

AUTOMATED EXTRACTION OF DATA CONSTRAINTS  
FROM SOFTWARE DOCUMENTATION

by

Ying Zhou

APPROVED BY SUPERVISORY COMMITTEE:

---

Andrian Marcus, Chair

---

Lawrence Chung

---

Shiyi Wei

Copyright © 2023

Ying Zhou

All rights reserved

AUTOMATED EXTRACTION OF DATA CONSTRAINTS  
FROM SOFTWARE DOCUMENTATION

by

YING ZHOU, BS

THESIS

Presented to the Faculty of  
The University of Texas at Dallas  
in Partial Fulfillment  
of the Requirements  
for the Degree of

MASTER OF SCIENCE IN  
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2023

## ACKNOWLEDGMENTS

I would like to express my sincere thanks to:

- My parents, sibling, and Zizhen who have provided love and support, serving as a constant source of encouragement throughout my studies.
- My advisor, Prof. Andrian Marcus, for not only introducing me to the world of academic research but also generously shared his vast knowledge, provided invaluable guidance, and offered unwavering support. His dedication to the future careers of his students is evident as he consistently strives to assist them in attaining their goals and accomplishments. I take pride in being his student and having the opportunity to learn from him.
- Prof. Shiyi Wei for his collaborative support, which has played a pivotal role in the realization of this thesis.
- Many professors who have provided invaluable guidance in nurturing my growth as a student, including Prof. Andrian Marcus, Prof. Shiyi Wei, Prof. Lawrence Chung, Prof. Benjamin A Raichel, Prof. Dan Ioan Moldovan, Prof. Anjum Chida, Prof. Min Chen, and others.
- My collaborators, Oscar Chaparro, Juan Manuel Florez, Jonathan Perry, Miao Miao, and Vlad Virsan, for their great contributions and active involvement throughout the entirety of my research.

This research was supported in part by grants from the US National Science Foundation: CCF-1910976 and CCF-1955837.

July 2023

AUTOMATED EXTRACTION OF DATA CONSTRAINTS  
FROM SOFTWARE DOCUMENTATION

Ying Zhou, MSCS  
The University of Texas at Dallas, 2023

Supervising Professor: Andrian Marcus, Chair

Data constraints encompass crucial business rules that specify the values allowed or required for the data utilized within 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.

This thesis contribute to the existing knowledge by studying 548 data constraints described in the documentation of nine systems. We identified and documented 15 linguistic discourse patterns employed by stakeholders to describe data constraints in natural language. In a comprehensive extensive study, we explore the use of the discourse patterns we discovered, along 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).

The discoveries made in this thesis represent a significant advancement in the automated identification and extraction of data constraints from natural language text, which in turn is essential for enabling the automation of traceability to code and facilitating test generation associated with these constraints.

## TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . .	iv
ABSTRACT . . . . .	v
LIST OF FIGURES . . . . .	viii
LIST OF TABLES . . . . .	ix
CHAPTER 1 INTRODUCTION . . . . .	1
1.1 Problem Description and Motivation . . . . .	1
1.2 Research Goal . . . . .	2
CHAPTER 2 BACKGROUND ON DATA CONSTRAINTS . . . . .	3
CHAPTER 3 HOW ARE DATA CONSTRAINTS DESCRIBED IN NATURAL LANGUAGE? . . . . .	7
3.1 Data Collection . . . . .	7
3.2 Discourse Pattern Coding Protocol . . . . .	11
3.2.1 Results - Discourse Patterns . . . . .	12
3.3 Threats to Validity . . . . .	14
3.4 Acknowledgments . . . . .	16
CHAPTER 4 DATA CONSTRAINTS CLASSIFICATION . . . . .	17
4.1 Data . . . . .	17
4.2 Labeling negative sentences . . . . .	17
4.3 Machine Learners . . . . .	21
4.4 Features . . . . .	22
4.5 Training, Testing, and Measures . . . . .	23
4.6 SMOTE . . . . .	24
4.7 Evaluation Metrics . . . . .	24
4.8 Results . . . . .	25
4.8.1 The Effect of Data Balancing . . . . .	26
4.8.2 The Effect of Feature Selection . . . . .	27
4.8.3 Analysis of False Negatives . . . . .	29
4.9 Threats to Validity . . . . .	31

4.10 Acknowledgments . . . . .	32
CHAPTER 5 RELATED WORK . . . . .	33
CHAPTER 6 CONCLUSION AND FUTURE WORK . . . . .	37
APPENDIX DISCOURSE PATTERNS CATALOG . . . . .	38
REFERENCES . . . . .	41
BIOGRAPHICAL SKETCH . . . . .	53
CURRICULUM VITAE	

## LIST OF FIGURES

2.1	Examples of data constraints (highlighted) . . . . .	4
3.1	Agreements and disagreements vs. final label . . . . .	9
4.1	Examples of utilizing "conj" to extract fragments . . . . .	19
4.2	F1 scores with different SMOTE ratios on 10CV eval. . . . .	25
4.3	F1 scores with different SMOTE ratios on CSV eval. . . . .	26
4.4	F1 scores using TF and AF feature sets . . . . .	28



## LIST OF TABLES

2.1	CIP Catalog (Florez et al., 2022a) Excerpt. . . . .	6
3.1	Systems, sentences, and constraints. . . . .	8
3.2	Three most commonly used discourse patterns. . . . .	13
3.3	Distribution of DPs and constraint types. . . . .	15
4.1	Best configurations for the three best learners. . . . .	30
4.2	False negatives vs. DP and constraint type. . . . .	31
A.1	Catalog of 15 discourse patterns . . . . .	38
A.2	Catalog of 15 discourse patterns(continued) . . . . .	39
A.3	Catalog of 15 discourse patterns(continued) . . . . .	40

# CHAPTER 1

## INTRODUCTION

### 1.1 Problem Description and Motivation

Business rules constitute a critical component of software systems, as they provide a well-defined and concise set of directives that govern the system's behavior and operations. Such rules specify the policies, procedures, and constraints that regulate the software system's behavior, thus ensuring its reliable and predictable functioning. *Data constraints* are implemented as rules or restrictions on the given data, specifying which values are permissible or required to meet specific requirements or standards (Brooks and Weber, 2011; Wiegers and Beatty, 2013a; Ross, 2012). Typically, such requirements are expressed in natural language.

Recent research suggests that keeping track of data constraints in software systems is a costly process, particularly as the systems become more complex with a greater number of data elements and intricate interrelationships between them (Mirzaie et al., 2019). In addition, the task is further complicated by the fact that software systems are constantly evolving and require regular updates to reflect those changes (Cohen et al., 2018). In this context, we argue that an automated approach that can extract data constraints from software documents is necessary, not only to enable traceability of these data constraints, but also to facilitate the automatic generation of test cases, among others.

While previous research has applied machine learning and natural language processing techniques to text classification problems, the automatic identification of data constraints in text has yet to be explored. Bilge et al. (Bilge and Ozyurt, 2019) demonstrated that incorporating text properties, such as n-grams or part of speech, as features in machine learning models can result in improved accuracy in text classification tasks such as sentiment analysis. In addition, previous research showed that for classifying elements of bug reports, more complex linguistic features such as discourse patterns are more effective than simpler

text properties like n-grams or part of speech (Chaparro et al., 2017). With that in mind, the implementation of an automated system that utilizes linguistic features, such as the discourse pattern, becomes crucial for ensuring the traceability of data constraints within software systems.

## 1.2 Research Goal

Our long-term research aims to achieve end-to-end automated fine-grained constraint traceability in software systems. To achieve this goal, our research will initially focus on the automated extraction of data constraints, which is the main focus of the thesis. Specifically, the main goals of the research supporting this thesis are:

- *G1: Collecting labeled data constraints.* This involves determining whether a sentence fragment represents a data constraint and identifying its operands and type, as defined in previous research studies (Florez et al., 2022b). To start, we use a set of 299 constraints from 15 systems that were previously collected (Florez, 2019) (defined in Chapter 3).
- *G2: Extracting linguistic features.* We extract linguistic features from the collected data constraints, such as n-grams, part-of-speech tags, and discourse patterns. (defined in Chapter 4)
- *G3: Automatic classification of data constraints.* We build a constraint predictor using different machine learning models, such as decision trees, random forests, support vector machines, neural networks, and ensemble models. We select the best model based on common evaluation metrics, specifically precision, recall, and F1 score. (defined in Chapter 4)

## CHAPTER 2

### BACKGROUND ON DATA CONSTRAINTS

This chapter presents important terminology, definitions and application regarding data constraints with a set of examples. Data constraints refer to a set of rules or limitations imposed on data, specifying the acceptable values that data must conform to, ensuring that it aligns with the predefined criteria and remains valid. Figure 2.1 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 Chapter 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*. As an illustration, the statement `The target file name is identical to the source file name.` represents a data constraint that enforces the requirement that the *target file name* should be the same as the *source file name*, with *value-comparison* as the type of constraint, and *target file name* and *source file name* as the operands involved. Likewise, for the constraint `set the frequency axis to log mode` in Figure 2.1 (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.* (Florez et al., 2022b) classified data constraints into four *types* (`binary-value`, `concrete-value`, `value-comparison`, and `categorical-value`). These constraint types

**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 2.1: Examples of data constraints (highlighted).

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  $C$ ). While equality operators apply to values of all types, relational operators do not apply to all types of values (*e.g.*, binary or categorical). 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.* (Florez et al., 2022a), 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  $V$ .

In their empirical study, Florez *et al.* (Florez et al., 2022b) defined 30 constraint implementation patterns (CIPs) based on the implementation of data constraints in Java. The definition of implementing a constraint involves two components: *an enforcing statement* and *data definition statements*. The study classified each enforcement statement based on the type of code construct it appeared in (*e.g.*, expression, if statement, return statement, etc.) and the number of operands it utilized, which results in the identification of 30 CIPs,

Table 2.1: CIP Catalog (Florez et al., 2022a) Excerpt.

---

<p><b>CIP name:</b> binary-comparison.</p> <p><b>Description:</b> Two values are compared using an operator such as <i>equals</i>, <i>does not equal</i>, <i>greater than</i>, <i>etc.</i></p> <p><b>Statement type:</b> Expression.</p> <p><b>Parts:</b> <math>\{variable1, op \in \{&gt;, \geq, &lt;, \leq, =, \neq\}, variable2\}</math></p> <p><b>Example:</b> <i>Instance:</i> <code>if(maxFreq &gt; wave.getNyquist())</code>  <i>Parts:</i> <math>\{\text{maxFreq}, &gt;, \text{wave.getNyquist()}\}</math>.</p>
---

---

<p><b>CIP name:</b> if-chain.</p> <p><b>Description:</b> A chain of ifs is used like a switch on a field, checking against the possible values of the variable.</p> <p><b>Statement type:</b> If statement.</p> <p><b>Parts:</b> <math>\{variable\}</math></p> <p><b>Example:</b> <i>Instance:</i> <code>if(onset == EMERGENT) {...} else if(onset == IMPULSIVE) {...} else if ...</code>  <i>Parts:</i> <math>\{\text{onset}\}</math>.</p>
---

---

with a line-of-code level of granularity. Furthermore, Florez *et al.* (Florez et al., 2022b) identified the most commonly used CIPs for implementing constraints of each constraint type. For instance, when it comes to constraints involving dual value comparison, the boolean property CIP was found to be the most frequently used, followed by the null check CIP as the next commonly employed pattern.

Previous work (Florez et al., 2022) has leveraged manually extracted information, including *constraint types*, *operands*, *CIP* to automatically locate the method and specific line of code that enforces a given data constraint. This approach serves as a motivation for our research, where we aim to automatically identify data constraints and extract their constituent elements.

## CHAPTER 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

Table 3.1 summarizes the data we collected (and discarded along the way). We started with a set of 299 constraints from 15 systems, which were collected in previous research (Florez et al., 2022b). 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 3.1 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 2.1, using the MITRE Annotation Toolkit (*a.k.a.* MAT) version 3.3 (MITRE Corporation, 2019). 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 3.1 column *Labeled/+ sent.*).



Table 3.1: Systems, sentences, and constraints.

System	Labeled			Rep-pack		Paper	
	+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		
<b>Total</b>	<b>1,502 (2,338)</b>	<b>14,659</b>	<b>2,884</b>	<b>921</b>	<b>1,011</b>	<b>475 (548)</b>	<b>12,236 (36,839)</b>

We labeled 14,659 sentences as *negative*, which did not contain any data constraints (Table 3.1 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 3.1 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

Final label <i>ignore</i>   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 <i>negative</i>   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 <i>not explicit</i>   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 <i>explicit</i>   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 3.1: Agreements and disagreements vs. final label

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.

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

## 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 (Florez et al., 2022b), we used the following process for labeling.

Using the aforementioned MAT software (MITRE Corporation, 2019), 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 Chapter 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 3.1 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 (Florez et al., 2022b), we added more labels. All constraints were reviewed by an author and re-categorized as *explicit*, *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 (Stenström and Aijmer, 2004; Polanyi, 2003a) of the 548 labeled constraints, based on open coding (Miles et al., 2014). 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.

### 3.2.1 Results - Discourse Patterns

The first and last columns in Table 3.3 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 3.2.

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*.

Table 3.2: Three most commonly used discourse patterns.

---

<p><b>Pattern code:</b> NP_BE_BINARY_VALUE</p> <p><b>Description:</b> A data element is assigned a binary value (e.g., <i>true/false, on/off, out/inout...</i>) by using an auxiliary verb.</p> <p><b>Rule:</b> [data] [be] [binary-value]</p> <p>[data] ∈ {NP}</p> <p>[be] ∈ {<i>be, allow, remain, show</i>}</p> <p>[binary-value] ∈ {JJ, VVN, \$binary-word}</p> <p>[\$binary-word] ∈ {<i>true/false, on/off, out/inout...</i>}</p> <p><b>Example:</b>[Background mode] [is] [on] (from jedit-5.6)</p>
---

---

<p><b>Pattern code:</b> NP_BE_VALUE</p> <p><b>Description:</b> A data element is assigned a concrete value by using an auxiliary verb.</p> <p><b>Rule:</b> [data] [be] [value]</p> <p>[data] ∈ {NP}</p> <p>[be] ∈ {<i>be, allow, remain, show</i>}</p> <p>[value] ∈ {NP, CD}</p> <p><b>Example:</b>[Default maxwait] [is] [3 minutes] (from apache-ant-1.10.6)</p>
--

---

<p><b>Pattern code:</b> NP_CD_TO_CD</p> <p><b>Description:</b> 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., <i>from...to, between...and</i>).</p> <p><b>Rule:</b> [data] [num] [to] [num]   [data] [from] [num] [to] [num]</p> <p>[data] ∈ {NP}</p> <p>[from] ∈ {<i>from, between</i>}</p> <p>[num] ∈ {CD}</p> <p>[to] ∈ {<i>to, and</i>}</p> <p><b>Example1:</b> [South latitude extent] [-90] [to] [90] (from swarm-2.8.11)</p> <p><b>Example2:</b> [hour] [from] [00] [to] [23] (from joda-time-2.10.3)</p>
--

---

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 i= 0 and port number j= 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 3.2, 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; 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 3.3 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 (see Chapter 4).

### **3.3 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

Table 3.3: 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

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.



### 3.4 Acknowledgments

I would like to express my sincere gratitude to the following collaborators for their valuable contributions and support throughout the completion of this chapter:

- **Juan Manuel Florez:** Juan shared a collection of 299 data constraints extracted from 15 systems. He encompassed the definition of data constraint types, operands, and implementation patterns, played a pivotal role in guiding our efforts to gather comprehensive and detailed information about data constraints.
- **Jonathan Perry:** Jonathan made significant contributions to data collection and management.
- **Miao Miao and Vlad Virsan:** They reviewed documents labeled by students coders, which greatly enhanced the accuracy and reliability of the gathered data.

## CHAPTER 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 ???. 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.

#### 4.2 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 (Honnibal and Montani, 2018) 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. (Zhang et al., 2021) used it to extract dependency patterns of syntax-semantics constraints from source and target languages, while Monszpart et al. (Monszpart et al., 2019) 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 (Roscoe et al., 2015), 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.
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

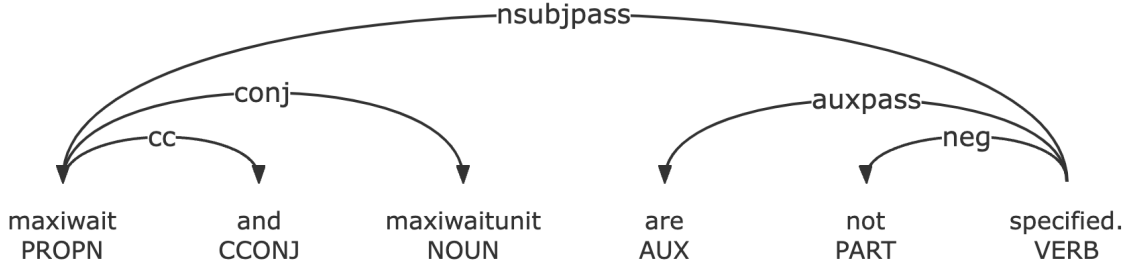


Figure 4.1: Examples of utilizing "conj" to extract fragments

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 4.1. 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 Chapter 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 `$word-list` occurs in the sentence, then we will assign the `value-comparison` type to the fragment.  
`$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`.

### 4.3 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 (Menard, 2010), Naive Bayes Classifier (Manning and Schütze, 2009) , linear Support Vector Machine (Hsu and Lin, 2003), non-linear Support Vector Machine (Hsu and Lin, 2003), and Decision Tree (Quinlan, 1988).

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* (Fan et al., 2008). The Naive Bayes Classifier uses a smoothing parameter  $\alpha$  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 transform 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.

#### 4.4 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* (Wang and Manning, 2012) 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.

## 4.5 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 (Dougherty, 2012). 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 (Pedregosa et al., 2011) creates a grid of all possible combinations of predefined 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 4.7). 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 (Chawla et al., 2002).

## 4.6 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) (Chawla et al., 2002). SMOTE is a well-known over-sampling 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. (Chawla et al., 2002) showed that SMOTE consistently outperformed the other techniques in terms of both sensitivity and specificity. In addition, Fernández et al. (Fernández et al., 2018) 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.

## 4.7 Evaluation Metrics

In order to evaluate effectiveness of the classifiers, we employ standard metrics, specifically precision, recall, and F1 score (Dougherty, 2012). 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

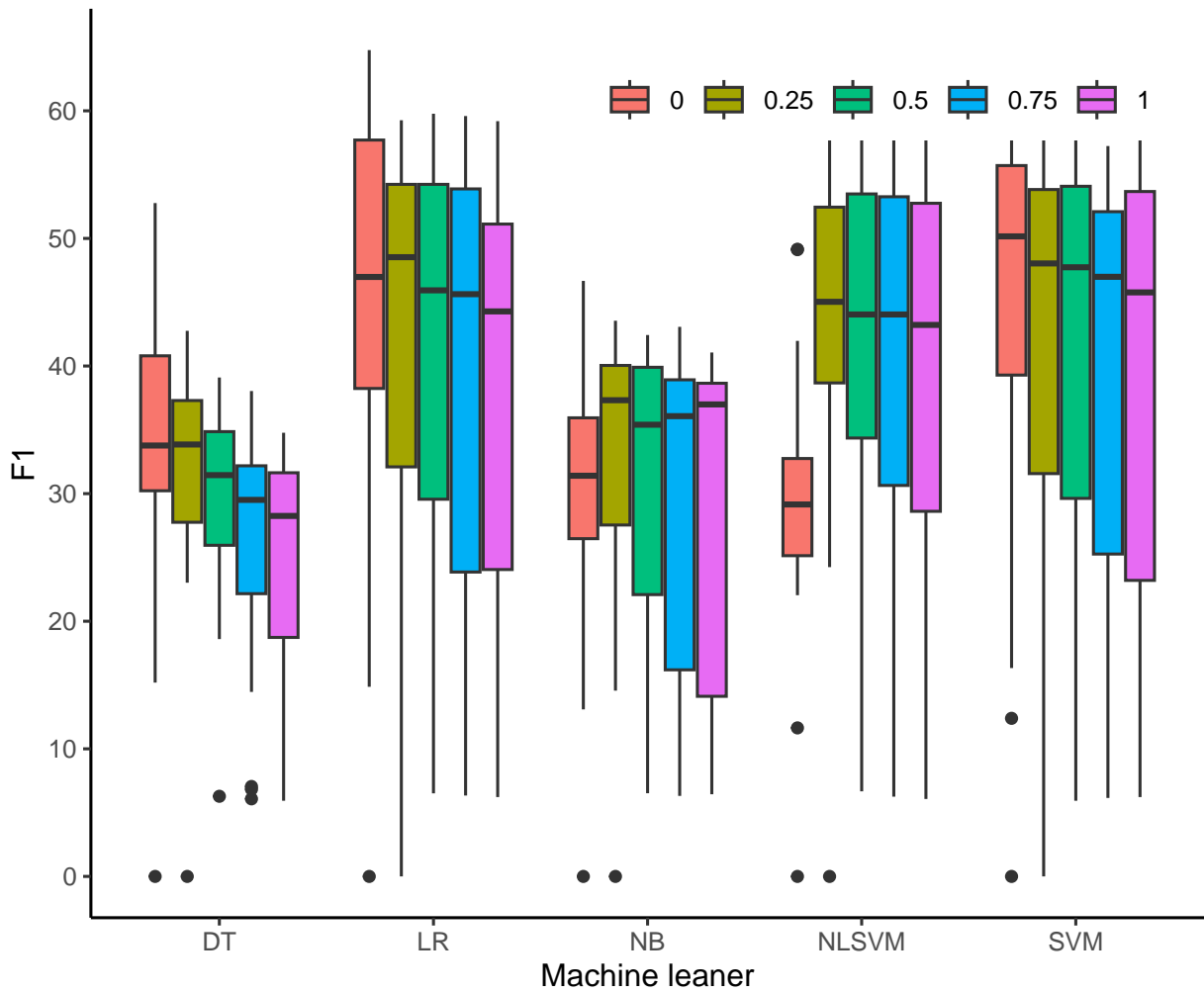


Figure 4.2: F1 scores with different SMOTE ratios on 10CV eval.

and recall (*i.e.*,  $F1 = 2 * (precision * recall) / (precision + recall)$ ) and we use it as the main performance measure.

## 4.8 Results

We evaluated each of the five classifiers with all combinations of the five features, using all five SMOTE ratios. For each classifier we evaluate 155 different configurations, using 10-fold cross validation and cross-system validation.

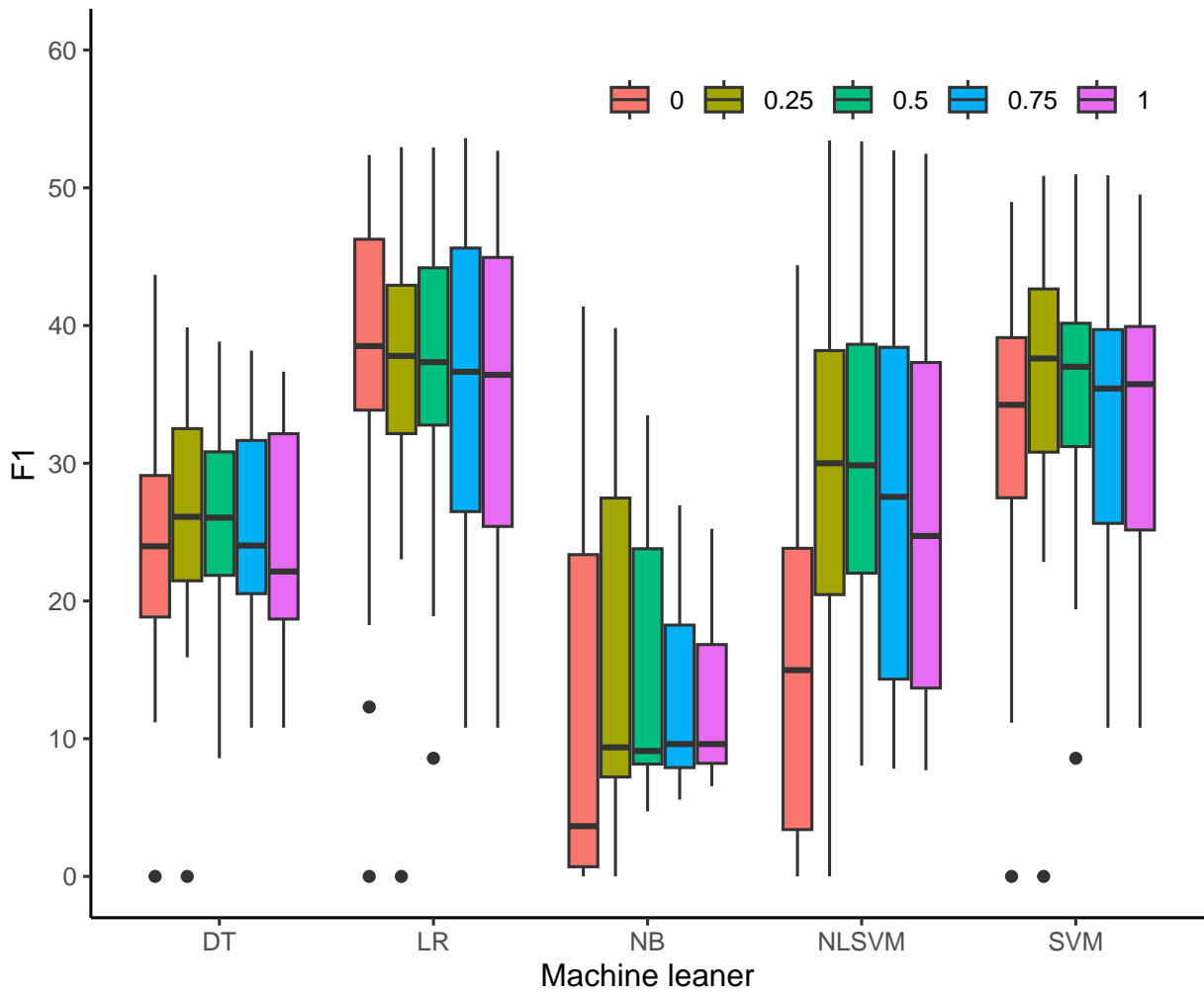


Figure 4.3: F1 scores with different SMOTE ratios on CSV eval.

#### 4.8.1 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 4.2 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 4.3 shows the same information for the cross-system validation. We observe the same phenomenon as above for the Naive Bayes and non-linear SVM, but for the cross-system validation, the Naive Bayes is less sensitive to data balancing.

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.

#### 4.8.2 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 4.4 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

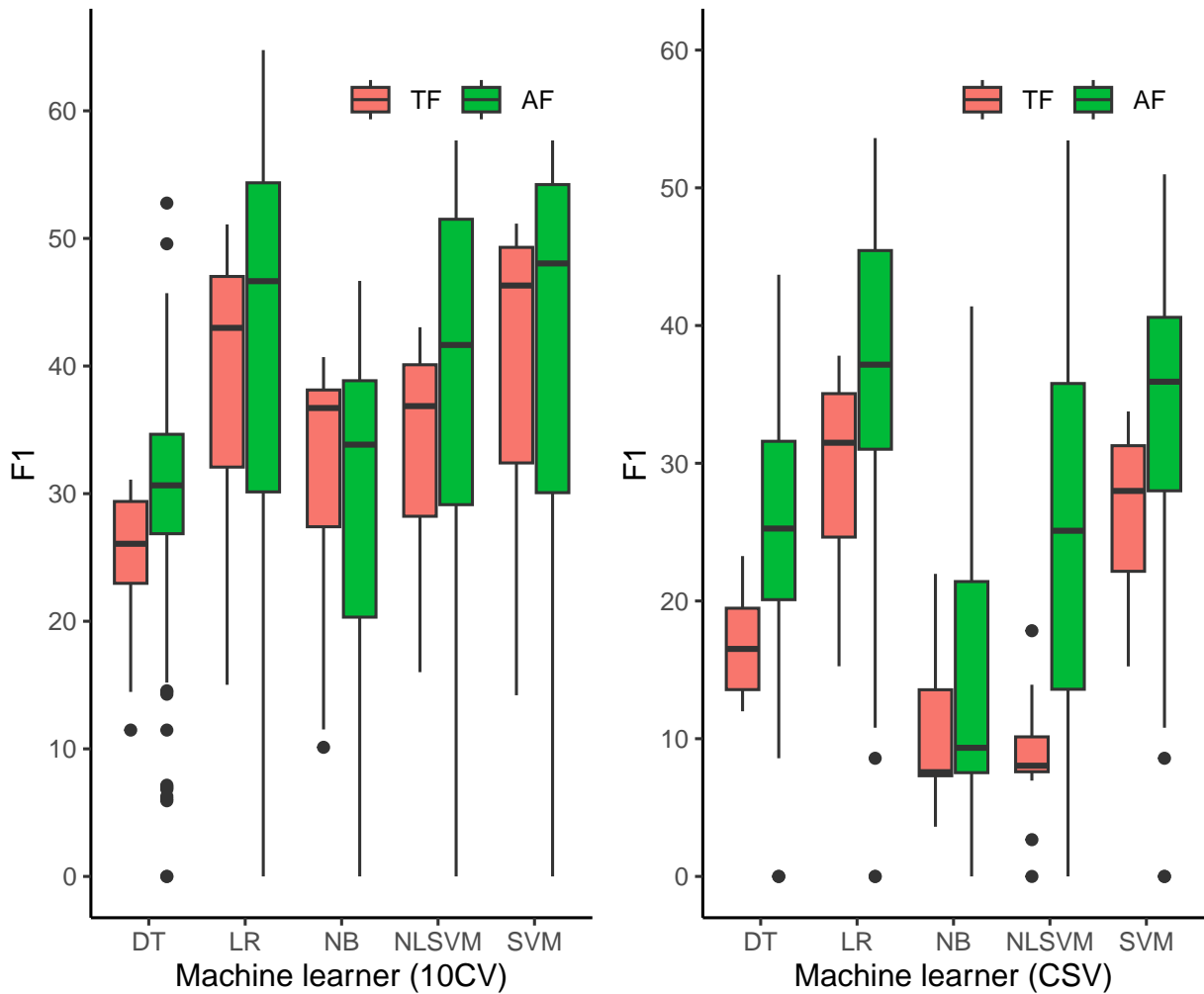


Figure 4.4: F1 scores using TF and AF feature sets

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 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 following 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 4.1 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 4.1, 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

### 4.8.3 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

Table 4.1: 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	

configurations for the three best learners (from Table 4.1) 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 4.2 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 4.2, 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.

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 4.2).

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).

Table 4.2: 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%

## 4.9 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.

#### 4.10 Acknowledgments

I would also like to extend my appreciation to the following collaborators throughout the completion of this chapter:

- **Oscar Chaparro:** Oscar shared open-source code to aid in the development of classifiers for identifying descriptions of data constraints.
- **Miao Miao:** Miao provided assistance in the implementation of discourse pattern matchers and the creation of heuristics for extracting operands associated with data constraints.

## CHAPTER 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 (Witt, 2012; Wiegers and Beatty, 2013b; Florez et al., 2022a; Yang et al., 2020), 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 (Florez, 2019) and such constraints can be traced automatically to methods and lines of code (Florez et al., 2022b). Our work is motivated by this prior work and fill in the knowledge gap on how to automatically identify data constraints. 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 (Yang et al., 2020). 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) (Hatano et al., 2016; Cosentino et al., 2013; Huang et al., 1996; Sneed and Erdös, 1996; Wang et al., 2004; Cosentino et al., 2012; Sneed, 2001). 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.

**Analysis 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 (Ko et al., 2006). The results showed that distinct types of words, such as verbs or prepositions, play a particular role in conveying the problem described in issues titles, as well as the context where it occurred. Sureka *et al.* executed a similar analysis on issue titles to find vocabulary patterns useful for predicting the issue severity (Sureka and Jalote, 2010). 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 (Chaparro et al., 2017; Song and Chaparro, 2020).

Xiao *et al.* described linguistic patterns to identify access control policies in requirement documents (Xiao et al., 2012).

Di Sorbo *et al.* identified linguistic patterns in sentences from mobile app reviews, developer email threads, and issue titles to perform content classification (Di Sorbo et al., 2019). Via textual analysis, other work has identified types of information in API documents via textual analysis (Maalej and Robillard, 2013), studied API privacy policy information, and identified information relevant to development activity summaries (Treude et al., 2015). Liu *et al.* (Liu et al., 2022) 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 (Polanyi, 2003b; Miles et al., 2013), 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) (Beyer et al., 2018; Catolino et al., 2019; Panichella et al., 2015; Di Sorbo et al., 2015; Kallis et al., 2019; Guo et al., 2010; Zhou et al., 2016; Antoniol et al., 2008; Thung et al., 2012; Xu et al., 2015; Martin et al., 2016; Zhou et al., 2014; Petrosyan et al., 2015; Guzman et al., 2015) and sentences or fragments in such documents (Abad et al., 2017; Baker et al., 2019; Kurtanović and Maalej, 2017; Sonbol et al., 2022; Dalpiaz et al., 2019; Krasniqi and Do, 2023; Chaparro et al., 2017; Song and Chaparro, 2020; Rani et al., 2021; Pandita et al., 2012; Pascarella

et al., 2019; Zhai et al., 2020; Robillard and Chhetri, 2015; Kumar and Devanbu, 2016; Monperrus et al., 2012; Jiang et al., 2017; Di Sorbo et al., 2019; Prana et al., 2019; Marques and Murphy, 2022; Marques et al., 2021; Chaparro et al., 2019; Arya et al., 2019), for diverse purposes, including assessing the code comment (Zhai et al., 2020; Pascarella et al., 2019) and bug report quality (Chaparro et al., 2017, 2019), prioritizing requirements (Abad et al., 2017; Baker et al., 2019; Kurtanović and Maalej, 2017; Sonbol et al., 2022; Dalpiaz et al., 2019), reproducing reported bugs (Fazzini et al., 2018; Liu et al., 2020), and recommending APIs (Robillard and Chhetri, 2015; Jiang et al., 2017).

Most approaches for fine-grained textual classification have relied on canonical supervised learning models (da Silva Maldonado et al., 2017; Abad et al., 2017; Baker et al., 2019; Kurtanović and Maalej, 2017; Sonbol et al., 2022; Dalpiaz et al., 2019; Krasniqi and Do, 2023; Chaparro et al., 2017; Song and Chaparro, 2020; Rani et al., 2021; Pascarella et al., 2019; Zhai et al., 2020; Robillard and Chhetri, 2015; Kumar and Devanbu, 2016; Monperrus et al., 2012; Jiang et al., 2017; Di Sorbo et al., 2019; Prana et al., 2019), 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 (Marques and Murphy, 2022; Marques et al., 2021; Chaparro et al., 2019; Liu et al., 2020; Chatterjee et al., 2021), such as Word2Vec, LSTMs, or BERT, are scarce (so are unsupervised approaches (Jiang et al., 2017; Abad et al., 2017)) 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) (Abad et al., 2017) for classification, and dependency parsing of text or heuristics (Pandita et al., 2012; Chaparro et al., 2017, 2019; Fazzini et al., 2018, 2022; Song et al., 2022).

Features used by canonical learning models include heuristics (*e.g.*, keywords, regular-expression-based features, or word count) (Chaparro et al., 2017; Krasniqi and Do, 2023;

Pascarella et al., 2019), textual properties (*e.g.*, nouns, n-grams, part of speech, sentiment, and term/document frequency) (Arya et al., 2019; da Silva Maldonado et al., 2017; Rani et al., 2021; Abad et al., 2017; Baker et al., 2019; Chaparro et al., 2017; Song and Chaparro, 2020; Kurtanović and Maalej, 2017; Krasniqi and Do, 2023; Zhai et al., 2020), and discourse/linguistic patterns (Chaparro et al., 2017; Abad et al., 2017; Dalpiaz et al., 2019).

Sentences and fragments often represent information valuable for developers, including conditions, facts, restrictions, arguments, explanations, and other elements found in requirements (Abad et al., 2017; Baker et al., 2019; Kurtanović and Maalej, 2017; Sonbol et al., 2022; Dalpiaz et al., 2019), issues and bug reports (Arya et al., 2019; Krasniqi and Do, 2023; Chaparro et al., 2017; Song and Chaparro, 2020; Chaparro et al., 2019; Fazzini et al., 2018, 2022; Liu et al., 2020; Song et al., 2022), code comments (Rani et al., 2021; Pandita et al., 2012; Pascarella et al., 2019; da Silva Maldonado et al., 2017), API documentation (Zhai et al., 2020; Robillard and Chhetri, 2015; Kumar and Devanbu, 2016; Monperrus et al., 2012; Jiang et al., 2017), app user reviews (Di Sorbo et al., 2019), project README files (Prana et al., 2019), and programming forum posts (Marques and Murphy, 2022; Marques et al., 2021).

While none of this research focuses on identifying data constraints in software artifacts, Kiziltan *et al.* (Kiziltan et al., 2016) 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.

## CHAPTER 6

### CONCLUSION AND FUTURE WORK

Through an analysis of 548 explicit data constraints on the natural language descriptions in the documentation of 9 systems, our findings revealed that these data constraints are expressed using a total of 15 distinct linguistic discourse patterns. We utilized these discourse patterns to assist us in automatically identifying the operands and constraint type of explicit data constraints that are described in natural language.

We conducted a comprehensive empirical study to evaluate 155 different configurations for each of five learners for classifying data constraint descriptions. The results demonstrated that by utilizing the identified discourse patterns, the constraint operands, their type, their description, and the sentence containing the constraints as features for training, leads to best classification results (70.87% precision and 59.73% recall, 64.76% F1). We also discovered correlations between the constraint type, the discourse patterns, and the ratio of positive to negative training data, which enabled to establish guidelines for future collection efforts of training data, with the expectation of enhancing the classification performance.

Taking these findings and conclusions into consideration, they represent a significant step towards the automatic identification and extraction of data constraints from natural language text. Apart from the mentioned future work aimed at enhancing the classification, a crucial next phase involves achieving the extraction of constraints to be incorporated into a pipeline alongside traceability or test generation tools.

## APPENDIX

### DISCOURSE PATTERNS CATALOG

Table A.1: Catalog of 15 discourse patterns

---

<p><b>Pattern code:</b> SET_NP_TO_BINARY_VALUE</p> <p><b>Description:</b> A data element is assigned a binary value (e.g., <i>true/false, on/off, out/inout...</i>) by using a phrase composed of an auxiliary verb and a preposition. The data element is positioned between the auxiliary verb and the preposition.</p> <p><b>Rule:</b> [set] [data] [to] [binary-value]   [data] [to] [binary-value] [set] ∈ {<i>set, define, mark</i>} [data] ∈ {NP} [to] ∈ {<i>to, as, =</i>} [binary-value] ∈ {JJ, VVN, \$binary-word} [\$binary-word] ∈ {<i>true/false, on/off, out/inout...</i>}</p> <p><b>Example:</b> [set] [the usefile attribute] [to] [false] (from apache-ant-1.10.6)</p>
<hr/> <p><b>Pattern code:</b> NP_SET_TO_BINARY_VALUE</p> <p><b>Description:</b> A data element is assigned a binary value (e.g., <i>true/false, on/off, out/inout...</i>) by using a phrase composed of an auxiliary verb and a preposition. The value assignment phrase is positioned after the data element.</p> <p><b>Rule:</b> [data] [set to] [binary-value]   [data] [be] [set to] [binary-value] [data] ∈ {NP} [be] ∈ {<i>is, are, can be</i>} [set to] ∈ {<i>set to, mark as, require to be</i>} [binary-value] ∈ {JJ, VVN, \$binary-word} [\$binary-word] ∈ {<i>true/false, on/off, out/inout...</i>}</p> <p><b>Example:</b> [ignoreContents] [is] [set to] [true] (from apache-ant-1.10.6)</p>
<hr/> <p><b>Pattern code:</b> CD_NP</p> <p><b>Description:</b> A data element is assigned a numeric value. The data element appears directly after the value.</p> <p><b>Rule:</b> [num] [data] [num] ∈ {CD} [data] ∈ {NP}</p> <p><b>Example:</b> [100] [edits] (from jedit-5.6pre0)</p>
<hr/> <p><b>Pattern code:</b> NP_CD</p> <p><b>Description:</b> A data element is assigned a numeric value. The value appears directly after a data element.</p> <p><b>Rule:</b> [data] [num] [data] ∈ {NP} [num] ∈ {CD}</p> <p><b>Example:</b> [the status code] [501] (from apache-httpcomponents-4.5.9)</p>

---

Table A.2: Catalog of 15 discourse patterns(continued)

---

**Pattern code:** NP\_EXIST  
**Description:** Check whether a data element exists by using the word “*exist*”.  
**Rule:** [data], exist  
[data] ∈ {NP}  
**Example:** [output file], exists (from apache-ant-1.10.6)

---

**Pattern code:** NP\_COMP\_NP  
**Description:** Compare the value of a data element with the value of another data element using the defined comparison words or symbols.  
**Rule:** [data] [compare to] [data]  
[data] ∈ {NP}  
[compare to] ∈ {j=, j≠, =, above, before, below, greater than, lower than, newer than, greater than or equal to, less than, more than, less than or equal to, at or before, equal to or earlier than, equal to or later than, equal to or greater than, equal to or less than, the same as...}  
**Example:** [signal] [above] [corner frequency] (from swarm-2.8.11)

---

**Pattern code:** SET\_NP\_TO\_VALUE  
**Description:** A data element is assigned a concrete value by using a phrase composed of an auxiliary verb and a preposition. The data element is positioned between the auxiliary verb and the preposition.  
**Rule:** [set] [data] [to] [value] | [data] [to] [value]  
[set] ∈ {set, define, mark}  
[data] ∈ {NP}  
[to] ∈ {to, as, =}  
[value] ∈ {NP, CD}  
**Example:** [set] [the frequency axis] [to] [log mode] (from swarm-2.8.11)

---

**Pattern code:** NP\_IN\_VALUESET  
**Description:** A categorical data element is assigned a finite set of values by using a preposition or a symbol.  
**Rule:** [data] [in] [valueset]  
[data] ∈ {NP}  
[in] ∈ {in, include, be, “:”, “(”, “)”, “-”}  
[valueset]: {[value]...[value]}  
[value] ∈ {NP, CD}  
**Example:** [Encoding considerations] [:] [”7bit”, ”8bit”, or ”binary”] (from apache-httpcomponents-4.5.9)

---

**Pattern code:** NP\_OF\_VALUE  
**Description:** A data element is assigned a concrete value by using an preposition “*of*”. The value appears after the data element.  
**Rule:** [data] of [value]  
[data] ∈ {NP}  
[value] ∈ {NP, CD}  
**Example:** [RSAM period] of [10 seconds] (from swarm-2.8.11)

---



Table A.3: Catalog of 15 discourse patterns(continued)

---

**Pattern code:** NP\_IF\_BINARY\_VALUE  
**Description:** Check whether the value of a data element equals a binary value (e.g., *true/false*, *on/off*, *out/inout...*) with the conjunction “*if*”. The conjunction “*if*” is positioned after the data element and before its binary value.  
**Rule:** [data], if [binary-value]  
[data] ∈ {NP}  
[binary-value] ∈ {JJ, VVN, \$binary-word}  
[\$binary-word] ∈ {*true/false*, *on/off*, *out/inout...*}  
**Example:** [”pages” field], if [set] (from jabref-5.0)

---

**Pattern code:** VALUE\_FOR\_NP  
**Description:** A data element is assigned a concrete value by using an preposition “*for*”. The value appears before a data element.  
**Rule:** [value] for [data]  
[value] ∈ {NP, CD}  
[data] ∈ {NP}  
**Example:** [NULL] for [the content type] (from apache-httpcomponents-4.5.9)

---

**Pattern code:** NP\_SET\_TO\_VALUE  
**Description:** A data element is assigned a concrete value by using a phrase composed of an auxiliary verb and a preposition. The value assignment phrase is positioned after the data element.  
**Rule:** [data] [set to] [value] | [data] [be] [set to] [value]  
[data] ∈ {NP}  
[be] ∈ {*is*, *are*, *can be*}  
[set to] ∈ {*set to*, *mark as*, *require to be*}  
[value] ∈ {NP, CD}  
**Example:** [the fileIndex attribute] [is] [set to] [”min”] (from log4j-2.13.3)

---

## REFERENCES

- Abad, Z. S. H., O. Karras, P. Ghazi, M. Glinz, G. Ruhe, and K. Schneider (2017). What works better? a study of classifying requirements. In *Proceedings of the IEEE 25th International Requirements Engineering Conference (RE)*, pp. 496–501. IEEE.
- Antoniol, G., K. Ayari, M. Di Penta, F. Khomh, and Y.-G. 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*, pp. 304–318.
- Arya, D., W. Wang, J. L. Guo, and J. 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)*, pp. 454–464. IEEE.
- Baker, C., L. Deng, S. Chakraborty, and J. 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)*, Volume 2, pp. 610–615. IEEE.
- Beyer, S., C. Macho, M. Pinzger, and M. Di Penta (2018). Automatically classifying posts into question categories on stack overflow. In *Proceedings of the 26th Conference on Program Comprehension*, pp. 211–221.
- Bilge, A. and F. Ozyurt (2019). Feature selection for sentiment analysis with n-grams. In *2019 5th International Symposium on Innovative Approaches in Scientific Studies and Education (ISIASE)*, pp. 74–79. IEEE.
- Brooks, A. and M. Weber (2011). An empirical study of data constraint implementations in java. *Software: Practice and Experience* 41(3), 253–289.

- Catolino, G., F. Palomba, A. Zaidman, and F. Ferrucci (2019). Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152, 165–181.
- Chaparro, O., C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyanyk, and V. 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)*, pp. 86–96.
- Chaparro, O., J. Liu, F. Zampetti, L. Moreno, M. DiPenta, A. Marcus, G. Bavota, and V. 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, New York, NY, USA, pp. 396–407. ACM.
- Chaparro, O., J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng (2017). Detecting missing information in bug descriptions. ESEC/FSE 2017, New York, NY, USA, pp. 396–407. Association for Computing Machinery.
- Chatterjee, P., K. Damevski, and L. Pollock (2021). Automatic extraction of opinion-based q&a from online developer chats. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1260–1272. IEEE.
- Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer (2002). Smote: Synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16, 321–357.
- Cohen, R. A., N. Kostyuk, and S. L. Stewart (2018). Data quality and record linkage costs in the national health interview survey. *Journal of Survey Statistics and Methodology* 6(1), 109–131.

- Cosentino, V., J. Cabot, P. Albert, P. Bauquel, and J. Perronnet (2012). A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. In A. Bikakis and A. Giurca (Eds.), *Rules on the Web: Research and Applications*, Lecture Notes in Computer Science, pp. 17–31. Springer, Berlin, Heidelberg.
- Cosentino, V., J. Cabot, P. Albert, P. Bauquel, and J. Perronnet (2013, October). Extracting business rules from COBOL: A model-based framework. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pp. 409–416.
- da Silva Maldonado, E., E. Shihab, and N. Tsantalis (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43(11), 1044–1062.
- Dalpiaz, F., D. Dell’Anna, F. B. Aydemir, and S. Çevikol (2019). Requirements classification with interpretable machine learning and dependency parsing. In *Proceedings of the IEEE 27th International Requirements Engineering Conference (RE’2019)*, pp. 142–152. IEEE.
- Di Sorbo, A., S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. 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)*, pp. 12–23.
- Di Sorbo, A., S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall (2019). Exploiting natural language structures in software informal documentation. *IEEE Transactions on Software Engineering* 47(8), 1587–1604.
- Dougherty, G. (2012). *Pattern recognition and classification: an introduction*. Springer Science & Business Media.

- Fan, R.-E., K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin (2008). Liblinear: A library for large linear classification. *The Journal of Machine Learning Research* 9, 1871–1874.
- Fazzini, M., K. P. Moran, C. Bernal-Cardenas, T. Wendland, A. Orso, and D. Poshyvanyk (2022). Enhancing mobile app bug reporting via real-time understanding of reproduction steps. *IEEE Transactions on Software Engineering*.
- Fazzini, M., M. Prammer, M. d’Amorim, and A. 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*, pp. 141–152.
- Fernández, A., S. García, M. Galar, R. C. Prati, B. Krawczyk, and F. Herrera (2018). Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 7(1), 1–19.
- Florez, J. M. (2019). Automated fine-grained requirements-to-code traceability link recovery. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 222–225. IEEE.
- Florez, J. M., L. Moreno, Z. Zhang, S. Wei, and A. Marcus (2022a). An empirical study of data constraint implementations in java. *Empirical Software Engineering* 27(5), 119.
- Florez, J. M., L. Moreno, Z. Zhang, S. Wei, and A. Marcus (2022b). An empirical study of data constraint implementations in java. *Empirical Software Engineering* 27(5), 119.
- Florez, J. M., J. Perry, S. Wei, and A. Marcus (2022). Retrieving data constraint implementations using fine-grained code patterns. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1893–1905.

- Guo, P. J., T. Zimmermann, N. Nagappan, and B. 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)*, pp. 495–504.
- Guzman, E., M. El-Haliby, and B. 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)*, pp. 771–776. IEEE.
- Hatano, T., T. Ishio, J. Okada, Y. Sakata, and K. Inoue (2016). Dependency-Based Extraction of Conditional Statements for Understanding Business Rules. *IEICE Transactions on Information and Systems E99.D(4)*, 1117–1126.
- Honnibal, M. and I. 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)*, 191–200.
- Hsu, C.-W. and C.-J. Lin (2003). A practical guide to support vector classification. In *Advances in Kernel Methods - Support Vector Learning*, pp. 169–208. MIT Press.
- Huang, H., W.-T. Tsai, S. Bhattacharya, X. Chen, Y. Wang, and J. Sun (1996, August). Business rule extraction from legacy code. In *Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC)*, pp. 162–167.
- Jiang, H., J. Zhang, Z. Ren, and T. 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)*, pp. 38–48. IEEE.
- Kallis, R., A. Di Sorbo, G. Canfora, and S. Panichella (2019). Ticket tagger: Machine learning driven issue classification. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*, pp. 406–409.

- Kiziltan, Z., M. Lippi, P. Torrioni, et al. (2016). Constraint detection in natural language problem descriptions. In *IJCAI*, Volume 2016, pp. 744–750. Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI’16).
- Ko, A. J., B. A. Myers, and D. H. Chau (2006). A Linguistic Analysis of How People Describe Software Problems. In *Visual Languages and Human-Centric Computing (VL/HCC’06)*, pp. 127–134.
- Krasniqi, R. and H. Do (2023). A multi-model framework for semantically enhancing detection of quality-related bug report descriptions. *Empirical Software Engineering* 28(2), 1–62.
- Kumar, N. and P. Devanbu (2016). Ontocat: Automatically categorizing knowledge in api documentation. *arXiv preprint arXiv:1607.07602*.
- Kurtanović, Z. and W. Maalej (2017). Automatically classifying functional and non-functional requirements using supervised machine learning. In *Proceedings of the IEEE 25th International Requirements Engineering Conference (RE)*, pp. 490–495. Ieee.
- Liu, H., M. Shen, J. Jin, and Y. 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*, pp. 128–140.
- Liu, M., X. Peng, A. Marcus, C. Treude, J. Xie, H. Xu, and Y. 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*, pp. 306–318.
- Maalej, W. and M. P. Robillard (2013). Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering* 39(9), 1264–1282.

- Manning, C. D. and H. Schütze (2009). An empirical study of naive bayes classifier. In *Proceedings of the 14th Conference on Empirical Methods in Natural Language Processing*, pp. 41–50.
- Marques, A. and G. 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)*, pp. 240–251. IEEE.
- Marques, A., G. Viviani, and G. 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)*, pp. 13–24. IEEE.
- Martin, W., 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).
- Menard, S. (2010). Logistic regression: Why we use it and how it works. *SAS Global Forum*.
- Miles, M. B., A. M. Huberman, and J. Saldaña (2013, Apr). *Qualitative Data Analysis: A Methods Sourcebook* (3rd ed.). SAGE Publications, Inc.
- Miles, M. B., M. Huberman, and J. Saldana (2014). Qualitative data analysis, a methods sourcebook. saeg publication.
- Mirzaie, M., B. Behkamal, and S. Paydar (2019). Big data quality: A systematic literature review and future research directions. *CoRR abs/1904.05353*.
- MITRE Corporation (2019). Mitre annotation toolkit (mat). <https://github.com/mitre/mitre-annotation>. Accessed: March 29, 2023.



- Monperrus, M., M. Eichberg, E. Tekes, and M. Mezini (2012). What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering* 17, 703–737.
- Monszpart, A., D. Papp, and T. 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)*, pp. 357–369. Springer.
- Pandita, R., X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar (2012). Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th international conference on software engineering (ICSE’12)*, pp. 815–825. IEEE.
- Panichella, S., A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. 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)*, pp. 281–290. IEEE.
- Pascarella, L., M. Bruntink, and A. Bacchelli (2019). Classifying code comments in java software systems. *Empirical Software Engineering* 24(3), 1499–1537.
- Pedregosa, F., 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, 2825–2830.
- Petrosyan, G., M. P. Robillard, and R. De Mori (2015). Discovering information explaining api types using text classification. In *Proceedings of the International Conference on Software Engineering (ICSE’15)*, pp. 869–879.

- Polanyi, L. (2003a). 14 the linguistic structure of discourse. *The Handbook of Discourse Analysis* 18, 265.
- Polanyi, L. (2003b). *The Handbook of Discourse Analysis*, Volume 18, Chapter The Linguistic Structure of Discourse, pp. 265. Wiley-Blackwell.
- Prana, G. A. A., C. Treude, F. Thung, T. Atapattu, and D. Lo (2019). Categorizing the content of github readme files. *Empirical Software Engineering* 24, 1296–1327.
- Quinlan, J. R. (1988). A comparative study of decision tree id3 and c4.5. In *Proceedings of the 5th International Workshop on Machine Learning*, pp. 47–64. Morgan Kaufmann.
- Rani, P., S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz (2021). How to identify class comment types? a multi-language approach for class comment classification. *Journal of Systems and Software* 181, 111047.
- Robillard, M. P. and Y. B. Chhetri (2015). Recommending reference api documentation. *Empirical Software Engineering* 20, 1558–1586.
- Roscoe, T., A. Martin, B. Shao, and K. Amorosa (2015). Parfait: Designing a scalable bug checker. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, Berkeley, CA, USA, pp. 163–174. USENIX Association.
- Ross, G. (2012). *Writing Effective Business Rules: A Practical Method* (1 ed.). Waltham, MA: Morgan Kaufmann.
- Sneed, H. M. (2001, May). Extracting business logic from existing COBOL programs as a basis for redevelopment. In *Proceedings of the 9th IEEE Workshop on Program Comprehension (IWPC)*, pp. 167–175.

- Sneed, H. M. and K. Erdös (1996, March). Extracting business rules from source code. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC)*, Berlin, Germany, pp. 240–247.
- Sonbol, R., G. Rebdawi, and N. Ghneim (2022). The use of nlp-based text representation techniques to support requirement engineering tasks: A systematic mapping review. *IEEE Access*.
- Song, Y. and O. 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)*, pp. 1551–1555.
- Song, Y., J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. 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)*, pp. 344–356.
- Stenström, A.-B. and K. Aijmer (2004). Discourse patterns in spoken and written corpora. *Discourse patterns in spoken and written corpora*, 1–287.
- Sureka, A. 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)*, pp. 366–374.
- Thung, F., D. Lo, and L. Jiang (2012). Automatic defect categorization. In *Proceedings of the Working Conference on Reverse Engineering (WCRE’12)*, pp. 205–214.

- Treude, C., F. Figueira Filho, and U. Kulesza (2015). Summarizing and measuring development activity. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*, pp. 625–636.
- Wang, S. and C. 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)*, pp. 90–94. Association for Computational Linguistics.
- Wang, X., J. Sun, X. Yang, Z. He, and S. Maddineni (2004, March). Business rules extraction from large legacy systems. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 249–258.
- Wieggers, K. and J. Beatty (2013a). *Software Requirements* (3 ed.). Redmond, WA: Microsoft Press.
- Wieggers, K. E. and J. Beatty (2013b). *Software Requirements* (Third ed.). Redmond, WA: Microsoft Press.
- Witt, G. C. (2012). *Writing Effective Business Rules : A Practical Method*. Waltham, MA: Morgan Kaufmann.
- Xiao, X., A. Paradkar, S. Thummalapenta, and T. 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*, pp. 1–11.
- Xu, B., D. Lo, X. Xia, A. Sureka, and S. Li (2015). Efspredictor: Predicting configuration bugs with ensemble feature selection. In *Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC'15)*, pp. 206–213.

- Yang, J., U. Sethi, C. Yan, A. Cheung, and S. Lu (2020, June). Managing data constraints in database-backed web applications. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, ICSE '20, New York, NY, USA, pp. 1098–1109. Association for Computing Machinery.
- Zhai, J., X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. 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)*, pp. 1359–1371.
- Zhang, M., Y. Liu, Y. Cheng, and M. 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*, pp. 1781–1796. Association for Computational Linguistics.
- Zhou, B., X. Xia, D. Lo, C. Tian, and X. Wang (2014). Towards More Accurate Content Categorization of API Discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC'14)*, pp. 95–105.
- Zhou, Y., Y. Tong, R. Gu, and H. Gall (2016). Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28(3), 150–176.

## BIOGRAPHICAL SKETCH

Ying Zhou is currently pursuing a master's degree in Computer Science at The University of Texas at Dallas, under the supervision of Dr. Andrian Marcus. Prior to this, she completed her BEng degree in Computer Science and Engineering at the University of Science and Technology, in Shenzhen, China.

Her research interests are in software engineering, with a specific focus on software maintenance and evolution. The main goal of her research is to develop advanced tools, methodologies, and practices that empower developers to build and maintain software more effectively. Her thesis centers on the automation of data constraints extraction, a crucial step towards achieving a comprehensive, fine-grained requirements-to-code traceability link recovery. Additionally, this automation will play a significant role in facilitating test generation, further enhancing the overall software development process. This endeavor leverages insights derived from qualitative methods and machine learning. The focus is to understand how stakeholders describe data constraints, and turn these insights into actionable techniques via the use of static analysis, text retrieval, heuristics and machine learning.

## CURRICULUM VITAE

# Ying Zhou

June 15, 2023

### Contact Information:

Department of Computer Science  
The University of Texas at Dallas  
800 W. Campbell Rd.  
Richardson, TX 75080-3021, U.S.A.

Email: [ying.zhou@utdallas.edu](mailto:ying.zhou@utdallas.edu)

### Educational History:

BS, Computer Science and Technology (3.63/4.00), 2020

Bachelor Thesis

IntelliSpotBugs: Reducing False-positive of SpotBugs Using Method Names

Advisor: Dr. Shin Hwei Tan

MS, Computer Science, The University of Texas at Dallas (4.00/4.00), 2023

*Automated Extraction of Data Constraints from Software*

Master Thesis

Advisor: Dr. Andrian Marcus

### Employment History:

Research Assistant, The University of Texas at Dallas, August 2021 – present

Research Assistant, Southern University of Science and Technology, June 2020 – August 2021

Software Engineer in Test Intern, Tencent, June 2019 – August 2019

### Professional Recognitions and Honors:

1<sup>st</sup> Prize, National Software Testing Finals for College Student, 2019

3<sup>rd</sup> Place, Tencent WeTest Cup. Dec. 2019

2<sup>nd</sup> Place, The Third IEEE International Software Testing Contest at QRS. Jul. 2019

4<sup>th</sup> Place, International Software Testing Contest at ICST, Apr. 2019

Campus Outstanding Student Scholarship, Oct. 2019

Outstanding Freshman Scholarship, SUSTech, Sep. 2016