector Machine [29], and Decision Tree [59].

We establish hyperparameters with predefined values as following for each classifier. Logistic Regression employs several parameters, including two regularization techniques - *L1* and *L2*, a penalty parameter *C* with values 0.01, 0.1, 1, 5, 10, 15 and 20, where larger values of *C* correspond to a higher penalty imposed on errors. In addition, the optimization of the logistic regression model's parameters is carried out using the solver *liblinear* [19]. The Naive Bayes Classifier uses a smoothing parameter α to prevent zero probabilities, with possible values of 0, 0.005, 0.01, 0.1, 0.5, 1.0, and 2.0. The Linear Support Vector Machine employs parameters such as *L1* and *L2* regularization techniques, a penalty parameter C with values of 0.1, 1, and 10. The Non-linear Support Vector Machine also uses a penalty parameter C of 1, and a kernel type of *rbf* which is a popular kernel function to transforms the data into a higher-dimensional space. For the Decision Tree algorithm, commonly used criteria such as *gini impurity* and *entropy* are applied, as well as strategies for determining the best way to split the data at each node of the tree using either *best* or *random*. Additionally, the maximum depth of the decision tree can be set to either 6, 8, or 10.

Using other, more complex classifiers (*e.g.,* deep learners) or language models is subject of future work. Such classifiers likely require additional training data.

4.3 Features

We use five features for classifying sentence fragments as data constraint descriptions: *fragments*, *sentences*, *discourse patterns*, *operands*, and *constraint types*.

Fragments, *sentences*, and *operands* are a set of words extracted from software artifacts. We use *n-grams* [72] to capture the vocabulary of these unstructured text features. N-grams are contiguous sequences of n terms in the text. We use unigrams, bigrams, and trigrams, where each n-gram is defined as a boolean feature indicating the presence or absence of such an n-gram in any of the textaul features. In addition, we use *PoS tags* to capture the type of vocabulary used in fragment, sentence and operands. We use contiguous sequences of *n-PoS* tags in the text. We define 1,2,3-PoS tags as boolean features indicating the presence or absence of a tag combination in any of the text of fragment, sentence and operands features.

Discourse patterns and *constraint types* are dict-like features which can be mapped to numeric feature values. Each discourse pattern or constraint type is defined as an integer feature indicating which pattern and type correspond to any of the fragments.

4.4 Training, Testing, and Measures

For each of the five classifiers we used two types of training and testing with all combination of the five features: 10-fold cross validation (10CV) and cross-system validation (CSV).

For 10CV, we shuffle the dataset randomly and select data from 70% of the sentences for training and parameter calibration and 30% for testing, to avoid over-fitting [18]. The balance between positive and negative samples is preserved in both the training and testing sets. We repeat the 70-30 split 5 times and each time we do the following steps:

(1) The 70% of the data set is randomly split into ten equal subsets (folds) by preserving the proportion of samples in each fold.
(2) GridSearchCV [54] creates a grid of all possible combinations of pre-defined hyperparameters. For each combination of hyperparameters in the predefined grid, the learner is trained using nine of the subsets as training data, and validated on the tenth fold.
(3) We repeat the previous steps ten times, each time using a different fold for validation.

GridSearchCV selects the combination of hyperparameters that achieved the best average performance across 10 folds. Finally, each learner is trained using the entire 70% of the dataset with the selected hyperparameters. We then evaluate each classifier on the 30% testing dataset and compute precision, recall, and F1 scores (defined in Section 4.4.2). The average performance of the learner across 5 experiments is computed for each combination of features and SMOTE ratio (explained below).

For CSV, we perform a leave-one-out system split iterating nine times. In each iteration, we select data from eight systems to train and calibrate hyperparameters, and reserve the data from the ninth system for evaluation. We follow a process similar to 10CV for selecting the combination of hyperparameters that yields the best average performance on the training sets of eight systems, repeating this process for each iteration of the leave-one-out split. Finally, each learner is then trained using the entire training data from the eight systems, with the selected hyperparameters, and tested on the ninth system. The performance of the learner is evaluated across all nine experiments, and the average performance is calculated for each combination of features and SMOTE ratio [11].

4.4.1 *SMOTE.*

To tackle the issue of imbalanced datasets where the number of constraints is significantly less than tool-generated fragments (*i.e.,* 548 versus 36839), we use the Synthetic Minority Over-sampling Technique (SMOTE) [11]. SMOTE is a well-known oversampling technique in the field of machine learning that generates synthetic samples for the minority class by interpolating new instances between existing nearest neighbors. Chawla et al. [11] showed that SMOTE consistently outperformed the other techniques in terms of both sensitivity and specificity. In addition, Fernández et al. [22] proved that SMOTE resulted in better or equivalent results in terms of classification performance, while also reducing the likelihood of overfitting. In our study we apply SMOTE to the training data, which yields positives (minority class) accounting for the negatives (majority class) by a ratio of either 0 (when SMOTE is not used), 0.25, 0.5, 0.75, or 1. In other words, for a SMOTE ratio of 1, the number of positive and negative samples in the training data is the same.

4.4.2 *Evaluation Metrics.*

In order to evaluate effectiveness of the classifiers, we employ standard metrics, specifically precision, recall, and F1 score [18]. Precision is defined as the percentage of accurately classified constraints out of the total number of instances classified as constraints with reference to a gold set (*i.e.,* $Precision = TP/(TP + FP)$). Recall is the percentage of constraints can be correctly classified out of the total number of constraints in the reference dataset. (*i.e.,* $Recall = TP/(TP + FN)$). F1 score is the harmonic means of the precision and recall (*i.e.,* $F1 = 2 * (precision * recall)/(precision + recall)$) and we use it as the main performance measure.

4.5 Results

We evaluated each of the five classifiers with all combinations of the five features, using all five SMOTE ratios. For each classifier we

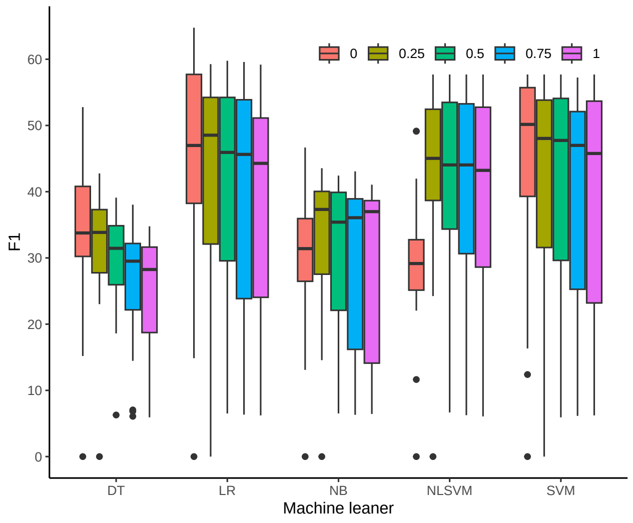

Figure 4: F1 scores with different SMOTE ratios on 10CV eval.

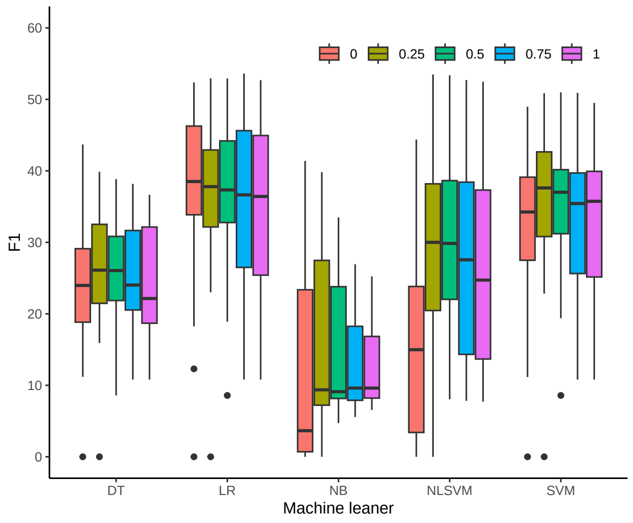

Figure 5: F1 scores with different SMOTE ratios on CSV eval.

evaluate 155 different configurations, using 10-fold cross validation and cross-system validation.

4.5.1 The Effect of Data Balancing. We start our analysis of the results by looking at the effect of the SMOTE ratio. We focus our analysis on the F1 value, which maximizes precision and recall, equally. Figure 4 shows the performance of the five learners for the five SMOTE ratios, across all 31 feature combinations. We observe that Decision Tree is the most sensitive to data balancing, the Naive Bayes and non-linear SVM perform significantly worse without SMOTE (0 ratio) than with data balancing (*i.e.,* the higher SMOTE ratios).

Similarly, Figure 5 shows the same information for the cross-system validation. We observe the same phenomenon as above for the Naive Bayes and and non-linear SVM, but for the cross-system validation, the Naive Bayes is less sensitive to data balancing.

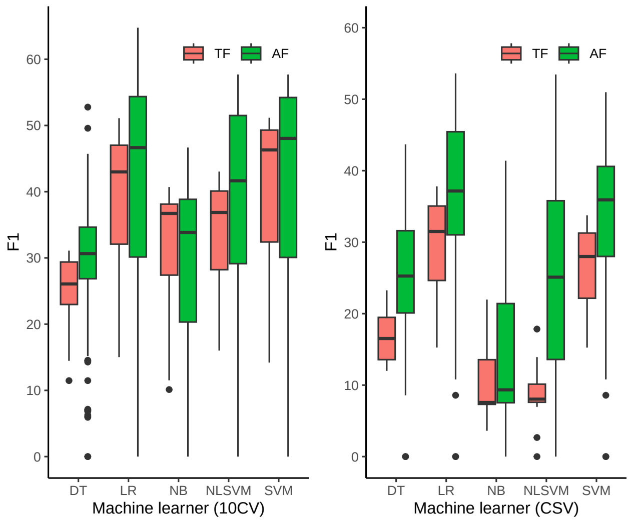

Figure 6: F1 scores using TF and AF feature sets

Overall, the 0.25 SMOTE ratio yields the best results (on average) across learners and types of evaluation. Nevertheless, as we will see further, the best performance for each learner and feature set are not always achieved with this SMOTE ratio.

4.5.2 The Effect of Feature Selection. We further analyze the results of using the *sentence* and *fragment* features only (denoted as TF - textual features), compared with using all the five features (denoted as AF - all features). Figure 6 shows the performance (F1) of the five classifiers, using the TF and AF feature sets, across all SMOTE ratios, using the 10 fold cross validation (10CV - on the left) and the cross system validation (CSV - on the right).

The average F1 for each feature set and learners for the 10 fold cross validation are:

- Decision Tree: 29.46 using AF and 24.81 using TF
- Linear Regression: 41.94 using AF and 38.44 using TF
- Naive Bayes: 29.28 using AF and 31.12 using TF
- SVM: 41.65 using AF and 40.14 using TF
- nonlinear SVM: 39.56 using AF and 33.80 using TF

The average F1 for each feature set and learners for cross system validation are:

- Decision Tree: 25.09 using AF and 16.75 using TF
- Linear Regression: 35.97 using AF and 29.23 using TF
- Naive Bayes: 13.81 using AF and 10.09 using TF
- SVM: 33.95 using AF and 26.56 using TF
- nonlinear SVM: 25.63 using AF and 8.68 using TF

We make several observations about the use of the feature sets, the learners, and the evaluation types. Regarding *feature selection*, we observe that (on average) for each learner and each evaluation type, using all features improves the classification performance, except for Naive Bayes for 10CV. Overall, using AF, across the five learners, for 10CV, the average F1 is 36.38 and for the cross system validation, the average F1 is 26.89. Using the TF only, across the five learners, for the 10 fold cross validation, the average F1 is 33.66 and for the cross system validation, the average F1 is 18.26.

Regarding the choice of *learners*, we observe that using Naive Bayes and Decision Trees lead to worse results than using any of

Table 4: Best configurations for the three best learners.

FR	S	OP	DP	CT	SM	Eval	PR	RE	F1	MODEL
1	1	1	1	1	0	10CV	70.87	59.73	64.76	LR
0	1	1	1	1	0.75	CSV	57.92	55.23	53.61	
1	0	1	1	0	0.25	10CV	60.95	61.34	61.12	NLSVM
1	0	1	1	1	0.25	CSV	61.69	51.09	53.82	
1	1	1	1	1	0.25	10CV	62.72	59.27	60.92	SVM
0	1	1	1	1	0.5	CSV	66.01	45.44	50.98	

the other three learners, for both types of validation, across all feature sets.

Regarding the *validation method*, 10CV shows better performance across all five learners and feature sets, than using CSV. This is likely due to the distribution of the constraints across systems. We further analyze the results of the cross system validation in the flowing subsection.

We focus now on the best performing configurations (*i.e.,* features and SMOTE ratio) from the three best performing learners (*i.e.,* LR, SVM, NLSVM). Table 4 shows these configurations. All three learners achieve F1 values above 60 for the 10 fold cross validation and over 50 for the cross system validation. Linear Regression and SVM achieve the best results with the same combination of features: all five features for 10 fold validation and all but fragments for the cross system validation, albeit with different SMOTE ratios. As for the nonlinear SVM, its best combination of features for 10-fold cross validation is without constraint type and sentence (with 0.25 SMOTE), whereas for the cross validation is without sentence (0.25 SMOTE). Further analysis of the data not shown in Table 4, reveals that the combination with all features (0.25 SMOTE) achieves 70.39 precision, 49.27 recall, and 57.93 F1 values (being the 5th best configuration) for the nonlinear SVM (for 10CV). Likewise, for cross system validation, the combination of all features without fragments (0.50 SMOTE) is the 5th best for the nonlinear SVM (86.6 precision, 32.67 recall, 44.43 F1).

We conclude that these two configurations are the best:

- features: fragment+sentence+DP+operand+type (0.25 or 0.0 SMOTE) for 10CV
- features: sentence+DP+operand+type (0.50 SMOTE) for CSV

4.5.3 Analysis of False Negatives. As mentioned above, the classifiers show lower performance in the CSV than in the 10CV. While we focused on F1 as indicator for classifier performance, we observe that the best configurations for the three best learners (from Table 4) achieve good precision for the cross system validation (*i.e.,* 55.05, 66.46, and 60.75, respectively). However, their recall values are lower (*i.e.,* 55.05, 45.63, and 51.71, respectively), meaning that nearly half of the constraints are classified incorrectly (*i.e.,* false negatives). We analyzed the features of the false negative constraints, specifically their *discourse patterns* and *constraint type*. Table 5 shows the false negative rate (column 2), and the number of positive (column 4) and negative (column 3) fragments, for each DP or constraint type. For constraints with the top 8 DPs in Table 5, the false negatives rates range from 69% to 100%. As it turns out, the former group of constraints make up a very small percentage of the total number of fragments (positives and negatives) with the same DPs (*i.e.,* less than 2%), with the exception of `SET_NP_TO_VALUE`, which account for 7.9% of the total fragments, but there are only 15 of them.

Table 5: False negatives vs. DP and constraint type.

DP / Constraint type	FN rate	-frag	+frag	% pos / total
`NP_CD`	100.0%	715	2	0.3%
`VALUE_FOR_NP`	100.0%	1,196	7	0.6%
`NP_OF_VALUE`	88.9%	4,660	12	0.3%
`SET_NP_TO_VALUE`	86.7%	176	15	7.9%
`NP_BE_VALUE`	85.3%	13,370	52	0.4%
`NP_COMP_NP`	79.5%	2,868	17	0.6%
`NP_IN_VALUESET`	76.5%	1,469	26	1.7%
`CD_NP`	69.3%	3,093	25	0.8%
`NP_IF_BINARY_VALUE`	56.3%	103	16	13.4%
`NP_EXIST`	50.0%	70	30	30.0%
`SET_NP_TO_BINARY_VALUE`	43.3%	8,994	236	2.6%
`NP_BE_BINARY_VALUE`	41.4%	62	10	13.9%
`NP_SET_TO_VALUE`	26.2%	25	14	35.9%
`NP_CD_TO_CD`	22.2%	18	48	72.7%
`NP_SET_TO_BINARY_VALUE`	8.8%	20	38	65.5%
concrete-value	86.5%	9,249	79	0.8%
categorical-value	76.5%	2,868	17	0.6%
value-comparison	50.0%	20,512	122	0.6%
binary-value	38.8%	4,210	330	7.3%

For the other seven DPs, the false negatives rates range from 8% to 56%. The ratio of the positive fragments with these DPs is much higher, ranging from 13% to 65%, with the exception of `SET_NP_TO_BINARY_VALUE`, which has a large number of instances (*i.e.,* 236). We observe a similar phenomenon regarding the constraint type of the false negatives (bottom part of Table 5).

In conclusion, in order to reduce the false negative rates (*i.e.,* improve recall) for constraints of a given type or DP, we need to either have a higher positive to negative ratio, or a large number of positives (at least 100).

4.6 Threats to Validity

Internal Validity: Our tool-based approach to create the set of negative fragments for classification affects the internal validity of our results. It is possible that other ways to identify fragments produce a different set, thus yielding different classification results. More so, the fragments of the sentences that were not matched to discourse patterns, in the negative sentences, were ignored. Likewise, the fragments of the positive that do no described explicit constraints are ignored.

Conclusion Validity: We chose F1 as the metric to assess the performance of the classifiers. While widely used for binary classifications, had we used other measures, the conclusions of our study would have been different. For example, if we would aim to maximize recall given a certain precision threshold, other configurations might have been the best in that scenario.

External Validity: The classification results may not generalize for other datasets, corresponding to different systems than the ones studied. Our future work will include additional data from other systems.

5 RELATED WORK

Software data constraints and business rules. Data constraints specify the values that are allowed or required for the data used in a software system [23, 74, 75, 78], and are typically described in natural language software artifacts (*e.g.,* requirements and design documents, or user manuals). Florez *et al.* found that developers implement data constraints in Java code using 31 well-defined patterns [23] and such constraints can be traced automatically to methods and lines of code [24]. Yang *et al.* found that data constraints can be implemented across multiple architectural layers in web applications, including the front-end, application, and database layers [78]. The authors focused on detecting implementation inconsistencies of these data constraints across layers. Other work has focused on automatically extracting business rules from source code (which may include constraints) [12, 13, 27, 30, 63, 64, 73]. These approaches aim to assist developers in documenting, verifying, and understanding the code as well as in supporting tasks such as code change impact analysis. Our work is motivated by this prior work.

Analysis of natural language software artifacts. Researchers have studied textual properties of a wide range of natural language software artifacts. Ko *et al.* analyzed the types and frequency of the words appearing in bug report titles to understand how stakeholders express software problems [34]. Sureka *et al.* executed a similar analysis on issue titles to find vocabulary patterns useful for predicting the issue severity [69]. Chaparro *et al.* found 154 patterns that reporters use to describe bugs in bug reports, and used them to perform automated bug report quality verification [9, 66]. Xiao *et al.* described linguistic patterns to identify access control policies in requirement documents [76]. Di Sorbo *et al.* identified linguistic patterns in sentences from mobile app reviews, developer email threads, and issue titles to perform content classification [17]. Via textual analysis, other work has identified types of information in API documents via textual analysis [40], studied API privacy policy information, and identified information relevant to development activity summaries [71]. Liu *et al.* [39] studied the structure of programming task descriptions in StackOverflow and constructed a knowledge graph that provides a hierarchical conceptual structure for such tasks in terms of [actions], [objects], and [constraints]. The knowledge graph was used to help formulate high quality how-to questions related to programming tasks.

Our work studies and documents the patterns that stakeholders use to describe data constraints in software documents. To the best of our knowledge, this is a first-of-its-kind study, yet it is possible that prior work derived patterns that indirectly capture data constraints. Similar to prior work [46, 57], we adopt an open-coding-based methodology for deriving such patterns.

Classification of textual software artifacts. Extensive research has focused on automatically classifying entire artifacts (*e.g.,* issues, comments, or app reviews) [3, 6, 7, 16, 25, 26, 32, 44, 52, 55, 70, 77, 81, 82] and sentences or fragments in such documents [1, 4, 5, 8, 9, 15, 17, 31, 35–37, 42, 43, 49, 51, 53, 58, 60, 61, 65, 66, 79], for diverse purposes, including assessing the code comment [53, 79] and bug report quality [8, 9], prioritizing requirements [1, 5, 15, 37, 65], reproducing reported bugs [21, 38], and recommending APIs [31, 61].

Most approaches for fine-grained textual classification have relied on canonical supervised learning models [1, 5, 9, 14, 15, 17, 31, 35–37, 49, 53, 58, 60, 61, 65, 66, 79], such as SVMs, Random Forests, and Logistic Regression. The advantage of such models is their interpretability; this is one of the reasons why we leverage them in our work. Deep learning approaches [8, 10, 38, 42, 43], such as Word2Vec, LSTMs, or BERT, are scarce (so are unsupervised approaches [1, 31]) since they require expensive training or fine-tuning with large labeled datasets. The relatively limited amount of data we used for evaluation prevented us from exploring these models, which are also hard to interpret. Other work proposed to use IR-based techniques (*e.g.,* LDA or BTM) [1] for classification, and dependency parsing of text or heuristics [8, 9, 20, 21, 51, 67].

Features used by canonical learning models include heuristics (*e.g.,* keywords, regular-expression-based features, or word count) [9, 35, 53], textual properties (*e.g.,* nouns, n-grams, part of speech, sentiment, and term/document frequency) [1, 4, 5, 9, 14, 35, 37, 60, 66, 79], and discourse/linguistic patterns [1, 9, 15].

Sentences and fragments often represent information valuable for developers, including conditions, facts, restrictions, arguments, explanations, and other elements found in requirements [1, 5, 15, 37, 65], issues and bug reports [4, 8, 9, 20, 21, 35, 38, 66, 67], code comments [14, 51, 53, 60], API documentation [31, 36, 49, 61, 79], app user reviews [17], project README files [58], and programming forum posts [42, 43].

While none of this research focuses on identifying data constraints in software artifacts, Kiziltan *et al.* [33] identified general constraints in textual problem descriptions via machine learning, in the context of solving constraint problems via formal solvers.

Inspired by all this prior work, we perform text classification on software artifacts to classify data constraint descriptions via canonical machine learning models and novel features.

6 CONCLUSIONS AND FUTURE WORK

By analyzing the natural language descriptions of 548 explicit data constraints from the documentation of 9 systems, we found that the data constraints are described using 15 linguistic discourse patterns. We leveraged these discourse patterns to help us automatically identify the operands and constraint type, of explicit data constraints described in natural language. A comprehensive empirical study evaluated 155 different configurations for each of five learners for classifying data constraint descriptions. The results revealed that using the discourse patterns we identified, the constraint operands, their type, their description, and the sentence containing the constrains as features for training, leads to best classification results (70.87% precision and 59.73% recall, 64.76% F1). We also found correlations between the constraint type, the discourse patterns, and the ratio of positive to negative training data, which allowed to formulate guidelines for future training data collection efforts, which will likely lead to better classification performance.

With that in mind, these findings and conclusions amount to a promising step towards the automatic identification and extraction of data constraints from natural language text.

In addition to the future work mentioned throughout the paper, focusing on improving the classification, the important next step to achieve the extraction of constraints to be used in a pipeline with traceability or test generation tools. The main challenge there, in addition to improving the classification performance, is to establish the impact of the false positives on the tools downstream from the classifier.

REFERENCES

[1] Zahra Shakeri Hossein Abad, Oliver Karras, Parisa Ghazi, Martin Glinz, Guenther Ruhe, and Kurt Schneider. 2017. What works better? a study of classifying requirements. In *Proceedings of the IEEE 25th International Requirements Engineering Conference (RE)*. IEEE, 496–501.

[2] Anonymous. 2023. towards-automated-identification-of-data-constraints. https://anonymous.4open.science/r/towards-automated-identification-of-data-constraints-8FC1/.

[3] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. 304–318.

[4] Deeksha Arya, Wenting Wang, Jin LC Guo, and Jinghui Cheng. 2019. Analysis and detection of information types of open source software issue discussions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 454–464.

[5] Cody Baker, Lin Deng, Suranjan Chakraborty, and Josh Dehlinger. 2019. Automatic multi-class non-functional software requirements classification using neural networks. In *Proceedings of the IEEE 43rd annual computer software and applications conference (COMPSAC'19)*, Vol. 2. IEEE, 610–615.

[6] Stefanie Beyer, Christian Macho, Martin Pinzger, and Massimiliano Di Penta. 2018. Automatically classifying posts into question categories on stack overflow. In *Proceedings of the 26th Conference on Program Comprehension*. 211–221.

[7] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.

[8] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 86–96.

[9] Oscar Chaparro, Jing Liu, Fiorella Zampetti, Laura Moreno, Massimiliano DiPenta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2017)*. ACM, New York, NY, USA, 396–407.

[10] Preetha Chatterjee, Kostadin Damevski, and Lori Pollock. 2021. Automatic extraction of opinion-based Q&A from online developer chats. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1260–1272.

[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[12] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. 2012. A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. In *Rules on the Web: Research and Applications (Lecture Notes in Computer Science)*, Antonis Bikakis and Adrian Giurca (Eds.). Springer, Berlin, Heidelberg, 17–31.

[13] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. 2013. Extracting Business Rules from COBOL: A Model-Based Framework. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. 409–416. https://doi.org/10.1109/WCRE.2013.6671316

[14] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[15] Fabiano Dalpiaz, Davide Dell'Anna, Fatma Basak Aydemir, and Sercan Çevikol. 2019. Requirements classification with interpretable machine learning and dependency parsing. In *Proceedings of the IEEE 27th International Requirements Engineering Conference (RE'2019)*. IEEE, 142–152.

[16] Andrea Di Sorbo, Sebastiano Panichella, Corrado A. Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C. Gall. 2015. Development Emails Content Analyzer: Intention Mining in Developer Discussions (T). In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*. 12–23.

[17] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. 2019. Exploiting natural language structures in software informal documentation. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1587–1604.

[18] Geoff Dougherty. 2012. *Pattern recognition and classification: an introduction*. Springer Science & Business Media.

[19] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *The Journal of Machine Learning Research* 9 (2008), 1871–1874.

[20] Mattia Fazzini, Kevin Patrick Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing Mobile App Bug Reporting via Real-time Understanding of Reproduction Steps. *IEEE Transactions on Software Engineering* (2022).

[21] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.

[22] Alberto Fernández, Salvador García, Mikel Galar, Roberto C Prati, Bartosz Krawczyk, and Francisco Herrera. 2018. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 7, 1 (2018), 1–19.

[23] Juan Manuel Florez, Laura Moreno, Zenong Zhang, Shiyi Wei, and Andrian Marcus. 2022. An empirical study of data constraint implementations in Java. *Empirical Software Engineering* 27, 5 (2022), 119.

[24] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. 2022. Retrieving Data Constraint Implementations Using Fine-Grained Code Patterns. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1893–1905.

[25] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. 495–504.

[26] Emitza Guzman, Muhammad El-Haliby, and Bernd Bruegge. 2015. Ensemble methods for app review classification: An approach for software evolution (n). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 771–776.

[27] Tomomi Hatano, Takashi Ishio, Joji Okada, Yuji Sakata, and Katsuro Inoue. 2016. Dependency-Based Extraction of Conditional Statements for Understanding Business Rules. *IEICE Transactions on Information and Systems* E99.D, 4 (2016), 1117–1126. https://doi.org/10.1587/transinf.2015EDP7202

[28] Matthew Honnibal and Ines Montani. 2018. Industrial-Strength Natural Language Processing with Python and spaCy. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2018), 191–200.

[29] Chih-Wei Hsu and Chih-Jen Lin. 2003. A Practical Guide to Support Vector Classification. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 169–208.

[30] Hai Huang, Wei-Tek Tsai, Sourav Bhattacharya, Xiaoping Chen, Yamin Wang, and Jianhua Sun. 1996. Business Rule Extraction from Legacy Code. In *Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC)*. 162–167. https://doi.org/10.1109/CMPSAC.1996.544158

[31] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 38–48.

[32] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2019. Ticket Tagger: Machine Learning Driven Issue Classification. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*. 406–409.

[33] Zeynep Kiziltan, Marco Lippi, Paolo Torroni, et al. 2016. Constraint detection in natural language problem descriptions. In *IJCAI*, Vol. 2016. Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI'16), 744–750.

[34] Andrew J. Ko, Brad A Myers, and Duen Horng Chau. 2006. A Linguistic Analysis of How People Describe Software Problems. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 127–134.

[35] Rrezarta Krasniqi and Hyunsook Do. 2023. A multi-model framework for semantically enhancing detection of quality-related bug report descriptions. *Empirical Software Engineering* 28, 2 (2023), 1–62.

[36] Niraj Kumar and Premkumar Devanbu. 2016. OntoCat: Automatically categorizing knowledge in API Documentation. *arXiv preprint arXiv:1607.07602* (2016).

[37] Zijad Kurtanović and Walid Maalej. 2017. Automatically classifying functional and non-functional requirements using supervised machine learning. In *Proceedings of the IEEE 25th International Requirements Engineering Conference (RE)*. Ieee, 490–495.

[38] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 128–140.

[39] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Jiazhan Xie, Huanjun Xu, and Yanjun Yang. 2022. How to formulate specific how-to questions in software development?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 306–318.

[40] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[41] Christopher D. Manning and Hinrich Schütze. 2009. An Empirical Study of Naive Bayes Classifier. In *Proceedings of the 14th Conference on Empirical Methods in Natural Language Processing*. 41–50.

[42] Arthur Marques and Gail C Murphy. 2022. Evaluating the Use of Semantics for Identifying Task-relevant Textual Information. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22)*. IEEE, 240–251.

[43] Arthur Marques, Giovanni Viviani, and Gail C Murphy. 2021. Assessing semantic frames to support program comprehension activities. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, 13–24.

[44] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. 2016. A Survey of App Store Analysis for Software Engineering. *IEEE Transactions on Software Engineering* (to appear), 99 (2016).

[45] Scott Menard. 2010. Logistic Regression: Why We Use It and How It Works. *SAS Global Forum* (2010).

[46] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. 2013. *Qualitative Data Analysis: A Methods Sourcebook* (3rd ed.). SAGE Publications, Inc.

[47] Mettew BA Miles, Michael Huberman, and Johny Saldana. 2014. Qualitative Data Analysis, A Methods Sourcebook. Saeg Publication.

[48] MITRE Corporation. 2019. MITRE Annotation Toolkit (MAT). https://github.com/mitre/mitre-annotation. Accessed: March 29, 2023.

[49] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17 (2012), 703–737.

[50] Aron Monszpart, Daniel Papp, and Tamas Gyimothy. 2019. Extraction of Technical Terminologies from a Medical Corpus. In *Proceedings of the 20th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing 2019)*. Springer, 357–369.

[51] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th international conference on software engineering (ICSE'12)*. IEEE, 815–825.

[52] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the IEEE international conference on software maintenance and evolution (ICSME'15)*. IEEE, 281–290.

[53] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019), 1499–1537.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[55] Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. 2015. Discovering Information Explaining API Types Using Text Classification. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*. 869–879.

[56] Livia Polanyi. 2003. 14 The Linguistic Structure of Discourse. *The Handbook of Discourse Analysis* 18 (2003), 265.

[57] Livia Polanyi. 2003. *The Handbook of Discourse Analysis*. Vol. 18. Wiley-Blackwell, Chapter The Linguistic Structure of Discourse, 265.

[58] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. 2019. Categorizing the content of github readme files. *Empirical Software Engineering* 24 (2019), 1296–1327.

[59] J Ross Quinlan. 1988. A comparative study of decision tree ID3 and C4.5. In *Proceedings of the 5th International Workshop on Machine Learning*. Morgan Kaufmann, 47–64.

[60] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of Systems and Software* 181 (2021), 111047.

[61] Martin P Robillard and Yam B Chhetri. 2015. Recommending reference API documentation. *Empirical Software Engineering* 20 (2015), 1558–1586.

[62] Timothy Roscoe, Alastair Martin, Bing Shao, and Kevin Amorosa. 2015. Parfait: Designing a Scalable Bug Checker. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Berkeley, CA, USA, 163–174. https://doi.org/10.1145/2830961.2830973

[63] Harry M. Sneed. 2001. Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment. In *Proceedings of the 9th IEEE Workshop on Program Comprehension (IWPC)*. 167–175. https://doi.org/10.1109/WPC.2001.921728

[64] Harry M. Sneed and Katalin Erdös. 1996. Extracting Business Rules from Source Code. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC)*. Berlin, Germany, 240–247. https://doi.org/10.1109/WPC.1996.501138

[65] Riad Sonbol, Ghaida Rebdawi, and Nada Ghneim. 2022. The use of nlp-based text representation techniques to support requirement engineering tasks: A systematic mapping review. *IEEE Access* (2022).

[66] Yang Song and Oscar Chaparro. 2020. BEE: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 1551–1555.

[67] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (Android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. 344–356.

[68] Anna-Brita Stenström and Karin Aijmer. 2004. Discourse patterns in spoken and written corpora. *Discourse patterns in spoken and written corpora* (2004), 1–287.

[69] A. Sureka and P. Jalote. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *Proceedings of the Asia Pacific Software Engineering Conference (ASPEC'10)*. 366–374.

[70] Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Automatic Defect Categorization. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*. 205–214.

[71] Christoph Treude, Fernando Figueira Filho, and Uirá Kulesza. 2015. Summarizing and Measuring Development Activity. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*. 625–636.

[72] Sida Wang and Christopher D Manning. 2012. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 90–94.

[73] Xinyu Wang, Jianling Sun, Xiaohu Yang, Zhijun He, and Srini Maddineni. 2004. Business Rules Extraction from Large Legacy Systems. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR)*. 249–258. https://doi.org/10.1109/CSMR.2004.1281426

[74] Karl E. Wiegers and Joy Beatty. 2013. *Software Requirements* (third ed.). Microsoft Press, Redmond, WA.

[75] Graham C. Witt. 2012. *Writing Effective Business Rules : A Practical Method*. Morgan Kaufmann, Waltham, MA.

[76] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. 2012. Automated Extraction of Security Policies from Natural-Language Software Documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[77] Bowen Xu, David Lo, Xin Xia, Ashish Sureka, and Shanping Li. 2015. EFSPredictor: Predicting Configuration Bugs with Ensemble Feature Selection. In *Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC'15)*. 206–213.

[78] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. 2020. Managing Data Constraints in Database-Backed Web Applications. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1098–1109. https://doi.org/10.1145/3377811.3380375

[79] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. 1359–1371.

[80] Meng Zhang, Yang Liu, Yong Cheng, and Maosong Sun. 2021. Exploring the Syntax-Semantics Interface in Neural Machine Translation with Dependency Parsing Constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 1781–1796.

[81] Bo Zhou, Xin Xia, David Lo, Cong Tian, and Xinyu Wang. 2014. Towards More Accurate Content Categorization of API Discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC'14)*. 95–105.

[82] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28, 3 (2016), 150–176.