THESIS

AUTOMATIC DETECTION OF CONSTRAINTS IN SOFTWARE DOCUMENTATION

Submitted by

Joy Ghosh

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2021

Master's Committee:

Advisor: Laura Moreno Cubillos

Sudipto Ghosh
Leo Vijayasarathy

Copyright by Joy Ghosh 2021

All Rights Reserved

ABSTRACT

AUTOMATIC DETECTION OF CONSTRAINTS IN SOFTWARE DOCUMENTATION

Software documentation is an important resource when maintaining and evolving software, as it supports developers in program understanding. To keep it up to date, developers need to verify that all the constraints affected by a change in source code are consistently described in the documentation. The process of detecting all the constraints in the documentation and cross-checking the constraints in the source code is time-consuming. An approach capable of automatically identifying software *constraints* in documentation could facilitate the process of detecting constraints, which are necessary to cross-check documentation and source code.

In this thesis, we explore different machine learning algorithms to build binary classification models that assign sentences extracted from software documentation to one of two categories: *constraints* and *non-constraints*. The models are trained on a data set that consists of 368 manually-tagged sentences from four open-source software systems. We evaluate the performance of the different models (Decision tree, Naive Bayes, Support Vector Machine, Fine-tuned BERT) based on precision, recall and F1-score. Our best model (*i.e.,* a decision tree featuring bigrams) was able to achieve 74.0% precision, 83.8% recall and an F1-score of 0.79. This suggests that our results are promising and that it is possible to build machine learning based models for the automatic detection of constraints in the software documentation.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Laura Moreno, for her continuous guidance and support. Without her supervision, this thesis would not have been completed.

I would also like to thank Dr. Sudipto Ghosh for his input and feedback on the thesis. Thanks to Dr. Leo Vijayasarathy for agreeing to be one of my thesis committee members.

I want to express my deepest gratitude to my wife and parents for their constant support throughout graduate school, and to my friends, who helped me to cope up with my new life in a different country.

DEDICATION

I would like to dedicate this thesis to God Almighty.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
DEDICATION iv
LIST OF TABLES vi
LIST OF FIGURES vii

Chapter 1 Introduction 1

Chapter 2 Methodology 4
2.1 Defining the term "Constraint" 4
2.2 Building the data set 6
2.3 Building constraint classification models 10
2.4 Evaluating the performance of the models 23
2.5 Threats to validity 24

Chapter 3 Results 26
3.1 Testing the performance of the models with two strategies 26
3.2 Strategy 1: Train and test the models on the same data set 27
3.3 Strategy 2: Train and test the models on different data sets 41
3.4 Summary and discussion 44

Chapter 4 Related work 47
4.1 Constraints 47
4.2 Software requirements 52
4.3 Business Rules 57
4.4 Automatic detection and classification of business rules 64
4.5 Summary 68

Chapter 5 Conclusions 69
5.1 Lessons learned 69
5.2 Future Work 70

Bibliography 72

LIST OF TABLES

2.1 Software systems used to build the data set 6
2.2 Composition of the data used in the study. Proportions of constraints and non-constraints with respect to sentences in analyzed section are shown in parenthesis. 8
2.3 Composition of the training and testing data sets for each system. Proportions of constraints and non-constraints with respect to data points in training/testing data set are shown in parenthesis. 9
2.4 System-wise parameters of decision tree models with onegram embedding 12
2.5 System-wise parameters of decision tree models with bigram embedding 13
2.6 System-wise parameters of decision tree models with n-gram (n={1,2}) embedding . . 13
2.7 System-wise parameters of best Naive Bayes models 13
2.8 System-wise parameters of best SVM models 15

3.1 Accuracy, precision, recall and F1-score comparison between different models across all systems for testing strategy 1 28
3.2 Accuracy, precision, recall and F1-score comparison between different models across all systems for testing strategy 2 43

4.1 Partial taxonomy by Maalej and Ghaisas [1]. The sub-classes for which the authors did not mention any definition have been omitted 67
4.2 New types of business rules proposed by Anish et al. [2]. Definitions are verbatim from the original publication. 67

LIST OF FIGURES

2.1 Sample diagram of a decision tree 11
2.2 SVM uses kernel trick to uplift the dimension of data points 15
2.3 Encoding procedure of the BERT algorithm 17
2.4 Example of encoding procedure of the BERT algorithm 17
2.5 Fine-tuned BERT optimized with Adam learning rate 21
2.6 Processing sentences before feeding into models 22

3.1 Top four tokens used by decision tree with bigram in HttpCore data set 36
3.2 Top four tokens used by decision tree with bigram in jEdit data set 37
3.3 Top four tokens used by decision tree with bigram in Swarm data set 37
3.4 Top four tokens used by decision tree with bigram in ArgoUML data set 38
3.5 Top four tokens used by decision tree with bigram in “All Systems” data set 38
3.6 Precision trend of the decision tree model with unigram 46

4.1 Constraint classification by Breaux and Antón [3] 48
4.2 Automatic extraction of Access Control Policies by Xiao et al. [4] 52
4.3 Taxonomy of nonfunctional requirements by Sommerville [5] 53
4.4 Taxonomy of functional requirements by Sharma and Biswas [6] 56
4.5 Business rule taxonomy by Ross [7] 57
4.6 Business rule taxonomy by Weiden et al. [8] 60
4.7 Automatic classification approach by Anish et al. [2]. 66

Chapter 1

Introduction

Software maintenance can constitute up to 90% of the total cost of the whole software life cycle [9]. Software documentation can help in reducing this cost by supporting software understanding, which is essential when maintaining and evolving software. According to a study by Lethbridge et al. [10], 61% of software engineers consider documentation as effective or extremely effective in learning a new software system. The same study reports that 50% of software engineers consider software documentation to be effective in solving system failures when other experienced team members are unavailable for help.

In this thesis, we use the term *software documentation* to refer to various types of documents describing different aspects of software, including requirements, internal or external design, and usage instructions. Let us consider the next two scenarios, where software documentation can be leveraged:

- A developer wants to find the package or class where a new feature should be implemented.
- A developer needs to update the implementation of a constraint (*e.g.,* “Users must be above 18 years old” changes to “Users must be above 21 years old” due to a modification in a government regulation).

A good software engineering practice is to implement a new feature into a package that contains similar functionalities. In the first scenario, identifying existing features or constraints related to the new feature in an architectural design document might hint where they are implemented, which can help developers to decide the best location for the new feature. However, finding the location of a constraint in source code is not always an easy task. In the second scenario, when a change in a constraint is requested, a developer must search for the location of the concept (*e.g.,* “minimum age”) in the code and then make the necessary changes to solve the request. In more complex cases, a change in a constraint can lead to changes in multiple parts of the source code.

In theory, developers could manually keep track of the code fragments that implement features or constraints in a traceability link matrix (TLM), for example. In practice, this could work in new software projects: as soon as a feature or constraint is implemented in a new location, it could be added to the TLM. In existing projects, however, the task might not be that simple—the features or constraints and their implementations must be identified by developers in existing documentation (maybe outdated) and a large codebase. In any case, creating and maintaining traceability links is an effort-consuming activity that is impacted by the resource limitations of software projects and always-growing code bases. Under these circumstances, an approach that *automatically tracks the source code locations where a constraint is implemented* can be helpful. Using this approach, a developer could easily identify the code fragments that might be affected by a change. Ideally, the approach should perform the following tasks:

- Automatically detect *constraints* in software documentation.
- Trace the *constraints* to their implementation in the source code.

In this thesis, we focus on the first task, *i.e.,* automatically detecting constraints in software documentation. We consider a *constraint* as a sentence that imposes some limitations on the design solution or implementation of a software system [11]. Any other sentence is considered as a *non-constraint*. Based on these definitions, we formulate the constraint detection task as a binary classification problem: given a sequence of sentences from a software document, we want to classify each sentence into one of two categories, namely *constraint* or *non-constraint*.

The second task, *i.e.,* tracing the constraints to their implementation in source code, is left for future work. As different kinds of constraints might be traced to source code in different ways, for the second task we might need to characterize constraint types, which would require a more complex classification system than a binary one. But the second task can leverage the output from our proposed binary classification approach to differentiate constraints from non-constraints in software documentation before tracing the constraints to the source code.

Instead of using pattern- or rule-driven approaches that require significant manual effort devoted to the syntactic analysis of constraints, we leveraged **machine learning** algorithms to build binary classification models that are to determine whether a sentence is a constraint or not. To construct the data set needed to train and validate the models, we used the documentation (*e.g.,* user manuals or guides, tutorials) of four open-source software (OSS) systems: HttpCore[1], jEdit[2], Swarm[3], and ArgoUML[4]. We selected OSS because we were unable to acquire documentation of proprietary software for research purposes. Two coders independently analyzed sentences from randomly selected sections of the collected OSS documentation and tagged them as *constraints*. The untagged sentences were extracted later as *non-constraints*. After discussing and solving the discrepancies, we used this coded data set as our ground truth to train and verify different machine learning models derived from SVM, Decision Tree, Naive Bayes, and Fine-tuned BERT. Our data set contained 167 constraint and 201 non-constraint sentences in total. Our best performing model (a decision tree with bigrams) was able to achieve 74.0% precision and 83.8% recall in the constraint detection task.

To the best of our knowledge, ours is the first study on automatic detection of *constraints* in the context of software documents. Similar studies have been conducted to automatically classify requirements [12, 13, 14, 15] and business rules [16, 17, 2] in software documentation. Other studies (*e.g.,* [3]) have focused on automatic constraint classification of privacy and security rules in the health sector.

[1]https://hc.apache.org/httpcomponents-core-4.4.x/index.html

[2]http://www.jedit.org/

[3]https://volcanoes.usgs.gov/software/swarm/index.shtml

[4]https://argouml.en.softonic.com/

Chapter 2

Methodology

The goal of our research is to automatically identify software constraints described in the documentation of software systems. As a first approach to achieve this goal, we formulate the detection of constraints in software documents as a binary classification problem: given a sentence S extracted from a software documentation artifact, a classification model must assign S to one of two categories: *constraint* or *non-constraint*.

In this study, we explore different machine learning algorithms to generate constraint classification models. We use a machine learning based approach rather than a pattern-based or rule-building approach because building such patterns or rules is a manual, effort-intensive and time-consuming task. Instead, to build our machine learning based classification models, we only need a data set consisting of constraint and non-constraint sentences for model training and testing. Although building the data set is mandatory for both the pattern-based approach and the machine learning based approach, the former requires extra manual effort to determine the patterns from the data set.

In this chapter, we describe the methodology we followed to build a constraint data set derived from the collection and analysis of documentation of open-source software systems. We also describe the machine learning algorithms and implementations that we leveraged to build our classification models, as well as the language models we selected for experimentation. At the end of the chapter, we describe the metrics we use to compare the performance of the models and discuss the threats to the validity of our study.

2.1 Defining the term "Constraint"

The first step in our study is to define what a *constraint* is in the context of software documentation. Different definitions for this concept can be found in the literature. The ISO/IEC/IEEE 29148 standard [11] defines a software constraint as an "externally imposed limitation on the system, its design, or implementation or on the process used to develop or modify a system". Similarly, Young

[18] defines a constraint as a "necessary attribute of a system that specifies legislative, legal, political, policy, procedural, moral, technology, or interface limitations". Søberg et al. [19] refers to software constraints as "rules and conventions commonly agreed to in a given programming environment" (for example, variable naming conventions in the source code).

Considering the existing definitions, we explored diverse documentation artifacts to extrapolate and get a hands-on understanding on what kind of sentences are considered as software constraints in these documents. After analyzing existing definitions and diverse documentation artifacts we built a working definition of constraints. For the purpose of this thesis, *a constraint is a sentence that conveys restrictions on design solutions or limitations on the implementation of features of a software system*. Some constraints apply to all requirements of a software system, and some others are related to a specific requirement or set of requirements. In general, constraints reduce the degree of freedom a developer has while building the system. For instance, a constraint can restrict the technology stack a developer can use, as it can be observed in the next excerpt extracted from the user manual[5] of Swarm.

Example 2.1.1. "*Swarm is platform independent* (will run on any operating system) but *requires a graphical display* and a *Java Virtual Machine 1.8 or greater*."

Example 2.1.1 is a constraint because it conveys the following restrictions or limitations:

- The system is able to run on all the operating systems, which indicates technology stack limitations.
- The system can not be used without a graphical display, which indicates hardware limitations.
- The system needs Java Virtual Machine with a version of at least 1.8, which indicates technology stack limitations.

[5] https://volcanoes.usgs.gov/software/swarm/index.shtml

Table 2.1: Software systems used to build the data set

System name	System description	Analyzed document type	System version
Swarm[6]	Seismic wave analysis tool	User manual and reference guide	2.8.10
HttpCore[7]	Minimal HTTP client	Tutorial book	4.4.5
jEdit[8]	Java based text editor	User guide	5.5.0
ArgoUML[9]	UML diagramming application	User manual	0.34

2.2 Building the data set

2.2.1 Subject systems and documentation

We used open-source software (OSS) systems as source of documentation to derive our data set. Ideally, we would have also included documentation of enterprise or proprietary software to strengthen the generalizability of our study. Unfortunately, it is difficult to obtain access to this kind of resources due to privacy and copyright concerns. We further discuss this issue in Section 2.5.

The OSS systems and the type of documents we used in our experiment are listed in Table 2.1. Three of these documents are user manuals, and one of them is a tutorial book. User manuals and user guides contain instructions on how end-users should use software systems [20], but the latter ones are meant to be more precise and concise than the former ones. Although the documentation of jEdit is referred to as a user guide, it also contains interfaces and functionality of the software for add-on developers. Add-ons are additional features that are developed by third-party developers to enhance the functionality of a software.

2.2.2 Constraint coding

We followed *deductive coding* [21] to manually identify constraints in the subject documents. Deductive coding is a top-down approach to code qualitative data, where a predefined set of codes is applied to the data set. In our case, we had a single code, namely *constraint*.

[6]https://volcanoes.usgs.gov/software/swarm/index.shtml

[7]https://hc.apache.org/httpcomponents-core-4.4.x/index.html

[8]http://www.jedit.org/

[9]https:‘//argouml.en.softonic.com/

To facilitate the coding task, we utilized the Mitre Annotation Toolkit (MAT[10]). MAT offers various features such as hand annotation, alignment, comparison and reconciliation of textual documents. We randomly selected sections of the documents to be manually analyzed (see Table 2.2). These sections were independently and incrementally annotated by two coders.

To understand how the coders annotated the documents, let us consider the Example 2.1.1 from Section 2.1. In this sentence, the segments in italics are considered constraints. From this example, we can observe that:

- A part of a sentence can represent one constraint.
- One sentence can contain multiple constraints.

We asked the coders to annotate only the part of the sentence containing a constraint. However, while preparing the training data set for the machine learning models, we took the whole sentence as one data point to preserve the constraint's context. The rationale behind this design decision is twofold. First, similar classification models in the literature make use of sentences as data points to build their models. Future studies using our data set might benefit from a uniform granularity level for comparison. Second, the machine learning algorithms selected for our study might not be well suited for segments of sentences as one data point. For example, BERT is considered a language model. The BERT model we used in our study was pre-trained on the whole sentences from the English language.

Once the coders completed their analysis of a document, they used MAT's annotation alignment to compare their annotated texts. Whenever a disagreement was identified, the coders solved it through a discussion. In the cases where the coders were not able to reach a consensus, a third person made the final decision for those data points.

[10] http://mat-annotation.sourceforge.net/

Table 2.2: Composition of the data used in the study. Proportions of constraints and non-constraints with respect to sentences in analyzed section are shown in parenthesis.

System	Sentences in document	Sentences in analyzed sections	Constraints in analyzed sections	Non-constraints available in analyzed sections	Non-constraints extracted from analyzed sections
SWARM	555	226	65 (28.8%)	161 (71.2%)	84
HttpCore	738	89	38 (42.7%)	51 (57.3%)	44
jEdit	2757	118	43 (36.4%)	75 (63.6%)	50
ArgoUML	8600	58	21 (36.2%)	37 (63.8%)	23
Total	12650	491	167 (34.1%)	324 (65.9%)	201

2.2.3 Data set consolidation

Using MAT, we generated a reconciled JSON file with the final annotations of the documents. Then, we transformed it into a CSV file for better consumption in the next steps of our study. As mentioned before, we considered the entire sentence containing a constraint as a data point in our final data set, rather than just the part of the sentence mentioning the constraint.

The output from MAT provided us with sentences that contain at least one constraint. Nevertheless, to train models for binary classification, we require an equal number of sentences that do not contain any constraints. This is because a balanced training data set (*i.e.,* a data set whose classes have roughly an equal number of data points) results in better accuracy in machine learning models [22]. The composition of the testing data set is different. Notice from Table 2.2 that the distribution of constraints and non-constraints is not equal in the documentation we analyzed. To ensure that the sample on which we evaluate our models truly represents the population (hence strengthening the validity of our study), we must preserve this distribution in the testing data set.

The process that we followed to obtain an equal number of constraints and non-constraints for the training data set, and to obtain the same ratio of constraints and non-constraints as in analyzed sections in the testing data set is described in Algorithm 1.

To exemplify this process, let us consider the user manual of Swarm, which consists of 555 sentences (see Table 2.2). The coders analyzed 226 of these sentences and tagged 65 as constraints. This means that there are $226 - 65 = 161$ non-constraints in the analyzed sections. We added 80%

Algorithm 1 Process followed to build the training and the test data set

1: **for each** $system \in Subject\, Systems$ **do**
2: Assign 80% of $constraints$ to the training data set
3: Extract an equal number of $non\text{-}constraints$ and assign them to the training data set
4: Assign the rest of the $constraints$ to the testing data set
5: Calculate the ratio of $constraints$ and $non\text{-}constraints$ in the analyzed sections
6: Extract and assign $non\text{-}constraints$ to the test data set such that the ratio of $constraints$ and $non\text{-}constraints$ in the test data set is similar to that of the analyzed sections.
7: **end for**

of the constraints to the training data set of Swarm, *i.e.,* 52 sentences. The rest of the constraints (*i.e.,* 13 sentences) were added to the testing data set. We completed the training data set with the same number of constraints it already contains (*i.e.,* 52 sentences). For the testing data set, we considered the proportions of constraints and non-constraints in the analyzed sections of the Swarm user manual, *i.e.,* 29% and 71% respectively. To maintain similar proportions, we added 32 non-constraints into the testing data set. Therefore, our training and testing data sets together consist of 65 constraints and 84 non-constraints for Swarm.

The number and distribution of constraints and non-constraints for each software system are shown in Table 2.2. Similarly, the number and distribution of constraints and non-constraints in the training and testing data sets of each system are shown in Table 2.3.

Table 2.3: Composition of the training and testing data sets for each system. Proportions of constraints and non-constraints with respect to data points in training/testing data set are shown in parenthesis.

System	Training data set			Testing data set			Total data points
	Constraints	Non-constraints	Data points	Constraints	Non-constraints	Data points	
SWARM	52 (50%)	52 (50%)	104	13 (28.9%)	32 (71.1%)	45	149
HttpCore	30 (50%)	30 (50%)	60	8 (42.1%)	11 (57.9%)	19	82
jEdit	34 (50%)	34 (50%)	68	9 (36.0%)	16 (64.0%)	25	93
ArgoUML	16 (50%)	16 (50%)	32	5 (38.5%)	8 (61.5%)	13	44
Total	132 (50%)	132 (50%)	264	35 (34.7%)	67 (66.3%)	101	368

2.3 Building constraint classification models

We used the scikit-learn library [23] and Tensorflow [24] to build our classification models. We explored different configurations of machine learning algorithms and encoding algorithms to generate various classification models and select the best-performing model for our particular task.

2.3.1 Machine learning algorithms

We experimented with four machine learning algorithms (*i.e.,* Decision Tree, Naive Bayes, Support Vector Machine, and Fine-tuned BERT) to construct our constraint classification models.

We used the randomized search algorithm[11] from the scikit learn library to search the hyperparameter space instead of grid search when tuning hyperparameters of decision tree, Naive Bayes and SVM. Grid search tries every possible combination of values for the hyperparameters and is slow relative to the randomized search. Randomized search tries out a number of random combinations of the parameters using a specified distribution. The combination that works the best is selected as the best parameter setting.

A brief description of each of the machine learning algorithms and the used implementations follows.

2.3.1.1 Decision trees

A decision tree (DT) is a combination of multiple conditional statements (also called nodes). Each internal node in the tree can be thought of as a conditional statement. Thus, the path from root to a leaf node represents a certain decision path. Decision trees are supervised learning methods mainly used for classification tasks. They are supervised algorithms because they require to be fed with the expected output while training the model. Figure 2.1 shows a sample diagram of a decision tree.

[11] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

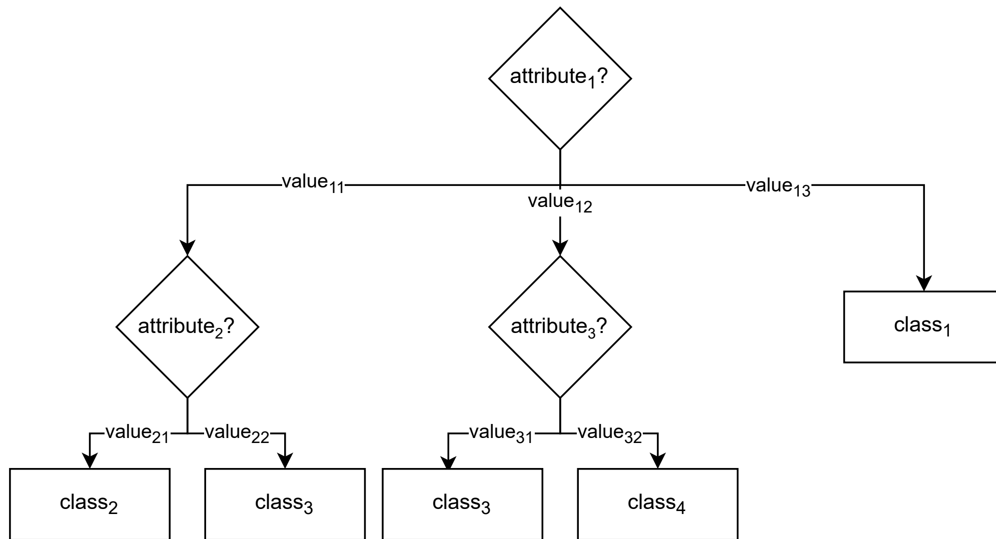


Figure 2.1: Sample diagram of a decision tree

We utilized the DecisionTreeClassifier[12] class from the scikit learn library to generate the decision tree based classifiers. This implementation requires, among others, the following parameters:

- *criterion*. It indicates the objective function that will be used to measure the effect of splitting a node into two nodes at the training phase. The value "gini" measures the effect of the split by calculating the impurity of the node and the value "entropy" measures the effect of the split by calculating the information gain.

- *max_depth*. It represents the maximum possible depth of the learned tree. A value of "None" indicates no set restriction on the depth of the tree. The higher the depth of a tree is, the more complex classification scheme the tree can learn in the training phase.

- *min_samples_split*. It indicates the minimum number of samples needed to split an internal node of the tree. If the samples indicate different classes, the internal node will be divided into two new nodes. A high value of *min_samples_split* indicates that the tree will be more resistant to the noise in the training data set.

[12]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

- *min_samples_leaf*. It indicates the minimum number of samples needed to create a new leaf node. That node will represent that particular sample. A high value of *min_samples_leaf* indicates that the tree will be more resistant to the noise in the training data set. The value of *min_samples_leaf* can be as low as 1.

- *max_features*. It indicates the maximum number of features or attributes the model will use to train itself. A value of "None" indicates the tree to learn from the maximum number of available features from the data set.

- *min_impurity_decrease*. It indicates the minimum value of impurity decrease for a node to be split into more nodes. If a data point can decrease the impurity of a node by a value greater than the minimum value, the node will be split into two nodes. A lower value of *min_impurity_decrease* indicates that a node will be split into two nodes even if there is a smaller decrease in the impurity of the node. The value of *min_impurity_decrease* can be as low as 0.

Table 2.4, 2.5 and 2.6 contain the tuned values of these hyper parameters for unigram, bigram and unigram+bigram (*i.e.,* n-gram where $n = \{1, 2\}$) embeddings respectively.

2.3.1.2 Naive-Bayes

A Naive-Bayes based model (NB) is a probabilistic classifier based on the Bayes' theorem. Bayes [25] first formulated this probability theorem in 1763 as:

$$Posterior = \frac{Prior \times likelihood}{Evidence},$$

Table 2.4: System-wise parameters of decision tree models with onegram embedding

System	criterion	max_depth	min_samples_split	min_samples_leaf	max_features	min_impurity_decrease
HttpCore	entropy	100	4	1	80	0.01
jEdit	gini	100	3	1	150	0.01
Swarm	gini	200	4	1	None	0.01
ArgoUML	entropy	200	4	3	80	0.01
All Systems	entropy	200	4	1	80	0

Table 2.5: System-wise parameters of decision tree models with bigram embedding

System	criterion	max_depth	min_samples_split	min_samples_leaf	max_features	min_impurity_decrease
HttpCore	entropy	None	2	1	100	0
jEdit	entropy	500	2	1	100	0
Swarm	gini	100	4	1	50	0
ArgoUML	entropy	100	2	1	150	0.01
All Systems	gini	None	4	1	20	0

Table 2.6: System-wise parameters of decision tree models with n-gram (n={1,2}) embedding

System	criterion	max_depth	min_samples_split	min_samples_leaf	max_features	min_impurity_decrease
HttpCore	gini	500	2	1	100	0
jEdit	entropy	50	4	1	150	0.01
Swarm	gini	None	2	1	100	0
ArgoUML	entropy	50	2	2	80	0.01
All Systems	gini	100	3	1	None	0

which can be translated into:

$$P(A|B) = \frac{P(A) \times P(B|A)}{P(B)}$$

$P(A|B)$ represents the probability for the occurrence of an event A provided that event B has already occurred. This is also called the posterior. $P(A)$ represents the probability for the occurrence of event A. NB uses this formula to classify an input analyzing the feature set. In our thesis, $P(A|B)$ can be translated into $P(sentence\ contains\ constraint \mid feature_i\ exists)$.

We used the multinomial variant[13] of Naive Bayes from the scikit learn library for our experiment. We chose this variant because it is well suited for text classification. The parameters relevant to this implementation are the following:

Table 2.7: System-wise parameters of best Naive Bayes models

System	Onegram		Bigram		Ngram	
	alpha	fit_prior	alpha	fit_prior	alpha	fit_prior
HttpCore	0.2	True	0	True	1	True
jEdit	0.05	True	0.2	False	0.2	True
Swarm	0	True	0	False	0.5	True
ArgoUML	0	True	0	True	0	True
All Systems	0.2	True	0	True	0.2	True

[13]https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

- *alpha*. This is the Laplace smoothing parameter. If a word from the testing data set is not present in the training data set, the smoothing parameter helps to assign a small probabilistic value to that word. The higher the value of alpha, the greater the smoothing operation will be. The value of *alpha* can be as high as 1 and as low as 0.

- *fit_prior*. This parameter indicates whether the model should learn the class prior probabilities. A value of “False” indicates that a uniform prior value will be used.

- *class_prior*. This parameter provides the prior probabilities of the classes to the model as input. We used a value of “None” to let the model learn prior probabilities only from the training data.

Table 2.7 shows the tuned values of these hyperparameters for our study.

2.3.1.3 Support Vector Machine

Support Vector Machine (SVM) [26] is also a supervised learning model. SVM was first introduced by Boser et al. [27] and by Cortes and Vapnik [28]. SVM has an associated learning algorithm that it uses to classify data points. In the training phase, the model tries to maximize or minimize the output value of the learning algorithm for the given data points to find the best snapshot of itself. To do so, it builds a hyperplane that can separate data points from different classes. If the data points are not separable by the hyperplane, SVM tries to uplift the dimension of the data set. The process is called kernel tricks (see Figure 2.2). SVM then tries to place this hyperplane as far as possible from the nearest data points, which in turn minimizes the generalization error.

The parameters relevant to the SVM algorithm are the following:

- *C*. It is a regularization parameter. A lower value of C indicates the presence of higher regularization. Regularization tells the model to give less importance to the misclassifications in order to reduce over-fitting.

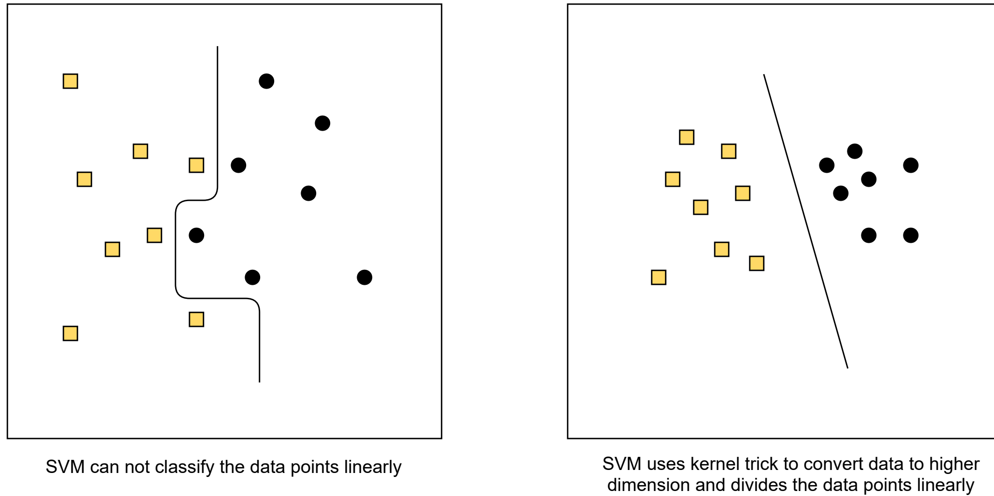


Figure 2.2: SVM uses kernel trick to uplift the dimension of data points

- *Kernel*. It specifies the function which will be used as the kernel for the model. A nonlinear data set is converted to a linear one in a higher dimension by passing the data points through these kernel functions.
- *Degree*. It indicates the degree of the polynomial kernel function.
- *Gamma*. It is a coefficient for the kernel function.

The values of the parameters that gave us the best results are mentioned in Table 2.8.

2.3.1.4 Bidirectional Encoder Representations from Transformers (BERT)

BERT is a recent algorithm developed by Devlin et al. [29] at Google. Most of the previous text encoding algorithms were unidirectional. As BERT is a transformer, it can learn textual contexts and dependencies of each word in both directions (from left to right and from right to left). As

Table 2.8: System-wise parameters of best SVM models

System	Onegram				Bigram				Ngram			
	C	Degree	Gamma	Kernel	C	Degree	Gamma	Kernel	C	Degree	Gamma	Kernel
HttpCore	146.834	6	0.025	rbf	24.648	1	0.039	rbf	24.151	1	0.113	rbf
jEdit	135.239	4	0.070	linear	12.476	1	0.011	linear	53.787	3	0.249	linear
Swarm	89.126	1	0.089	rbf	19.232	3	0.740	linear	114.641	4	0.045	rbf
ArgoUML	46.364	2	0.264	linear	102.390	1	0.034	linear	79.196	3	0.128	rbf
All Systems	128.498	5	0.013	rbf	57.758	3	0.025	rbf	186.938	4	0.039	rbf

a result, it has the potential to perform better than other encoding algorithms (*e.g.,* Bi-LSTM). Although BERT is more than an encoding algorithm, this is our focus in this subsection.

Bidirectional Encoder Representation from Transformers (BERT) is a transformer-based, context-aware masked language model (MLM). Transformers in BERT contain only encoders (*i.e.,* they do not contain any decoder). It is also called a language model because it can be thought of as a vector representation of a specific natural language. Each word from a natural language will have a row and a column represented in a two-dimensional vector space. A cell in the vector space will contain the probabilistic relation between two words (one word represented by the row of the cell and another word represented by the column of the cell). This relation signifies the probability of two words appearing together in a sentence. For this reason, BERT can differentiate the meaning of homonyms, *e.g.,* it can differentiate between *playing in a "park"* and *"park"ing a car*. BERT learns about the context dependency bidirectionally (from left to right and from right to left) at the same time. While pre-training the model for a language, each word is sent to the model masked at a time. The model tries to guess the incoming word (using the probabilistic relations with the surrounding words) and thus tries to improve itself by correcting any errors.

BERT is more popular than LSTM these days because LSTM takes a long time to train, whereas BERT comes as a pre-trained model over the internet. Also, BERT is genuinely bidirectional as it learns about the context dependency in both directions simultaneously. Even though LSTM has a bidirectional variant called BiLSTM, it is not truly bidirectional. The BiLSTM model is trained from left to right and from right to left separately and then concatenated at the end.

BERT uses softmax as an activation function. If there are values in the output layer of a machine learning model, the softmax function will turn them into probabilities with a sum of 1.0. Softmax will give higher (exponential) weight in the probability distribution of those bigger numbers.

$$softmax(z_i) = \frac{e^{z_i}}{\int_{j=1}^{K} e^{z_j}}$$

The encoding part of the algorithm is shown in Figure 2.3. The example of this procedure is shown in Figure 2.4 for the input sentence:

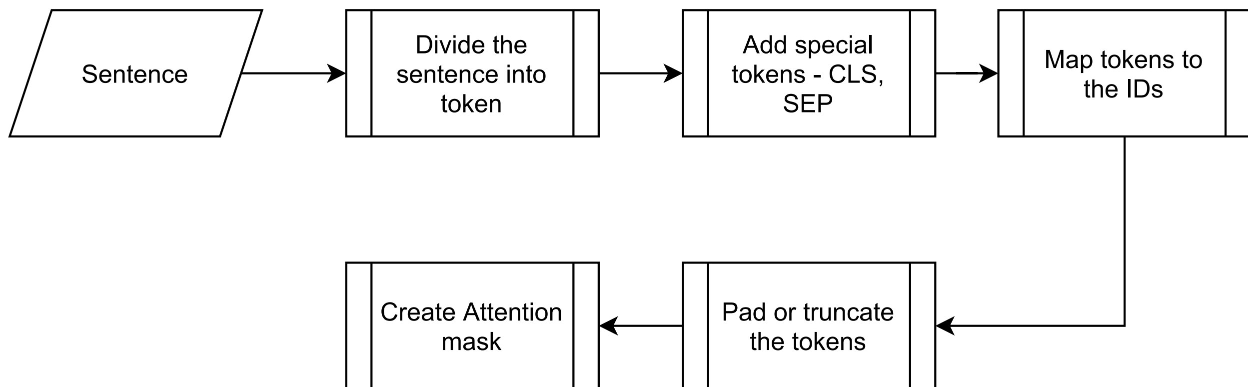


Figure 2.3: Encoding procedure of the BERT algorithm

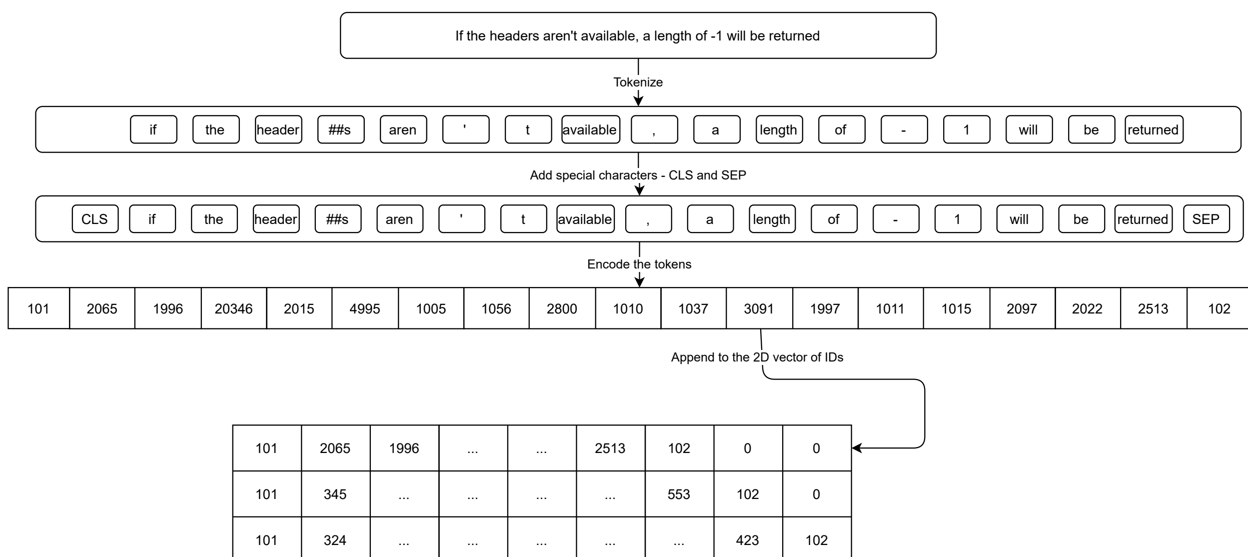


Figure 2.4: Example of encoding procedure of the BERT algorithm

Example 2.3.1. “If the headers aren’t available, a length of -1 will be returned”.

First, each word of the sentence is divided into separate tokens. For the classification task, the special tokens “CLS” and “SEP” are respectively added at the beginning and end of the sentence to delimit it. Each word is then encoded to a unique ID. Since the lengths of sentences are different in a language, padding is added to the end of the sentence. Later, all the sentences are added to a 2D grid.

BERT is still evolving. There is no built-in library yet for finer tuning specific to different tasks. We built our models with both generic base BERT model and Fine-tuned BERT. The generic base BERT is the smallest version of BERT (in terms of the number of internal parameters in the

model) available as pre-trained. Fine-tuned BERT models are constructed by tuning these internal parameters with an optimization algorithm for a specific task.

We used Adam [30] as an added layer on top of the base BERT to fine tune the pre-trained BERT model for the task of text classification. Resources[14] are available on the Tensorflow website to help in the fine tuning of the base BERT model for specific tasks.

Adam is an optimization algorithm that helps reconfigure the parameters of an algorithm by running the model multiple times and moving the parameters in the directions that gives the best result from the model. It uses adaptive learning rate and Stochastic gradient descent with momentum. We did not use the original large (in terms of the number of internal parameters in the model) version of BERT, as it is quite extensive for our purposes and has a large set of parameters. Optimizing those parameters with limited resources would have been challenging, although it would have resulted in higher accuracy and precision values. Larger BERT models require a significant amount of memory and CPU to be trained into a full-scale Fine-tuned BERT model. We leveraged the Tensorflow 2 [24] framework to implement BERT in our experiments. We chose the Small uncased BERT for the English language. This model was published by Turc et al. [31]. A later version of the model[15] was released with Tensorflow 2. This is a small version of the original BERT model with fewer parameters. Uncased version of BERT does not contain any accent markers (markers on the alphabets of certain language for example, Latin language).

Even the smaller model used in our experiment had more than 110 million internal parameters. The file size of the downloaded pre-trained model was 103.18 MB. The largest BERT model can have up to 2.7 billion internal parameters. The file size of the biggest pre-trained model is 390.13 MB. Below we discuss different parameters of the BERT model.

- *L*. It indicates the number of hidden layers, *i.e.,* the number of transformer blocks. L can have a minimum value of 2 and a maximum value of 24. Our model used a value of 4 for L.

[14] https://www.tensorflow.org/text/tutorials/classify_text_with_bert

[15] https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1

- *H*. It indicates the size of each hidden layer. H can have a minimum value of 128 and a maximum value of 1,024. Our model used a value of 512 for H.

- *A*. It indicates the number of attention heads per transformer layer. A can have a minimum value of 2 and a maximum value of 16. Our model used a value of 8 for A.

We tried to run a slightly bigger model in our local machines, but it was consuming a significant amount of resources to achieve a little increase in accuracy (1% - 1.5%). Therefore, we opted for the smaller model.

Every machine learning model needs some level of hyperparameter tuning. The hyperparameters associated with Fine-tuned BERT are listed below.

- *Batch size*. The authors of the BERT model [29] suggest the next values for the batch size: $8, 16, 32, 64, 128$. We used a value of 32 for batch size at it was giving us the best accuracy score.

- *Learning rate*. The authors [29] suggest the next values for learning rate[16]: 3×10^{-4}, 1×10^{-4}, 5×10^{-5}, 3×10^{-5}, 2×10^{-5}. They also suggest to use smaller values (in the range of 10^{-5}) while fine tuning for a text classification task. We used the value of 3×10^{-5}, as it gave us the best accuracy score.

- *Epoch*. It refers to the number of epochs needed to fine tune the BERT model, and it usually differs from study to study. This parameter is also related to overfitting, a common phenomenon in machine learning. It occurs when a model is trained on the training data set for so long that it gets over aligned with the training data and becomes oversensitive to any noise in the testing data set. When overfitting occurs, the training accuracy gradually increases and the validation accuracy gradually decreases. Figure 2.5 shows the learning rate of Fine-tuned BERT for all the systems in our data set. In this figure, we can observe that after epoch 10, the validation accuracy rate becomes constant for the "All systems" data set. After the same

[16]https://github.com/google-research/bert

point, the validation loss begins to increase gradually. This point is a good indicator for halting the training before overfitting the model. We followed the same procedure for all system-wise data sets and their individual validation curve is shown in Figure 2.5. The trend of these graphs cannot be guessed beforehand. For example, In the case of validation accuracy of the Swarm data set, the curve seems to drop after epoch 10. But as the validation loss kept dropping, we continued to train the model. Then the accuracy increased again after epoch 14.

2.3.2 Encoding text into numerical data

Machine learning algorithms do not directly work with textual data. Therefore, we need to encode the textual data into numerical data through tokenization and vectorization before feeding it to the machine learning algorithms. We used the n-gram model to tokenize the sentences in our data set, and TF-IDF to vectorize those tokens.

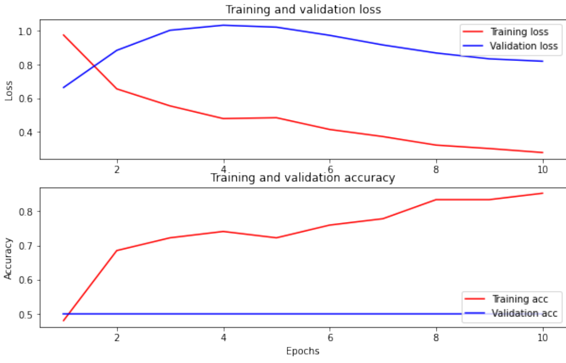
2.3.2.1 N-gram model

In the n-gram model, a sentence is divided into contiguous sequences of n words. When $n = 1$, the model is called *unigram*. Similarly, when $n = 2$, the model is called *bigram*, and when $n = 3$, it is called *trigram*. In this thesis, when we refer to the composition of tokens from both *unigram* and *bigram* as $n = \{1, 2\}$ or unigram+bigram.

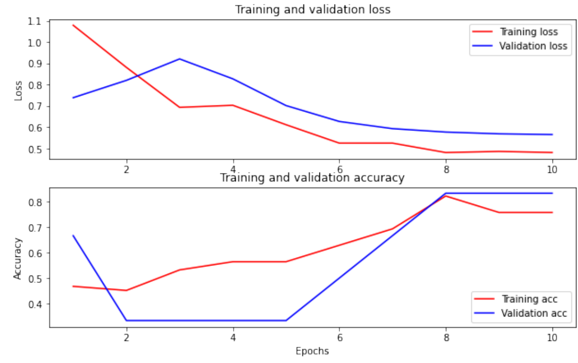
2.3.2.2 Term Frequency - Inverse Document Frequency (TF-IDF)

TF-IDF is one of the most popular encoding algorithms. It indicates the relevance of a word (or a group of words) to a document [32]. It achieves this based on the frequency of the word in the document with respect to the inverse document frequency in a collection of documents.

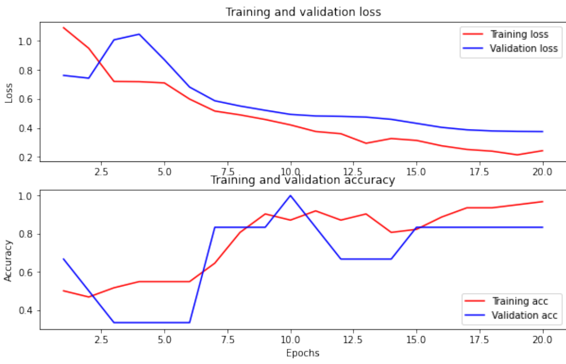
Let us assume that we want to calculate the TF-IDF score of a word or term t in a document d that belongs to the set of documents $\mathcal{D}$, which contains a total of N documents. The formula to



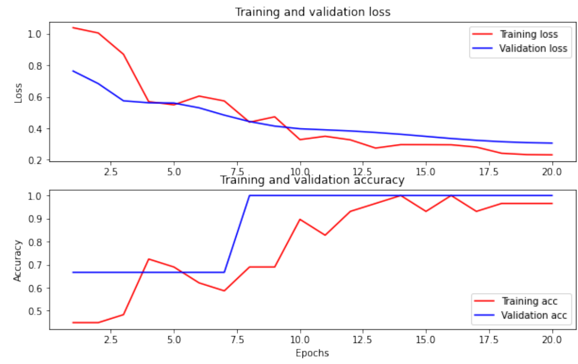
(a) HttpCore



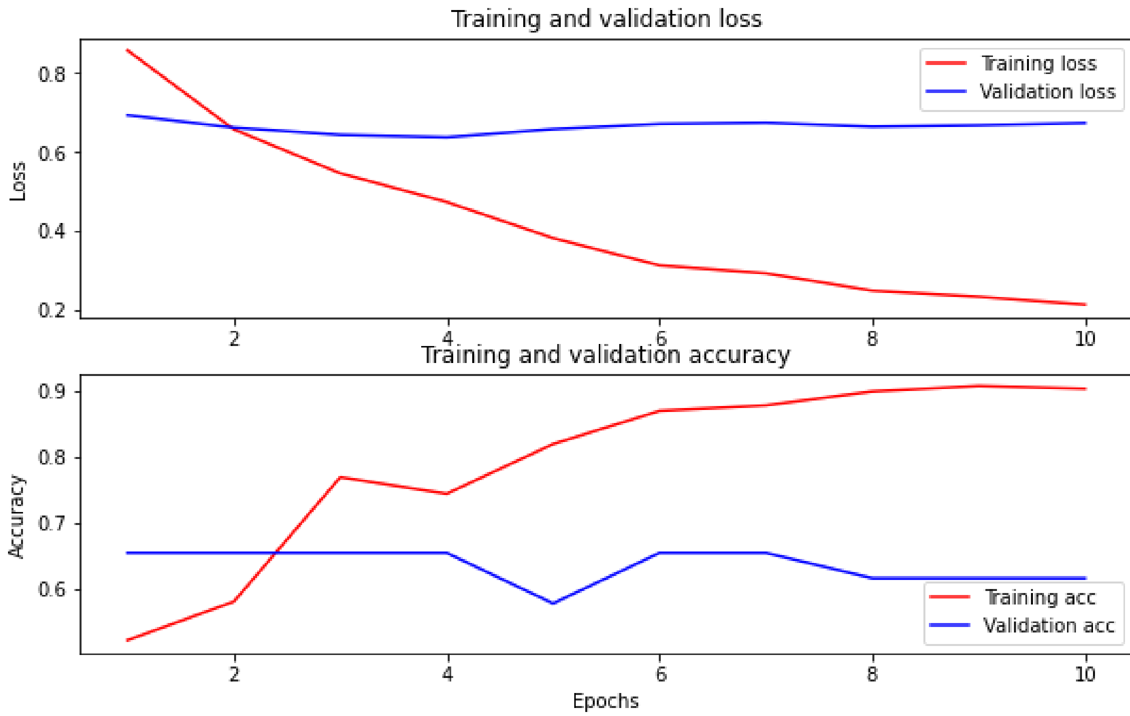
(b) jEdit



(c) Swarm



(d) ArgoUML



(e) All systems

Figure 2.5: Fine-tuned BERT optimized with Adam learning rate

compute TF-IDF for t is given by:

$$tfidf(t, d, \mathcal{D}) = tf(t, d) \times idf(t, \mathcal{D})$$

$$tf(t, d) = log(1 + freq(t, d))$$
$$idf(t, \mathcal{D}) = log\left(\frac{N}{|\{d \in \mathcal{D} : t \in d\}|}\right)$$

where the function $freq(t, d)$ computes the occurrences of the term t in the document d, and $|\{d \in \mathcal{D} : t \in d\}|$ indicates the number of documents in $\mathcal{D}$ where the term t appears.

Figure 2.6 shows the basic steps for converting a sentence into vectors of numbers to feed into machine learning algorithms.

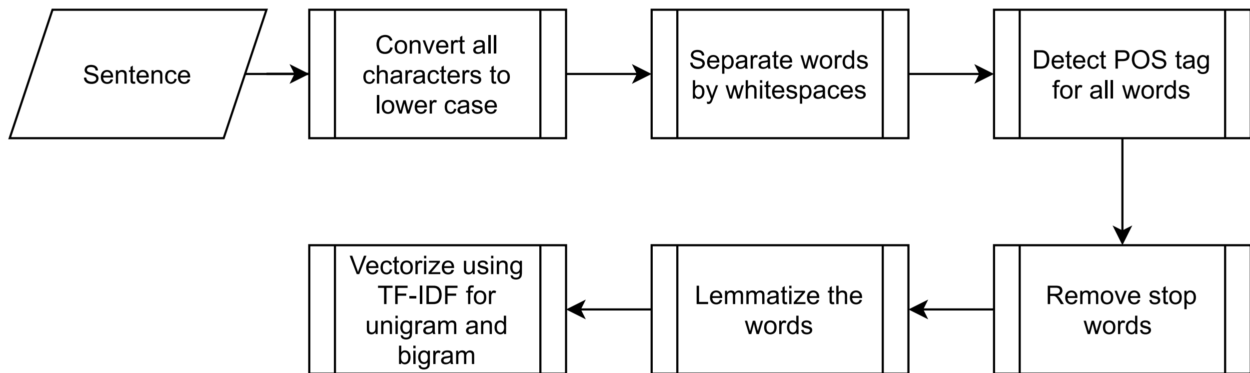


Figure 2.6: Processing sentences before feeding into models

We demonstrate this process with Example 2.3.2 from the tutorial book of HttpCore.

Example 2.3.2. “A HTTP message consists of a header and an optional body”

1. Convert characters to lower case: “a http message consists of a header and an optional body”.

2. Separate words by white spaces: “a”, “http”, “message”, “consists”, “of”, “a”, “header”, “and”, “an”, “optional”, “body”.

3. Detect Parts of Speech (POS[17]) tags for each word: <“a”, DT>, <“http”, NN>, <“message”, NN>, <“consists”, VBZ>, <“of”, IN>, <“a”, DT>, <“header”, NN>, <“and”, CC>, <“an”, DT>, <“optional”, JJ>, <“body”, NN>.

4. Remove stop words: “http”, “message”, “consists”, “header”, “optional”, “body”.

5. Lemmatize the words: “http”, “message”, “consist”, “header”, “optional”, “body”.

6. Vectorize using TF-IDF for unigram, bigram.

2.4 Evaluating the performance of the models

To evaluate the performance of the resulting classification models, we use the four metrics described next. These metrics are standard in the evaluation of classifiers and have been used in studies similar to ours [12, 33].

- *Accuracy*. It computes the fraction of data points a model is able to correctly classify. The accuracy of a model can be as low as 0, if the model misclassifies all the data points in the testing data set, and as high as 1, if the model correctly classifies all the data points in the testing data set.

$$Accuracy = \frac{Data\ points\ correctly\ classified}{Total\ number\ of\ data\ points}$$

- *Precision*. It signifies the fraction of data points positively classified by the model that is actually correct. Precision values are in the range $[0, 1]$. A value of 0 indicates that all the data points positively classified by the model are incorrect. A value of 1 indicates that all data points positively classified by the model are actually correct.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

[17] DT indicates Determiner, NN indicates Noun, VBZ indicates Verb (3rd person singular present), IN indicates Preposition or subordinating conjunction, CC indicates Coordinating conjunction, DT indicates Determiner, JJ indicates Adjective

- *Recall.* It describes the fraction of actual positive data points the model is able to correctly predict. Recall values are in the range $[0, 1]$. A value of 0 indicates that the model is not able to recognize any of the positive data points in the data set. A value of 1 indicates that the model is able to recognize all the positive data points.

$$
\begin{aligned}
Recall &= \frac{Data\ points\ predicted\ positive\ by\ the\ model}{Number\ of\ actual\ positive\ data\ points\ in\ the\ dataset} \\
&= \frac{True\ Positive}{True\ Positive + False\ Negative}
\end{aligned}
$$

- *F1-score.* It represents the harmonic mean between precision and recall, *i.e.,* it combines precision and recall into a single value. In some situations, both precision and recall have to be considered to compare the performance of different models, and this is when F1-score comes in handy. A model can have an F1-score as high as 1, when both the precision and recall are 1. It can also be as low as 0 indicating that either precision or recall are 0.

$$
F1\text{-}score = \frac{Precision \times Recall}{Precision + Recall}
$$

2.5 Threats to validity

Threats to *internal validity* refer to experimental conditions that might affect the outcomes of our study. One of these threats is the small size of the data set we built to train the classification model, which consists of 368 data points. Similar binary classification models [12] were trained and tested in data set containing almost twice as many data points as we have. Given the size of our data set, our model might miss some patterns of constraints that are not present in the training data set. The testing data set was also small. As a result, if the model fails to predict the right class of a single sentence, the value of the resultant metric could fluctuate.

When using data sets derived from deductive coding, subjectivity in the manual analysis is another threat to the internal validity. To mitigate this threat, we had two coders independently annotating sentences in the documentation. Still, while analyzing the results obtained with the

Fine-tuned BERT model, we came across an interesting situation, where the sentence "Auto-scale toggles helicorder auto-scaling on and off" was correctly classified as *constraint*, although it was originally—and incorrectly—tagged as *non-constraint*. To understand the cause of this error, it is important to remember how we collected the non-constraint sentences. The coders were asked to label the constraints in the given documents. This is an effort-intensive task that requires concentration. When continuously executing this task for a while, it is possible for coders to overlook a constraint due to several reasons such as tiredness. This particular sentence was missed by both coders, and in the later steps, was picked up as a non-constraint sentence. In future studies and if the financial budget allows it, we will recruit a larger set of coders to reduce subjectivity.

We only analyzed a subset of the sections of the documents in our sample. It is likely that the types of contents of other sections are different. To reduce selection bias, we randomly chose a subset of sections to be used in our study. However, since this subset is small, we might not have covered all the types of contents from the analyzed documents. Therefore, our model might not recognize constraints from a section with other types of content that we have not coded.

Threats to *external validity* refer to experimental conditions that affect the generalizability of the outcomes of our study. We only analyzed publicly available documentation of open-source software systems. Getting access to the documents of enterprise or proprietary software was not possible for us. Different from proprietary software, open-source software projects rarely devote time to requirements engineering, and therefore, do not produce requirements specification documents. The documentation from enterprise-level software might differ on purpose, types of contents and structure, so our models might not perform properly to identify constraints in such documents. This also applies to other forms of open-source software documentation, whose types of contents and structure might differ from the ones in our sample.

Chapter 3

Results

Following the methodology described in the previous chapter, we build different classification models and compare their performance in the detection of *constraint* and *non-constraint* sentences. To recapitulate, we leveraged Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT) and Fine-tuned BERT with Adam to generate the models. We used n-grams ($n = \{1, 2, 3, \{1, 2\}\}$) with TF-IDF and BERT as embedding and language models.

In this chapter, we discuss the performance of the generated constraint classification models based on the evaluation metrics described in Section 2.4, *i.e.,* accuracy, precision, recall, and F1-score. We often refer to a random classifier as a baseline for comparison. A random classifier is a model in which the prediction score of a class is randomly assigned. Notice that the accuracy, precision, and recall values reached by a random classifier trained on a balanced two-class data set like ours would be 50%.

3.1 Testing the performance of the models with two strategies

To measure the performance of the models we followed two strategies:

- **Strategy 1**: *Train the model on a portion of a data set* (80% of the constraints and an equal number of non-constraints) *and test the model on the other portion of the same data set*. The results of this testing strategy demonstrate how our models will perform when detecting new constraints in documentation already seen by the models.

- **Strategy 2**: *Train the model on three data sets and then test the model on a fourth data set*. The results of this testing strategy indicate how out models will perform when detecting constraints in documentation that is totally new to the models.

3.2 Strategy 1: Train and test the models on the same data set

An overview of the performance of the different machine learning models and embedding schemes used in our experiments for all of our subject systems is shown in Table 3.1. Overall, decision tree based models performed better than the other models.

3.2.1 System-wise comparison of models

Given a software document, our primary goal is to detect all the constraints it describes, so that developers can easily find them. This could save time in the development and maintenance of software. Therefore, our classifiers should detect as many software constraints as possible while avoiding false positives. Leaving out one constraint undetected (and thus not implemented and tested) might become problematic when delivering the software product to the customer. Therefore, when comparing the performance of two models, we give more importance to recall than to precision as an evaluation metric (see underlined values in Table 3.1). A higher recall value indicates that a model is able to capture more constraints in the sequence of sentences than a model with a lower recall value. If the performance of two models in terms of recall is similar, we also consider their precision. A higher precision value indicates that a sentence predicted as a constraint by a model has a higher chance of being an actual constraint. If both precision and recall values of two models are close to each other, we consider the F1-score.

Table 3.1 reports the metrics for all the models across all the systems. Next, we analyze the models that performed the best for each subject system.

Table 3.1: Accuracy (AC), precision (PC), recall (RC) and F1-score (F1) comparison between different machine learning models across all systems for testing strategy 1. Abbreviations used for the sake of brevity: Model (MDL), Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), BERT model fine-tuned with Adam (FT-BERT), embedding (EMD) unigram (1GM), bigram (2GM), trigram (3GM), unigram+bigram (1-2GM). The underlined values represent the highest recall value obtained for a system by a particular model and embedding. The **values in bold** indicate the highest recall value obtained for a system.

MDL	EMB	System																			
		HttpCore				jEdit				Swarm				ArgoUML				All Systems			
		AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1
SVM	1GM	68.2	70.6	85.7	0.77	80.0	86.7	81.2	0.84	64.4	83.3	62.5	0.71	75.0	75.0	85.7	0.80	72.5	85.7	70.6	0.77
	2GM	72.7	72.2	92.9	0.81	68.0	70.0	87.5	0.78	42.2	75.0	28.1	0.41	66.7	66.7	85.7	0.75	70.6	77.9	77.9	0.78
	3GM	68.2	66.7	100	0.80	36.0	0	0	0	66.7	69.8	93.8	0.80	66.7	80.0	57.1	0.67	67.6	68.0	**97.1**	0.80
	1-2GM	68.2	68.4	92.9	0.79	80.0	86.7	81.2	0.84	62.2	82.6	59.4	0.69	75.0	75.0	85.7	0.80	73.5	91.8	66.2	0.77
	BERT	85.7	83.3	71.4	0.77	79.2	72.7	80.0	0.76	84.2	71.4	83.3	0.77	81.8	71.4	89.1	0.79	75.0	71.4	73.2	0.72
NB	1GM	72.7	100	57.1	0.73	84.0	92.9	81.2	0.87	66.7	81.5	68.8	0.75	75.0	75.0	85.7	0.80	70.6	89.6	63.2	0.74
	2GM	59.1	66.7	71.4	0.69	52.0	66.7	50.0	0.57	71.1	80.6	78.1	0.79	66.7	71.4	71.4	0.71	64.7	76.7	67.6	0.72
	3GM	45.5	100	14.3	0.25	36.0	50.0	37.5	0.43	64.4	80.8	65.6	0.72	66.7	100	42.9	0.60	53.9	69.8	54.4	0.61
	1-2GM	54.5	83.3	35.7	0.50	84.0	92.9	81.2	0.87	68.9	82.1	71.9	0.77	75.0	75.0	85.7	0.80	73.5	91.8	66.2	0.77
	BERT	66.7	50.0	100	0.67	58.3	50.0	10.0	0.17	47.4	37.5	100	0.55	81.8	71.4	89.1	0.79	56.5	50.6	100	0.67
DT	1GM	59.1	69.2	64.3	0.67	72.0	100	56.2	0.72	62.2	82.6	59.4	0.69	50.0	60.0	42.9	0.50	66.7	81.5	64.7	0.72
	2GM	77.3	73.7	**100**	0.85	64.0	66.7	87.5	0.76	71.1	74.4	**90.6**	0.82	66.7	66.7	85.7	0.75	69.6	74.0	83.8	0.79
	3GM	68.2	66.7	100	0.80	72.0	69.6	**100**	0.82	66.7	69.8	93.8	0.80	41.7	0	0	0	67.6	68.0	**97.1**	0.80
	1-2GM	63.6	71.4	71.4	0.71	68.0	90.0	56.2	0.69	57.8	93.3	43.8	0.60	58.3	66.7	57.1	0.62	70.6	79.7	75.0	0.77
	BERT	81.0	100	42.9	0.60	62.5	55.6	50.0	0.53	55.3	30.8	33.3	0.32	63.6	60.0	60.0	0.60	67.4	62.8	65.9	0.64
FT - BERT	-	77.3	61.5	100	0.76	80.0	66.7	88.9	0.76	92.0	100	77.8	0.88	76.7	77.2	**90.7**	0.83	76.9	61.90	82.9	0.71

3.2.1.1 HttpCore

On the HttpCore data set, we found that SVM with bigram performed better than SVM with all other embeddings. Even though SVM with trigram achieved a higher recall value (100%) than the one of SVM with bigram (92.9% recall), SVM with bigram achieved higher precision value (72.2%) and F1-score (0.81) than SVM with trigram (66.7% precision and 0.80 F1-score). In other words, SVM with trigram was able to detect all the constraints in the HttpCore data set, but SVM with trigram also classified higher number of non-constraint sentences as constraints than SVM with bigram. Thus, we concluded that SVM with trigram achieved better performance than SVM with bigram.

Naive Bayes with bigram achieved a recall value of 71.4%, a precision value of 66.7% and an F1-score of 0.69. Even though Naive Bayes with BERT had a perfect recall value, it scored a precision value (50%) similar to a random classifier and a lower F1-score (0.67) than Naive Bayes with bigram. So, we concluded that Naive Bayes with bigram embedding performed the best among all the variants of Naive Bayes.

The decision tree classifier with bigram performed better than all other variants of decision tree in this system. It reached a 100% recall, 73.7% precision and 0.85 F1-score.

Fine-tuned BERT achieved 100% recall and 61.5% precision value. Even though this model achieved a perfect recall value, it achieved lower precision than decision tree with bigram.

Considering the overall results, we concluded that decision tree with bigram embedding performed the best on the HttpCore data set, as it achieved a perfect recall value and the highest F1-score.

3.2.1.2 jEdit

On the jEdit data set, SVM with unigram performed better than SVM with bigram, reaching a recall value of 81.2% and a precision value of 86.7%. Even though SVM with bigram embedding achieved a higher recall value (87.5%) than SVM with unigram, SVM with bigram achieved a lower precision value (70%) and lower F1-score (0.78) than SVM with unigram. Another aspect to highlight here is that SVM with unigram and SVM with unigram+bigram (*i.e.,* n-gram where

$n = \{1, 2\}$) resulted in an identical performance. This means that in this case, SVM was able to find all the necessary keywords to classify the incoming sentences with unigrams, and adding bigrams to the mix did not make any difference. Lastly, SVM with trigram performed poorly in jEdit achieving 0% precision and recall. It indicates that SVM with trigram did not find any helpful three-word length keyword which can be used to differentiate constraints from non-constraints. As a result, SVM with trigram failed to detect all of the constraints present in the testing data set.

Naive Bayes with bigram, trigram and BERT embeddings performed equal or worse than a random classifier for jEdit in terms of recall (no value greater than 50%). Naive Bayes with unigram embedding and unigram+bigram embedding performed better than other variants of Naive Bayes, both achieving a recall value of 81.2%, a precision value of 92.9% and an F1-score of 0.87. Even though Naive Bayes with unigram and unigram+bigram embedding achieved the same performance, we concluded that Naive Bayes with unigram performed better than the other because the unigram embedding uses a lower number of keywords than unigram+bigram to achieve the same performance.

In the case of the decision tree based models, the trigram variation was able to achieve the highest recall value (100%) with a precision value of 69.6% and an F1-score of 0.82. Even though decision tree with unigram and unigram+bigram achieved higher precision values (100% and 90% respectively) than decision tree with trigram, we still selected the trigram as the best performing variation of decision tree as we focused more on the higher recall values than higher precision values. The BERT embedding variation achieved a recall value of 88.9% and a precision value of 66.7%.

Overall, decision tree with trigram embedding performed the best in this data set, achieving a perfect recall, 69.6% precision and 0.82 F1-score. This model achieved the highest recall value in the jEdit data set.

3.2.1.3 Swarm

SVM with trigram embedding performed better than other SVM variations on the Swarm data set. It was able to achieve 93.8% recall, 69.8% precision and an F1-score of 0.80. Although other

variations of Swarm obtained a higher precision value than trigram variation, their recall values were lower than that of the trigram variation.

The BERT variation of the Naive Bayes model reached the highest recall value (100%) obtained for this system but obtained a precision value of only 37.5%. Instead, the bigram variation of Naive Bayes achieved a recall value of 78.1%, a precision value of 80.6% and an F1-score of 0.79. As the bigram variation achieved better recall value than all other embeddings (except Naive Bayes with BERT), we concluded that Naive Bayes with bigram performed better than all other variants of Naive Bayes.

In the case of the decision tree classifiers, the bigram variation performed better (90.6% recall, 74.4% precision and 0.82 F1-score) than other embeddings. Even though decision tree with trigram achieved a higher recall value (93.8%), decision tree with trigram achieved a lower precision (69.8%) and F1-score (0.80) than decision tree with bigram. Fine-tuned BERT achieved a recall value of 77.8% and a precision value of 100%.

Overall, the decision tree with bigram performed better than all the other classifiers. Decision tree with bigram achieved the second highest recall value and the second highest F1-score in the Swarm data set.

3.2.1.4 ArgoUML

ArgoUML was the smallest data set in our study. Still, SVM and Naive Bayes managed to perform at a similar level in ArgoUML compared to the other data sets. Decision tree suffered in this small data set, whereas Fine-tuned BERT performed the best among all the classifiers.

SVM performed better with BERT than with other embeddings on the ArgoUML data set. It achieved 89.1% recall, 71.4% precision and an F1-score of 0.79. All the embeddings of SVM (except trigram) achieved recall values higher than 85%.

The Naive Bayes model with BERT embedding achieved the highest recall value (89.1%) among all the embeddings of Naive Bayes in this data set, a precision value of 71.4% and an F1-score of 0.79. On this data set, the decision tree models did not perform well (except with

the bigram embedding). Decision tree with bigram was able to achieve 85.7% recall and 66.7% precision, but with trigram it scored a value of 0% in both recall and precision.

On the other hand, Fine-tuned BERT performed well on the ArgoUML data set. It achieved a recall value of 90.7%, a precision value of 77.2% and an F1-score of 0.83. This model achieved the highest recall value and the highest F1-score in ArgoUML data set. So, we selected Fine-tuned BERT as the best performing model in this data set.

3.2.2 Model performance across “All Systems” data set

Considering all the subject systems as a single data set, we constructed the “All Systems” data set. In this data set, SVM performed better with the trigram embedding than with the bigram and unigram one, achieving an F1-score of 0.80, a recall value of 97.1% and a precision value of 68.0%. Even though the rest of the embeddings achieved higher precision values than trigram, those embeddings achieved lower precision values than trigram. All of the SVM based models achieved F1-score higher than 0.72 in this data set.

In the case of the Naive Bayes models, the BERT embedding achieved a perfect recall value (100%) and an F1-score of 0.67. However, its precision value (50.6%) is comparable to that of a random classifier. Thus, we selected the Naive Bayes model with unigram+bigram (66.2% recall, 91.8% precision and 0.77 F1-score) to be the best performing model on this data set.

In the case of the decision tree based models, we selected the trigram embedding as the best variation (with 97.1% recall, 68% precision and 0.80 F1-score). Decision tree with trigram and SVM with trigram achieved identical performance in this data set. Both of these models achieved the highest recall value and the highest F1-score in this data set.

Fine-tuned BERT achieved a recall value of 82.9%, a precision value of 61.9%, an accuracy value of 76.9% and an F1-score of 0.71.

We concluded that decision tree with trigram and SVM with trigram performed the best in this data set. Both of these models achieved the highest recall value and the highest F1-score in this data set.

3.2.3 Overall comparison between classification models

Even though decision trees are some of the most trivial models in the machine learning field, they performed unexpectedly well in our study. Our decision tree based models were able to achieve the best performance on all the data sets (except ArgoUML). This result is inline with other studies [34] that have also found decision tree based models to perform well for text classification. Nevertheless, these classifiers did not perform consistently well across all the data. For example, the decision tree model with unigram on the jEdit data set (56.2% recall), the decision tree model with BERT on the jEdit data set (50% recall), and the decision tree model with trigram on the ArgoUML data set (0% recall) performed close or worse than a random classifier. Dalal and Zaveri [35] argues that decision tree based models generally perform well in text classification but suffers when the number of attributes in the data is too large.

We found that Naive Bayes based models cannot perform as well as decision tree for textual classification tasks. Naive Bayes assumes conditional independence, *i.e.,* it assumes that "features are independent of each other given the class"[18]. However in natural language, most features are not independent of one another.

Overall, the performance of the Fine-tuned BERT models was consistently good throughout all the systems. The recall values for these models ranged in between 77.8% and 100%, and their precision values ranged in between 61.5% and 100%. Even though some of the other models performed better than Fine-tuned BERT on some particular cases, in other cases those models performed even worse than a random classifier. For example, decision tree with trigram embedding achieved the best performance in the jEdit and "All Systems" data sets (with recall values of 100% and 97.1% respectively) among all the models. However, in the case of ArgoUML data set, decision tree with trigram achieved 0% recall value. In case of SVM, SVM with trigram achieved the best performance in "All Systems" data set (97.1% recall and 68.0 precision) but performed poorly (0% recall and 0% precision) in the jEdit data set. Fine-tuned BERT can perform better than other models in small data sets, such as our ArgoUML data set.

---

[18]https://nlp.stanford.edu/IR-book/html/htmledition/properties-of-naive-bayes-1.html

In general, all the models performed better on the "All Systems" data set than on the single-system data sets. On the single-system data sets, even though a model performed well, some other models clearly failed. The main reason for the models to perform better in the "All Systems" data set is—again—the number of data points available for training. The higher the number of data points is, the better the overall model performance. For example, in the "All Systems" data set, the Fine-tuned BERT model achieved an F1-score of 0.71, the variants of SVM obtained F1-scores in the range of 0.72 to 0.80, the variants of decision tree obtained F1-score in the range of 0.64 to 0.80, and the variants of Naive Bayes obtained F1-score in the range of 0.61 to 0.77.

ArgoUML was the hardest data set to classify for the decision tree based models. In ArgoUML, the decision tree based models achieved F1-scores in the range of 0 to 0.62 (except bigram embedding). We attribute this situation to the size of the ArgoUML data set, which is the smallest one in our study (see Table 2.2). Because of this, the algorithms did not get enough data points to build a proper classification model and hence behaved poorly in the testing data set.

The trigram encoding did not perform well in the ArgoUML data set. SVM, Naive Bayes and decision trees achieved poor recall values (57.1%, 42.9% and 0% respectively) with trigram embeddings. Compared to trigram, unigram and bigram performed well, with recall values ranging from 71.4% to 85.7% (except decision tree with unigram). This indicates that there are very few keywords with three-word length that exists in the ArgoUML data set.

3.2.4 Choosing the best model

In Table 3.1, we have marked the model performing best for each data set in bold. Decision tree with bigram achieved the best performance in HttpCore and Swarm data set. Decision tree with trigram achieved the best performance in jEdit and "All Systems" data set. Fine-tuned BERT achieved the best performance in ArgoUML data set. SVM with trigram achieved the best performance in the "All Systems" data set.

Even though decision tree with trigram achieved the best performance in jEdit and "All Systems" data set, decision tree with trigram performed poorly (0% precision and recall) in ArgoUML.

The same situation occurred in the case of SVM with trigram. Even though SVM with trigram achieved the best performance in "All Systems" data set, SVM with trigram performed poorly (0% precision and recall) in jEdit. Thus, we discarded decision tree with trigram and SVM with trigram as options to be selected as the best performing model.

The remaining two choices for choosing the best performing model were decision tree with bigram and Fine-tuned BERT. Decision tree with bigram achieved recall values in the range of 83.8% and 100%, precision values in the range of 66.7% and 74.4%. Whereas, Fine-tuned BERT achieved recall values in the range of 77.8% and 100%, precision values in the range of 61.5% and 100%. If we compare these two choices data set by data set, we can see that decision tree with bigram performed better than Fine-tuned BERT in three data sets (HttpCore, Swarm and "All Systems"). Fine-tuned BERT performed better than decision tree with bigram in two data sets (jEdit and ArgoUML). As decision tree with bigram performed better than Fine-tuned BERT in higher number of data sets, we concluded that decision tree with bigram performed the best among all the models.

3.2.4.1 Keywords used by decision tree with bigram

One advantage of decision tree based models is that the internal structure of the model is transparent, which means that we can understand the decision making process of the model by analyzing the structure of the tree. Figures 3.1, 3.2, 3.3, 3.4 and 3.5 show the four splitting nodes closest to the root node of the decision tree with bigram based models for every data set. Even though the height of these trees was greater than four, we present only the top four levels for the sake of simplicity.

Let us consider the structure of the decision tree with bigram embedding for the HttpCore data set (see Figure 3.1). The most frequent bigram token in the HttpCore data set was "repeatable entity", and all of the sentences containing this keyword were marked as constraints. As a result, the decision tree algorithm can predict with high confidence that a sentence containing this token is a constraint, and does not require any other token in the sentence to compare. Thus, the trained decision tree based model placed this token in the root node. The numeric value on the right side

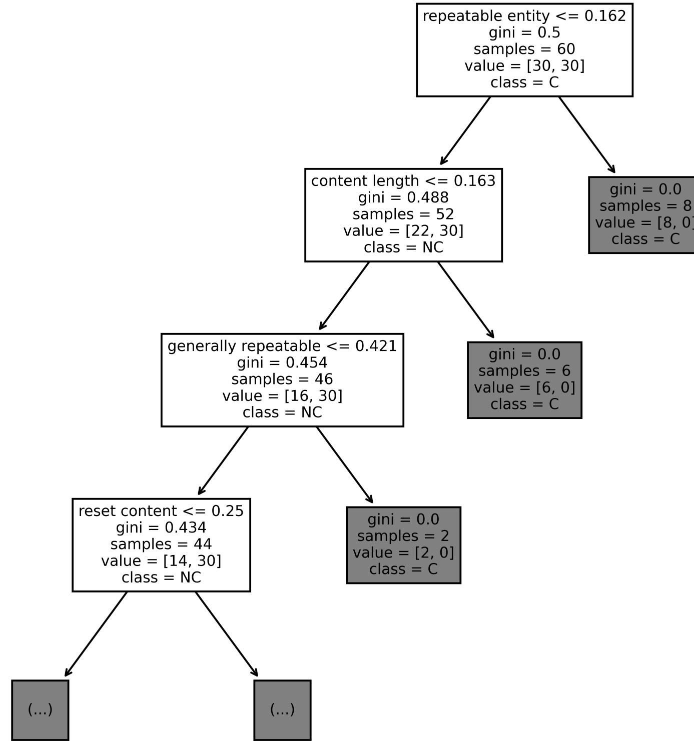


Figure 3.1: Top four tokens used by decision tree with bigram in HttpCore data set

of the inequality referring to the token (*i.e.,* "repeatable entity") indicates the threshold value for the token. This means that if the TF-IDF score of this keyword in an input sentence is lower than the threshold value (0.162), the model will confidently classify that sentence as a constraint. The attribute "value" in the nodes represents the distribution of constraints and non-constraints in the sub-tree rooted at that node. The attribute "class" represents the majority class present in that node.

3.2.5 In-depth analysis of misclassified cases

In this section, we analyze some example sentences where one of the models made some mistakes in the classification process, but other models were able to correctly classify the sentence.

Example 3.2.1. "If nothing happens, you can run the application from a command (or DOS) prompt to see if there are any errors that can be used for troubleshooting."

Example 3.2.1 is part of the user manual of Swarm. Although it was originally labeled as a non-constraint sentence, Fine-tuned BERT classified it as a constraint. From a grammatical perspective,

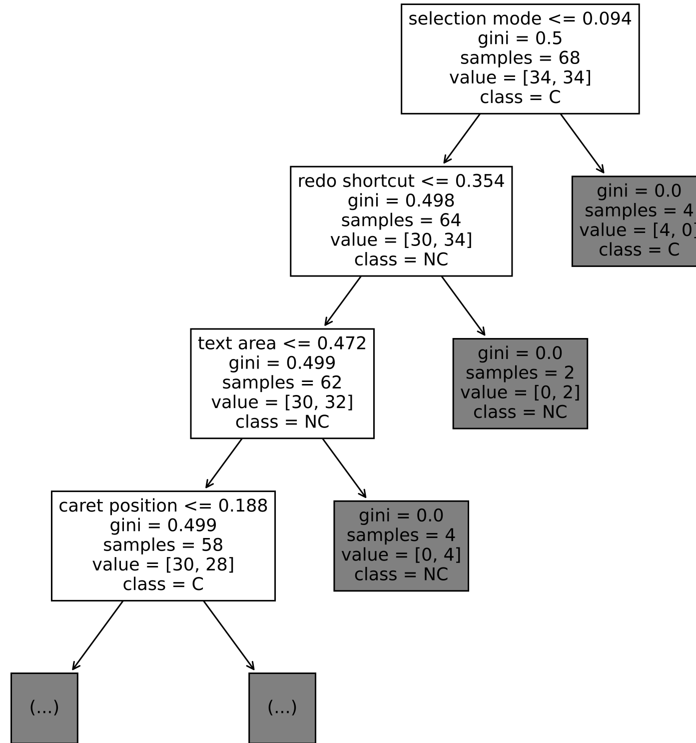


Figure 3.2: Top four tokens used by decision tree with bigram in jEdit data set

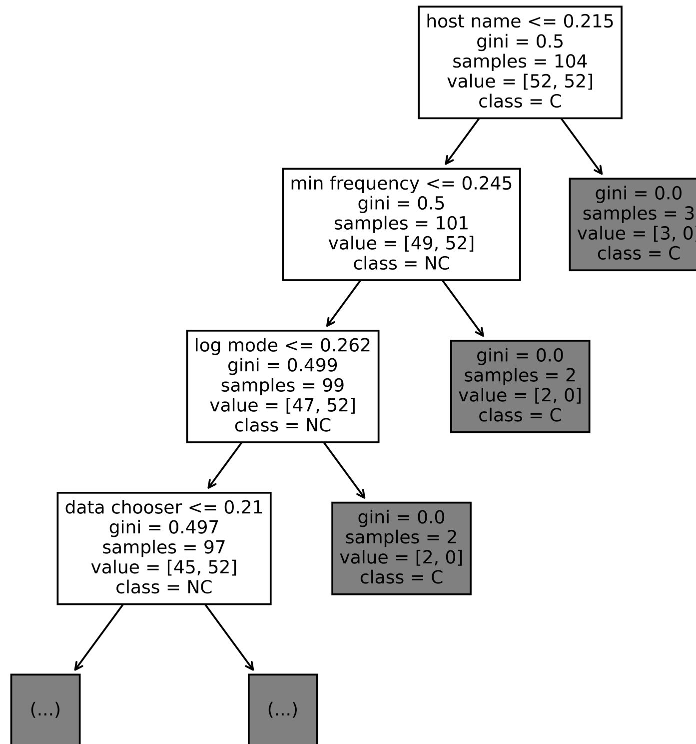


Figure 3.3: Top five four used by decision tree with bigram in Swarm data set

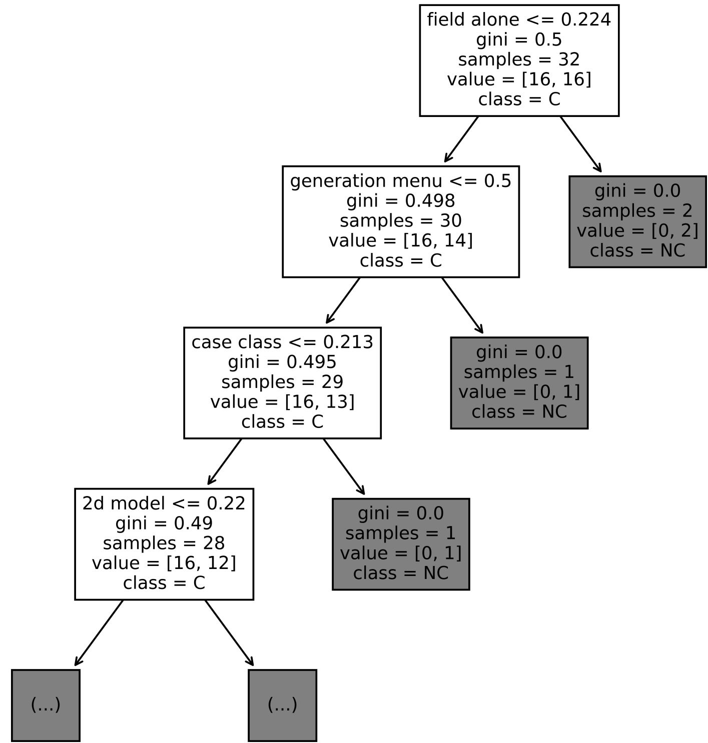


Figure 3.4: Top five four used by decision tree with bigram in ArgoUML data set

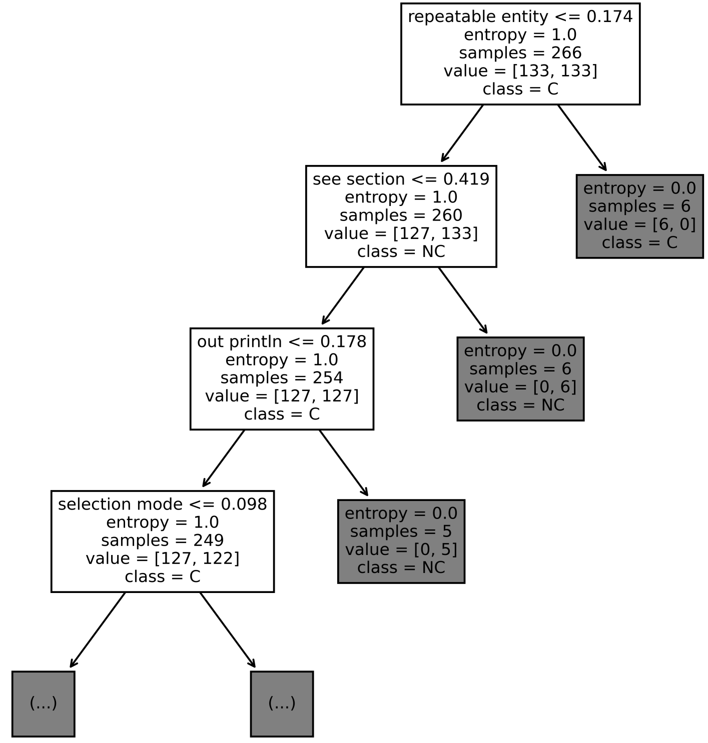


Figure 3.5: Top four tokens used by decision tree with bigram in “All Systems” data set

this is a conditional sentence. In our training data set, many of the sentences containing constraints followed a conditional structure, where an action might or might occur depending on a condition. So naturally, Fine-tuned BERT considered this sentence as a constraint, even though it is merely suggesting the user to try another process if one fails. This situation indicates that although Fine-tuned BERT is able to find relational patterns between the words of a sentence, it cannot understand the meaning of that relation, and thus, it failed in this case. The same phenomenon was observed with other similar sentences (*e.g.,* “For example, if some text was inserted, Undo will remove it from the buffer").

Example 3.2.2. “Range selections are equivalent to selections in most other text editors; they cover text between two points in a buffer.”

Example 3.2.2 is an excerpt from the user guide of jEdit. This sentence is not a constraint, because it only compares a feature of jEdit with that of other systems and does not describe any limitation of the feature in jEdit. Nonetheless, the decision tree model with bigram classified this sentence as a constraint. There is a sentence containing constraint in the training data set “Range Selection Dragging the mouse creates a range selection from where the mouse was pressed to where it was released” (also excerpted from jEdit). As a result, when decision tree with bigram found the keyword “Range Selection” in if Example 3.2.2, it classified it as a constraint. This is one of the problems of n-gram based models: they look for keywords to classify a sentence and do not try to understand their meaning.

Example 3.2.3. “Multiple Selection Edit > More Selection > Multiple Selection (keyboard shortcut: C+/) turns multiple selection mode on and off.”

Example 3.2.3 is another excerpt from the user guide of jEdit, which is a constraint because it restricts the possible values for multiple selection mode. All the models except Fine-tuned BERT classified it as a constraint. The n-gram models found the popular keywords of constraints: “on” and “off”, and classified this sentence as a constraint. When Fine-tuned BERT analyzed the structure of the sentence, the model got confused at the beginning of the sentence: “Multiple Selection

Edit > More Selection > Multiple Selection". This portion of the sentence describes the menu path to change the multiple selection mode in jEdit. This is not a common sentence grammatically speaking and thus the Masked Language Model of Fine-tuned BERT failed to recognize it as a constraint. A similar phenomenon was observed in another example sentence from the user guise of jEdit: "HOME is bound to Smart Home". This sentence is describing the shortcut functionality of the key "HOME" in the jEdit application. Without the context of its parent paragraph, Fine-tuned BERT failed to parse this sentence correctly. Based on these examples, we can see that Fine-tuned BERT sometimes fails to properly classify sentences that do not follow a common structure. However, n-gram based models were able to recognize those constraints with the help of popular keywords.

Example 3.2.4. "Earthworm data provides raw wave data only."

Example 3.2.4 is part of the user manual of Swarm. This sentence is a constraint because it limits the possible output formats by the Earthworm sub-system. However, the decision tree model with bigram marked this sentence as a non-constraint. Although there were multiple constraint sentences in the training data set containing the token "Earthworm" (*e.g.,* "Connection to Earthworm requires the IP address or host name of the server, port number, and communication time out in seconds"), there was also one non-constraint sentence in the train data containing this token ("Figure 4 Adding new Earthworm data source 3"). In the case of the decision tree with bigram embedding, the produced bigram was "Earthworm data". This token was not present in the training samples of constraints but was present in the training samples of non-constraints. After the vectorization process, there is no way to detect that the token "Earthworm data" contains the token "Earthworm" in it. As a result, the decision tree model failed to classify this sentence as a constraint.

Example 3.2.5. "*Visibility* This checkbox allows to hide the visibility of a package."

Example 3.2.5 is an excerpt from the user manual of ArgoUML. This sentence is a constraint because it describes the functionality of a checkbox that is capable of controlling the visibility of a

package. All the sentences containing the token “checkbox” in the training data set are constraints (*e.g.,* “Operation This checkbox allows to hide or show the operations compartment of a class or interface”). As a result, all the decision tree based models were able to classify this sentence as a constraint. Decision tree with bigram based models found the bigram token “checkbox allows” and decision tree with trigram based models found the token “this checkbox allows”. These tokens helped the models to detect this sentence as a constraint. Instead, Fine-tuned BERT failed to classify this sentence as a constraint. The reason lies behind the text styling adopted by the documentation of ArgoUML. Whenever describing an item in a list, the authors of the documentation did not add any line break to differentiate the title of the item from the description of the item. Rather the authors just changed the style of the text for the portion of the title of the item. When we built our data set for training purposes of the models, we discarded the styling of the text. We have mentioned before that Fine-tuned BERT depends on the masked language model and fails to classify a sentence correctly when the grammatical structure of the sentence is not correct. The first word of the Example 3.2.5 is the title of an item in a list. As this kind of sentence structure is not common in the English literature, BERT was not pre-trained on this kind of sentences, so it failed to classify this sentence as a constraint.

3.3 Strategy 2: Train and test the models on different data sets

As mentioned before, we considered four data sets in our study. In Section 3.2, we presented the results of training and testing the models on the same data set. In this section, we present the results of training our models on three of these data sets and testing them on the other one. This will help us understand how our models would perform on the documentation of an OSS system for which the models were not trained on.

Table 3.2 reports on the performance of the models when trained on the data sets of the other three software systems and tested on the one at hand. For example, the metric values listed in the column “HttpCore” represent the performance of the models trained on the full data sets of jEdit, Swarm, and ArgoUML, and tested on the full data set of HttpCore.

We can observe that decision tree with bigram was able to achieve the best performance in the testing data sets of HttpCore, jEdit and Swarm. This model also achieved 100% recall in all the four testing data sets (HttpCore, jEdit, Swarm and ArgoUML), but its precision was rather low (in the range between 52.3% and 56.8%). These high recall and low precision values indicate that decision tree with bigram was able to detect all the constraints in the testing data sets, but misclassified many non-constraints as constraints.

Fine-tuned BERT achieved the best performance in the ArgoUML testing data set with 73.1% recall and 70.6% precision. Even though decision tree with bigram was able to achieve a perfect recall value, its precision (52.3%) was much lower than that of Fine-tuned BERT (70.6%). In the jEdit testing data set, Fine-tuned BERT achieved only 48.8% recall value. This is the only time we observed Fine-tuned BERT performing worse than a random classifier.

SVM with trigram was also able to achieve 100% recall in all of the testing data sets, but similarly to the decision tree model with bigram, its precision also suffered (ranging from 52.3% to 56.4%).

In the case of Naive Bayes based models, none of its variants were able to perform consistently well in terms of recall across the four testing data sets (unlike decision with bigram and SVM with trigram). Naive Bayes suffered more in the HttpCore testing data set compared to the other testing data sets (with a highest recall value of 52.3%). This indicates that the assumption of feature independence in the sentences of the HttpCore testing data set did not hold well in these cases.

As the decision tree model with bigram achieved the highest recall values in all the testing data sets and also achieved higher precision than SVM with trigram, we concluded that decision tree with bigram is the best performing model in this testing strategy.

As mentioned earlier, we consider the performance of a random classifier as the baseline of comparison. Overall, our best performing models in each of the testing data sets achieved better recall and precision values than a random classifier. This indicates that with the addition of more data sets from other open-source software systems, our models will be able to perform even better on the data sets on which the models were not trained.

Table 3.2: Accuracy (AC), precision (PC), recall (RC) and F1-score (F1) comparison between different machine learning models across all systems for testing strategy 2. Abbreviations used for the sake of brevity: Model (MDL), Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), BERT model fine-tuned with Adam (FT-BERT), embedding (EMD) unigram (1GM), bigram (2GM), trigram (3GM), unigram+bigram (1-2GM). The underlined values represent the highest recall value obtained for a system by a particular model and embedding. The **values in bold** indicate the highest recall value obtained for a system.

MDL	EMB	System															
		HttpCore				jEdit				Swarm				ArgoUML			
		AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1	AC (%)	PC (%)	RC (%)	F1
SVM	1GM	56.1	57.7	68.2	0.62	62.4	59.3	96.0	0.73	53.0	57.1	47.6	0.52	52.3	52.9	78.3	0.63
	2GM	52.4	53.1	97.7	0.69	54.8	54.3	100	0.70	57.7	57.1	**100**	0.73	52.3	52.3	100	0.69
	3GM	53.7	53.7	100	0.70	53.8	53.8	100	0.70	56.4	56.4	100	0.72	52.3	52.3	100	0.69
	1-2GM	53.7	55.2	72.7	0.63	61.3	58.3	**98.0**	0.73	54.4	56.2	85.7	0.68	52.3	53.1	73.9	0.62
	BERT	55.6	62.4	71.8	0.67	56.7	72.9	66.7	0.70	52.1	55.4	74.6	0.64	52.3	56.4	75.1	0.64
NB	1GM	41.5	43.8	31.8	0.37	57.0	56.8	84.0	0.68	50.3	57.1	47.6	0.52	61.4	63.6	60.9	0.62
	2GM	61.0	67.6	52.3	0.59	58.1	61.2	60.0	0.61	54.4	55.6	94.0	0.70	52.3	52.3	100	0.69
	3GM	52.4	56.8	47.7	0.52	53.8	53.8	100	0.70	54.4	56.5	83.3	0.67	52.3	52.3	100	0.69
	1-2GM	41.5	44.1	34.1	0.38	59.1	58.1	86.0	0.69	48.3	53.5	63.1	0.58	61.4	63.6	60.9	0.62
	BERT	38.6	45.8	48.9	0.47	53.5	62.3	78.7	0.69	45.0	56.3	72.5	0.63	63.4	64.9	55.3	0.64
DT	1GM	62.2	59.4	**93.2**	0.73	49.5	51.9	82.0	0.64	55.0	56.3	90.5	0.69	59.1	56.8	91.3	0.70
	2GM	53.7	53.7	100	0.70	53.8	53.8	100	0.70	57.0	56.8	**100**	0.72	52.3	52.3	100	0.69
	3GM	53.7	53.7	100	0.70	53.8	53.8	100	0.70	56.4	56.4	100	0.72	52.3	52.3	100	0.69
	1-2GM	46.3	50.0	13.6	0.21	45.2	49.2	60.0	0.54	55.0	57.0	82.1	0.67	52.3	52.4	95.7	0.68
	BERT	53.7	61.4	72.9	0.67	43.8	46.7	62.3	0.53	59.1	55.7	67.2	0.61	51.8	71.2	52.9	0.61
FT - BERT	-	69.5	70.9	57.9	0.64	48.4	44.7	48.8	0.47	68.5	59.8	84.6	0.70	68.1	70.6	**73.1**	0.71

3.4 Summary and discussion

We formulated a working definition of software constraints after studying existing literature and exploring software documentation. We built a data set containing 167 constraints and 199 non-constraint sentences. This data set is meant to be used in future studies.

We analyzed the performance of several constraint classification models generated with four machine learning algorithms. In our study, the decision tree based model with bigram embedding performed the best among all the models, with recall values ranging from 83.8% to 100% and precision values ranging from 66.7% to 74.4%.

On our smallest data set (*i.e.,* ArgoUML), Fine-tuned BERT achieved 90.7% recall and 77.2% precision. BERT has been pre-trained on thousands of sentences from the English language beforehand, thus it was able to achieve such a performance. We also found that if a sentence is not properly structured, Fine-tuned BERT can fail to classify the sentence correctly.

Some studies [36] suggest that SVM models generally achieve high precision and poor recall, but we did not find any such relation in our study. In some cases, we found SVM to achieve high precision with low recall, and in some others, low precision with high recall. For example, the SVM model with trigram achieved high recall (100%) but low precision (66.7%) on the HttpCore data set. But on the same data set, SVM with bigram achieved low precision (72.2%) with high recall (92.9%).

Even though Naive Bayes models are trivial classifiers, the multinomial variant performs well in text classification problems[19], which was confirmed in our study. For example, Naive Bayes with BERT embedding performed better than the decision tree with bigram embedding on the ArgoUML data set. Domingos and Pazzani [37] also observed the same result and reasoned that even if the probability estimation of Naive Bayes is slightly off, the correct class still gets a higher probabilistic value than the other classes. As we select the class with the highest probabilistic value, the result still remains correct. Naive Bayes based models can perform better than decision tree based models on small data sets (for example, ArgoUML data set), but latter ones perform

[19] https://scikit-learn.org/stable/modules/naive_bayes.html

better than the former on larger data sets (for example, "All Systems" data set). Kohavi et al. [38] and Domingos and Pazzani [37] also observed the same effect of data set size on the performance of Naive Bayes and decision tree models.

Decision tree based models generally need some key nodes for good performance. In Figures 3.1, 3.2, 3.3, 3.4 and 3.5 we have shown the top key nodes in each of the data sets. In the case of text classification, the model can suffer when the dimensionality of the data is high. Decision tree models performed worse (*e.g.,* Decision Tree with trigram on the ArgoUML data set) relative to Fine-tuned BERT when the size of the data set was small. However, we observed improvements in the performance of the decision tree classifiers when trained on larger data sets. For example, the decision tree model with bigram performed better than the Fine-tuned BERT model on the HttpCore data set. But when the size of the data set grew larger (in the "All systems" data set), the decision tree models resulted in lower precision than Fine-tuned BERT. Other studies observed a similar effect of the size of the data set on the performance of decision tree models. Catlett [39] states that the recall of decision tree models increases with larger data sets, but if the size of the data sets keeps increasing, the precision value and F1-score start to drop after some point. For example, in the case of decision tree with unigram we noticed that the precision values increased when the number of data points in the data set were less than 100 (see Figure 3.6), but when the number of data points increased to more than 100, the precision values declined.

We also found that, even if our models were not trained on the documentation of a software system, the models can still detect constraints better than the baseline random classifier in that software system. This in turn proves that our models will be able to perform better if the models are trained on more data sets from different software systems.

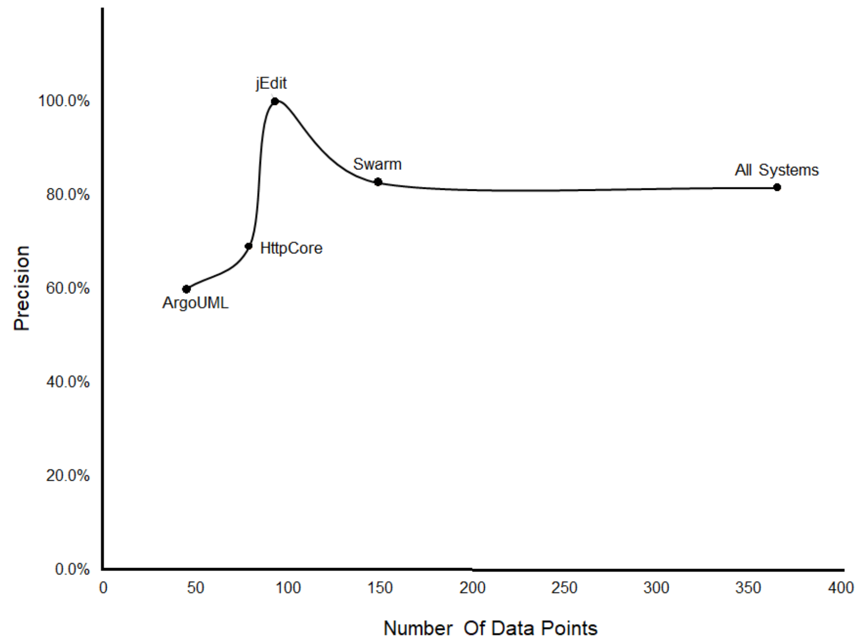


Figure 3.6: Precision trend of the decision tree model with unigram

Chapter 4

Related work

Relevant to our research is the work on constraint categorization and their automatic detection in the documentation. Also related to our work is the detection of other types of contents (*e.g.,* functional or non-functional requirements and business rules) in software documentation. In this chapter, we describe existing taxonomies and detection efforts focused on constraints, requirements and business rules, and how our work differs from them.

4.1 Constraints

4.1.1 Constraint categorization

Despite their importance, not many research efforts have focused on characterizing and categorizing software constraints.

4.1.1.1 Taxonomy of privacy constraints by Breaux and Antón

As part of their work on semantic parsing of privacy constraints from regulatory rules, Breaux and Antón [3] proposed a taxonomy for the constraints found in the Health Insurance Portability Act[20] (HIPAA). Their goal was to reduce ambiguity between constraints and prioritize them, to support requirement engineers in the analysis of constraints of healthcare software systems.

Breaux and Antón's approach used semantic parameterization to build formal models of constraints from natural language. Semantic parameterization is the process of parsing sentences from natural language form into a formal first-order predicate logic form. The constraints were divided into two classes namely, parameterized and non-parameterized constraints. At the end of the parameterization process, the policies in natural language form are parsed and mapped to different

[20]https://www.hhs.gov/hipaa/index.html

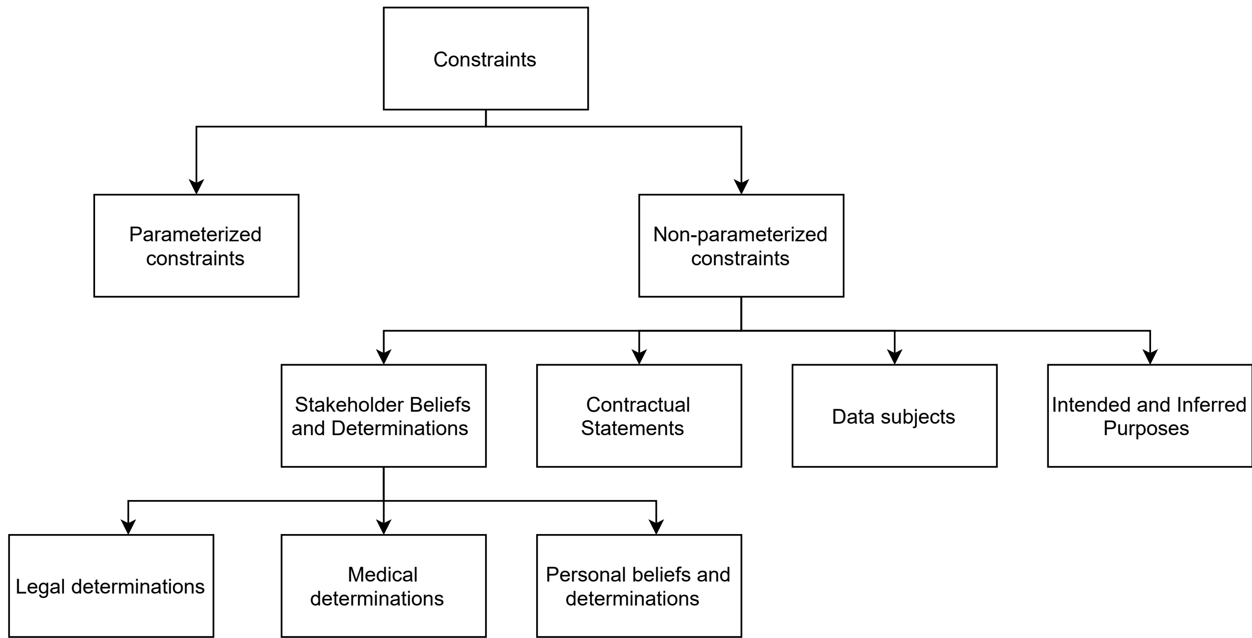


Figure 4.1: Constraint classification by Breaux and Antón [3]

properties such as subject, action, modality, object, target and purpose. The full taxonomy is shown in Figure 4.1 and discussed below:

- *Parameterized constraints.* After parsing these constraints, the aforementioned properties are mapped to a single word or a noun phrase from the sentence.
- *Non-parameterized constraints.* In this kind of constraints, the aforementioned properties cannot be mapped to a single word or a noun phrase from the sentence. Instead, these properties are mapped to the phrases with "wh" questions. Non-parameterized constraints can be divided into four sub-classes.
  - *Stakeholder beliefs and determinations.* These constraints involve a decision-making process, which gives access to some data. They can further be divided into three sub-classes depending on whether the decision is made by a legal authority, medical personal, or by an average person.
  - *Contractual statements.* These constraints restrain data access depending on whether a person has been legally granted access to the data.

- *Data subjects*. These constraints restrain data access permissions depending on the ownership of the data.
- *Intended and inferred purposes*. These constraints impose access permissions on the data depending on the task for which the data will be used.

To the best of our knowledge, this is the only taxonomy of constraints proposed in the literature so far. The focus of Breaux and Antón's taxonomy is on privacy and security constraints that protect patients' information. This differs from our research goal, which is to detect *all* software constraints in software documentation.

4.1.1.2 Types of constraints in the ISO/IEC/IEEE standards

The ISO/IEC/IEEE 29148 standard [11] does not explicitly describe a specific taxonomy of constraints. However, the following constraint types are mentioned along the document:

- *Business operational constraints*. Constraints that describe limitations on how a business process will be executed.
- *Operational constraints*. Constraints to successfully implement the business operation into the system. This may end up creating new functional requirements.
- *Project constraints*. Constraints describing the limitations on project timelines, financial budget, etc.
- *Design constraints*. Constraints describing the rules needed to be abide by the laws from government entities or to satisfy customer requirements. For example, a customer might impose conditions on design notation and coding standards.
- *Interface constraints*. Constraints describing the limitations of an external entity that the system needs to communicate with.
- *Physical size constraints*. Constraints describing the weight or volume limitations on the hardware itself.

- *Constraints derived from law*. Constraints that are imposed by government entities for the safety of the users.
- *Maintenance constraints*. Constraints imposed on the maintenance activity of the system.
- *User or operator limitations*. Constraints imposed on the system due to the limited work capacity or speed of humans.
- *System performance constraints*. Constraints defining a benchmark (*e.g.,* time, memory) for the system to be acceptable to the customers.
- *Safety constraints*. Constraints the system has to maintain to ensure the security of user data or the physical well-being of the user.
- *Data availability constraints*. Constraints determining how user data will be stored and accessed in the system.
- *Memory constraints*. Constraints on how much primary and secondary memory the system is permitted to use.
- *Integrity constraints*. Constraints on the integrity of the user data. It is similar to the *referential integrity constraints* in database systems.

In addition, Wiegers and Beatty [40] collected the next types of constraints from the ISO/IEC/IEEE 29148:2011(E) standard [41].

- *Technology stack constraint*. Constraints on the tool, language or framework to be used while building the software.
- *Platform constraint*. Constraints that are imposed on the system because of using a certain operating system or browser.
- *Compatibility constraints*. When a new version of the system comes up, it has to be compatible with its older versions such that customers using the older versions of the software do not face any issues.

- *Compliance constraints*. Mirror the *constraints derived from laws* discussed earlier.
- *Hardware constraints*. Mirror the *physical size constraints* and *memory constraints* discussed earlier.
- *Physical constraints*. Same as the *physical size constraints* discussed earlier.
- *Interface constraints*. Same as the *interface constraints* discussed earlier.
- *Data format constraints*. Constraints on the data standard or data format to be used to communicate with external entities.

The aforementioned constraints can be thought of as sub-classes of software constraints. These sub-classes are in no way an exhaustive list and they do not represent a complete taxonomy of *constraints*. Moreover, some of the sub-classes might even overlap with each other.

We explored these constraint types in the literature to find out what type of sentences are considered as constraints. Doing so helped us to understand the definition of constraints better which in turn reduced disagreements in the coding phase of the thesis.

4.1.2 Automatic detection of constraints in documentation

Our main goal was to automatically detect *software constraints* in natural language texts. In other words, we wanted to classify natural language sentences into *constraints* and *non-constraints* classes. There are not a lot of works existing in the literature on constraint detection in software documentation.

4.1.2.1 Automated extraction of access control policies by Xiao et al.

Security policies can be considered as a sub-class of *software constraints*. Xiao et al. [4] defined access control policies (ACP) as “principals such as users have access to which resources". ACP falls into the similar category of *safety constraints* discussed earlier. As these sentences limit the access permission of certain data, ACP is considered as sub-class of *software constraints*.

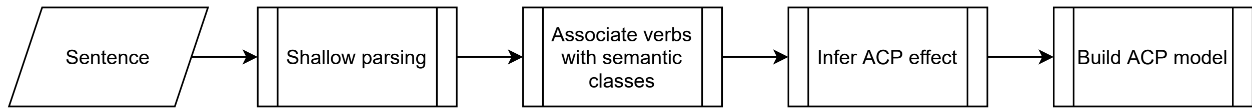


Figure 4.2: Automatic extraction of Access Control Policies by Xiao et al. [4]

The overall steps to build a formal ACP model from natural language text are shown in Figure 4.2. We are going to explain the steps mentioned in the figure using Example 4.1.1 shown below:

Example 4.1.1. “Admin should not access user’s personal financial data".

The authors analyzed the sentence with the shallow parser. The parser segmented the sentence into subject, object, verb groups, phrases, clauses. The example sentence would be analyzed as <Admin, subject>, <should not access, main verb group>, <user’s personal financial data, object>. Then the authors would compare the verbs with a pre-built dictionary of verbs to retrieve semantic classes. The example verb group would be matched with the “Deny” action effect. The subject, object and the action effect would then be encoded in formal XACML format. In a nutshell, the authors relied on the pre-built list of verbs to identify whether a sentence contains an access control policy (ACP) or not.

4.2 Software requirements

ISO/IEC TS 24748 [42] defines a software requirement as a “statement that translates or expresses a need and its associated constraints and condition”. Sommerville [5] classified software requirements into the next three major categories: *non-functional requirements*, *functional requirements*, and *domain specific requirements*. In this section, these requirement categories will be briefly discussed.

4.2.1 Non-functional requirements

ISO/IEC/IEEE 24765 defines a non-functional requirement as a "software requirement that describes not what the software will do but how the software will do it" [43]. Sommerville [5] clas-

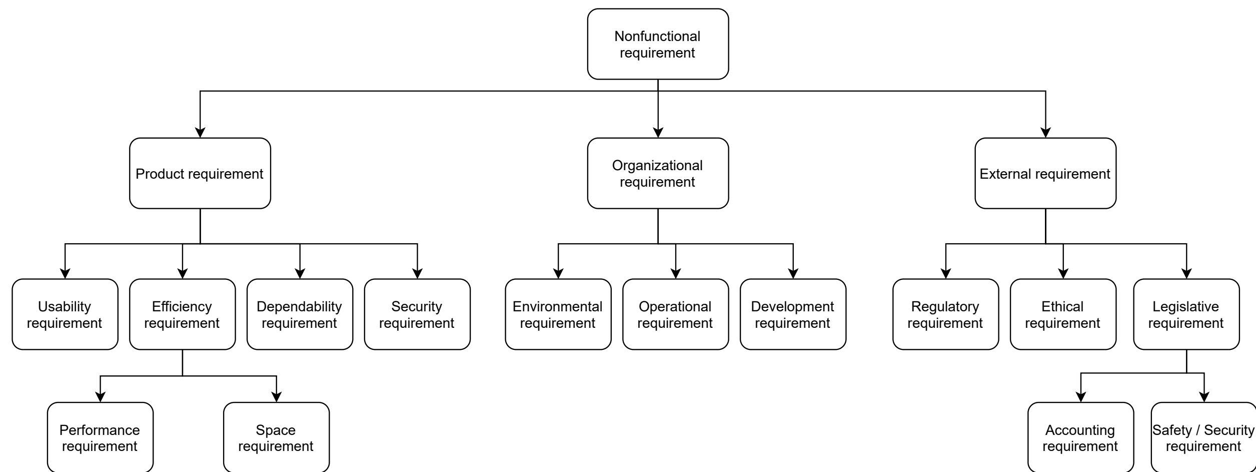


Figure 4.3: Taxonomy of nonfunctional requirements by Sommerville [5]

sified nonfunctional business requirements into three subcategories, namely product requirements, organizational requirements, and external requirements. The taxonomy is shown in Figure 4.3.

There is a large body of research literature [12, 13, 44, 14, 15] present on nonfunctional requirements categorization and how to automatically classify them. Hence, we do not concentrate on non-functional requirements.

4.2.2 Functional requirements

ISO/IEC/IEEE 24765 defines a functional requirement as a "statement that identifies what results a product or process shall produce" [43]. In general, functional requirements refer to features or functionality the software offers to its users. There have been relatively fewer research efforts on classifying functional requirements.

4.2.2.1 Taxonomy of functional requirements by Ghazarian [45]

Ghazarian [45] studied documentation from fifteen projects in the domain of enterprise systems [45]. 1217 functional requirements were found after analyzing the documentation of those projects Those requirements were classified into twelve classes. They are mentioned below:

- *Data output*. The requirements that describe the format of the output data. The output can be the final output itself or it can be the intermediate output between modules.

- *Data input*. The requirements that describe the format the system expects the input data to be. Similar to output requirements, the data input can describe intermediate data input formats between modules of the system.
- *Event trigger*. The requirements that describe the system response on the actions initiated by users. For example: Keypress events.
- *Business logic*. The requirements that describe the business processes of the system.
- *Data persistence*. The requirements that describe how data will be stored in the system.
- *User Interface navigation*. The requirements that describe the sequential flow of screens on different user activities.
- *External call*. The requirements that describe how the system will interact with other software and vice versa.
- *Communication*. The requirements that describe the requirements or contents in case of communicating with any external entities.
- *User Interface*. The requirements that describe the screen layout of the system.
- *User Interface logic*. The requirements which describe the choices of screen flow depending on the user interaction.
- *Data validation*. The requirements that describe the criteria that determine whether the input data valid.
- *External behavior*. The requirements that describe the reaction of any external sub-component of the system.

The author provided definitions of each of these classes but did not provide any examples. "Data output", "Data input", "Event trigger", "Business logic" and "Data persistence" were the

most common functional requirements in these fifteen projects. All together these classes comprised 85% of all the functional requirements found. “External behavior” and “Data validation” were the least common functional requirements. The author also argued that the low concentration of some of these requirement types can indicate the bad quality of the software itself.

We identified some problems with this taxonomy:

- The author listed business rules in the “Business logic” category. But we know that business rules are different from requirements.
- The proposed taxonomy is more implementation-oriented than documentation-oriented. Most of the popular classes are related to the developed code. For example - Data output, data input.

4.2.2.2 Taxonomy of functional requirements by Sharma and Biswas

Sharma and Biswas [6] presented a taxonomy of functional requirements in this study [6]. They built this taxonomy by following the grounded theory [46] approach. The grounded theory mainly focuses on verifying assumptions by analyzing the data. In other term, grounded theory tries to find clues grounded in the data.

The authors analyzed documentation from five enterprise software systems. These systems were in the domain of academic, finance and healthcare sectors. Three of these documents were in free flow format and the rest of them contained structured use cases. The authors found 3060 requirement data points from those documents. In the first step of analyzing the requirements, the authors followed the open coding approach. The findings were frequently compared with other members. In the next phase, the authors performed selective coding to identify the central category and integrate the other sub-categories to the central category to design an initial taxonomy. In the last step, they did theoretical coding to define relationships between the finalized categories. The final taxonomy is shown in Figure 4.4. The authors classified functional requirements into seven categories. Types of requirements that each of these categories contain, are listed below each of these categories in the figure.

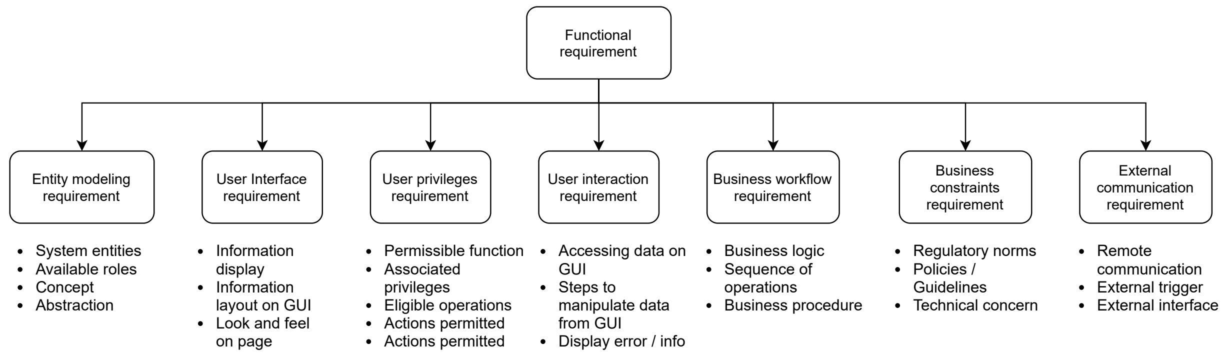


Figure 4.4: Taxonomy of functional requirements by Sharma and Biswas [6]

4.2.3 Automatic classification of functional and non-functional requirements by Kurtanović and Maalej

The authors devised a machine learning model that can automatically separate non-functional requirements from functional requirements [12]. They used the data set from "Second RE17 data challenge: the identification of requirement types using Quality attributes"[21]. There were 625 sentences in the data set and for each sentence in the data set, there was a label associated with it. The label indicated whether a sentence is a functional requirement or if it is a non-functional requirement what subclass does it belong to. There were eleven sub-classes in the non-functional requirement category: Availability, Fault tolerance, Look and feel, Maintainability, Operational, Performance, Portability, Scalability, Legal, Usability and Security.

The model was able to successfully differentiate between functional and non-functional requirements. In the case of binary functional and non-functional classification, the model achieved a precision value of around 0.92 and a recall value of around 0.92 while no automatic feature selection was being applied. In the case of multi-class classifications for non-functional requirements, the model achieved precision values in the range of 0.65 to 0.86 and recall values in the range of 0.77 to 0.82.

[21] http://ctp.di.fct.unl.pt/RE2017/downloads/data sets/nfr.arff

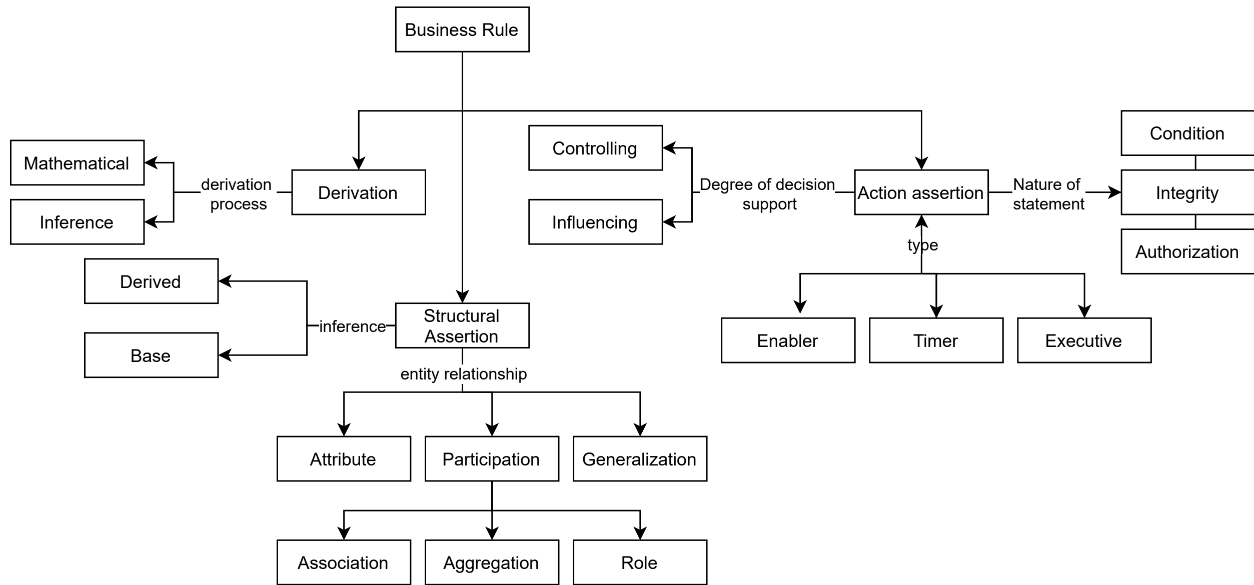


Figure 4.5: Business rule taxonomy by Ross [7]

4.3 Business Rules

A business rule is “a statement that defines or constrains some aspect of the business” [47]. Business rules define core regulations or policies of a company, which have to be maintained as guidelines to conduct the business activities of the company. These rules can be imposed by a government entity or by the company itself.

In this chapter, we explain previous works on business rules by providing a brief description of these taxonomies, as well as some of their pros and cons.

4.3.1 Taxonomy by Ross

Ross [7] is one of the pioneers in the field of classifying and modeling business rules. The GUIDE project [47] summarizes his proposed taxonomy. Business rules are common in almost every type of business industry and so this taxonomy was not restricted to the software industry. It uses a hypothetical EU-Rent’s car rental business to exemplify the rule categories in the taxonomy.

The taxonomy is represented in Figure 4.5. As the taxonomy is quite big and is not directly related to our study we will not dive deep into each sub-category. Ross classifies business rules into three major categories:

- *Structural Assertion (Fact).* A business rule that describes a concept or an existing entity in relation to other concepts or entities of interest [47]. An *entity* can be an actor or object in the system.

  Structural assertions can be classified into two subcategories based on their creation process: *Base* and *Derived*. *Base* rules are atomic and *Derived* rules are constructed from other rules by mathematical calculations or logical inference.

  Structural assertions can also be classified into three categories based on the relationship between terms. A *term* can be any word or phrase that the business uses when modeling its system. The three categories are: *Attribute* describing the property of a business entity, *Generalization* describing the subset or superset relationship among entities and *Participation* describing the aggregation or association relationships between entities.

- *Derivation.* A business rule that stems from one or more atomic business rules (by *mathematical* calculation or by *inference*) is called a *derived* business rule [47].

- *Action Assertion.* A business rule that specifies control over the possible results of an action [47].

  Action assertions can be divided into three categories based on the nature of the statement – *Conditional* action assertion leads to other action assertions based on the condition, *Integrity* action assertion that must be preserved, *Authorization* action assertion grants privileges to actors.

  Action assertions can also be categorized based on the action type. *Enabler* action assertion initiates another action assertion whereas *Executive* assertion acts as a trigger to execute another action assertion. *Timer* action assertion defines a threshold in time after which another action is auto-initiated.

  Action assertions can also be divided into two categories based on the flexibility of the resultant action: *Controlling* and *Influencing*.

Ross's was one of the first efforts towards building a taxonomy for business rules. The taxonomy attempted to classify every business rule into one or more categories. However, we found several problems when trying to adopt this taxonomy for our classification task.

- The taxonomy was published in 1997 and has not been updated since. Meanwhile, the software industry has evolved. Many modern requirements documents consist mostly of diagrams rather than text. Traditional databases (which inspired the original taxonomy) have been replaced with other technologies. Some parts of the taxonomy focus on the entity (Structural assertion) relationship similar to database systems. It means that these types of relations mainly focus on relations between entities in a sentence. This type of relation-focused taxonomy can not help much in automatic binary textual classifications as they do not focus on the sentence structure.

- The examples mentioned in the taxonomy are not related to the software industry.

- It is one of the most complex business rule taxonomy in existence. One rule can fit into multiple categories depending on the scheme of the classification. For example, a sentence might fall into the "Derived" subcategory of the "Structural Assertion" category if the relationship between entities is considered. But if the inference procedure is considered, then the same sentence falls into the "Derived" or the "Base" subcategory. This complexity became particularly problematic in the tagging step of our data collection process. As a single sentence was tagged by multiple coders, for most of the sentences there was at least one subcategory mismatch between the tagged sub-classes of the coders.

- Modern software practitioners consider definitions and non-functional requirements as business rules [1]. But the taxonomy does not consider them as business rules.

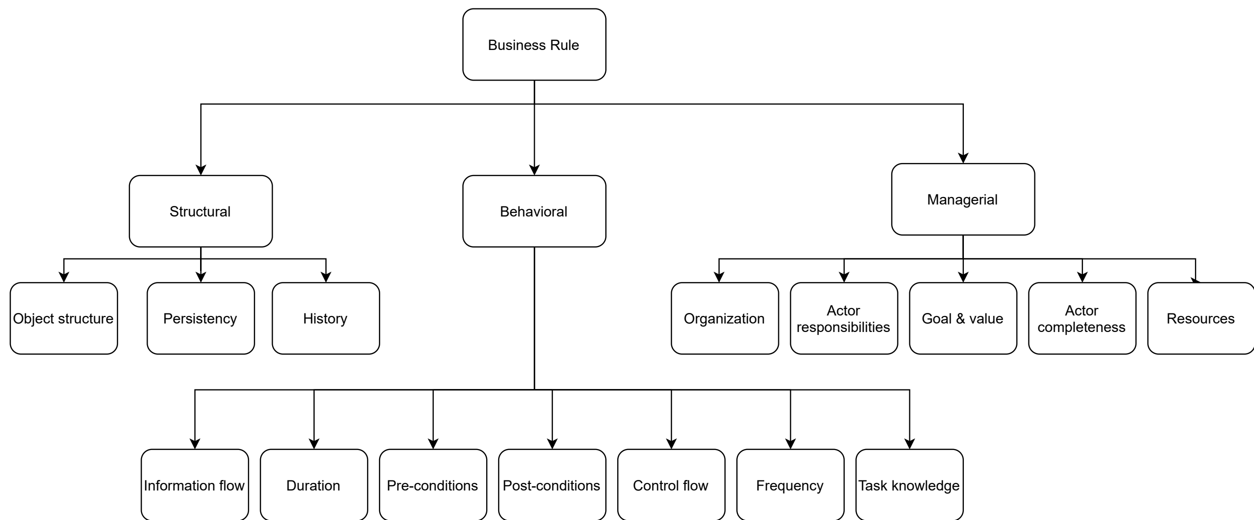


Figure 4.6: Business rule taxonomy by Weiden et al. [8]

4.3.2 Taxonomy by Weiden et al.

Weiden et al. [8] proposed a new taxonomy primarily based on the taxonomy by Ross [7] with some modifications. They classified the business rules into three main categories. The detailed taxonomy is shown in Figure 4.6 and is described below.

- *Structural*. This category focuses on business relations between objects, and how long the information or history of an object should be stored. Rule types in this category include:
  - *Object Structure*. A business rule that lists the objects or entities in the system. It can also describe the relationships between objects.
  - *Persistence*. A business rule that describes the data persistence timeline of objects.
  - *History*. A business rule that describes whether the system should store the history of some particular object.
- *Behavioral*. This category focuses on conditional relations and sequential relations between business tasks. Rule types in this category include:
  - *Information flow*. A business rule that describes the required information of a task from other tasks.

- *Preconditions*. A business rule that describes the conditions needed to be fulfilled for the current task to initiate.
- *Post-conditions*. A business rule that describes the conditions that will appear once the current task finishes executing.
- *Frequency*. A business rule that describes how frequently a task can be performed.
- *Duration*. A business rule that describes how long a task should last.
- *Control flow*. A business rule that controls the execution order of multiple tasks.
- *Task knowledge*. A business rule that describes the information or knowledge needed to perform a business task.

• *Managerial*. This category focuses on the organization and resources of the business.

- *Organization*. A business rule that describes the organizational policies.
- *Goal and value*. A business rule that describes the goal of the company.
- *Actor competencies*. A business rule that describes the quality of actors that make them competent for the business.
- *Actor responsibility*. A business rule that describes the tasks one actor should perform.
- *Resources*. A business rule that describes the policies to make good use of company resources.

Weiden et al. [8] conducted a case study to evaluate to what extent their taxonomy (see Figure 4.6) fits into an industry level software and to refine their initial taxonomy if necessary. The case study was based on interviews with the business analysts of a mortgage offering processing software. The identified 320 business rules were stored both in natural text and in a relational database, which were also categorized according to their proposed taxonomy. UML diagrams were also considered as semi-formatted business rules. The classification process, however, was a manual effort, and although some of the business rules fit within the software domain, the taxonomy

covered too many aspects of businesses in other fields. For example, we can consider the subcategory “Actor competency" in the category of “Managerial" rules. The authors added sentences to this sub-category that describe an actor’s skills necessary for the role. These sentences describing an actor’s skills are clearly business rules but they are not specifically related to any actor related to any software systems. Besides, the taxonomy was also old (published in 2002).

4.3.3 Taxonomy by Maalej and Ghaisas

Maalej and Ghaisas [1] surveyed different types of business rules by interviewing 11 experienced software practitioners from different fields of the software industry. The goal of this study was to find out the perceptions of business rules in the stakeholders of the software industry. The interviewees worked in various software systems related to financial services, insurance and telecom industries. The authors asked the interviewees questions such as the following:

- What types of sentences they consider as business rules.
- Questions related to common requirement engineering practices.
- What type of business rules they have used. They were also asked to give some examples for each of them.
- They were asked about the problems faced while maintaining these rules throughout the life cycle of the software.

The interviewees mentioned 27 different types of rules. The authors found that practitioners have a broad perception for this term, ranging from flow of business processes to directives for calling external system interfaces.

The authors were motivated by two phenomena:

- One simple change in a business rule results in a significant ripple effect. This means that a change in a single business rule might affect some other business rules that depended on the original rule. As a result, many requirements might change with them, which in turn can lead to an extended delay in the delivery timeline of the software.

- Most of these rules are “domain, region, or company specific rather than application specific” [1]. Thus, the authors suggested that it might be a good idea to design an automated approach to extract business rules from artifacts to increase code reusability.

The authors found that the understanding of business rules varies from one practitioner to another depending on their area of expertise, company culture and role. They mentioned that one of the software practitioners considers business rules as "non-negotiable constraints which are driven by corporate policy or regulations". Even some of the other practitioners think that non-functional rules are a certain category of business rules. These notions of business rules of software practitioners in the current software industry do not really reflect the formal definition of business rules found in the literature. The authors also mentioned that these types of business rules suggested by the practitioners, are not mutually exclusive, and are non-exhaustive. This implies that there are more types of business rules out there than mentioned by the practitioners. The top five types of business rules (ordered by how many practitioners mentioned them) are listed below.

- Validations rules and value ranges
- System or application specific rules
- Calculation rules
- Access control rules
- External system interfaces

However, the authors did not provide any well-organized definitions or examples for them. We have listed these definitions in Table 4.1 that were provided by the authors throughout the study. Some of the definitions were collected from other studies [16] conducted by the same authors [1]. The sub-classes for which the authors did not mention any definition have been omitted.

This taxonomy was one of the few modern efforts to classify business rules. One big plus point for this taxonomy is that the authors tried to build the taxonomy keeping the feedback from current software practitioners in mind.

Even though the authors have tried to make a list of the business rules, here are some problems we found while following their taxonomy:

- The authors only listed the names of the categories of the business rules they had collected by interviewing the software practitioners. They did not provide any definition for most of the categories. They only added examples for a few categories.
- The research questions the authors were trying to answer were not directed towards building a taxonomy of business rules. Their main goal was to find out how today's software industry uses business rules while building the software.

4.4 Automatic detection and classification of business rules

Several research efforts have focused on building approaches to automatically classify business rules. In this section, we describe the existing approaches to automatically classify business rules.

4.4.1 Automatic detection and classification of business rules by Ghaisas et al.

Ghaisas et al. [16] collected samples of business rules from documentation of 20 large projects in the insurance domain. To automatically classify them, they introduced rule intent patterns. Rule intent patterns are syntactic patterns consisting of phrases and keywords in business rules. These patterns contained parts of speech tags, keywords and wildcard characters. For example, consider a business rule mentioned by the authors.

Example 4.4.1. "During a call to the Service Router, the application in context will be locked to the active user."

The rule intent patterns of this sentence are:

- During $\star$,
- $\star$ call `TO` + `NN` $\star$,

- $\star$ application + `MD VB VBN` $\star$, and

- + `TO` $\star$ user $\star$;

where `TO` represents the preposition *to*, `NN` indicates noun, `MD` indicates modal, `VB` indicates verb, `VBN` indicates verb in past participle form, and '$\star$' is a wild character to indicate occurrence of any words with 0 or more frequency. Each of these patterns represents a rule intent. The analysis of the 20 documents yielded 517 rule intents.

After analyzing all the training sentences, the rule intents that appear together in the same types of business rule sentences, were grouped into a sub-class. To classify a sentence into categories of business rules, the approach detects all the rule intents present in that sentence first. Then, it finds the subclass that generally contains those rule intents in group in a sentence. That sub-class represents the category of business rules for that sentence.

We considered whether using rule based approach can achieve better performance than machine learning based classification approach. Although it was a good idea to find rule intents in the sentences, the task to detect and group rule intents needed heavy manual effort. When the data set grows larger, it becomes practically impossible to group rule intents from each of those sentences. So, this system was not scalable. Also, in the tagging step of finding those pattern-based rule intents, it is hard to keep track of the already found rule intents. While building those rule intents, If a person tags two sentences with the same rule intent pattern in a week apart, there is a high chance that he will not remember that he has already tagged a sentence with a similar structure into another rule intent. Considering these reasons, we decided to use machine learning based approach instead of rule based approach.

4.4.2 Automatic classification approach by Anish et al.

Anish et al. [2] devised a machine learning based approach that can automatically classify business rules. Ghaisas [1] was also a part of this research team. In their previous work, Ghaisas et al. [16] expressed each of those business rules into a composition of multiple rule intents. Each of these intents were associated with a part of speech (POS), keywords and wild character based pat-

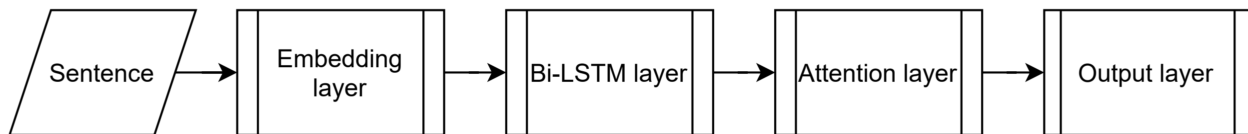


Figure 4.7: Automatic classification approach by Anish et al. [2].

terns. However, as mentioned earlier, it is difficult to build patterns for a large data set. Thus, Anish et al. [2] introduced machine learning algorithms and new types of business rules (see Table 4.2) on top of the ones proposed by Maalej and Ghaisas [1].

For their data set, they collected 4,043 sentences from 40 Software Requirement Specification (SRS) documents in the insurance domain. Their model is shown in Figure 4.7. It has five logical layers.

In the embedding layer, they used Word2Vec algorithm [48] to convert words into numbers. In the next layer, those numbers were fed into a BiLSTM model [49] to preserve contextual dependencies of words in both directions. An aggregated value is collected from all the nodes in a weighted manner. In the output layer, different sigmoid functions were used to classify sentences into a specific business rule category.

For binary classification (business rule sentence vs. non-business rule sentence), the proposed approach achieved 88.11% precision and 87.6% recall value. In the case of classifying sentences according to their categories, the approach achieved precision values in the range of 55.36% to 99.9%, while the recall value was in the range of 44.44% to 83.87%. Some of the categories (*i.e., Deadline*, *User Interface*, *Data Protection*) were exceptionally easy to classify for the model and had a high precision value. However, for some of the other categories (*i.e., Data Validation*, *Financial Transaction*), the model performed similarly to a random classifier model. For the low recall values, the authors argued that they had very few data points for those categories in their data set. Also in the case of *Data Validation*, the authors mentioned that this category sentence contains a wide range of information and it can have multiple sub-categories within itself.

Even though this approach was a good option for our binary classification task between constraints and non-constraints, we did not use this approach in our thesis. This model did not consider

Table 4.1: Partial taxonomy by Maalej and Ghaisas [1]. The sub-classes for which the authors did not mention any definition have been omitted

ID	Type	Definition
1	Validations rules and value ranges	"Rules that validate data processed by the system or used by stakeholders to perform some task" [16]
2	System / application specific rules	"Restricting the deployment of the application, how it should be used" [1] and rules that may change from systems to systems
4	Calculation rules	How to calculate a particular value [1]
5	Access control rules	"Rules that restrict stakeholder's access or constraints activities performed by them" [16]
8	Sequencing/ control flow	"Rule that restricts the functionality of the system, its behavior or the process" [1].
12	User interface rules	"Rules related to user interface describing various screen elements, screen layout etc" [16].
17	Non-functional rules	"Non-functional rules include availability restrictions, performance rules, and the number of concurrent users at a time" [1].

Table 4.2: New types of business rules proposed by Anish et al. [2]. Definitions are verbatim from the original publication.

Business rule type	Definition
Deadline	Rules that restrict time duration, date.
Conditional Execution	Rules describing checks to be performed by user or system while executing process steps.
Data Protection	Rule act that mandates the practices to be followed to protect sensitive data.
Documentation Mandate	Rule act that mandates the information that needs to be provided in a document.
Financial Transaction	Rule act that restricts the monetary transactions.
User Responsibility	Rule act that mandates the user to perform some action.

every type of business rule found in their previous and new studies (it only considered 10 types of business rules). Anish et al. used BiLSTM to train their model. At that time BiLSTM was one of the state of the art machine learning models. BiLSTM is slower than BERT as BiLSTM can input one word at a time at the time of training, whereas BERT comes as pre-trained over the internet. Also, BiLSTM tries to understand the bidirectional context by concatenating two unidirectional training phases. So, BiLSTM is not a true bidirectional context aware model.

4.5 Summary

In this chapter, we described taxonomies and automatic classification approaches on constraints, requirements and business rules related to our work. Although we have not used any of the taxonomies discussed in this chapter, readers can use it as a starting point to learn about existing approaches for text classification in software documentation.

Chapter 5

Conclusions

5.1 Lessons learned

In this thesis, we developed an approach to identify software constraints from software documentation. With this purpose, we constructed a working definition of constraints based on existing works on constraints in the literature. We analyzed the documentation of four open-source software systems and built a data set containing examples of constraint and non-constraint sentences.

We also built machine learning based models that can classify sentences from software documentation into two categories: constraint and non-constraint. The machine learning model based on decision tree algorithm with bigram embedding was able to achieve 74% precision, 83.8% recall and 0.79 F1-score. Considering the small size of our data set (consisting of 368 data points), this was a good result. Other binary classification models, such as the classification model of functional and non-functional requirements (see Section 4.2.3) by Kurtanović and Maalej [12] achieved 92% precision and recall on a data set consisting of 625 data points. With a larger data set, we believe that our models could have achieved similar or better results. Kurtanović and Maalej [12] also used SVM with n-gram based models similar to ours (see Section 2.3.2.2). To the best of our knowledge, ours is the first attempt to automatically detect constraints in software documentation artifacts.

We observed some interesting trends in the behavior of the machine learning algorithms and derived models in our testing strategy 1. These behaviors have been discussed (see Section 3.4) in detail. Our decision tree with bigram and Fine-tuned BERT models performed consistently better than the other models in all the data sets. Fine-tuned BERT was able to achieve 90.7% recall in a small data set like ArgoUML, because the model was already pre-trained on thousands of English language sentences. However, Fine-tuned BERT cannot perform correctly if the sentence structure is uncommon. Unlike other studies, we did not find any co-relation between precision and recall

values in the case of SVM based models. In the case of decision trees, we observed improvements in the performance of the models when the size of the training data set increased, just like in other similar studies. In fact when the size of the data set is small, Naive Bayes can perform better than the decision tree based models. However, with the increase of the size of the data set, decision trees gradually outperform Naive Bayes based models.

5.2 Future Work

In the short term, training the model on a larger data set is likely to produce better results. To increase the size of our data set, we must annotate more sentences and maybe other documents. To execute this task and mitigate the threats to validity discussed in Section 2.5, we plan to recruit more coders in the future.

Although we performed an intrinsic evaluation of our approach with testing strategy 2, we are yet to evaluate how much it helps developers when working on real-world software systems. Running the approach on the documentation of industry-level software is necessary to assess its usefulness and is left as future work.

Currently, our model can classify sentences into one of two classes: constraints and non-constraints. We believe that this model can be extended to be a multimodal classifier. For example, the model could detect sentences that express constraints, requirements or business rules.

In the long term, the next step of the research is to develop an approach to trace constraint sentences to their implementation in source code. This step is challenging because of various reasons, such as irregularities in identifier names and complex source code structure. Moreover, in current practice, values of environmental and configuration variables for live systems are stored in a separate portal rather than directly in the source code. Also, the documentation of software systems is often not well maintained and hence could be inconsistent with respect to the code.

Ideally, our tracing approach should be able to link sub-classes of constraints to code. To do so, we must understand how developers implement different types of constraints. If we find that the implementation of different types of constraints differs and we can detect the type of constraint

to trace, we might be able to apply a strategy that is suitable to the constraint at hand. In that case, a taxonomy for different constraint types and a model capable of classifying those constraints are necessary.

Bibliography

[1] Walid Maalej and Smita Ghaisas. Capturing and sharing domain knowledge with business rules lessons learned from a global software vendor. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 364–373. IEEE, 2014.

[2] Preethu Rose Anish, Abhishek Sainani, Abdul Ahmed, and Smita Ghaisas. Implementation-centric classification of business rules from documents. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 227–233. IEEE, 2019.

[3] Travis Breaux and Annie Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE transactions on software engineering*, 34(1):5–20, 2008.

[4] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[5] Ian Sommerville. Software engineering 9th edition. *ISBN-10*, 137035152:18, 2011.

[6] Richa Sharma and Kanad K Biswas. Functional requirements categorization grounded theory approach. In *2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 301–307. IEEE, 2015.

[7] Ronald Ross. *The business rule book: classifying, defining and modeling rules: Ross method*. Database Research Group, Boston, Mass, 1997. ISBN 9780941049030.

[8] Marcel Weiden, Leo Hermans, Guus Schreiber, and Sven van der Zee. Classification and representation of business rules. *Universiteit van Amsterdam*, 2002.

[9] Petr Marounek. Simplified approach to effort estimation in software maintenance. *Journal of systems integration*, 3(3):51–63, 2012.

[10] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003.

[11] ISO/IEC/IEEE international standard - systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, pages 1–104, 2018. doi: 10.1109/IEEESTD.2018.8559686.

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

[13] John Slankas and Laurie Williams. Automated extraction of non-functional requirements in available documentation. In *2013 1st International workshop on natural language analysis in software engineering (NaturaLiSE)*, pages 9–16. IEEE, 2013.

[14] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. Automated classification of non-functional requirements. *Requirements engineering*, 12(2):103–120, 2007.

[15] Mengmeng Lu and Peng Liang. Automatic classification of non-functional requirements from augmented app user reviews. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 344–353, 2017.

[16] Smita Ghaisas, Manish Motwani, and Preethu Rose Anish. Detecting system use cases and validations from documents. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 568–573. IEEE, 2013.

[17] Preethu Rose Anish, Abhishek Sainani, Abdul Ahmed, and Smita Ghaisas. Implementation-centric classification of business rules from documents. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 227–233. IEEE, 2019.

[18] Ralph Rowland Young. *The requirements engineering handbook*. Artech House, 2004.

[19] D. I. K. Søberg, R. Welland, and M. P. Atkinson. Software constraints for large application systems. *The Computer Journal*, 40(10):598–616, 1997. doi: 10.1093/comjnl/40.10.598.

[20] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[21] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. *Qualitative Data Analysis: A Methods Sourcebook*. SAGE Publications, Inc, Thousand Oaks, Califorinia, third edition, 2014. ISBN 978-1-4522-5787-7.

[22] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

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

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[25] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. 1763. *MD computing: computers in medical practice*, 8(3):157–171, 1991.

[26] William S Noble. What is a support vector machine? *Nature biotechnology*, 24(12):1565–1567, 2006.

[27] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[28] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3): 273–297, 1995.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[30] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[31] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

[32] Juan Ramos. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48. Citeseer, 2003.

[33] K Mouthami, K Nirmala Devi, and V Murali Bhaskaran. Sentiment analysis and classification based on textual reviews. In *2013 international conference on Information communication and embedded systems (ICICES)*, pages 271–276. IEEE, 2013.

[34] Arundhati Navada, Aamir Nizam Ansari, Siddharth Patil, and Balwant A Sonkamble. Overview of use of decision tree algorithms in machine learning. In *2011 IEEE control and system graduate research colloquium*, pages 37–42. IEEE, 2011.

[35] Mita K Dalal and Mukesh A Zaveri. Automatic text classification: a technical review. *International Journal of Computer Applications*, 28(2):37–40, 2011.

[36] M Ikonomakis, Sotiris Kotsiantis, and V Tampakas. Text classification using machine learning techniques. *WSEAS transactions on computers*, 4(8):966–974, 2005.

[37] Pedro Domingos and Michael Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classi er. In *Proc. 13th Intl. Conf. Machine Learning*, pages 105–112. Citeseer, 1996.

[38] Ron Kohavi et al. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Kdd*, volume 96, pages 202–207, 1996.

[39] Jason Catlett. Overprvning large decision trees. In *IJCAI*, pages 764–769. Citeseer, 1991.

[40] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[41] ISO/IEC/IEEE international standard - systems and software engineering – life cycle processes –requirements engineering. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94, 2011. doi: 10.1109/IEEESTD.2011.6146379.

[42] ISO/IEC TS 24748-1:2016. ISO/IEC TS 24748-1:2016 Systems and software engineering – Life cycle management – Part 1: Guidelines for life cycle management. Standard, International Organization for Standardization, 2016.

[43] Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017. doi: 10.1109/IEEESTD.2017.8016712.

[44] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[45] Arbi Ghazarian. Characterization of functional software requirements space: The law of requirements taxonomic growth. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 241–250. IEEE, 2012.

[46] Ciarán Dunne. The place of the literature review in grounded theory research. *International journal of social research methodology*, 14(2):111–124, 2011.

[47] David Hay, Keri Anderson Healy, and J Hall. Defining business rules-what are they really. *Final report*, 34, 2000.

[48] Tomas Mikolov, Kai Chen, Gregory S Corrado, and Jeffrey A Dean. Computing numeric representations of words in a high-dimensional space, May 19 2015. US Patent 9,037,464.

[49] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. doi: 10.1109/78.650093.