



# Towards the automated identification of data constraints in software documents

Ying Zhou<sup>1</sup> · Miao Miao<sup>2</sup> · Vlad Birsan<sup>2</sup> · Oscar Chaparro<sup>3</sup> · Shiyi Wei<sup>2</sup> · Andrian Marcus<sup>1</sup>

Received: 11 November 2025 / Accepted: 15 September 2026  
© The Author(s) 2026

## Abstract

Data constraints are important business rules that specify the values allowed or required for the data used in a software system. These constraints are typically described in textual software artifacts (*e.g.*, requirements, developer documents, user manuals, *etc.*). In this paper, we focus on automating the task of identifying data constraints in natural language software documents. In order to automate this task, we first study how data constraints are described in natural language. Specifically, we studied 546 data constraints described in the natural language artifacts of nine software systems. We uncovered and documented nine linguistic discourse patterns that stakeholders use to describe data constraints in natural language. We cast the constraint identification and extraction as a text matching and classification problem. We explore the suitability of traditional machine learning techniques, using the discourse patterns we discovered, together with other linguistic elements, as features, for automatically classifying sentence fragments as data constraint descriptions. The best combination of features and learner achieves 63.63% precision and 61.40% recall (62.34% F1). Automating the extraction of data constraints from natural language text is essential for automating their traceability to code, test generation, compliance analysis, *etc.*

**Keywords** Requirements extraction · Machine learning · Natural language processing · Data constraint

## 1 Introduction

Software systems perform operations on data that represent aspects of the application domain or parts of computing components. Functional requirements, domain business rules, and design decisions define constraints on such data, often described in natural language. *Data constraints* specify what values are allowed or required for the given data [23, 78, 79]. For example, the highlighted parts of the sentences in Table 1 show four data constraints described in natural language.

Keeping track of data constraints during software evolution is essential for generating test cases, checking how they are enforced in the source code, or making code changes when the constraints change. Many data constraints are formulated based on regulatory requirements and are important for ensuring system compliance with such requirements.

Recent research examined how they are implemented in code [23] and how to trace them to their implementations [24, 82]. This prior work has focused on manually extracted data constraints [24] or constraints extracted from databases [82], making the automated analysis of

---

✉ Ying Zhou  
yzhou29@gmu.edu

Miao Miao  
mmiao@utdallas.edu

Vlad Birsan  
vlad.birsan@utdallas.edu

Oscar Chaparro  
oscarch@wm.edu

Shiyi Wei  
swei@utdallas.edu

Andrian Marcus  
amarcus7@gmu.edu

<sup>1</sup> George Mason University, Fairfax, United States

<sup>2</sup> The University of Texas at Dallas, Richardson, United States

<sup>3</sup> William and Mary, Williamsburg, United States

**Table 1** Examples of data constraints (highlighted)

|                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Example1</b> (from apache-ant-1.10.6)<br>The <b>target file name is identical to the source file name.</b><br><b>Constraint type:</b> value-comparison<br><b>Operands:</b> target file name, source file name<br><b>Discourse pattern:</b> NP_COMP_NP                                                                                                                                                                 |
| <b>Example2</b> (from log4j-2.13.3)<br>The set of <b>built-in levels includes TRACE, DEBUG, INFO, WARN, ERROR, FATAL.</b><br><b>Constraint type:</b> categorical-value<br><b>Operands:</b> Built-in levels, TRACE, DEBUG, INFO, WARN, ERROR, and FATAL<br><b>Discourse pattern:</b> NP_IN_VALUESET                                                                                                                       |
| <b>Example3</b> (from swarm v2), two constraints in one sentence<br><b>Log frequency, if checked, will set the frequency axis to log mode.</b><br><b>Constraint1 type:</b> binary-value<br><b>Operands1:</b> log frequency, checked<br><b>Discourse pattern1:</b> NP_BINARY_VALUE<br><b>Constraint2 type:</b> concrete-value<br><b>Operands2:</b> the frequency axis, log mode<br><b>Discourse pattern2:</b> NP_BE_VALUE |

constraints currently impractical. For example, the traceability approach by Florez et al. [24] cannot extract data constraints from existing documents in a software project. The same work revealed that the manual identification and extraction of data constraints from software documents is a time-consuming and error-prone task. Data constraints are defined at different stages and documented in various software documents. For example, some data constraints are defined directly in requirements and captured in requirement documents, while others are defined later, as a result of design decisions, and are document in other type of textual software artifacts. Keeping track of data constraints across various documents and software engineering tasks (e.g., compliance check, testing, documentation, etc.) is far from trivial. One way to support the definition and maintenance of data constraints is to automate their identification and extraction from exiting software documents. This paper proposes the use natural language processing (NLP) techniques, together with traditional machine learning (ML), for automatically identifying and extraction of data constraint descriptions.

At a high level, our work is akin to other research on automatically extracting and classifying different types of requirements [1] (e.g., functional vs. non-functional [6, 12, 38], security-related requirements [7, 80], regulatory requirements, etc.)

Previous work on tracing data constraints to their implementations Florez et al. [24] leveraged on structural elements of the constraints (e.g., type and operands). We hypothesize that data constraints descriptions are likely to exhibit regularities related to their type and operands. Furthermore, we posit that these patterns can be used for classifying natural language text as data constraints. With this in mind, we address two research questions:

**RQ1:** How are data constraints described in natural language?

**RQ2:** Which machine learning strategy most effectively classifies text fragments as data constraint descriptions?

To characterize how data constraints are described in natural language (RQ1), we manually collected and labeled 1,560 data constraint descriptions from the textual artifacts of 15 software systems, which we make available for future research. We analyzed a subset of 546 *explicit* data constraints from nine of these systems and found that they are described using a set of nine linguistic discourse patterns, which we documented in a catalog (Sect. 3). For example, the four data constraints shown in Table 1 are described using four different discourse patterns (i.e., NP\_COMP\_NP, NP\_IN\_VALUESET, NP\_IF\_BINARY\_VALUE, NP\_BE\_VALUE). These discourse patterns have not been documented in any previous research. For each pattern, the catalog also records the constraint fragment, its operands, and its constraint type, which serve as features for the machine learning classifier that identifies data constraints. The identified constraints then support downstream tasks such as traceability and test generation.

Based on the discourse pattern catalog, we developed *S2F*, a tool that automatically identifies sentence fragments matching the discourse patterns in a given natural language sentence and extracts their operands and constraint types (Sect. 4.1). Our goal is to automatically determine whether the text fragments extracted with *S2F* are data constraint descriptions. To find the most effective machine learning strategy for classifying text fragments as data constraint descriptions (RQ2), we examined the use of *discourse patterns*, *operands*, and *data constraint types* as novel features for the classification of text fragments into data constraints, in addition to the textual features of the constraint descriptions (Sect. 4). We classified the above-mentioned 546 data

constraint descriptions, and 26,641 additional sentence fragments with no data constraints, using four different traditional machine learning techniques (*i.e.*, Logistic Regression, Linear SVM, Multinomial Naive Bayes, and Decision Trees). We explored all combinations of the features and applied two feature encodings, TF-IDF transformation [14] and BERT embeddings [17]. Given the unbalanced nature of the data we used, we also examined the effect of different data balancing strategies on the classification performance. We found that using the novel features, operands and fragments, led to improvement in classification. The best combination of features, learner, and BERT embeddings without data balancing achieved 63.63% precision and 61.40% recall (62.34% F1). In practice, the automated classification can reduce the amount of manual inspection needed before any downstream analysis. The classifier can act as a pre-filter that prioritizes likely constraint fragments from a large pool of candidate fragments and provides candidate typed constraints and operands for traceability and test-generation pipelines.

The main contributions of the paper are:

- A catalog of nine discourse patterns that characterize how stakeholders describe data constraints in natural language, which provides the linguistic features for automated identification.
- S2F, a tool that uses the catalog to automatically extract candidate fragments together with their operands and constraint types.
- An empirical study that identified which features, feature encodings, learners, and data balancing strategies are best for automatically classifying text fragments as data constraints.
- A data set of 1,560 labeled data constraints from 15 software systems and a replication package including all the data and the complete results [2].

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

## 2 Background

This section introduces important terminology and definitions regarding data constraints, with a set of examples. Table 1 shows four highlighted examples of data constraints, which 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* (defined below) of each constraint, as well as the *discourse pattern* (defined in Sect. 3).

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

Florez et al. [23] classified data constraints into four *types* (value-comparison, binary-value, categorical-value, and concrete-value). These constraint types are relevant in this work, and we reuse the definitions introduced by their work.

For value-comparison constraints, the value of a data element *X* is constrained by the value of another data element *Y* (or constant *C*). The relationship between the data elements translates to equality or relational operators used to determine if *X* is greater than, less than, equal to or not equal to *Y* (or *C*). Constraints of this type always have two operands: the two data elements, *X* and *Y* (or *C*).

For binary-value constraints, a data element *X* has one of only two possible, mutually-exclusive binary values *BV* (*e.g.*, true/false, on/off, and null/not-null). These are equivalent to value-comparison constraints with the equality operator and only two possible values. This constraint type always has two operands: the data element *X* and the binary value *BV*.

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 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 have two operands: the data element *X* and the concrete value *V*.

## 3 How are data constraints described?

We performed a study to understand how data constraints are described in natural language software documents, answering the following research question:

**RQ1:** How are data constraints described in natural language?

Specifically, we aimed to identify discourse patterns (if any) used in describing the constraints.

### 3.1 Data collection

Table 2 summarizes the data set we constructed and used in the empirical studies from this paper; details of the collected data are included in the replication package [2]. The dataset used in this study includes constraints from nine systems in different domains and from multiple forms of documentation, including user manuals, tutorials, and specifications. The analyzed artifacts also vary in size and format: Joda-Time uses a user manual with 18 HTML pages, ArgoUML uses a user manual with 403 pages, and HTTPComponents uses 51 tutorial pages and a specification document. Note that we collected more data, some of which was not used in the studies from this paper, as explained below. Nonetheless, they are valuable for future research and we make them available.

We started with a set of 299 constraints from 15 systems, collected by Florez et al. [24]. We then labeled additional sentences from the textual artifacts of these systems (user manuals, requirements, design specifications, *etc.*). In total, we labeled 22,237 sentences as follows. Each sentence was labeled with one of the following labels: *ignore*, *negative*, or *positive*. The sentences of the 299 constraints from previous work did not have any of these labels, as they were used for a different purpose (*i.e.*, tracing to code). *Ignore* sentences are those that are not in usual natural language format, specifically, they include content lists (*i.e.*, bullet or enumerated lists), log/outputs, regular expressions, source code, tables, and XML-related information. We labeled 4,746 sentences (21.3%) as *ignore* and they were not used further in our study.

Each sentence that contained at least one constraint was labeled as *positive*. The constraint description was highlighted, as shown in the examples from Table 1, using the MITRE Annotation Toolkit (MAT) version 3.3 [50]. For each constraint, the starting and ending characters of its

description in the sentence were recorded. We labeled 1,052 sentences (4.7%) as *positive*, with 1,560 constraints (some sentences describe more than one constraint). We labeled 16,439 sentences (73.9%) as *negative*, which did not contain any data constraints.

The focus of our work is on *explicit* data constraint descriptions in natural language. We define an *explicit* constraint description as 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* and *them*). Automatically classifying such descriptions would require complex and notoriously imprecise NLP techniques such as automated reference resolution [58, 62]. Consequently, we excluded 791 constraints (50.7% of the 1,560 labeled constraints) that were not explicit: 270 constraints (17.3%) span multiple sentences and 521 constraints (33.4%) had potentially ambiguous references via indefinite pronouns. Constraint descriptions that span multiple sentences often state the data element in one sentence and the permitted value or condition in a following sentence, while pronoun-based descriptions refer to the constrained data element or value using terms such as *it* or *them*. Handling such constraints is subject of future work.

Furthermore, we excluded 171 constraints (11.0%) that refer to GUI elements. We consider these a separate category because the restriction is often expressed through widget behavior or screen context, such as whether a field is enabled, which options are available, or which value can be selected in a particular window. Mapping such UI behavior to the underlying constrained data element and value requires UI context in addition to the linguistic patterns that can be identified within a single sentence. After excluding non-explicit and GUI-related constraints, 598 explicit non-GUI constraints remained. We further excluded 52 constraints (3.3%) from 6 systems where we had left fewer than 20 constraints. We also excluded the *negative* sentences from these systems. All these excluded constraints can still

**Table 2** Systems, sentences, and explicit constraints used

| System     | Domain                     | Textual artifact | +sent (+cons; -frags) | -sent (-frags) |
|------------|----------------------------|------------------|-----------------------|----------------|
| Joda-Time  | Date/time library          | User Manual      | 36 (52; 17)           | 544 (1,682)    |
| Apache Ant | Build manager              | User Manual      | 137 (158; 315)        | 2,863 (8,617)  |
| HTTPComp   | HTTP Client/Server library | Tutorial, Specs  | 39 (45; 73)           | 728 (2,578)    |
| jEdit      | Text editor                | User Manual      | 65 (67; 127)          | 1,043 (3,330)  |
| ArgoUML    | UML Modeler                | User Manual      | 47 (48; 50)           | 2,350 (7,108)  |
| Swarm      | Seismic wave visualizer    | User Manual      | 45 (58; 42)           | 167 (538)      |
| Checkstyle | Source Code Style Checker  | User Manual      | 18 (22; 24)           | 286 (838)      |
| JabRef     | Citation Manager           | User Manual      | 36 (39; 78)           | 195 (521)      |
| Log4j      | Logging Framework          | User Manual      | 51 (57; 107)          | 242 (596)      |
| Total      |                            |                  | 474 (546; 833)        | 8,418 (25,808) |

“+” denotes positive sentences or constraints, and “-” represents negative sentences or fragments

be used in future research and we make them available in our replication package [2], which includes all 1,560 labeled constraints from 15 systems. We filtered out negative sentences from systems with fewer than 20 constraints because those systems lack sufficient data for machine learning, with 20 chosen as a practical threshold for this study.

For this study, we were left with 474 positive sentences, containing 546 explicit constraints across nine systems (4th column in Table 2). These 546 constraints were also labeled with their *constraint type* and *operands*. Note that positive sentences may also have negative fragments (*i.e.*, fragments that do not describe constraints), in addition to the positive ones. The 474 positive sentences included 833 negative fragments.

Among the 14,016 negative sentences within the nine remaining systems, 149 consisted of two words or less, making them unsuitable for natural language analysis. Of the remaining 13,867 negative sentences, 8,418 were used in the classification studies from Sect. 4, which explains this last data filtering step in detail.

### 3.1.1 Constraint labeling protocol

We employed 10 computer science (CS) graduate and undergraduate students, and four authors of this paper to label sentences and constraints. We used the following process for labeling each document. Using the aforementioned MAT software [50], a labeler read through the entire document and marked each sentence with one label (*ignore*, *positive*, or *negative*). First, if the sentence was deemed to be ignored (based on the above definition), it was labeled as *ignore*. Otherwise, if at least one data constraint was described in the sentence, the labeler would label it and copy the operands into a shared table, as well as assign one of the four types defined in Sect. 2. In addition, for each constraint, the labeler would determine if the constraint met the criteria for being an *explicit* data constraint, and if not, provide a reason. Finally, if the sentence contained no constraints, it was labeled as *negative*.

Each sentence was labeled either by one paper author or two of the 10 trained CS students, separately. The authors had more experience with this task, so we had more confidence in their work. Conversely, the students were less experienced, hence the need to have two labelers per sentence. Using two students to label the same data independently also allowed us to capture agreements and assess the difficulty of the task. 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 student-labeled data, of the 4,746 sentences finalized as *ignore*, the reviewer corrected 202 cases where one student disagreed and 28 cases where

both students disagreed; of the 11,109 sentences finalized as *negative*, the reviewer corrected 930 cases where one student disagreed and 43 cases where both students disagreed; of the 221 constraints finalized as *not explicit*, the reviewer corrected 145 cases where one student disagreed and 38 cases where both students disagreed; of the 247 constraints finalized as *explicit*, the reviewer corrected 170 cases where one student disagreed and 5 cases where both students disagreed. Out of the 16,323 labels across all categories (4,746 ignored sentences, 11,109 negative sentences, 221 non-explicit constraints, and 247 explicit constraints), the annotators achieved agreement on 14,762 labels ( $\approx 90.44\%$  agreement, with a Cohen's Kappa coefficient [15] of 0.79).

While our protocol was designed to ensure high quality of the labeled data, these 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.

### 3.2 Discourse pattern coding protocol

We analyzed the 546 *explicit* constraints from nine systems to determine 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 used in their descriptions. These patterns define the parts of speech of the constraints and the operands in the constraints.

In order to extract these patterns, we performed a qualitative discourse analysis [59, 71] of the 546 labeled constraints, based on open coding [49]. Two authors 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 15 patterns. During the analysis, some initial patterns were merged due to conceptual overlap. For instance, NP\_BE\_BINARY\_VALUE, which describes the assignment of a binary value to a data element through an auxiliary verb, and NP\_SET\_TO\_BINARY\_VALUE, which describes the same assignment using a phrase composed of an auxiliary verb and a preposition with the value assignment phrase positioned after a data element, were merged as they both describe the same binary value assignment despite their syntactic differences. A code was defined for each pattern and used for labeling. In the following steps, each of the two coders 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 then they discussed with the other coder. Once they both agreed on the new pattern, its code was defined

**Table 3** Distribution of DPs and constraint types

| ID  | Pattern code       | conv | vcom | binv | catv | Total |
|-----|--------------------|------|------|------|------|-------|
| DP1 | NP_BE_BINARY_VALUE |      |      | 301  |      | 301   |
| DP2 | NP_BE_VALUE        | 58   | 22   |      |      | 80    |
| DP3 | NP_CD_TO_CD        |      | 48   |      |      | 48    |
| DP4 | NUM_NP             | 11   | 6    |      |      | 17    |
| DP5 | NP_COMP_NP         |      | 43   |      |      | 43    |
| DP6 | NP_IF_BINARY_VALUE |      |      | 17   |      | 17    |
| DP7 | NP_IN_VALUESET     |      |      |      | 18   | 18    |
| DP8 | NP_OF_NP           | 9    | 3    |      |      | 12    |
| DP9 | NP_EXIST           |      |      | 10   |      | 10    |

conv=concrete-value, vcom=value-comparison, binv=binary-value, and catv=categorical-value

**Table 4** Distribution of discourse patterns across the nine systems

| System     | DP1 | DP2 | DP3 | DP4 | DP5 | DP6 | DP7 | DP8 | DP9 | Total |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------|
| Joda-Time  | 3   | 16  | 25  | 2   | 5   |     | 1   |     |     | 52    |
| Apache Ant | 120 | 19  |     |     | 7   | 3   | 3   | 3   | 3   | 158   |
| HTTPComp   | 22  | 10  |     | 2   | 7   | 1   | 1   | 2   |     | 45    |
| jEdit      | 52  | 4   |     | 4   | 2   | 1   | 2   | 1   | 1   | 67    |
| ArgoUML    | 33  | 5   |     |     | 2   | 1   | 4   | 1   | 2   | 48    |
| Swarm      | 15  | 5   | 14  | 3   | 8   | 6   | 3   | 4   |     | 58    |
| Checkstyle | 14  | 4   |     | 1   | 1   |     |     | 1   | 1   | 22    |
| JabRef     | 17  | 2   | 5   | 3   | 9   | 3   |     |     |     | 39    |
| Log4j      | 25  | 15  | 4   | 2   | 2   | 2   | 4   |     | 3   | 57    |
| Total      | 301 | 80  | 48  | 17  | 43  | 17  | 18  | 12  | 10  | 546   |
| #Systems   | 9   | 9   | 4   | 7   | 9   | 7   | 7   | 6   | 5   | —     |

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 met to discuss and resolve any discrepancies, refining the codes by combining, splitting, or modifying them. This process was repeated for an additional hundred constraints at a time until all 546 constraints were labeled.

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

### 3.3 Results - discourse patterns

We summarize the resulting nine discourse patterns from three perspectives. Table 3 shows the frequency of each pattern and its distribution across constraint types. The ID column in Table 3 provides the DP labels used in Table 4. To help interpret the pattern codes, Table 5 provides descriptions, rules, definitions, and examples for the three most commonly used patterns. The complete discourse-pattern catalog is provided in the replication package [2].

The distribution in Table 4 shows that nine discourse patterns recur across multiple systems. The two most frequent patterns, NP\_BE\_BINARY\_VALUE and NP\_BE\_VALUE, appear in all systems, and five additional patterns appear in at least six systems. This distribution indicates that many

constraints in the studied systems are expressed through recurring linguistic patterns.

The NP\_BE\_BINARY\_VALUE discourse pattern represents that a binary value has a dependency on the modal verbs “be”, “allow”, “remain”, “show”, or the equality symbol “=” with a data element. For example, “*fork = true*” is the constraint with operands *fork* and *true*. This relationship is alternatively expressed through the term “set to” alongside its synonyms “define as”, “specify as”, “marked as”, and “default to”. For instance, “*usedefaultexcludes attribute defaults to false*” has the constraint *usedefaultexcludes = false* with operands *usedefaultexcludes* and *false*.

The NP\_BE\_VALUE pattern is similar to NP\_BE\_BINARY\_VALUE, except that the value is not binary. For example, “*default maxwait is 3 minutes*” has the constraint *maxwait = 3 minutes* with operands *maxiwait* and *3 minutes*; “*the margin is set to 0*” has the constraint *margin = 0* with operands *margin* and *0*.

The third most frequent pattern is NP\_CD\_TO\_CD, which represents a range of numeric values, expressed using a preposition/conjunction pair (e.g., “*from...to*” and “*between...and*”), having a dependency with a data element. For example, “*a valid port number between 0 and 65,535*” has the constraints *port number ≥ 0* and *port number ≤ 65535* with operands *port number*, *0*, and *65535*.

**Discourse pattern rules and operands.** The rules of the discourse patterns indicate which parts of the data

**Table 5** Three most commonly used discourse patterns

---

**Pattern code:** NP\_BE\_BINARY\_VALUE

**Description:** An attribute is assigned a binary value (e.g., *true/false, on/off, out/inout...*).

**Rules:**

1. data [be] binary-value
2. data [set] [to] binary-value
3. data [be] [set] [to] binary-value
4. [set] data [to] binary-value
5. data [be] turn binary-value

**Definitions:**

data  $\in \{\text{NP}\}$ ;

[be]  $\in \{\text{be, allow, remain, show, '='}\}$ ;

[set]  $\in \{\text{set, define, specify, mark, default}\}$ ; [to]  $\in \{\text{to, as, with}\}$ ;

binary-value  $\in \{\text{VBN, ADJ, on, off, true, false, ...}\}$

NP is noun phrase; VBN and ADJ are PoS, denoting the past participle verbs and adjectives, respectively;

**Examples:**

1. fork [=] true (from apache-ant-1.10.6)
  2. useDefaultExcludes attribute [defaults] [to] false (from apache-ant-1.10.6)
  3. [setting] setterCanReturnItsClass property [to] true (from checkstyle-8.35)
  4. [set] ant.build.clonevm [to] true (from apache-ant-1.10.6)
  5. overwrite [is] turned off (from apache-ant-1.10.6)
- 

**Pattern code:** NP\_BE\_VALUE

**Description:** An attribute is assigned a concrete value.

**Rules:**

1. data [be] [value]
2. with data [set] [to] value
3. data [be] [set] [to] value
4. [set] data [to] value
5. require data to [be] [set] [to] value

**Definitions:**

data  $\in \{\text{NP}\}$ ;

[be]  $\in \{\text{be, allow, remain, show, '='}\}$ ;

[set]  $\in \{\text{set, define, specify, mark, default}\}$ ;

[to]  $\in \{\text{to, as, with}\}$ ;

value  $\in \{\text{NP, CD}\}$ ;

CD is PoS, denoting the cardinal numbers

**Examples:**

1. default maxwait [is] 3 minutes (from apache-ant-1.10.6 / Tasks)
  2. with its slider [set] by default [to] priority 1 (from argouml-0.35.4)
  3. the margin [is] [set] [to] 0 (from jedit-5.6pre0)
  4. [defines] a null instant [as] the current time (from joda\_time-2.10.3)
  5. requires fork to [be] [set] [to] true (from apache-ant-1.10.6)
- 

**Pattern code:** NP\_CD\_TO\_CD

**Description:** A attribute is assigned a range of numeric values. The lower bound and upper bound are connected by a preposition/conjunction pair (e.g., *from...to, between...and*).

**Rules:**

1. data value1 to value2
2. data [from] value1 [to] value2
3. value1 to value2 data

**Definitions:**

data  $\in \{\text{NP}\}$ ;

[from]  $\in \{\text{from, between}\}$ ;

[to]  $\in \{\text{to, and}\}$ ;

value1, value2  $\in \{\text{NP, CD}\}$

**Examples:**

1. a valid port number between 0 and 65535 (from log4j-2.13.3)
  2. hour [from] 00 [to] 23 (from joda\_time-2.10.3)
  3. two to four authors (from jabref-5.0)
-

constraints are the *operands*. For the rules shown in Table 5, the terms matching [data] and [value] are the operands. Specifically, for the NP\_BE\_BINARY\_VALUE pattern example *fork* matches [data] in the rule and *true* 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 [value1] and [value2] in the rule, making them the three operands.

**Correlation between discourse patterns and constraint types.** Table 3 shows the distribution of discourse patterns over the constraint types. In columns 2-5, we refer to concrete-value as *conv*, value-comparison as *vcom*, binary-value as *binv*, and categorical-value as *catv*. We observed that six out of the nine discourse patterns are used to describe constraints of a single type. For example, all the 301 constraints described with the NP\_BE\_BINARY\_VALUE discourse pattern are of binary-value type. The other three discourse patterns (NP\_BE\_VALUE, NUM\_NP, or NP\_OF\_NP) are used to describe constraints of two types: concrete-value and value-comparison. For example, of the 80 constraints described with the NP\_BE\_VALUE discourse pattern, 58 are of concrete-value type and 22 are of value-comparison type. This observation indicates that we can use the discourse pattern of a constraint to automatically infer its type, which we actually do in the classification study (Sect. 4).

### 3.4 Threats to validity

**Construct Validity.** Subjectivity in identifying constraints was mitigated by following a well-defined protocol to label sentences and constraints. The labeling process was performed in iterations, and included reviews and discussion of disagreements to refine the rules. The reported agreement summarizes the initial annotation workflow. We did

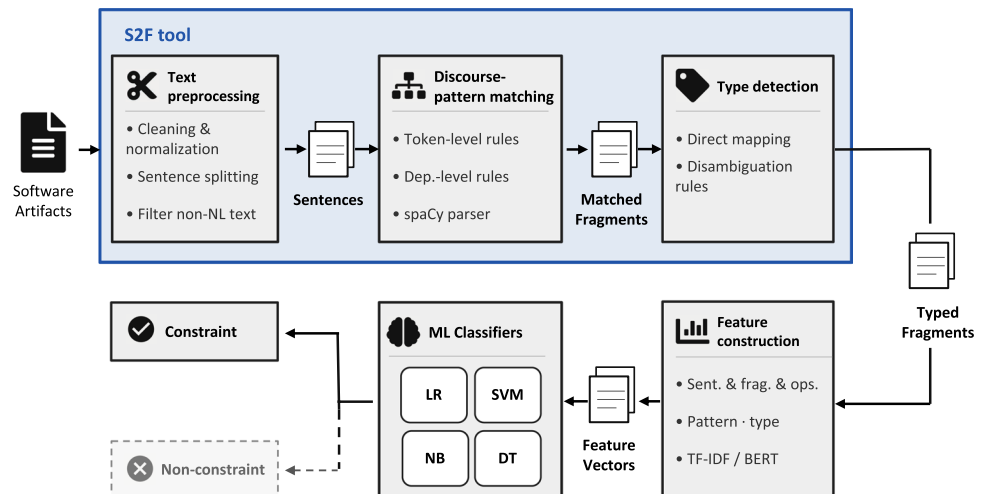
not separately evaluate agreement for each later annotation decision, such as constraint boundaries or constraint types. Subjectivity in deriving the discourse patterns from identified constraints was mitigated by following an open coding procedure with discussion of ambiguous cases, refinement of the pattern catalog and coded data, and assessment of the coding process.

**External Validity.** The identified discourse patterns may not generalize to explicit constraints descriptions beyond the data we analyzed. That said, the systems we used are of different domains and kinds (*e.g.*, desktop apps and libraries), and the constraints are from different documents (user manuals, design specifications, *etc.*); this strengthens the generalization of our patterns. Discovering new patterns from more data would not invalidate the ones we documented in this dataset. In fact, our discourse pattern catalog is meant to evolve with new research.

## 4 Classifying text fragments as data constraints with machine learning

Given a sentence fragment, we want to automatically determine whether the fragment is a data constraint description or not. Fig. 1 provides an overview of our approach for identifying data constraints in software documentations. The approach first preprocesses the textual artifacts by cleaning and normalizing the text, splitting it into sentences, and filtering sentences that are not in natural language format, such as code snippets and tables (Sect. 3.1). The Sentence-to-Fragment (S2F) tool then identifies fragments that match discourse patterns, extract the corresponding operands, and assigns constraint type using direct mappings or disambiguation rules (Sect. 4.1.1). The resulting fragments are encoded as feature vectors using sentence, fragment, operands, discourse pattern and constraint type features with TF-IDF or BERT (Sect. 4.3). Finally, machine learning

**Fig. 1** Overview of classifying text fragments as data constraints



classifiers classify the feature vectors as constraint or non-constraint fragments (Sect. 4.5).

To support our data constraint extraction and identification approach we conducted an empirical study to determine how machine learning techniques can most effectively classify natural-language sentence fragments as data constraint descriptions. The study aims to answer the following research question.

**RQ2:** Which machine learning strategy most effectively classifies text fragments as data constraint descriptions? We further refine the **RQ2** to address several specific aspects of the classification:

- **RQ2.1:** How do feature encoding methods affect machine learning-based data constraint classification?
- **RQ2.2:** How does the best machine learning-based classifier compare with a fine-tuned BERT classifier?
- **RQ2.3:** Which features are the most important for machine learning-based data constraint classification?
- **RQ2.4:** How do data balancing techniques affect machine learning-based data constraint classification?

## 4.1 Data

We used the 474 positive sentences from nine systems, containing 546 explicit data constraint descriptions, described in Sect. 3. We also used the 13,867 labeled negative sentences, which did not contain any labeled data constraints.

### 4.1.1 Sentence fragment extraction

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

**Discourse pattern matching.** We use the spaCy Matcher version 3.4.3 [31] to identify the nine discourse patterns, applying a set of linguistic features: part-of-speech tags, dependencies, and entities. Before processing sentences, we use a text cleaner to ensure that the input adheres to a standardized format for subsequent processing. This includes rectifying incorrectly encoded text to remove unreadable characters and removing specific phrases like “a value of” and “the default of” preceding binary values. Using the text cleaner helps prevent scenarios where fragments that were intended to match NP\_BE\_BINARY\_VALUE are no longer matched. Additionally, the text cleaner eliminates words like “only” and “therefore” to maintain accurate identification of operands within fragments.

To split a sentence into fragments that match the discourse patterns, we adopt a layered analysis approach [65], which involves defining each pattern with two levels: one for token matching and one for dependency matching between the tokens.

First, to perform token 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.

Second, to perform dependency matching, we define 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 two patterns (*i.e.*, NUM\_NP and NP\_OF\_NP) to extract complex nouns and noun phrases from constraints using spaCy. These patterns serve as potential noun phrases for facilitating boundary detection of noun phrases. Our approach first identifies the above patterns in the text, then merges the tokens from matched patterns into a single token, and finally identifies the other patterns in the text.

**Operand 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 Table 3, S2F assigns a constraint type to each matched fragment in one of two ways. If a discourse pattern is associated only one constraint type, S2F directly maps the fragment to that type, which applies to six of the nine discourse patterns. Otherwise, if a discourse pattern can correspond to multiple constraint types, S2F applies a set of disambiguation rules to infer the type, which applies to the other 3 discourse patterns (*i.e.*, NP\_BE\_VALUE, NUM\_NP, and NP\_OF\_NP) correspond to two constraint types: *value-comparison* and *concrete-value*.

For the three ambiguous patterns, we implemented four rules to infer the constraint type for a fragment sequentially, based on our analysis of the data.

- **Rule 1:** If the fragment text starts with *set*, then the `concrete-value` type will be assigned to the fragment.
- **Rule 2:** If any word from the a list of words (*i.e.*, *at least*, *up to*, *lower*, *larger*, *more*, *negative*, *positive*, *multi*, *not*, *n't*) appears in the sentence, then the `value-comparison` type will be assigned to the fragment.
- **Rule 3:** If the conjunction *if* or *when* is 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.
- **Rule 4:** If the fragment fails to meet the criteria in rules 1, 2 and 3, then the fragment type will be assigned as `concrete-value`.

For example, a fragment starting with *set* (Rule 1) would bypass subsequent checks for keywords (Rule 2) or conjunctions (Rule 3). While no overlaps occurred in our dataset, this sequential design ensures unambiguous assignments.

#### 4.1.2 S2F Evaluation

We manually evaluated the accuracy of the S2F tool. We randomly sampled 374 sentences (24 positives and 350 negatives) from 13,867 negative and 474 positive sentences labeled in Sect. 3, with stratification preserving the systems and applying a 95% confidence level. S2F identified 1,211 fragments from the 374 sentences. Each sentence was reviewed by two authors independently, and any discrepancies were resolved through discussion. The review process for each sentence involved the following steps: the reviewers received a list of fragments within the sentence, along with the corresponding discourse pattern and operands for each fragment, identified by S2F. Their task is to verify the *accuracy* of both the discourse pattern and operands assigned to each fragment. A correct discourse pattern for a given fragment indicates that the identified fragment text conforms to one of the predefined rules for that pattern. Moreover, ensuring correct operands for the fragment signifies that the operands are extracted from the matched tokens, specifically those with ID names containing the keywords “data”

and “value”, as stipulated by the corresponding pattern rule. A fragment is considered *falsely identified* if either the discourse pattern or the operands do not meet these criteria.

Table 6 presents the accuracy obtained by S2F, on the sample data. For each discourse pattern, the second column shows the number of fragments identified by S2F from the sampled sentences. The third column shows the number of correctly identified fragments, while the last column calculates accuracy.

As shown in Table 6, the overall accuracy is very high (98.6%), across all patterns, which gives us high confidence in the labels generated by S2F. While analyzing the cases where S2F misclassified the fragments, we identified two main causes: errors in the spaCy parser and the rules describing the discourse patterns. The spaCy errors can be further classified into three types: inaccurate extraction of noun phrases, incorrect part-of-speech assignment for terms, and errors in the dependency parsing between terms. The inaccurate extraction of noun phrases occurs when spaCy identifies a superset of noun phrases. For instance, the phrase “*the Utilities > Options dialog box*” was mistakenly labeled as a noun phrase, leading to the incorrect identification of the fragment “pane of the Utilities>Options dialog box”. The accurate noun phrase should have been “the Utilities”, that is the data element of the fragment “pane of the Utilities”. Incorrect part-of-speech assignments for terms arise when a word has multiple parts of speech depending on the context. For example, in the constraint “*If the logfile attribute is set to fail*”, spaCy erroneously assigns the part-of-speech ‘VERB’ to ‘fail’. But in this context, ‘fail’ should be categorized as a noun value, causing S2F to erroneously extracting corresponding fragments for the constraints. Errors in dependency parsing between terms can be exemplified in the sentence “Q: Is there a FAQ on BibTeX?” Here, spaCy incorrectly parsed an *nsubj* relationship between “Q.” and the verb “is”, resulting in the inaccurate identification of the fragment “Q: Is there a FAQ on BibTeX?” with its operands “Q” and “FAQ”.

The discourse pattern rules we defined also have some limitations. For instance, we only utilized two rules to extract nouns, which are insufficient to describe all noun phrases. Currently, neither the rules nor spaCy can identify ‘years divisible by 100’ as a noun phrase in the constraint “*years divisible by 100 are not leap*”. Another example is the case of the NP\_BE\_BINARY\_VALUE pattern, where we observed that the majority of adjectives could be interpreted as `binary-value` (such as ‘empty’, ‘present’, and ‘specified’). However, certain adjectives, such as ‘several’, did not conform to this pattern.

**Table 6** S2F accuracy

| Pattern code       | Fragment | Correct | Accuracy (%) |
|--------------------|----------|---------|--------------|
| NP_BE_BINARY_VALUE | 623      | 616     | 98.88        |
| NP_BE_VALUE        | 108      | 106     | 98.15        |
| NP_CD_TO_CD        | 4        | 4       | 100.00       |
| NUM_NP             | 76       | 72      | 94.74        |
| NP_COMP_NP         | 36       | 33      | 91.67        |
| NP_IF_BINARY_VALUE | 4        | 4       | 100.00       |
| NP_IN_VALUESET     | 3        | 3       | 100.00       |
| NP_OF_NP           | 347      | 346     | 99.71        |
| NP_EXIST           | 10       | 10      | 100.00       |
| Total              | 1,211    | 1,194   | 98.60        |

### 4.1.3 Negative fragments filtering

S2F extracted 833 *negative* fragments from 474 positive sentences (Table 2, column *Paper/+sent(+cons, -frags)*). It also extracted 25,808 *negative* fragments from the 8,418 negative sentences (Table 2 column *Paper/-sent(-frags)*). Since S2F did not identify any of the defined nine patterns from 5,418 negative sentences, these sentences were not used in our studies for classifying text fragments as data constraints. The classification is performed on S2F-extracted fragments, as a single sentence can contain zero, one, or multiple constraints. Therefore, negative sentences from which S2F extracts no fragment do not provide classifier instances with discourse patterns, operands, or constraint types.

## 4.2 Classifiers

We use four common classifiers, which are frequently used for text classification: Logistic Regression [48], Multinomial Naive Bayes [44], Linear Support Vector Machine [32], and Decision Tree [61]. For each classifier, we describe its classification logic and hyperparameter tuning through grid search over predefined ranges as follows:

- Logistic Regression (LR) predicts the probability of a fragment being constraint or non-constraint by modeling input features as a weighted linear combination of their log-odds (logit). These probabilities are then classified using a threshold (e.g., 0.5) to assign final labels. We use two regularization techniques – *L1 regularization* and *L2 regularization* and 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* [21].
- Multinomial Naive Bayes (NB) models categorical feature distributions (e.g., word frequencies) to estimate constraint/non-constraint probabilities using Bayes' theorem. NB computes posterior class likelihoods by combining prior probabilities with multinomial feature counts, then assigns labels based on maximum posterior prediction. We use a smoothing parameter  $\alpha$  to prevent zero probabilities, with values of 0, 0.005, 0.01, 0.1, 0.5, 1.0, and 2.0.
- Linear Support Vector Machine (SVM) identifies an optimal hyperplane by analyzing training samples closest to the class boundary (support vectors) to maximize separation margin. Classification depends on the fragment's position relative to this hyperplane. We employ

the *L1* and *L2* regularization techniques, and a penalty parameter  $C$  with values of 0.1, 1, and 10.

- Decision Trees (DT) classify fragments as constraint or non-constraint using a hierarchical tree structure. Each internal node in the tree represents a decision based on a feature derived from the input representation. Each branch corresponds to the outcome of the decision, and each leaf node indicates the final class label. We employ *gini impurity* and *entropy*, as well as strategies for determining the best way to split the data at each node of the tree using either *best* or *random*. The maximum depth of the decision tree is set to 6, 8, or 10.

## 4.3 Features and feature encoding

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. To represent unstructured textual features, we compare the use of Term Frequency-Inverse Document Frequency (TF-IDF) and Bidirectional Encoder Representations from Transformers (BERT).

- *Context-independent Transformation with TF-IDF* [14] is a representative traditional statistical approach for converting text into vectors based on word frequencies. This approach calculates the importance of each word in a document relative to its occurrence across a collection. We use *n-grams* [76] 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 textual features. We also use part-of-speech tags (*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 fragment, sentence, and operands. We then apply TF-IDF to transform n-grams and n-PoS tags into a matrix representation of the textual features.
- *Contextualized Embedding Generation with BERT* [17] is a language model built on transformer architecture, trained on a large and diverse corpus, that generates word representations depending on the surrounding context. Each sentence is tokenized and encoded using the pre-trained 'bert-base-cased' model [17], a 12-layer transformer architecture with 768 hidden units and 12 attention heads. The model was pretrained on two large text corpora: BooksCorpus (800 M words) and English

Wikipedia (2.5B words), using masked language modeling (MLM) and next-sentence prediction (NSP) objectives. Sentences are capped at a maximum sequence length of 512 tokens, with padding or truncation applied as needed. BERT generates two outputs: (1) the [CLS] token embedding that provides a context-aware summary of the entire sentence; and (2) a series of token embeddings, where each embedding reflects the meaning of individual words based on their contextual surroundings. We utilize the [CLS] token's final hidden state as the sentence representation embedding, which captures global sentence-level semantics and is frequently utilized in classification tasks [17]. To represent fragments and operands, we extract the token embeddings for constituent words and compute their average. Specifically, we retrieve the embeddings for words in a fragment or operand using their indices from the BERT token embeddings, then calculate the mean to form a 768-dimensional vector representing these features. In this encoding, BERT acts as a frozen feature extractor whose pretrained parameters are not updated, and the fixed contextual vectors are consumed by the traditional classifiers. These vectors complement the sparse TF-IDF representation by capturing meaning that depends on the surrounding sentence.

*Discourse patterns* and *constraint types* are categorical features that can be mapped to numeric values. Each discourse pattern or constraint type is represented as an integer, indicating which pattern or type corresponds to each fragment. Sect. 4.4 describes the separate trainable embedding-layer encoding used by the fine-tuned BERT classifier.

#### 4.4 Fine-tuned BERT classifier

To answer RQ2.2, we also fine-tune `bert-base-cased` [17] for the same binary classification task. The fine-tuned classifier uses the same sentence, fragment, operand, discourse pattern, and constraint type features described in Sect. 4.3. The sentence, fragment, and operand vectors are obtained using the same contextualized BERT encoding approach described above. Fine-tuning changes the classifier in three ways. First, instead of using BERT as a fixed encoder to generate text embeddings for the traditional machine learning classifiers, we fine-tune BERT by updating its parameters during classifier training. Second, the categorical features are also trained through embedding matrices, as described below. Third, the traditional downstream classifier is replaced by a neural classification head attached to BERT. The RQ2.1 classifiers are trained on feature vectors after BERT has already generated them, which makes BERT a fixed feature generator in that setting.

For fine-tuning, the classifier must be trained together with BERT so the prediction error can update both the classifier weights and the BERT parameters. Therefore, the fine-tuned model concatenates the feature representations for each experimental feature set and passes the concatenated vector to a neural classification head. The classification head contains a fully connected layer with 256 hidden units, a ReLU activation, a dropout layer with rate 0.1, and a final output layer for the two binary classes.

The fine-tuned classifier encodes categorical features as learned vector representations. For each categorical feature, we map each possible category value to an integer ID. The model uses the integer ID as a lookup key into a trainable embedding matrix. For a categorical feature with  $K$  possible values, the embedding matrix has  $K$  rows, and row  $i$  is the vector representation for category ID  $i$ . In our implementation, each row has length 8. We use this small embedding size as a practical setting because the categorical features have few possible values and do not require high-dimensional representations. At the beginning of training, the entries in the matrix are randomly initialized. During training, instances that contain a category use that category's row in the forward pass, and backpropagation updates the row values according to the classification loss. In this way, discourse patterns and constraint types become learned vector representations rather than fixed integer codes. The learned categorical vectors are therefore optimized jointly with the BERT encoder and the classification head.

#### 4.5 Training, testing, and measures

For each of the four classifiers, we shuffle the dataset randomly and select data from 70% of the sentences for training and parameter calibration and 30% for testing to measure model performance by avoiding over-fitting [20]. 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 follow the next three steps:

- The 70% of the data set is randomly split into ten equal folds by preserving the proportion of samples in each fold.
- GridSearchCV [57] creates a grid of all possible combinations of pre-defined hyperparameters. For each combination of hyperparameters in the predefined grid, the learner is trained using nine of the subsets as training data, and validated on the tenth fold.
- 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 Sect. 4.7). The average performance of the learner across 5 experiments is computed for each combination of features and data balancing strategies.

For the fine-tuned BERT classifier, we use the same 70-30 train-test split and repeat the experiment five times. Within the 70% training portion, we reserve 10% as a validation set and use the remaining 90% to update the model weights. After each training epoch, we evaluate the model on the validation set and keep the model state with the highest validation F1 score. We then evaluate that selected model on the preserved 30% test set. We evaluate the fine-tuned bert-base-cased model with every combination of the five features described in Sect. 4.3, resulting in 31 combinations. For each feature combination, we use batch size 16 and maximum sequence length 128, and we compare three epoch settings: 3, 5, and 10 epochs. To account for the strong class imbalance in the dataset, we use weighted cross-entropy loss. For each split, the non-constraint class receives weight 1.0, and the data-constraint class receives a weight computed from the subtraining set as the number of non-constraint fragments divided by the number of data-constraint fragments. The weighting scheme makes positive errors more costly in proportion to the imbalance observed in the training data. In the full dataset, the class-imbalance ratio is approximately 48.79 non-constraint fragments per data-constraint fragment.

## 4.6 Data balancing

To tackle the imbalanced dataset—the number of constraints (*i.e.*, 546 positive fragments) is much less than tool-generated fragments (*i.e.*, 26,641 negative fragments)—we use three data balancing techniques during the training process:

- a. Duplicating instances from the minority class (data constraints) to balance the class distribution.
- b. Random sampling of instances from the majority class (negative fragments) to mitigate its dominance.
- c. Intelligent over-sampling strategies provided by the imbalanced-learn Python library [39].

**Duplicating data constraints.** Given the significant variance observed in the distribution of discourse patterns in our data, spanning from 10 (NP\_EXIST) to 301 (NP\_BE\_BINARY\_VALUE), we duplicate instances of data constraints based on discourse patterns rather than random duplication. This duplication aims to ensure a minimum

number of constraints for each discourse pattern in the training set, with thresholds set at 50, 100, and 200 instances.

For each 70-30 split and threshold, the positive instances in the training set are grouped by their discourse patterns, and the frequency of each pattern is counted. For patterns with a count below the predefined threshold, additional instances are generated by sampling with replacement from the existing instances of that pattern to meet the threshold. These sampled instances are then added to the training dataset. The 70-30 split and duplicating process were repeated 25 times, and the evaluation results were obtained by averaging the results of these runs.

**Sampling Negative Fragments.** The positive ratio in our data, calculated by dividing the 546 constraints by the sum of constraints and 26,641 negative fragment instances, is to 2.01%. We sample *negative* fragments to achieve higher positive ratios of: 3%, 10%, 20%, and 50%.

For each 70-30 split, we initially compute the positive ratio for the training set. The positive ratio is calculated as the proportion of positive instances to the total number of instances in the training set. Then, we randomly sample negative fragments from the negative subset of the training data to achieve the desired positive ratio. Similar to duplication, the 70-30 split and sampling process was repeated 25 times, and the results were averaged for evaluation.

**Intelligent Over-Sampling Techniques.** We used four oversampling techniques from the imbalanced-learn library: the Synthetic Minority Over-sampling Technique (SMOTE) [13]; the Support Vector Machine SMOTE (SVMSMOTE) [52]; the Borderline SMOTE (BorderlineSMOTE) [30]; and the Random Over-Sampling (RandomOverSampler) [39].

SMOTE is a well-known oversampling technique in the field of machine learning that generates synthetic samples for the minority class by interpolating new instances between existing nearest neighbors. BorderlineSMOTE [30] and SVMSMOTE [52] are specialized variants of SMOTE designed to identify samples for resampling. RandomOverSampler [39] is a technique in which the minority class is over-sampled by randomly duplicating instances from the minority class until the class distribution is balanced. Unlike SMOTE and its variants, RandomOverSampler does not generate synthetic samples as it merely duplicates existing instances.

During the training process, these four over-sampling techniques yield positives (minority class) accounting for the negatives (majority class) by a ratio of either 0 (when sampling strategy is not used), 0.33, 0.66, or 1. In other words, for a sampling 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: precision, recall, and F1 score [20]. 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 that are correctly classified out of the total number of constraints in the reference dataset (*i.e.*,  $Recall = TP / (TP + FN)$ ). F1 score is the harmonic mean of the precision and recall (*i.e.*,  $F1 = 2 * (precision * recall) / (precision + recall)$ ).

## 4.8 Results

In this section, we present the results and answer the research questions defined earlier. The experimental results and datasets are available in the replication package [2].

We evaluated each of the four classifiers with every combination of the five features (fragments, sentences, discourse patterns, operands, and constraint types), resulting in 31 combinations. The Multinomial Naive Bayes (NB) classifier cannot produce results with BERT embeddings or intelligent oversampling techniques because it requires features to be non-negative integer counts. BERT embeddings contain real-valued vectors with negative values, and intelligent oversampling techniques may generate continuous or negative values, both of which violate NB's assumptions. We analyze the results of each classifier, using the three data balancing techniques described before. For all classification performance evaluations, we focus on the maximum F1 scores because they are less sensitive to class imbalance and provide a balanced consideration of precision and recall [29].

### 4.8.1 RQ2.1: TF-IDF vs BERT in feature encoding

Fig. 2 presents the maximum F1 scores achieved by various learners using two feature encoding approaches: TF-IDF and BERT without data balancing. We observe that BERT leads to better classification performance (in terms of F1) than TF-IDF for both the Logistic Regression (LR) and Linear Support Vector Machine (LSVM) learners, rising from 52.10% to 62.34% and from 47.22% to 62.04%, respectively. This improvement can be attributed to BERT's context-aware embeddings, which capture contextual details in natural language, helping learners interpret complex sentence structures and word meanings more accurately. For the Decision Tree (DT) learner, using BERT leads to lower F1 score (*i.e.*, from 39.74% to 34.21%). Similar trends have been observed in text classification tasks, where Decision Trees performed better with linguistic features (e.g., TF-IDF) compared to embeddings (e.g., BERT) [26, 68]. One possible explanation for this trend is that Decision Trees rely on discrete decision boundaries, which align more effectively with the sparse, high-dimensional feature space of TF-IDF. In contrast, BERT embeddings capture dense, contextual information that may introduce complexity not well-suited for the hierarchical splitting criterion in tree-based algorithms.

### 4.8.2 RQ2.2: Fine-tuned BERT classifier

We compare the best traditional machine learning configuration from RQ2.1 with a fine-tuned bert-base-cased classifier. The fine-tuning experiment uses the BERT model and evaluates every combination of the five features, resulting in 31 combinations, under three epoch settings: 3, 5, and 10 epochs. Table 7 reports the ten configurations with the highest average F1 scores across the five repeated train-test splits. The best configuration uses all five feature groups and 10 training epochs, achieving 61.31% F1. The remaining top

**Fig. 2** Maximum F1 scores for TF-IDF and BERT feature encodings

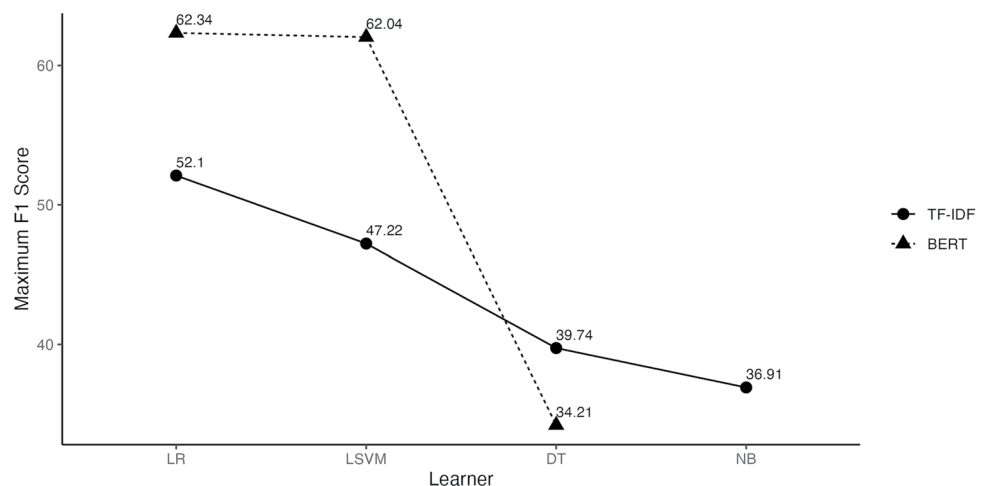


**Table 7** Top ten fine-tuned bert-base-cased configurations by average F1

| Rank | Feature set            | Epochs | Precision | Recall | F1           |
|------|------------------------|--------|-----------|--------|--------------|
| 1    | frag+ops+sent+pat+type | 10     | 61.38     | 61.48  | <b>61.31</b> |
| 2    | frag+ops+sent+type     | 3      | 59.63     | 63.13  | 61.19        |
| 3    | ops+type               | 3      | 62.29     | 61.10  | 61.07        |
| 4    | ops+sent+pat           | 3      | 64.45     | 57.98  | 60.78        |
| 5    | frag+ops               | 3      | 57.52     | 65.38  | 60.75        |
| 6    | ops+sent               | 10     | 61.79     | 60.11  | 60.62        |
| 7    | frag+sent+pat          | 3      | 57.83     | 63.95  | 60.51        |
| 8    | frag+sent              | 3      | 56.31     | 65.34  | 60.37        |
| 9    | ops+type               | 10     | 61.99     | 58.98  | 60.32        |
| 10   | frag+ops+pat           | 3      | 58.69     | 62.75  | 60.18        |

**Table 8** Average F1 scores on the test set for each learner after excluding each feature from the training set using BERT embeddings

| Learner | Excluded Feature  | F1 Score(Diff) |
|---------|-------------------|----------------|
| LR      | None (base trial) | 49.86          |
|         | operands          | 40.59(−9.27)   |
|         | fragment          | 40.62(−9.24)   |
|         | sentence          | 47.80(−2.06)   |
|         | discourse pattern | 50.21(+0.35)   |
|         | constraint type   | 51.65(+1.79)   |
| SVM     | None (base trial) | 49.68          |
|         | fragment          | 39.63(−10.05)  |
|         | operands          | 40.86(−8.82)   |
|         | sentence          | 48.35(−1.33)   |
|         | discourse pattern | 49.57(−0.11)   |
|         | constraint type   | 51.17(+1.49)   |
| DT      | None (base trial) | 25.29          |
|         | operands          | 21.28(−4.01)   |
|         | fragments         | 22.47(−2.82)   |
|         | discourse pattern | 24.08(−1.21)   |
|         | sentence          | 25.40(+0.11)   |
|         | constraint type   | 25.69(+0.40)   |

configurations are close, ranging from 60.18% to 61.19% F1, and they use either 3 or 10 epochs. Across the top ten configurations, operands appear in eight configurations, fragments in six, sentences in six, discourse patterns in four, and constraint types in four. The discourse-pattern and constraint-type features appear in several high-performing configurations, but only in combination with fragment, operand, or sentence features. Overall, fine-tuned BERT does not improve the average F1 over the best frozen-BERT traditional classifier from RQ2.1, where Logistic Regression with BERT embeddings achieved 62.34% F1. Thus, using BERT as a frozen encoder with Logistic Regression remains the best-performing machine learning strategy in this study.

#### 4.8.3 RQ2.3: ablation study on feature selection

We conduct feature ablation studies for each learner using BERT embeddings to assess the impact of individual features on learner performance, quantified by the F1 score.

Feature ablation, a specific type of ablation study, involves systematically removing one or more features from the learner observing the results changes in performance [66]. For each learner, we document the F1 scores before and after excluding each feature, as shown in Table 8, except for the Multinomial Naive Bayes (NB) classifier, which cannot handle BERT embeddings or intelligent oversampling techniques due to its requirement for non-negative integer count features. For Logistic Regression (LR), excluding the fragment and operands features resulted in substantial drops in F1 score from 49.86% to 40.59% and 40.62%, respectively. In contrast, removing the constraint type improved performance increasing to 51.65%. A similar trend was observed for both linear Support Vector Machine (LSVM) and Decision Tree (DT), where the exclusion of fragment and operands features led to drops in F1 score, while the removal of constraint type led to an improvement.

These findings indicate that the *fragment* and *operands* features, identified by the S2F tool, are critical for constraint classification in traditional machine learning approaches. The *constraint type* feature provides no benefit and may even detract from performance.

#### 4.8.4 RQ2.4: data balancing techniques

We discuss the results from evaluating the effect of balancing techniques on data constraint classification.

**Duplicating Data Constraints.** Table 9 shows the maximum F1 values obtained across the 31 feature combinations for one learner, one feature encoding, and one duplication threshold. Bold values indicate the maximum F1 score for each learner. We observe that duplicating constraints results in lower F1 scores compared to no duplication (N/A) across all learners and feature encodings.

**Sampling Negative Fragments.** Table 10 presents the results achieved from sampling negative fragments using five positive ratios, across four learners and 31 feature combinations. Bold values indicate the maximum F1 score for each learner. The results show that sampling negative fragments yields lower F1 scores compared to no sampling (N/A) across all learners and feature encodings.

**Table 9** Maximum F1 scores obtained by duplicating constraints

| Learner | TF-IDF       |       |       |       | BERT         |       |       |       |
|---------|--------------|-------|-------|-------|--------------|-------|-------|-------|
|         | N/A          | 50    | 100   | 200   | N/A          | 50    | 100   | 200   |
| LR      | 52.10        | 25.27 | 26.63 | 25.8  | <b>62.34</b> | 60.96 | 60.61 | 60.36 |
| LSVM    | 47.22        | 23.63 | 25.58 | 26.97 | <b>62.04</b> | 60.84 | 60.04 | 58.28 |
| DT      | <b>39.74</b> | 31.8  | 33.33 | 31.4  | 34.21        | 30.15 | 28.48 | 27.73 |
| NB      | <b>36.91</b> | 22.12 | 21.62 | 20.39 | –            | –     | –     | –     |

**Table 10** Maximum F1 scores obtained by sampling negative fragments

| Learner | TF-IDF       |       |       |       |       | BERT         |       |       |       |       |
|---------|--------------|-------|-------|-------|-------|--------------|-------|-------|-------|-------|
|         | N/A          | 0.03  | 0.1   | 0.2   | 0.5   | N/A          | 0.03  | 0.1   | 0.2   | 0.5   |
| LR      | 52.10        | 30.81 | 32.75 | 24.61 | 10.97 | <b>62.34</b> | 60.27 | 55.74 | 46.67 | 29.21 |
| LSVM    | 47.22        | 25.14 | 33.68 | 25.66 | 11.3  | <b>62.04</b> | 60.06 | 52.69 | 44.08 | 28.38 |
| DT      | <b>39.74</b> | 31.78 | 27.6  | 20.48 | 10    | 34.21        | 32.11 | 26.26 | 19.44 | 12.06 |
| NB      | <b>36.91</b> | 20.06 | 18.01 | 12.65 | 3.46  | –            | –     | –     | –     | –     |

**Table 11** Maximum F1 scores using intelligent over-sampling techniques

| Learner | Technique         | TF-IDF       |       |       |       | BERT         |       |       |       |
|---------|-------------------|--------------|-------|-------|-------|--------------|-------|-------|-------|
|         |                   | N/A          | 0.33  | 0.66  | 1.0   | N/A          | 0.33  | 0.66  | 1.0   |
| LR      | SMOTE             | 52.10        | 49.45 | 47.9  | 45.99 | <b>62.34</b> | 60.9  | 59.93 | 58.4  |
|         | SVMSMOTE          |              | 48.94 | 48.64 | 46.11 |              | 60.01 | 59.47 | 58.78 |
|         | BorderlineSMOTE   |              | 48.48 | 46.5  | 47.06 |              | 61.71 | 59.04 | 59.85 |
|         | RandomOverSampler |              | 48.08 | 45.29 | 43.86 |              | 60.79 | 59.74 | 58.6  |
| LSVM    | SMOTE             | 47.22        | 46.79 | 45.41 | 44.58 | <b>62.04</b> | 60.15 | 60.6  | 59.43 |
|         | SVMSMOTE          |              | 48.03 | 47.31 | 47.09 |              | 60.55 | 60.04 | 57.73 |
|         | BorderlineSMOTE   |              | 48.69 | 47.39 | 45.6  |              | 60.29 | 61.44 | 59.27 |
|         | RandomOverSampler |              | 46.44 | 43.69 | 44.86 |              | 57.48 | 57.46 | 56.38 |
| DT      | SMOTE             | <b>39.74</b> | 34.18 | 31.86 | 30.24 | 34.21        | 30.64 | 28.1  | 27.12 |
|         | SVMSMOTE          |              | 35.75 | 33.66 | 33.3  |              | 30.93 | 29.03 | 28.47 |
|         | BorderlineSMOTE   |              | 33.07 | 30.84 | 29.47 |              | 31.13 | 28.34 | 26.92 |
|         | RandomOverSampler |              | 30.36 | 27.08 | 26.74 |              | 31.08 | 28.94 | 27.12 |
| NB      | SMOTE             | <b>36.91</b> | 31.21 | 32.67 | 32.63 | –            | –     | –     | –     |
|         | SVMSMOTE          |              | –     | –     | –     |              | –     | –     | –     |
|         | BorderlineSMOTE   |              | 32.22 | 31.04 | 31.95 |              | –     | –     | –     |
|         | RandomOverSampler |              | 32.02 | 32.44 | 31.2  |              | –     | –     | –     |

**Intelligent Over-Sampling Techniques.** Table 11 presents the maximum F1 scores obtained from 31 feature combinations. Each combination was evaluated using three ratios (i.e., 0.33, 0.66, and 1) as well as a baseline condition without oversampling (N/A). The ratio represents the balance between minority and majority class samples after resampling, and a ratio of 1 indicates equal numbers of samples in both classes. No results are reported for the Multinomial Naive Bayes (NB) learner with SVMSMOTE due to a technical limitation: the NB algorithm cannot process input data containing negative values. However, the SVMSMOTE oversampling strategy generates synthetic data vectors that include negative values. Bold values indicate the maximum F1 score for each learner.

The results indicate that none of the applied sampling strategies improved the performance of any classifiers, and no data balancing technique used with the traditional

ML-based approach for constraint classification performed best for any of the feature encoding methods.

#### 4.9 Threats to validity

We discuss the main threats to the validity of our findings and the mitigation strategies we adopted for these threats.

**Construct Validity.** The removal of 5,418 negative sentences that did not contain any of the nine defined patterns identified by S2F may have affected the results of our studies. These sentences could have contributed to a more diverse dataset for our traditional machine learning-based approach, or they may have contained fragments that were overlooked. However, these sentences were excluded because they did not contain fragments that followed the defined discourse patterns. Since these patterns are essential for identifying key attributes that serve as features in ML-based approaches, their absence could affect the evaluation

and analysis of these features' impact on classification performance. Moreover, no fragments were identified by the authors that fell outside the defined discourse patterns during the evaluation of S2F, which reduces the likelihood that important fragments were overlooked.

**Internal Validity.** We tackled selection bias by using stratified splitting to maintain balanced distributions of positive and negative sentences. To prevent data leakage, we applied data balancing methods only to the training set. To handle class imbalance, we applied oversampling techniques, including SMOTE and simple duplication of minority class samples, as well as undersampling for the majority class. We also used GridSearchCV with k-fold cross-validation and repeated the process five times for stability. To further ensure the validity of feature selection, we conducted an ablation study by systematically removing individual features and evaluating their impact on model performance.

**Construct Validity.** In order to answer our research questions related to the use of machine learning technique to classify natural language sentence fragments and the importance of the used features, data balancing, and embedding, we utilized standard measures used in similar classification works, namely: precision, recall, and F1. Since we are not favoring neither precision nor recall in our classification task, we use F1 as the main measure to determine the performance of a particular machine learning technique and/or configuration.

**External Validity.** The conclusions drawn from this study are dependent on the specific datasets collected and the use of traditional MLs. Given the distinct characteristics, complexities, and interrelationships of these datasets, as well as the inherent limitations and potential biases of the model employed, the findings may not fully capture the broader range of challenges that could arise in other requirements extraction and analysis tasks. To mitigate this threat, we used diverse datasets from nine real-world Java systems, covering a variety of software domains and documentation types, including libraries, tools, and frameworks across different application areas. For the ML models, we used commonly applied classifiers in text classification, including logistic regression, linear SVM, decision trees, and multinomial Naive Bayes.

## 5 Related work

We discuss related research along three themes: data constraints in software, information extraction from natural language requirements, and classification of natural language requirements.

**Data Constraints in Software.** Prior work focused on the implementation of data constraints. These works show

that such restrictions matter for correctness, maintenance, and runtime checking. Yang et al. [82] studied constraints on data values in database-backed web applications. The constraints include presence, length, numericality, uniqueness, inclusion, exclusion, and format restrictions. Their work examines how these constraints are specified and enforced across the front-end, application, and database layers. They found that developers struggle to keep constraints consistent across layers, and that inconsistencies can lead to reliability and usability problems. Florez et al. [23] conducted an empirical study of data constraint implementations in Java, analyzing 187 constraints from eight real-world systems and identifying 31 distinct implementation patterns. Their study further found that nearly half of all constraints are enforced in multiple code locations, resulting in code clones that complicate maintenance. Building on these findings, Florez et al. [24] introduced Lasso, the first technique to automatically retrieve the method and line of code where a given data constraint is enforced, improving method-level link recovery over text-retrieval baselines by up to 163%.

More recent work has examined data-related correctness problems in large-scale FinTech systems, where constraint violations can have serious consequences. Wang et al. [75] address the problem of semantically equivalent constraints being redundantly specified in production systems and propose a decision procedure to detect such equivalences efficiently. Wang et al. [77] mine arithmetic equality constraints from runtime database records and generate SQL reconciliation scripts to detect fund loss violations automatically. Tange et al. [73] study how formally specified constraints in FinTech systems evolve and propose a traceability analysis to identify code methods that may need updates when a constraint changes. In contrast to these works, our work focuses on the extraction of data constraints from natural language software documents.

**Information Extraction from Natural Language Software Documents.** Many prior research have shown that vocabulary, linguistic, and discourse patterns are useful signals for analyzing software artifacts. These patterns have been studied in bug reports [10, 19, 35, 70, 72], requirement documents [4, 53, 80], content classification [19, 43], and query formulation [41]. This body of research shows that recurring linguistic structures can help identify specific types of structured content in natural language software text.

A closely related group of work identifies rule-based statements in software documents by combining grammatical preprocessing with human defined sentence patterns. Xiao et al. [80] proposed Text2Policy to identify access control policy sentences. Their method uses POS tagging and finite-state phrase chunking to label grammatical roles, including subjects, main verb groups, and objects, and then matches the labeled structures against semantic patterns.

Pandita et al. [54] used a similar process for API documents by tagging words, grouping them into phrases using cascading finite-state rules, and applying semantic templates to infer formal method specifications. These works show that sentence grammar and human defined patterns can support rule identification in software documents. However, they focus on different kinds of rules. Access control work looks for who may perform an action on which resource, while API specification work looks for what must be true before or after a method is called. These rules are often expressed through relatively predictable language, such as modal verbs, permission expressions, null checks, and range checks. Our work, in contrast, looks for fragments that say what values a data item may, must, or must not have. Such fragments describe relationships among data items, restrict valid values, define required states, or assign specific values under particular conditions. Therefore, patterns designed to identify permissions or method conditions cannot be directly transferred to value restrictions over data items.

More broadly related requirements engineering work also use NLP to support automated analysis of requirements, but they target different kinds of information from ours. Arora et al. [4] used NLP pattern matchers to detect whether requirements conform to predefined structural templates. Their approach assumes known requirement templates, while value restrictions over data items can be written in many sentence forms. Nikora et al. [53] used POS tagging and machine learning to classify requirements by Linear Temporal Logic pattern type. Their approach is tailored to temporal requirements, where the relevant language is organized around event ordering and sequencing. Our setting requires different linguistic forms, including comparison, membership, and assignment expressions. Zhang et al. [84] used NLP and machine learning to extract and rank domain terminology from requirements for feature modeling. Their approach identifies noun phrases that name domain concepts. Our work instead identifies sentence fragments that restrict what values a data item may, must, or must not take. Thus, terminology extraction does not capture the value restriction fragments studied in our work.

Overall, prior work demonstrates that NLP techniques can identify structured information in requirements when the relevant natural language patterns are well-defined. However, many types of requirements remain difficult to extract because their linguistic expression has not been systematically characterized. Our work addresses one instance of this broader challenge by empirically deriving discourse patterns for data constraints, providing a basis for automated identification and a model for studying other forms of constraint knowledge in requirements documents.

**Classification of Natural Language Software Artifacts.** Classifying natural language content in software artifacts

has been widely studied across a range of tasks. Prior work classifies a wide range of text units: entire artifacts such as bug reports [3, 34, 36], app reviews [28, 42, 47, 55], and issues [8, 27, 74, 81, 85], as well as finer-grained ones such as individual requirement statements [1, 6, 7, 12, 16, 22, 38, 69] and sentences within development emails [18, 19], bug reports [9, 10, 70], issues [5], and source code [56, 63, 67, 83]. Training data is typically obtained by manually labeling instances from existing artifacts, though some recent work reduces this cost through weak supervision [12] or natural-language-inference-based zero-shot transfer [22]. The dominant features are lexical, primarily n-grams and TF-IDF representations [1, 6, 36, 38, 56, 60, 63, 83], often complemented by syntactic features such as part-of-speech tags [5, 10, 16, 70] and dependency-based linguistic patterns [1, 10, 16, 19]. Contextual embeddings such as BERT have more recently been adopted to better capture semantic variation across writing styles and domains [9, 11, 22, 45, 46].

The dominant learners are traditional classifiers such as SVMs, Naive Bayes, and Random Forests [1, 5, 6, 10, 16, 19, 33, 36–38, 51, 56, 60, 63, 64, 67, 69, 70, 83], which work well even with moderately sized labeled datasets. Deep learning models are less common precisely because they require substantially more labeled data [9, 11, 22, 40, 45, 46], and class imbalance between positive and negative instances is a recurring challenge that prior work addresses through resampling and reweighting strategies [1, 12, 36].

Several works go beyond flat category labels. Chaparro et al. [9, 10] use discourse patterns to identify sentences describing observed behavior, expected behavior, or steps to reproduce in bug reports and additionally extract individual reproduction steps as structured action-object tuples. Frattini et al. [25] classify requirement sentences as causal or non-causal and extract the embedded cause-effect structure. Our classification approach applies this established paradigm directly. We use commonly-used traditional ML classifiers that combine lexical features with BERT embeddings, motivated by their effectiveness in prior work. We also explicitly address class imbalance, since text describing data constraints is rare relative to text that does not describe constraints in software documents. The discourse patterns we derive are incorporated as features, similar as prior pattern-based approaches [10, 19]. What has not been studied before is data constraint classification. The artifacts prior work classifies are often relatively long and rich in context, carrying enough lexical evidence for keywords and n-grams to discriminate categories. Data constraint descriptions, by contrast, are short and highly compressed. A constraint can be just a few words, leaving keywords and n-grams with little evidence to use. What matters is how the data element, operator, and value work together to describe a constraint, rather than the individual words alone.

The classifier therefore needs representations that capture both the candidate fragment and the operands it contains. Our results support this explanation, fragment and operand features contribute the most to classification, while BERT representations outperform context-independent encodings by using the surrounding sentence context to interpret short constraint descriptions.

## 6 Conclusions

We found that data constraints described in natural language software documents follow patterns that are common across different systems and types of documents. By analyzing the natural language descriptions of 546 explicit data constraints from the documentation of nine systems, we identified nine linguistic discourse patterns commonly used to describe such constraints. Based on these patterns, we developed a tool (S2F) to automatically identify and extract sentence fragments corresponding to explicit data constraint descriptions, along with their operands and constraint types. We further learned that the data constraints pattern, their operands and types are useful features to be used for the automated classification of text fragments into data constraints, using traditional machine learning techniques. The best-performing traditional ML classifier (*i.e.*, Logistic Regression), using BERT embeddings, achieved 63.63% precision, 61.40% recall (62.34% F1), with Linear Supporting Vector Machine a close second. Fragment and operands are the most important features for ML-based constraint classification, and removing the constraint type feature led to improved performance. However, data balancing techniques did not improve the results of the ML-based approach. These findings are a promising step towards the automated identification and extraction of data constraints from natural language text, essential for automating traceability and test generation.

### 6.1 Limitations and implications

Our work focuses on explicit data constraint descriptions that can be interpreted from a contiguous fragment within a single sentence. We excluded constraints whose interpretation depends on information spread across multiple sentences, constraints that refer to data elements or values through pronouns such as *it* or *them*, and constraints expressed through GUI behavior or screen context. These excluded cases are still important in practice, but they require additional modeling of information spread across sentences, pronoun references, or user-interface context beyond the fragment-level linguistic patterns studied here.

Our approach can also be affected by NLP preprocessing errors. S2F uses PoS tags, noun-phrase boundaries, and dependency information to identify candidate fragments and operands. Errors in these preprocessing steps can lead to missed fragments, incorrect operands, or incorrect discourse-pattern assignments.

In addition, although the studied systems cover different software domains and document types, the discourse-pattern catalog was derived from a limited corpus and may need to be extended for other domains, writing styles, or documentation conventions. We therefore interpret the reported performance as evidence for the feasibility of identifying explicit data constraint descriptions in the studied setting, not as complete solution for all possible forms of data constraints in software documents.

Despite these limitations, automated identification of data constraints can support several RE and SE activities. Data constraints describe what values software data may, must, or must not take. Once identified, they can provide concrete units for traceability, compliance analysis, change analysis, verification, and testing. For traceability maintenance and change impact analysis, extracted constraints can help identify which documented rules should be linked to source code. For requirements verification and validation, they can help analysts inspect whether documented value restrictions are missing, conflicting, redundant, or compliant with regulations. For testing and input-validation audits, constraint operands can suggest boundary values, allowed-value sets, required fields, and invalid inputs that should be checked.

### 6.2 Future work

Future work should extend the current approach to the special excluded constraint cases described earlier. Constraints that span multiple sentences require methods that can connect the data element, value, and restriction when they are described across more than one sentence. Constraints that use pronouns such as *it* or *them* require reference resolution to identify the constrained data element or value [58, 62]. GUI-related constraints require methods that can connect textual descriptions with user-interface context and interaction behavior. We also plan to investigate AI-based identification and extraction methods beyond the frozen BERT feature encoding used in this study, such as prompt-based generative models that can analyze a sentence together with its surrounding text. Another future direction is to evaluate how the identified constraints can support downstream RE tasks, such as tracing data constraints to their implementation code, analyzing the impact of constraint changes, and deriving tests from extracted constraint operands.

**Supplementary Information** The online version contains supplementary material available at <https://doi.org/10.1007/s00766-026-00472-x>.

**Author contributions** All authors wrote the main manuscript text and reviewed the manuscript.

**Data availability** The experimental results and datasets are available in the replication package [2].

## Declarations

**Conflict of interest** The authors declare no competing interests.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Abad Z, Karras O, Ghazi P et al (2017) What works better? a study of classifying requirements. Proc. the IEEE 25th Int. Requirements Engineering Conf. (RE). pp 496–501
- Zhou Y, Miao M, Birsan V et al (2026) Towards the automated identification of data constraints in software documents (Replication Package).
- Antoniol G, Ayari K, Di Penta M et al (2008) Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. Proc. the Conf. of the Center for Advanced Studies on Collaborative Research: Meeting of Minds. pp 304–318
- Arora C, Sabetzadeh M, Goknil A et al (2015) Change impact analysis for natural language requirements: An nlp approach. In: 2015 IEEE 23rd International Requirements Engineering Conference (RE), IEEE, pp 6–15
- Arya D, Wang W, Guo JL et al (2019) Analysis and detection of information types of open source software issue discussions. 2019 IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE). pp 454–464
- Baker C, Deng L, Chakraborty S et al (2019) Automatic multi-class non-functional software requirements classification using neural networks. Proc. the IEEE 43rd annual computer software and applications conference (COMPSAC'19). pp 610–615
- Batool R, Naseer A, Maqbool A et al (2025) Automated categorization of software security requirements: an nlp and ml based approach. Requirements Eng 30(1):133–145. <https://doi.org/10.1007/s00766-025-00443-8>
- Beyer S, Macho C, Pinzger M et al (2018) Automatically classifying posts into question categories on stack overflow. Proc. the 26th Conf. on Program Comprehension. pp 211–221
- Chaparro O, Bernal-Cárdenas C, Lu J et al (2019) Assessing the quality of the steps to reproduce in bug reports. In: Proc. the 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'19), pp 86–96
- Chaparro O, Liu J, Zampetti F et al (2017) Detecting missing information in bug descriptions. Proc. of the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE). pp 396–407
- Chatterjee P, Damevski K, Pollock L (2021a) Automatic extraction of opinion-based q&a from online developer chats. In: 2021 IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE), pp 1260–1272
- Chatterjee R, Ahmed A, Rose Anish P, et al (2021b) A pipeline for automating labeling to prediction in classification of nfrs. In: 2021 IEEE 29th International Requirements Engineering Engineering Conference (RE), pp 323–323
- Chawla NV, Bowyer KW, Hall LO et al (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357
- Chowdhury GG (2010) Introduction to modern information retrieval. Facet publishing
- Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Measur 20(1):37–46
- Dalpiatz F, Dell'Anna D, Aydemir FB et al (2019) Requirements classification with interpretable machine learning and dependency parsing. In: Proc. the IEEE 27th Int. Requirements Engineering Conf. (RE'2019), pp 142–152
- Devlin J, Chang MW, Lee K et al (2019) Bert: Pre-training of deep bidirectional transformers for language understanding. Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT). pp 4171–4186
- Sorbo D, Panichella A, Visaggio S, CA et al (2015) Development Emails Content Analyzer: Intention Mining in Developer Discussions (T). Proc. the Int. Conf. on Automated Software Engineering (ASE'15). pp 12–23
- Di Sorbo A, Panichella S, Visaggio CA et al (2021) Exploiting natural language structures in software informal documentation. IEEE Trans Software Eng 47(8):1587–1604
- Dougherty G (2012) Pattern recognition and classification: an introduction
- Fan RE, Chang KW, Hsieh CJ et al (2008) Liblinear: a library for large linear classification. J Mach Learn Res 9:1871–1874
- Fazelnia M, Kosciński V, Herzog S et al (2024) Lessons from the use of natural language inference (nli) in requirements engineering tasks. Proc. of the 32nd IEEE International Requirements Engineering Conference. IEEE,(RE'24)
- Florez JM, Moreno L, Zhang Z et al (2022) An empirical study of data constraint implementations in java. Empir Softw Eng 27(5):119
- Florez JM, Perry J, Wei S et al (2022b) Retrieving data constraint implementations using fine-grained code patterns. In: Proc. of the 44th IEEE/ACM Int. Conf. on Software Engineering (ICSE), pp 1893–1905
- Fratini J, Fischbach J, Mendez D et al (2022) Causality in requirements artifacts: prevalence, detection, and impact. Requirements Eng 28(1):49–7. <https://doi.org/10.1007/s00766-022-00371-x>
- Fulari R, Rusert J (2024) Utilizing machine learning to predict question difficulty and response time for enhanced test construction. Proc. of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2024). pp 528–533
- Guo PJ, Zimmermann T, Nagappan N et al (2010) Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. Proc. the Int. Conf. on Software Engineering (ICSE'10). pp 495–504
- Guzman E, El-Haliby M, Bruegge B (2015) Ensemble methods for app review classification: An approach for software evolution (n). Proc. the 30th IEEE/ACM Int. Conf. on Auto. Software Engineering (ASE'15). pp 771–776

29. Haixiang G, Yijing L, Shang J et al (2017) Learning from class-imbalanced data: review of methods and applications. *Expert Syst Appl* 73:220–239
30. Han H, Wang WY, Mao BH (2005) Borderline-smote: a new over-sampling method in imbalanced data sets learning. *Int. Conf. on intelligent computing*. pp 878–887
31. Honnibal M, Montani I (2018) Industrial-strength natural language processing with python and spacy. *Proc of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* pp 191–200
32. Hsu CW, Lin CJ (2003) A practical guide to support vector classification. In: *Advances in Kernel Methods - Support Vector Learning*, pp 169–208
33. Jiang H, Zhang J, Ren Z et al (2017) An unsupervised approach for discovering relevant tutorial fragments for apis. *Proc. the IEEE/ACM 39th Int. Conf. on Software Engineering (ICSE'17)*. pp 38–48
34. Kallis R, Sorbo D, Canfora AG et al (2019) Ticket tagger: Machine learning driven issue classification. *Proc. the IEEE Int. Conf. on Software Maintenance and Evolution (ICSME'19)*. pp 406–409
35. Ko AJ, Myers BA, Chau DH (2006) A Linguistic Analysis of How People Describe Software Problems. *Visual Languages and Human-Centric Computing (VL/HCC'06)*. pp 127–134
36. Krasniqi R, Do H (2023) A multi-model framework for semantically enhancing detection of quality-related bug report descriptions. *Empir Softw Eng* 28(2):1–62
37. Kumar N, Devanbu P (2016) Ontocat: Automatically categorizing knowledge in api documentation. *arXiv:1607.07602 arXiv preprint*
38. Kurtanović Z, Maalej W (2017) Automatically classifying functional and non-functional requirements using supervised machine learning. *Proc. the IEEE 25th Int. Requirements Engineering Conf. (RE)*. pp 490–495
39. Lemaître G, Nogueira F, Aridas CK (2017) Imbalanced-learn: a python toolbox to tackle the curse of imbalanced datasets in machine learning. *J Mach Learn Res* 18(17):1–5
40. Liu H, Shen M, Jin J et al (2020) Automated classification of actions in bug reports of mobile apps. *Proc. the 29th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*. pp 128–140
41. Liu M, Peng X, Marcus A et al (2022) How to formulate specific how-to questions in software development? *Proc. of the 30th ACM Joint European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*. pp 306–318
42. Maalej W, Nabil H (2015) Bug report, feature request, or simply praise? On automatically classifying app reviews. *Requirements Engineering Conf. (RE), 2015 IEEE 23rd Int.* pp 116–125
43. Maalej W, Robillard MP (2013) Patterns of knowledge in api reference documentation. *IEEE Trans Software Eng* 39(9):1264–1282
44. Rish I (2001) An empirical study of naive bayes classifier. *Proc. of the 14th Conf. on Empirical Methods in Natural Language Processing*. pp 41–50
45. Marques A, Murphy GC (2022) Evaluating the use of semantics for identifying task-relevant textual information. *Proc. the IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER'22)*. pp 240–251
46. Marques A, Viviani G, Murphy GC (2021) Assessing semantic frames to support program comprehension activities. *Proc. the IEEE/ACM 29th Int. Conf. on Program Comprehension (ICPC'21)*. pp 13–24
47. Martin W, Sarro F, Jia Y (2016) A survey of app store analysis for software engineering. *IEEE Trans Software Eng* (99)
48. Menard S (2010) Logistic regression: Why we use it and how it works. *SAS Global Forum*
49. Miles MB, Huberman M, Saldana J (2014) *Qualitative data analysis, a methods sourcebook*. (sage publication)
50. MITRE Corporation (2019) Mitre annotation toolkit (mat). <https://github.com/mitre/mitre-annotation>, accessed: March 29, 2023
51. Monperrus M, Eichberg M, Tekes E et al (2012) What should developers be aware of? an empirical study on the directives of api documentation. *Empir Softw Eng* 17:703–737
52. Nguyen HM, Cooper EW, Kamei K (2011) Borderline over-sampling for imbalanced data classification. *Int J Knowl Eng Soft Data Paradigms* 3(1):4–21
53. Nikora AP, Balcom G (2009) Automated identification of ltl patterns in natural language requirements. *20th International Symposium on Software Reliability Engineering*. IEEE, pp 185–194
54. Pandita R, Xiao X, Zhong H et al (2012) Inferring method specifications from natural language api descriptions. *Proc. the 34th Int. conference on software engineering (ICSE'12)*. pp 815–825
55. Panichella S, Di Sorbo A, Guzman E et al (2015) How can i improve my app? classifying user reviews for software maintenance and evolution. *Proc. of the IEEE Int. Conf. on software maintenance and evolution (ICSME'15)*. pp 281–290
56. Pascarella L, Bruntink M, Bacchelli A (2019) Classifying code comments in java software systems. *Empir Softw Eng* 24(3):1499–1537
57. Pedregosa F, Varoquaux G, Gramfort A et al (2011) Scikit-learn: machine learning in Python. *J Mach Learn Res* 12:2825–2830
58. Peng H, Khashabi D, Roth D (2015) Solving hard coreference problems. In: *Proc. of the 2015 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. pp 809–819
59. Polanyi L (2003) *The Handbook of Discourse Analysis*, vol 18, Wiley-Blackwell, chap The Linguistic Structure of Discourse, p 265
60. Prana GAA, Treude C, Thung F et al (2019) Categorizing the content of github readme files. *Empir Softw Eng* 24:1296–1327
61. Hssina, Badr, Merbouha A, Ezzikouri H, Erritali M (2014) A comparative study of decision tree id3 and c4.5. *International Journal of Advanced Computer Science and Application*. <https://doi.org/10.14569/SpecialIssue.2014.040203>
62. Rahman A, Ng V (2012) Resolving complex cases of definite pronouns: the winograd schema challenge. *Proc. of the 2012 joint conf. on empirical methods in natural language processing and computational natural language learning*. pp 777–789
63. Rani P, Panichella S, Leuenberger M et al (2021) How to identify class comment types? a multi-language approach for class comment classification. *J Syst Softw* 181:111047
64. Robillard MP, Chhetri YB (2015) Recommending reference api documentation. *Empir Softw Eng* 20:1558–1586
65. Roscoe T, Martin A, Shao B et al (2015) Parfait: Designing a scalable bug checker. In: *Proc. of the 2015 USENIX Annual Technical Conf. (USENIX ATC)*, pp 163–177 <https://doi.org/10.1145/2830961.2830973>
66. Sheikholsami S, Meister M, Wang T et al (2021) Autoablation: Automated parallel ablation studies for deep learning. *Proc. of the 1st Workshop on Machine Learning and Systems*. pp 55–61
67. da Silva ME, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans Software Eng* 43(11):1044–1062
68. Sobhy M, El-Atta A, El-Sawy AA et al (2022) Word representation models for arabic dialect identification. *Proc. of the Seventh Arabic Natural Language Processing Workshop (WANLP)*. pp 474–478
69. Sonbol R, Rebdawi G, Ghneim N (2022) The use of nlp-based text representation techniques to support requirement engineering tasks: A systematic mapping review. *IEEE Access*
70. Song Y, Chaparro O (2020) Bee: a tool for structuring and analyzing bug reports. *Proc. the 28th ACM Joint Meeting on European*

- Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'21). pp 1551–1555
71. Stenström AB, Aijmer K (2004) Discourse Patterns in Spoken and Written Corpora, Pragmatics & Beyond New Series, vol 120. John Benjamins Publishing Company, Amsterdam / Philadelphia
  72. Sureka A, Jalote P (2010) Detecting Duplicate Bug Report Using Character N-Gram-Based Features. Proc. the Asia Pacific Software Engineering Conf. (ASPEC'10). pp 366–374
  73. Tange W, Wang C, Yao P et al (2023) Dclink: Bridging data constraint changes and implementations in fintech systems. 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 914–925
  74. Thung F, Lo D, Jiang L (2012) Automatic defect categorization. Proc. the Working Conf. on Reverse Engineering (WCRE'12). pp 205–214
  75. Wang C, Fan G, Yao P, et al (2023a) Verifying data constraint equivalence in fintech systems. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 1329–1341
  76. Wang S, Manning CD (2012) Baselines and bigrams: Simple, good sentiment and topic classification. Proc. of the 50th Annual Meeting of the Association for Computational Linguistics (ACL). pp 90–94
  77. Wang T, Zhi C, Zhou X et al (2023b) Data constraint mining for automatic reconciliation scripts generation. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 1119–1130
  78. Wiegers KE, Beatty J (2013) Software Requirements, 3rd edn. Redmond, WA
  79. Witt GC (2012) Writing Effective Business Rules: A Practical Method. Waltham, MA
  80. Xiao X, Paradkar A, Thummalapenta S et al (2012) Automated extraction of security policies from natural-language software documents. Proc. the ACM SIGSOFT 20th Int. Symposium on the Foundations of Software Engineering. pp 1–11
  81. Xu B, Lo D, Xia X et al (2015) Efspredictor: Predicting configuration bugs with ensemble feature selection. Proc. the Asia-Pacific Software Engineering Conf. (ASPEC'15). pp 206–213
  82. Yang J, Sethi U, Yan C et al (2020) Managing data constraints in database-backed web applications. In: Proc. of the 42nd ACM/IEEE Int. Conf. on Software Engineering (ICSE), pp 1098–1110 <https://doi.org/10.1145/3377811.3380375>
  83. Zhai J, Xu X, Shi Y et al (2020) Cpc: Automatically classifying and propagating natural language comments via program analysis. Proc. the ACM/IEEE 42nd Int. Conf. on Software Engineering (ICSE'20). pp 1359–1371
  84. Zhang J, Chen S, Hua J et al (2022) Automatic terminology extraction and ranking for feature modeling. 2022 IEEE 30th International Requirements Engineering Conference (RE). IEEE, pp 51–63
  85. Zhou Y, Tong Y, Gu R et al (2016) Combining text mining and data mining for bug report classification. J softw Evol Process 28(3):150–176

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.