An Intelligent Framework to Generate Use Case Diagrams and Class Diagrams from Requirements Documents

Thamer A. Alrawashdeh, Adnan A. Hnaif, Mustafa Alrifaee, and Mohammed S. Kamel

***Abstract*—Use case and class diagramming are essential requirements engineering techniques that play a pivotal role in modeling software specifications and facilitating the software development process. However, software requirements are often expressed in Natural Language (NL), which can be ambiguous, noisy, immeasurable, and open to interpretation. This research addresses these challenges by automatically extracting the required elements to generate use case and class diagrams from software requirements documents written in natural language. Accordingly, an automated framework is proposed based on Natural Language Processing (NLP) techniques—such as tokenization and part-of-speech tagging—to parse the software requirements syntactically using a set of heuristic rules. These rules facilitate the extraction of actors, use cases, entities, relationships, and attributes required for generating the corresponding diagrams. Furthermore, to enhance the framework's performance, the k-nearest neighbor (k-NN) algorithm is employed to predict previously processed requirements and reduce redundant computation. The framework's effectiveness was evaluated using two performance metrics: recall and precision. Experimental results show that the proposed approach achieves an average recall of 96% and an average precision of 92%, confirming its robustness and reliability.**

***Index terms*—artificial intelligence, software engineering, use case diagram, class diagram, requirements documents.**

I. Introduction

In software engineering, understanding system requirements is crucial for building effective and efficient software solutions. However, requirements are often expressed in natural language (NL), which is inherently ambiguous, noisy, difficult to quantify, and open to interpretation [1]–[3]. To address this challenge, software engineers employ various techniques and tools to extract and organize essential information from the requirements, enabling the creation of precise and structured artifacts, such as use case and class diagrams [4]–[7].

Despite these efforts, existing techniques in requirements analysis often exhibit limitations in processing time and accuracy [8], [9]. Moreover, most methods that convert NL requirements into use cases lack sufficient intelligence, speed, and accuracy, thereby reducing end-user confidence.

Manuscript received March 28, 2025; revised October 8, 2025. Date of publication July 1, 2026. Date of current version July 1, 2026.

Authors are with the Al-Zaytoonah University of Jordan, Jordan (e-mails: {thamer.r , adnan_hnaf , mrifae, MohammedKAMEL}@zuj.edu.jo).

Digital Object Identifier (DOI): 10.24138/jcomss-2025-0016

Additionally, many algorithms struggle to interpret and analyze the grammar of natural language effectively, leading to inconsistent or incomplete results [8].

This research proposes an intelligent framework for the automatic generation of use case and class diagrams from software requirements written in NL. It explores the underlying principles, algorithms, and tools necessary to support this automation. As a result, a solid foundation is established for effectively analyzing and representing both the functional and structural aspects of a system.

Use case diagrams are graphical representations that capture interactions between actors (i.e., users, external systems, or other software components) and the system under consideration. They help identify functional requirements by illustrating various use cases or scenarios in which users or external entities interact with the system. By defining these interactions, use case diagrams provide a clear overview of the system's behavior and boundaries [10], [11].

In contrast, class diagrams represent the static structure of a system, capturing key elements such as classes, attributes, and relationships [11], [12]. They allow software engineers to model the system's objects, their properties, and the associations among them. Class diagrams are fundamental for understanding data organization, identifying class relationships, and providing a solid basis for software implementation [13], [14].

The process of generating use case and class diagrams from NL requirements involves several steps. Initially, the engineer analyzes the textual requirements to identify key actors, their goals, and specific system interactions. This analysis defines the system's use cases, which are then organized into a use case diagram summarizing system behavior. Subsequently, entities, objects, and their characteristics are identified to construct a class diagram that represents the system's static structure. By examining the relationships and dependencies among these elements, the framework produces accurate and consistent models that support early software design.

The main contributions of this research can be summarized as follows:

- An intelligent automated framework combining NLP and AI techniques to generate UML artifacts from NL requirements.
- Improved extraction accuracy through TF-IDF vectorization and k-NN classification to detect and reuse similar requirement patterns.

1845-6421/09/2025-0016 © 2026 CCIS

- A rule-based linguistic model for identifying actors, use cases, entities, and relationships.
- Empirical validation using four case studies with 30 requirement documents, achieving 96.25% recall and 92% precision.
- A foundation for broader automation, enabling future extension toward generating additional UML models such as sequence and activity diagrams.

The remainder of this paper is organized as follows. Section II reviews related work on generating use case and class diagrams from natural-language requirements. Section III presents the research methodology. Section IV reports the experimental setup and results, while Section V provides the diagram validation, error analysis, and qualitative examination. Section VI discusses the limitations of the rule-based approach, and Section VII concludes the paper.

II. Literature Review

Generating use case and class diagrams from software requirements expressed in Natural Language (NL) is a crucial step in software development, as it helps bridge the gap between non-technical stakeholders and developers. This literature review explores key related research and methodologies in this area.

Abdelnabi et al. [6] conducted a survey that examined existing works on transforming textual requirements into Unified Modeling Language (UML) class models, outlining their strengths and limitations. The survey provided a comprehensive explanation and evaluation of the current approaches and tools, highlighting their degree of automation, efficiency, and completeness. The study emphasized the necessity of automating the transformation process by integrating artificial intelligence with requirements engineering and applying natural language processing (NLP) techniques to extract class diagrams from NL requirements.

Chen et al. [13] explored the use of Named Entity Recognition (NER) to identify entities and relationships in textual requirements. Zhang et al. [15] investigated methods for understanding the meaning of requirements through word embeddings and semantic similarity measures.

Kurtev et al. [16] proposed a framework to automate the generation of diagrams from NL requirements, aiming to improve the efficiency and accuracy of the process. Similarly, Kabbedijk et al. [17] focused on transforming NL into UML diagrams using rule-based approaches and model-driven techniques to generate both use case and class diagrams.

The quality and correctness of generated diagrams are critical factors. To address this, Wang et al. [18] proposed validation techniques to verify the consistency and completeness of diagrams produced from NL requirements.

Tan et al. [19] introduced a CNN–LSTM hybrid model for accurate text classification, demonstrating the effectiveness of deep learning architectures in processing complex linguistic patterns within requirements documents. Similarly, Yacoub et al. [20] employed multilingual deep learning models to enhance sarcasm and sentiment detection, illustrating how improved text semantics can lead to more precise interpretation of software requirements. Gupta et al. [1] developed an AI-augmented usability evaluation framework for software requirements specifications, emphasizing human–AI collaboration during the early stages of software design. Moreover, Wei et al. [2] presented an automatic generation and verification approach for software requirements specifications by combining machine learning with formal verification techniques.

Recent studies have also explored new frontiers in automating UML diagram generation through text mining, machine-learning, and large language models (LLMs). For instance, Setiyawan et al. [4] and Kumar et al. [21] employed text-mining and NLP techniques to derive class, sequence, and use-case diagrams directly from software requirement specifications, demonstrating the feasibility of linguistic pattern analysis in structural modeling. More recently, Ouaddi et al. [22] proposed a chatbot-assisted framework capable of discovering UML use-case diagrams from textual system descriptions, while Babaalla et al. [23], [24] introduced LLM-driven model-driven-architecture (MDA) pipelines that automatically generate class diagrams and corresponding source code. Similarly, Ojha et al. [25] and Shehata et al. [26] demonstrated how general-purpose LLMs can be leveraged to transform user stories and natural-language specifications into UML representations with minimal human intervention. These recent efforts underscore a growing shift from rule-based and traditional NLP methods toward AI- and LLM-powered frameworks that enhance automation, scalability, and semantic understanding in software modeling. The proposed framework in this study complements these developments by combining NLP heuristics with machine-learning prediction to achieve high precision and recall in generating use-case and class diagrams from natural-language requirements.

In conclusion, generating use case and class diagrams from NL requirements is a multifaceted research area that has witnessed significant advancements. The integration of NLP, artificial intelligence, semantic analysis, and rule-based approaches is effectively bridging the gap between textual requirements and structured diagrams, thereby improving communication between stakeholders and developers. Nonetheless, ongoing research continues to address challenges related to ambiguity and aims to further enhance the automation, accuracy, and scalability of this process.

III. Methodology

This section outlines the methodology employed to generate use case and class diagrams from Natural Language (NL) requirements using machine-learning techniques. The process demands considerable time and precision to ensure that the generated diagrams are both feasible and practical. To achieve this, a robust framework, illustrated in Figure 1, has been developed for generating use case and class diagrams through the analysis and understanding of user-provided scenarios. The framework consists of the following phases: text acquisition, sentence segmentation, prediction, data preparation, text tokenization, chunking, knowledge extraction, generation of use case and class diagrams, and finally, results extraction.

A. Text Acquisition

The text acquisition phase allows users to input software requirement scenarios written in English text. The proposed

framework continuously validates all user inputs to ensure that the entered content is written in the English language.

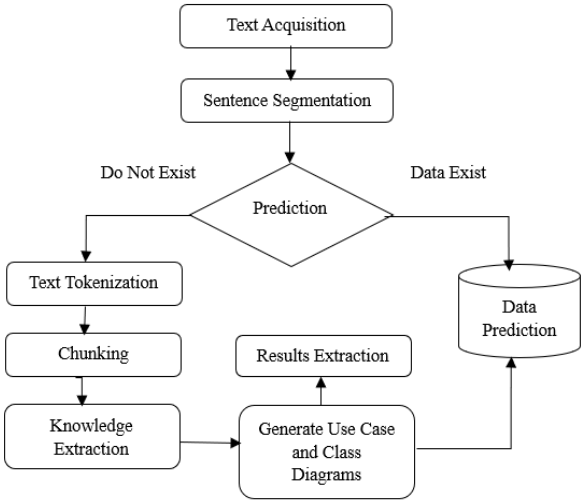

Fig. 1. Proposed Framework.

B. Text Segmentation

The text segmentation phase is designed to divide the acquired paragraphs into individual sentences so that the subsequent phase can process each sentence independently. In the English language, punctuation marks such as periods (full stops), exclamation points, and question marks generally indicate the end of a sentence—except when the period is used in abbreviations or decimal numbers, such as 3.3, Ms., or U.K. Exclamation marks ("!") and question marks ("?") clearly mark sentence boundaries because they are relatively unambiguous, whereas periods can be ambiguous since they may denote abbreviations or decimal numbers [27], [28]. To address this issue, a binary classifier is employed to determine whether a period marks the end of a sentence by producing a "yes" or "no" decision.

The simplest form of such a classifier is a decision tree, which operates as an if–then procedure that poses questions and branches based on the corresponding answers [29]. In this research, the Natural Language Toolkit (NLTK) library in the Python programming environment is used to implement the decision tree as a binary classifier. Figure 2 illustrates the decision tree that determines whether a word represents an end-of-sentence (E-O-S) marker.

To perform sentence separation, the decision tree relies on multiple linguistic rules. These include assessing whether the word containing the period is capitalized, all lowercase, all uppercase, or numeric—where an uppercase word is often indicative of an abbreviation. The decision also considers the case of the word following the period: if the next word begins with a capital letter, the period is likely to mark the end of a sentence. Additionally, numeric features such as the length of the word containing the period are crucial, as abbreviations and acronyms tend to be relatively short [30], [31].

The model also accounts for the higher probability that a word following a period typically starts a new sentence. The outputs of this phase consist of a list of sentences extracted from

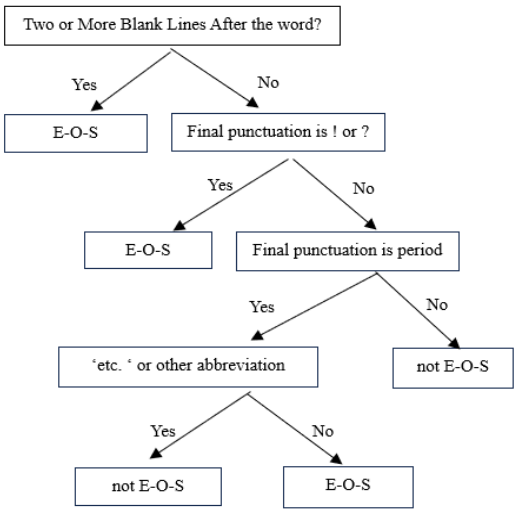

Fig. 2. Decision tree that decides whether a word is an end-of-sentence (E-O-S).

the acquired paragraphs. Algorithm 1 illustrates the decision-making process and evaluates each node within the decision tree.

Algorithm 1. Sentence segmentation method.

```
BEGIN
  TEXT Scenario, Sentence
  INT count_char = 0, count_Sentence = 0
  node = get_rules()
  count_node = node.size
  // decision tree
  WHILE (Scenario.char)
    WHILE (count_node)
      IF (Scenario.char == node[count_node]) THEN
        Sentence[count_Sentence]= Scenario.char[count_char]
        remove_between(0, count_char)
        count_Sentence++
      END IF
      count_node++
    END WHILE
    count_char++
  END WHILE
END
```

C. Prediction

The prediction phase aims to determine whether a given sentence already exists in the database—that is, whether it has previously passed through all processing phases and therefore does not need to be analyzed again. To achieve this, machine learning algorithms (MLAs) are employed, which operate exclusively on numerical feature spaces. Consequently, the expected input for such algorithms is a two-dimensional array in which rows represent instances and columns represent features [32].

Before applying the MLA, the textual data—comprising sentences, instances, and documents—must be converted into numerical vector representations. This process, known as feature extraction or more simply vectorization, enables the algorithms to process textual information effectively [32]. According to Fei and Zhang [33], one of the most widely used algebraic and mathematical models for representing text

numerically is Term Frequency–Inverse Document Frequency (TF-IDF), which is therefore adopted in this research.

Term frequency–inverse document frequency (TF-IDF) combines two key metrics. The first is the term frequency (TF), which measures how often a term appears in a specific document. The second is the inverse document frequency (IDF), which evaluates the importance of a term by considering how frequently it appears across the entire collection of documents. The inverse document frequency is calculated by dividing the total number of documents by the number of documents containing the term, followed by applying logarithmic scaling to the result. Equation (1) represents the computation of the inverse document frequency, while Equation (2) defines the TF-IDF feature vector [34].

$$idf_i = log\ (\frac{R_{total}}{R_{i\ total}}) \quad (1)$$

$$TF - IDF(i,j) = ft(i,j) \cdot fid(i,j) \quad (2)$$

where $ft(i,j)$ is the frequency of the term, $fid(i,j)$ represents the inverse of the frequency of the document for term i in document j, R_{total} is the total requirements, and $R_{i\ total}$ represents the total requirements with term i.

After applying this vectorization technique to the example in the previous phase and summing the rows of each matrix, the most influential words are determined, as shown in Table I.

TABLE I
SCORE OF EACH WORD

Words	TF-IDF
Customer	2.0
Dashboard	3.0
Outstanding	6.0
Bills	6.0
Registered	6.0
Billers	3.0

Consequently, to verify whether the entered requirements have already been analyzed and stored in the database, the proposed framework employs the acquired vectors to train and predict using a k-nearest neighbor (k-NN) classification model. This clustering algorithm is considered among the top ten data mining algorithms due to its high performance in real-world applications and its nonparametric nature [15], [35], [36].

The k-NN algorithm utilizes the Euclidean distance to measure the similarity between two data points. It calculates the distance between each training sample and the test sample in the dataset and then returns the k closest samples, known as the nearest neighbors [37]. Figure 3 illustrates an example showing how the three nearest neighbors are identified based on properties such as word similarity and the weight of each word within a sentence.

Figure 3 illustrates how neighbors are determined based on specific characteristics. Here, k = 3 represents the range or number of neighbors to be returned, y denotes the number of word repetitions within the sentence, and x indicates the weight of the similar word. The blue circles represent the training samples, while the red circle represents the test sample.

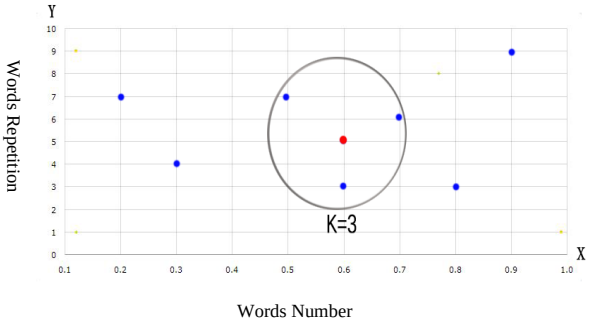

Fig. 3. Illustration of the nearest neighbor in the diagram [37].

Accordingly, this algorithm is applied in the proposed framework to predict which extracted sentences should proceed through the remaining phases of generating use case and class diagrams, and which should bypass these phases by retrieving the corresponding diagrams directly from the database.

Algorithm 2 presents the pseudocode of the k-nearest neighbor (k-NN) algorithm. The dataset consists of two classes, A and B, which serve as training samples, with k = 3 representing the number of nearest neighbors to be returned. The k-NN algorithm calculates the distance between each test sample and all training samples, subsequently returning the three closest neighbors.

In this research, the algorithm is applied to the output obtained from the vectorization phase, where these outputs serve as the training samples. This enables the retrieval of the three closest requirement sentences corresponding to a newly entered requirement. Consequently, the framework can skip the remaining processing phases for new requirements that already exist in the database, thereby improving efficiency.

Algorithm 2. k-nearest neighbor's pseudocode.

```
BEGIN
INPUT text
TEXT dataset = get_dataset()
Start a learning set with separate classes (C1, C2, ..., Cp)
Clear K-nearest neighbors sets (Cj(K)), j = 1, ..., p
Find an unclassified input sample x^new
Set K, 1 <= K <= n
FOR EACH CLASS Cj ∈ (C1, C2, ..., Cp)
   Set i = 0
   FOR EACH x ∈ C^j
      IF (i < K) THEN
         Assign x to the K-nearest neighbors sets (Cj(K))
         i = i + 1
      ELSE
         Calculate the OWA-distance between x and x^new
         IF (x is closer to x^new than any sample in class Cj(K)) THEN
            Delete the farthest sample from the set (Cj(K))
            Assign x to set (Cj(K))
         END IF
      END IF
   END FOR
   Calculate the OWA-distance d between x and the set Cj(K)
END FOR
Mark the class with minimum distance d_j as j*
Assign x to Cj*
END
```

D. Text Tokenization

Tokenization is the process of dividing sentences into smaller segments or units known as tokens [20], [38]. These tokens can represent words, numbers, or symbols. This phase is essential for identifying word boundaries, which mark the end of one word and the beginning of another. The proposed tokenization approach effectively recognizes such patterns and serves as a fundamental step in the process of knowledge extraction, enabling the identification of word types related to actors, use cases, and entities.

The following tasks are employed to define words and determine their boundaries, while Algorithm 3 illustrates the proposed tokenization module based on these rules [39]:

- turning all letters into lower or upper case,
- turning numbers into words or removing them,
- removing punctuations, accent marks, and other diacritics,
- removing white spaces,
- expanding abbreviations,
- removing stop words, sparse terms, and particular words,
- applying text canonicalization.

As illustrated in Figure 4, the proposed framework divides sentences into individual words. The user enters the software requirements text in the designated input area, and upon pressing the Text Tokenization button, the tokenized results are displayed in the output field below.

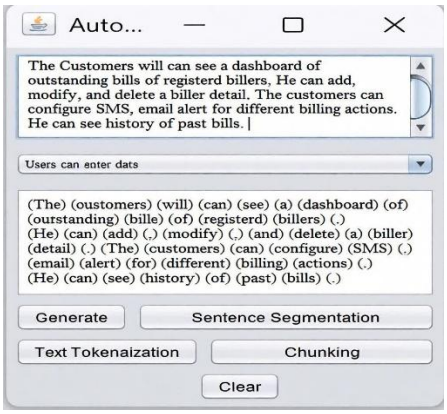

Fig. 4. Tokenization example.

Algorithm 3. Text tokenization pseudocode.

```
BEGIN
TEXT words
Sentence = Sentence_Segmentation(nlp)
// Chunking
WHILE (Sentence <= Sentence.size)
   words = converting_all_to_lower(Sentence)
   words = converting_numbers_into_words(Sentence)
   words = removing_punctuations(Sentence)
   words = removing_spaces(Sentence)
   words = removing_stop_words(Sentence)
   words = text_canonicalization(Sentence)
END WHILE
OUTPUT words
END
```

E. Chunking

In the chunking phase, individual units of information (tokens) are grouped into larger, meaningful units known as chunks using Part-Of-Speech (POS) tags. Each chunk represents a syntactic component, such as a noun phrase or a verb phrase. The POS tagging process identifies the grammatical category of each word—such as noun, verb, or adjective—independent of the sentence's overall structure or the specific phrase in which the word appears [40], [41].

F. Knowledge Extraction

Once all words have been categorized according to their types, the relevant information—such as word classifications and the relationships between words—is identified to generate the use case and class diagrams. This process is guided by a predefined set of identification rules, as outlined below:

Rule 1. Identify actors [40]:

- If a common name is found, it is most likely the name of an actor.
- If a proper noun is found, it is most likely the name of an actor.
- If a consecutive noun sequence is found and the last noun is not involved within the set of words, including number, date, address, birth, identification number, volume, type, and code, the consecutive name is most likely the name of an actor.
- Ignore proper names such as those of persons and locations.

Rule 2. Identify use cases [42]:

- If a main verb is found, it is most likely a use case.
- If a transitive verb is found, it is most likely a use case.
- If a name follows a verb, it is most likely a use case (such as creating a report).
- Ignore all verbs involved in the following list: contain, consist of, include, and involve.

Rule 3. Identify the class diagram [43].

- Nouns are converted to entity types.
- A gerund may indicate an entity type that is converted from a relationship type.
- A specialization's relationship may exist between entities: sentence structure "is a" can relate two nouns, A and B, to each other.
- A noun such as "database," "record," "system," "company," "information," "organization," or "detail" may not be considered a relevant candidate for an entity type since it indicates a business environment and, thus, does not need to be included in the entity's category.
- Every proper noun (e.g., person's name, location) is ignored and not considered a class.
- Nouns such as "vehicle_number," "group_no," "person_id," and "room_type" may refer to an attribute type.
- The genitive case, also called the possessive case, often shows ownership. Hence, instances of these can be used to extract attributes.
- If consecutive nouns are present, check the last noun. If it is not one of the words in set S where S = {number, no, code,

date, type, volume, birth, id, address, name}, it is most likely an entity type. Otherwise, it may indicate an attribute type.

- A noun phrase succeeding a "has/have" verb phrase may indicate the presence of attribute types.
- A transitive verb can be a candidate for relationship type.
- A verb followed by a preposition such as "in," "on," 'to," and "by" can indicate a relationship type.
- If a verb is equal to one of the list {include, involve, consists of, contain, comprise, divided to, embrace}, the relationship can be aggregation or composition.

Figure 5 illustrates the categorized words according to their types and morphological meanings obtained from the previous phase (chunking). This categorization facilitates the detection and extraction of representatives, use cases, and entities.

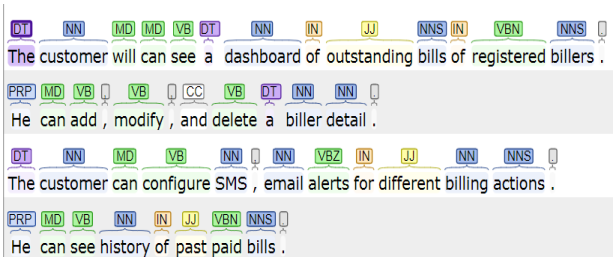

Fig. 5. Word analysis example [43].

As illustrated in Figure 5, the sentence is divided into a set of words, each annotated with its corresponding Part-Of-Speech (POS) tag—such as noun (NN), verb (VB), adjective (JJ), adverb (RB), pronoun (PRP), preposition (IN), conjunction (CC), interjection (UH), and determiner (DT). This tagging process facilitates the automatic generation of actors, use cases, entities, and attributes.

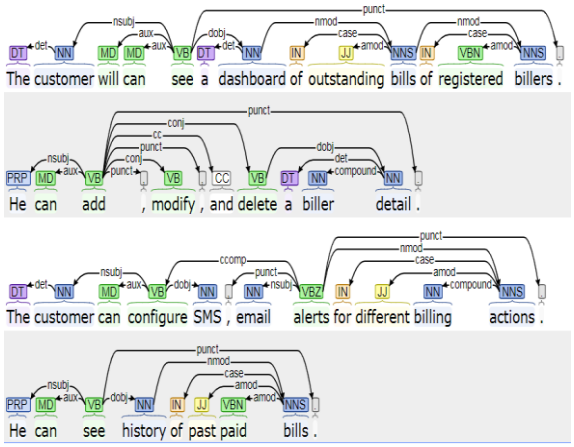

Fig. 6. Extracting relationships between words [43].

Figure 6 illustrates the extraction of relationships between words. Relationship extraction refers to the process of identifying and extracting semantic relationships within a text, typically occurring between two or more entities. These relationships are determined according to the following rules [43]:

1. Train a self-supervised classifier on a small corpus:
   - For every parsed sentence, locate every pair of noun phrases (*X*, *Y*) with a sequence of words *r* connecting them. Mark them as positive examples if they meet all of the constraints; if not, mark them as negative examples.
   - Map each triple (*X*, *r*, *Y*) to a feature vector representation (e.g., incorporating POS tags, number of stop words in *r*, NER tag).
   - Train a binary classifier to identify trustworthy candidates.
2. Pass over the entire corpus and extract possible relations:
   - Fetch potential relations from the corpus.
   - Keep or discard candidates if the classifier considers them trustworthy or not, respectively.
3. Perform rank-based assessment of relations based on text redundancy:
   - Normalize (omit non-essential modifiers) and merge relations that are the same.
   - Count the distinct sentences in which the relations are present and assign probabilities to each relation.

Algorithm 4 illustrates the process of verifying the rules after the words have been categorized.

Algorithm 4. Process of the rule checking

```
BEGIN
TEXT Acceptable
nlp = Text_Tokenization(nlp)
// Chunking
nlp_Types = Types_of_words(nlp)
WHILE (nlp_Types <= nlp_Types.Size)
   IF (Check_rules(nlp_Types)) THEN
      Acceptable = nlp_Types
   END IF
END WHILE
OUTPUT Acceptable
END
```

G. Generating the Use Case and Class Diagrams

In the diagram generation phase, the extracted information from the previous phase is utilized to apply the appropriate symbols and construct the use case and class diagrams. Figure 7 presents an example of the extracted information used in this process.

```
actor user ["signin", "Registration", "Add a product", "Delete a product", "Modify products", "Print a report"];
actor user.administrator["Delete a customer", "Disable a client"];
```

Fig. 7. Example of extracted information.

H. Results Extraction

Results extraction represents the final phase of the proposed framework, during which the outcomes are displayed based on the outputs generated in the previous phase. Figure 8 presents the graphical interface of the proposed tool, which is designed to be simple and user-friendly. The interface includes two input fields for entering text written in Natural Language (NL), while the lower output field displays the processed results. On the right side of the interface, four functional buttons are provided to facilitate user interaction.

When the Generation button is pressed, the algorithm sequentially executes all processing stages, after which the outputs are displayed, as illustrated in Figures 9 and 10. The Sentence Segmentation button divides the input text into individual sentences, while the Chunking button separates the words, analyzes each one, and determines its type and relationship to other words within the same sentence. Finally, the clear button removes all entered data from the interface.

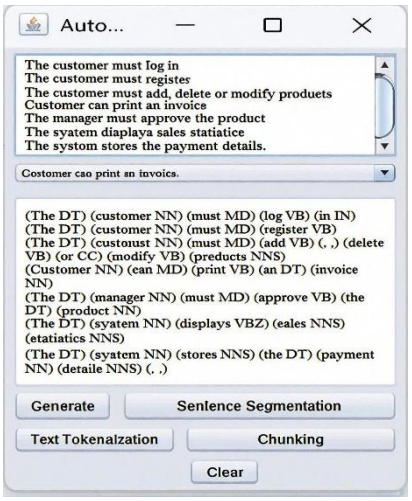

Fig. 8. Home page display.

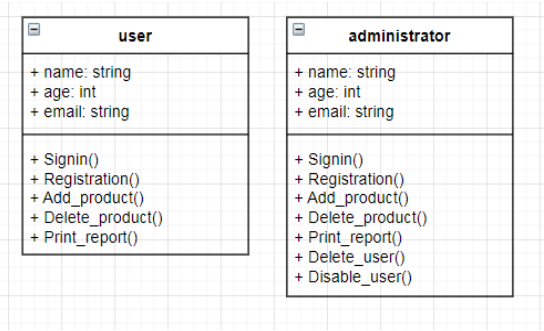

Fig. 9. Use case diagram.

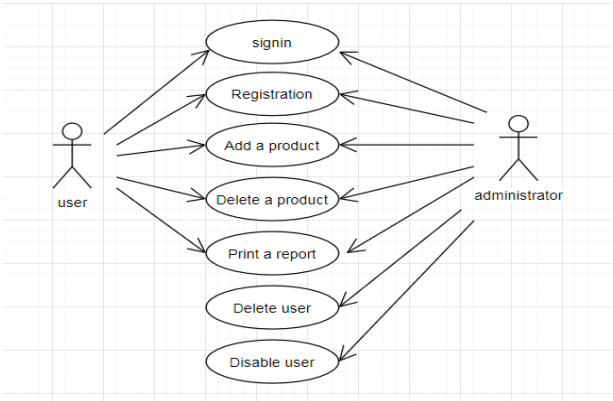

Fig. 10. Class diagram.

IV. Experiment and Results

A dataset of 30 natural-language software requirements documents was compiled to evaluate the proposed framework. The documents were intentionally selected to span multiple application domains, incorporate diverse writing styles, and exhibit varying levels of linguistic complexity, thereby creating a realistic and representative testing environment for both the rule-based and machine-learning components. The dataset covers three domains commonly encountered in software engineering practice: business and administrative systems, educational and university-support platforms, and service-oriented or transactional applications. These documents include publicly available requirement samples as well as instructional case studies used in software engineering courses, all written in English.

To ensure linguistic diversity, the dataset contains documents written in several stylistic formats, including structured requirement lists, narrative paragraphs, and mixed descriptive–instructional forms. Sentence structures range from simple declarative statements to compound and complex constructions. This variation was incorporated deliberately to assess the robustness of the extraction rules and the predictive components of the framework across differing writing conventions.

The annotation process followed a two-stage methodology. First, each document was manually examined to identify actors, use cases, entities, attributes, relationships, and interaction flows. These annotations served as the ground truth for evaluating extraction accuracy. In the second stage, each document was converted into a corresponding use case diagram and class diagram, constructed manually to reflect all annotated elements and their interrelations. These diagrams provided the reference models against which the correctness of the automatically generated diagrams was assessed.

All annotations and reference diagrams were created by two domain experts—a senior academic with extensive experience in UML modeling and a professional software engineer. Any discrepancies between their annotations were resolved through discussion to ensure consistency and reliability. The resulting dataset comprises 30 documents ranging from 30 to 270 words, representing diverse domains and sentence structures. All annotations and UML diagrams were expert-verified to ensure transparent and reproducible evaluation. Because several documents are instructional or internally developed, their full texts cannot be publicly released; however, summaries and representative samples are provided to maintain clarity and reproducibility. Table II presents the number of documents used in each experiment along with the corresponding word counts for each document.

TABLE II
Details of Requirements Documents

Case Study	Number of Documents	Range Size (Number of Words)
1	6	30–80
2	6	50–120
3	6	60–270
4	12	40-250

Two evaluation metrics precision and recall were employed to assess the performance of the proposed framework. These measures were originally designed to evaluate the effectiveness of information retrieval systems [44] and have recently been adopted to assess the performance of information extraction systems [45].

The recall metric represents the ratio of the number of correct results returned by the proposed framework to the total number of relevant results identified by human analysts, as expressed in Equation (3):

$$recall = \frac{Correct\ results}{Total\ correct\ results} \tag{3}$$

Precision refers to the accuracy of the proposed framework, defined as the proportion of correctly generated results to the total number of results produced by the framework, as expressed in Equation (4):

$$\text{Precision} = \frac{correct\ results}{all\ returned\ results} \quad (4)$$

Table III presents the performance scores of the proposed framework across the 30 evaluated documents, categorized into four distinct groups. Each row displays the score corresponding to a specific category, while the final row summarizes the overall performance of the proposed framework. Before incorporating the training data, the framework achieved a recall of 86.5% and a precision of 78.5%.

TABLE III
INITIAL RESULTS BEFORE ADDING THE TRAINING DATA

Case Study	Recall (%)	Precision (%)
1	88	79
2	85	80
3	86	77
4	87	78
Overall	86.5	78.5

As an artificial intelligence (AI)–based system, the proposed tool for generating use case and class diagrams required training to enhance its accuracy. Table IV presents the results obtained after repeating the experiments following the training phase.

TABLE IV
PERFORMANCE MEASUREMENT RESULTS

Case Study	Recall (%)	Precision (%)
1	96	93
2	98	94
3	95	90
4	96	91
Overall	96.25	92

As shown in Table IV, the experimental results improved significantly after adding the training data and activating the machine-learning component. The overall performance of the proposed framework reached 96.25% recall and 92% precision, demonstrating that increasing the amount of training data enhances accuracy.

To further assess the effectiveness of the proposed framework, its results were compared with existing studies on automated requirements analysis and information extraction. Previous works, such as [11] and [13], reported recall rates ranging between 80% and 92% and precision rates between 75% and 88% when extracting structured information from software requirements. In contrast, the proposed method achieved superior performance, particularly after training, with recall improving to 96.25% and precision to 92%.

To validate robustness under different data splits, we performed stratified 5-fold cross-validation. Each fold used 80% of documents for training (including AI-based learning phases: TF-IDF + k-NN) and 20% for testing. Stratification ensured that each fold maintained diversity in requirement complexity and sentence length.

Across the five folds, the framework achieved highly stable performance, with an average recall of 95.3% and average precision of 91.2%, demonstrating that accuracy is not dependent on any specific subset of documents. Table V summarizes the performance across all folds.

TABLE V
5-FOLD CROSS-VALIDATION RESULTS

Fold	Recall (%)	Precision (%)
Fold 1	94.7	90.8
Fold 2	95.9	91.5
Fold 3	94.8	92.1
Fold 4	95.6	90.4
Fold 5	95.3	91.0
Overall	95.3%	91.2%

These results confirm that the framework generalizes well to unseen natural-language requirement documents and maintains consistent extraction accuracy across varied scenarios. To emphasize the superiority of the proposed framework, its performance was compared with several previously published methods addressing the automatic generation of UML diagrams from natural language requirements. Table VI summarizes this comparison in terms of precision and recall.

TABLE VI
COMPARATIVE ANALYSIS WITH PREVIOUS STUDIES

Study	Approach	Precision (%)	Recall (%)	Limitations
Gosala [11]	Rule-based extraction	78	85	Lacks adaptability to complex text; limited semantic coverage
Chen et al. [13]	Machine-learning-based (semantic recognition)	85	90	Improved entity recognition but limited by dataset size
Malik et al. [5]	NLP + network-science extraction	88	91	Captures relational links but has higher processing cost
Elallaoui et al. [7]	NLP + heuristic rules (user stories)	86	89	Effective for short, structured sentences
Proposed Method	NLP + TF-IDF + k-NN (AI-based)	92	96.25	Combines rule-based and AI prediction; robust to varied requirement structures

As shown in Table VI, the proposed AI-based framework achieves superior performance, with 92% precision and 96.25% recall, surpassing all compared methods. Unlike traditional rule-based or purely heuristic approaches, the integration of TF-IDF vectorization with k-nearest neighbor (k-NN) learning enables the system to automatically recognize similar requirement patterns and adapt to previously unseen text structures. This hybrid design substantially enhances both the semantic understanding and reusability of the extracted

information, resulting in measurable improvements in automation accuracy and computational efficiency.

V. Diagram Validation, Error Analysis and Qualitative Examination

To complement the precision and recall metrics used for evaluating extracted linguistic elements, an additional validation procedure was performed to assess the correctness of the UML diagrams generated by the proposed framework. Two domain experts a senior academic specializing in requirements modeling and a professional UML practitioner independently reviewed the generated diagrams for all 30 requirement documents. Each generated use case diagram and class diagram was compared against the reference diagrams manually produced during the annotation phase. Discrepancies were discussed, and consensus was achieved to determine the final ground truth used for evaluation. Diagram correctness was assessed based on four standard criteria: semantic correctness, structural accuracy, completeness, and conformity to UML notational conventions. Using these criteria, diagram correctness was quantified by comparing the number of correctly generated elements and relationships to the total expected in the reference model. The results demonstrated that the framework achieved an average correctness of 90% for use case diagrams and 88% for class diagrams. Most deviations arose from ambiguous sentence structures, misinterpreted relationships, or confusion between domain entities and descriptive attributes.

To better understand these deviations, a qualitative error analysis was conducted to examine common extraction failures and identify systematic weaknesses in both the rule-based and machine-learning components. The analysis revealed that errors frequently originate from sentences with implicit subjects or actions, where the absence of an explicit actor leads to incorrect or missing extractions. Domain-specific terminology also posed challenges, as uncommon or specialized nouns were sometimes misclassified as attributes rather than entities. Multi-clause or compound sentences often resulted in fragmented or merged use cases, reflecting the difficulty of parsing actions expressed across multiple syntactic units. Ambiguous verbs—such as *supports, handles, or contains* further complicated extraction, as these terms do not consistently indicate system functions and may be interpreted incorrectly depending on context.

Several detailed examples illustrate the challenges encountered by the rule-based engine. In passive constructions such as “The report is generated automatically by the system,” the framework may incorrectly identify “report” as an actor, while in sentences like “The system contains a list of registered customers,” POS-tagging errors can cause the verb contains to be extracted as a use case despite explicit rule exclusions. Entity–attribute confusion also occurs in cases such as “Each order record has an order ID and date,” where “order ID” is mistakenly treated as an entity, and relationship extraction errors arise when prepositional cues are misinterpreted, as in “The system logs the user into the portal,” where the system may incorrectly infer an includes relationship rather than an authentication action. These examples highlight fundamental limitations of the rule-based approach: its reliance on surface grammatical cues makes it sensitive to POS-tagging inaccuracies, while implicit actors, passive structures, and nominalizations remain difficult to resolve without deeper semantic understanding. Moreover, the heuristic rules cannot consistently disambiguate verbs, domain-specific terminology, nested sentence structures, conditional or temporal expressions, or unresolved pronoun references. As summarized in Table VII, these limitations manifest in several error categories, including actor misidentification, use case mislabeling, entity–attribute confusion, relationship errors, POS-propagation errors, and failures arising from missing semantic inference.

TABLE VII
Categories of Common Errors and their Cases

Error Category	Description	Example	Primary Cause
Actor Misidentification	Wrong noun extracted as actor	“The report is generated…”	Over-general noun rules
Use Case Mislabeling	Verb incorrectly treated as a use case	“System contains a list…”	Verb ambiguity
Entity/Attribute Confusion	Attributes mistaken as entities	“Order ID”	Consecutive noun rule limitations
Relationship Errors	Incorrect mapping of semantic relations	“logs the user into”	Preposition-based heuristics
POS Tagging Propagation Errors	Downstream errors caused by incorrect tagging	Any complex sentence	NLP tool limitations
Implicit Actor Failures	Missing actor in passive constructions	“A message is sent”	Lack of semantic inference

Despite these challenges, the expert evaluation confirms that the framework consistently generates UML diagrams that closely approximate human-constructed models. The observed errors are largely attributable to inherent ambiguities in natural language and the limited semantic depth of heuristic methods. These findings point to promising future directions, including the integration of dependency parsing, semantic role labeling, and transformer-based contextual embeddings to enhance extraction accuracy and improve diagram correctness.

VI. Limitations of the Rule-based Approach

Although the proposed framework achieves strong extraction performance, its reliance on handcrafted linguistic rules imposes several inherent limitations. The approach remains domain-dependent, as rules designed around common software-engineering terminology may not generalize well to specialized or technical domains. It also struggles with complex or ambiguous sentence structures—such as passive voice, implicit actors, nested clauses, and nominalizations—where surface-level heuristics are insufficient for accurate interpretation. Furthermore, the rule set cannot consistently resolve contextual ambiguity or semantic nuances, especially when verbs or noun phrases carry domain-specific meanings. While modern NLP

techniques such as dependency parsing and transformer-based embeddings could address some of these challenges, they require substantial training data and reduce interpretability. Overall, the rule-based approach offers transparency and predictable behavior but lacks the flexibility and semantic depth needed for broader applicability.

VII. Conclusion

Developers and designers who create UML models often face significant challenges related to time, effort, and accuracy during software development. Despite the importance of these models, relatively few studies have employed artificial intelligence (AI) to automate their generation. To address this gap, this research proposed an automated framework for generating use case and class diagrams from software requirements documents written in a common Natural Language (NL).

The proposed NLP–AI framework demonstrated strong performance in generating UML diagrams from natural-language requirements. Moreover, the incorporation of the k-nearest neighbor (k-NN) algorithm enhances performance by predicting whether an entered requirement resembles an existing, pre-processed sentence in the database. The framework also demonstrates the capability to analyze and interpret input scenarios semantically, identifying essential linguistic features such as references, comparisons, and additive cohesive elements.

This intelligent integration of NLP and AI techniques contributes to a deeper understanding of how use case and class diagrams represent the overall system structure and behavior. Such enhanced understanding can lead to reduced development time and cost, better project planning, shortened software development lifecycles, and lower error rates.

For future work, additional empirical studies are recommended to further evaluate and validate the proposed framework. Furthermore, the framework can be extended to automatically generate additional UML models, such as sequence and activity diagrams, thereby broadening its applicability in model-driven software engineering.

Acknowledgements and Funding

The authors would like to thank Al-Zaytoonah University, Jordan, for providing the necessary scientific research supplies to implement the research, and for funding this work through the research fund No. (2022-17-33).

References

[1] S. Gupta, G. Epiphaniou, and C. Maple, "AI-augmented usability evaluation framework for software requirements specification in cyber physical human systems," *Internet of Things*, vol. 23, p. 100841, 2023, doi: 10.1016/j.iot.2023.100841.

[2] X. Wei, Z. Wang, and S. Yang, "An automatic generation and verification method of software requirements specification," *Electronics*, vol. 12, no. 12, p. 2734, 2023, doi: 10.3390/electronics12122734.

[3] Q. Zhi, L. Gong, J. Ren, M. Liu, Z. Zhou, and S. Yamamoto, "Element quality indicator: A quality assessment and defect detection method for software requirement specification," *Heliyon*, vol. 9, no. 5, 2023, doi: 10.1016/j.heliyon.2023.e15536.

[4] N. F. Setiyawan, Y. Priyadi, and W. Astuti, "Development of class diagrams based on use case and sequence diagrams using a text mining approach in SRS Penguin," in *Proc. 2023 IEEE World AI IoT Congress (AIIoT)*, 2023, pp. 70–76, doi: 10.1109/AIIoT56828.2023.10182765.

[5] M. I. Malik, M. A. Sindhu, and R. A. Abbasi, "Extraction of use case diagram elements using natural language processing and network science," *PLoS One*, vol. 18, no. 6, p. e0287502, 2023, doi: 10.1371/journal.pone.0287502.

[6] E. A. Abdelnabi, A. M. Maatuk, and M. Hagal, "Generating UML class diagram from natural language requirements: A survey of approaches and techniques," in *Proc. 2021 IEEE 1st Int. Maghreb Meeting MI-STA*, 2021, pp. 288–293, doi: 10.1109/MI-STA52233.2021.9464515.

[7] M. Elallaoui, K. Nafil, and R. Touahni, "Automatic transformation of user stories into UML use case diagrams using NLP techniques," *Procedia Computer Science*, vol. 130, pp. 42–49, 2018, doi: 10.1016/j.procs.2018.04.009.

[8] B. Hnatkowska and M. Cebinka, "Activity diagram generation based on use-case textual specification," *Computing and Informatics*, vol. 40, no. 4, pp. 772–795, 2021.

[9] B. Alturas, "Connection between UML use case diagrams and UML class diagrams: A matrix proposal," *Int. J. Comput. Appl. Technol.*, vol. 72, no. 3, pp. 161–168, 2023, doi: 10.1504/IJCAT.2023.129807.

[10] G. Bergström, "Evaluating the layout quality of UML class diagrams using machine learning," *J. Syst. Softw.*, vol. 192, p. 111413, 2022, doi: 10.1016/j.jss.2022.111413.

[11] B. Gosala, "Automatic classification of UML class diagrams using deep learning technique: Convolutional neural network," *Appl. Sci.*, vol. 11, no. 9, p. 4267, 2021, doi: 10.3390/app11094267.

[12] A. Al Thunibat, N. A. M. Zin, and N. Sahari, "Mobile government user requirements model," *Journal of E-Governance*, vol. 34, no. 2, pp. 104–111, 2011.

[13] F. Chen, L. Zhang, X. Lian, and N. Niu, "Automatically recognizing the semantic elements from UML class diagram images," *J. Syst. Softw.*, vol. 193, p. 111431, 2022, doi: 10.1016/j.jss.2022.111431.

[14] E. Planas and J. Cabot, "How are UML class diagrams built in practice? A usability study of two UML tools: MagicDraw and Papyrus," *Comput. Stand. Interfaces*, vol. 67, p. 103363, 2020, doi: 10.1016/j.csi.2019.103363.

[15] S. Zhang *et al.*, "Learning k for KNN classification," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 3, pp. 1–19, 2017, doi: 10.1145/3054912.

[16] I. Kurtev *et al.*, "Model-based component development and analysis with ComMA," *Sci. Comput. Program.*, vol. 103067, 2020, doi: 10.1016/j.scico.2020.103067.

[17] J. Kabbedijk *et al.*, "Customer involvement in requirements management: Lessons from mass market software development," in *Proc. 2009 17th IEEE Int. Requirements Eng. Conf.*, 2009, pp. 281–286, doi: 10.1109/RE.2009.39.

[18] C. Wang *et al.*, "Automatic generation of acceptance test cases from use case specifications: An NLP-based approach," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 585–616, 2020, doi: 10.1109/TSE.2020.2966364.

[19] N. S. Tan, M. L. Tham, S. Y. Chua, and Y. L. Lee, "Accurate and fast classification of natural disasters using CNN–LSTM and inference acceleration," *J. Commun. Softw. Syst.*, vol. 20, no. 1, pp. 58–68, 2024, doi: 10.24138/jcomss-2024-001.

[20] A. Yacoub, A. E. Aboutabl, and S. O. Slim, "Multilingual sarcasm detection for enhancing sentiment analysis using deep learning algorithms," *J. Commun. Softw. Syst.*, vol. 20, no. 4, pp. 278–289, 2024.

[21] S. Kumar, Aryaman, A. Aryan, and D. Yadav, "Natural language processing based automatic making of use case diagram," in *Proc. 2023 5th Int. Conf. Inventive Research in Computing Applications (ICIRCA)*, 2023, doi: 10.1109/ICIRCA58278.2023.10159739.

[22] C. Ouaddi, L. Benaddi, A. Souha, A. Jakimi, and B. Ouchao, "A sketch of an approach for discovering UML use-case diagrams from textual specifications of systems using a chatbot," in *Proc. 2024 IEEE 12th Int. Symp. Signal, Image, Video Commun. (ISIVC)*, 2024, doi: 10.1109/ISIVC62047.2024.10596185.

[23] Z. Babaalla, A. Jakimi, and M. Oualla, "LLM-driven MDA pipeline for generating UML class diagrams and code," *IEEE Access*, vol. 13, pp. 145872–145886, 2025, doi: 10.1109/ACCESS.2025.3498215.

[24] Z. Babaalla, E. M. Bouziane, A. Jakimi, and M. Oualla, "From text-based system specifications to UML diagrams: A bridge between words and models," in *Proc. 2024 Int. Conf. Circuit, Systems and Communication (ICCSC)*, 2024, doi: 10.1109/ICCSC62046.2024.10595321.

[25] M. Ojha, S. Gupta, and R. Sharma, "Towards class diagram generation from user stories using LLMs," in *Proc. 2025 Int. Conf. Next Generation Information System Engineering (NGISE)*, 2025.
[26] M. Shehata, B. Lepore, H. Cummings, and E. Parra, "Creating UML class diagrams with general-purpose LLMs," in *Proc. 2024 IEEE Working Conf. Softw. Visualization (VISSOFT)*, 2024, doi: 10.1109/VISSOFT62042.2024.10594761.
[27] H. Mohamed, E. H. El-Behaidy, W. Khoriba, and J. Li, "Improved white blood cells classification based on pre-trained deep learning models," *J. Commun. Softw. Syst.*, vol. 16, no. 1, pp. 37–45, 2020.
[28] E. S. Btoush and M. M. Hammad, "Generating ER diagrams from requirement specifications based on natural language processing," *Int. J. Database Theory Appl.*, vol. 8, no. 2, pp. 61–70, 2015, doi: 10.14257/ijdta.2015.8.2.06.
[29] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, "An introduction to decision tree modeling," *J. Chemometrics*, vol. 18, no. 6, pp. 275–285, 2004.
[30] C. R. Narawita, "UML generator: Use case and class diagram generation from text requirements," *Int. J. Adv. ICT Emerg. Regions*, vol. 10, no. 1, 2017.
[31] M. Muhairat, S. AlZu'bi, B. Hawashin, M. W. Elbes, and M. Al-Ayyoub, "An intelligent recommender system based on association rule analysis for requirement engineering," *J. Univers. Comput. Sci.*, vol. 26, no. 1, pp. 33–49, 2020.
[32] B. Bengfort, R. Bilbro, and T. Ojeda, *Applied Text Analysis with Python: Enabling Language-Aware Data Products with Machine Learning*. O'Reilly Media, 2018.
[33] N. Fei and Y. Zhang, "Movie genre classification using TF-IDF and SVM," in *Proc. 2019 7th Int. Conf. Information Technology: IoT and Smart City*, 2019, pp. 131–136.
[34] E. D. Canedo and B. C. Mendes, "Software requirements classification using machine learning algorithms," *Entropy*, vol. 22, no. 9, p. 1057, 2020.
[35] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," *Advances in Neural Information Processing Systems*, vol. 14, 2001.
[36] M. Filippone, F. Camastra, F. Masulli, and S. Rovetta, "A survey of kernel and spectral methods for clustering," *Pattern Recognit.*, vol. 41, no. 1, pp. 176–190, 2008.
[37] Z. Deng, X. Zhu, D. Cheng, M. Zong, and S. Zhang, "Efficient kNN classification algorithm for big data," *Neurocomputing*, vol. 195, pp. 143–148, 2016.
[38] A. S. Nayak, A. P. Kanive, N. Chandavekar, and R. Balasubramani, "Survey on pre-processing techniques for text mining," *Int. J. Eng. Comput. Sci.*, vol. 5, no. 6, pp. 16875–16879, 2016.
[39] S. Vijayarani and R. Janani, "Text mining: Open-source tokenization tools—An analysis," *Adv. Comput. Intell.*, vol. 3, no. 1, pp. 37–47, 2016.
[40] M. D. Anderson and D. Vilares, "Increasing NLP parsing efficiency with chunking," *MDPI Proc.*, vol. 2, no. 18, p. 1160, 2018.
[41] N. S. Tan, M. L. Tham, S. Y. Chua, and Y. L. Lee, "Accurate and fast classification of natural disasters using CNN–LSTM and inference acceleration," *J. Commun. Softw. Syst.*, vol. 20, no. 1, pp. 58–68, 2024.
[42] M. Z. Alksasbeh, B. A. Alqaralleh, T. A. Alramadin, and K. A. Alemerien, "An automated use case diagrams generator from natural language requirements," *J. Theor. Appl. Inf. Technol.*, vol. 95, no. 5, 2017.
[43] M. Rocha, A. Simão, and T. Sousa, "Model-based test case generation from UML sequence diagrams using extended finite state machines," *Softw. Qual. J.*, vol. 29, no. 3, pp. 597–627, 2021, doi: 10.1007/s11219-020-09517-3.
[44] D. Melamed, R. Green, and J. Turian, "Precision and recall of machine translation," in *Companion Volume Proc. HLT-NAACL 2003—Short Papers*, 2003, pp. 61–63.
[45] M. Buckland and F. Gey, "The relationship between recall and precision," *J. Amer. Soc. Inf. Sci.*, vol. 45, no. 1, pp. 12–19, 1994.

T. AL-Rawashdeh is an Associate Professor at the Software Engineering Department, Faculty of Science and information Technology, Al-Zaytoonah University of Jordan. He received his Ph.D. degree in Software Engineering from University Utara Malaysia in 2011. His researches focus on the Software Requirements, Software Architecture, Software testing and Quality.

A. Hnaif is a full Professor at the Cybersecurity Department, Faculty of Science and information Technology, Al-Zaytoonah University of Jordan. He received his Ph.D. degree in Network Security from University Sains Malaysia–National Advanced IPv6 Centre and Excellence (NAV6) in 2010. His researches focus on the network security, network monitoring and documentation, computer networks and communications, wireless networks, and parallel processing techniques.

M. Alrifaee was born in Zarqa Jordan, in 1978. He received the B.Sc. and M.Sc. degrees in information technology from the University of Sindh, Hyder-Abad, Pakistan, in 2000 and 2001, respectively. He received Ph.D. degree in Multimedia from the Faculty of Technology, University of De Montfort, Leicester, UK, in 2015. In 2002, he joined the Department of Computer Information System, University of Al-Zaytoonah, as a Lecturer, and in 2015 became an Assistant Professor. His current research interests include pattern recognition, signal processing and software testing. Dr. Mustafa is a member in the Department of Multimedia Technology, University of Al-Zaytoonah, Amman, Jordan.

M. Kamel is a Freelancer developer of web and mobile applications. He is a postgraduate student at Alzaytoonah University of Jordan, Software Engineering Department.