TU WIEN Informatics

# Data-Centric AI for Conceptual Modeling: Cleansed Data and a GNN-LLM based Recommender for UML Class Diagrams

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieurin**

im Rahmen des Studiums

**Data Science**

eingereicht von

**Anđela Đelić, BSc**
Matrikelnummer 12224128

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Mitwirkung: Assistant Prof. Mag.a rer.nat. Dr.in techn. Julia Neidhardt

Wien, 25. März 2026

| Anđela Đelić | Dominik Bork |
|---|---|

TU WIEN Informatics

# Data-Centric AI for Conceptual Modeling: Cleansed Data and a GNN-LLM based Recommender for UML Class Diagrams

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieurin**

in

**Data Science**

by

**Anđela Đelić, BSc**
Registration Number 12224128

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Assistant Prof. Mag.a rer.nat. Dr.in techn. Julia Neidhardt

Vienna, March 25, 2026

| Anđela Đelić | Dominik Bork |
|---|---|

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Erklärung zur Verfassung der Arbeit

Anđela Đelić, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 25. März 2026

Anđela Đelić

# Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dominik Bork and Julia Neidhardt, for their guidance throughout my scientific journey and for generously sharing their knowledge and expertise with me. Their advice was invaluable to the development of this thesis, and I am deeply appreciative of the time, care, and constructive feedback they invested in it.

I would also like to thank my colleagues from the Business Informatics and Data Science Research Units, and especially Charlotte Verbruggen, for their encouragement and kindness along the way. Their words of reassurance often meant more than they probably realized.

Finally, I would like to thank my family and friends for their unwavering support. Above all, I am especially grateful to my parents, without whom I would not have been able to complete my Master's degree. Their belief in me, encouragement, and love have meant more than words can express. I am also deeply thankful to my partner, who patiently listened to all my worries, frustrations, and complaints, and who was always there for me with love and understanding.

# Kurzfassung

Im Bereich des Model-Driven Engineering (MDE) können auf Machine Learning (ML) basierende Recommender-Systeme Entwicklerinnen und Entwickler während der konzeptuellen Modellierung unterstützen, indem sie auf Grundlage des aktuellen Zustands eines Modells plausible Modellierungselemente vorschlagen. Die für eine solche ML-Forschung üblicherweise verwendeten Datensätze konzeptueller Modelle, die häufig aus öffentlichen Software-Repositorien gewonnen werden, weisen jedoch oft erhebliche Qualitätsprobleme auf, darunter duplizierte Modelle, triviale Beispielmodelle und mehrsprachige Namen von Modellelementen.

Über die Datenqualität hinaus stellt die multimodale Natur konzeptueller Modelle, bei der Informationen sowohl durch semantische Beschreibungen als auch durch strukturelle Beziehungen vermittelt werden, eine zusätzliche Schwierigkeit für die Entwicklung robuster Werkzeuge zur Modellvervollständigung dar. Insbesondere führt die Transformation von Unified Modeling Language (UML)-Modellen in maschinenlesbare Repräsentationen häufig zu einem Informationsverlust: Ansätze, die auf textueller Linearisierung basieren, vernachlässigen strukturelle Abhängigkeiten, während graphbasierte Repräsentationen die semantische Reichhaltigkeit der Namen von Modellelementen verringern können.

Diese Studie stellt eine umfassende Data-Cleansing-Pipeline vor, die speziell für UML-Modelldatensätze entwickelt wurde und heuristikbasiertes Filtern von Dummy-Modellen, ähnlichkeitsbasierte Duplikatserkennung sowie Sprachfilterung kombiniert. Darüber hinaus wird ein Recommender-System für die Vervollständigung von UML-Klassendiagrammen entwickelt, indem eine hybride Architektur feinjustiert wird, die Graph Neural Networks (GNNs) und Large Language Models (LLMs) integriert. Das resultierende Modell kann fehlende Modellierungselemente empfehlen, darunter Klassennamen, Attribute, Operationen und Beziehungstypen.

Der vorgeschlagene Ansatz wird durch Reproduzierbarkeitsstudien evaluiert, die die Auswirkungen des Cleansing auf nachgelagerte ML-Aufgaben messen, sowie durch Experimente zur Vervollständigung von UML-Klassendiagrammen, in denen die Leistung des feinjustierten Modells mit einer Zero-Shot-Baseline verglichen wird. Die Ergebnisse verdeutlichen das Potenzial der Kombination einer UML-spezifischen Cleansing-Pipeline mit einem multimodalen GNN-LLM-Ansatz zur Auto-Completion und weisen damit auf eine stärker datenorientierte und strukturbewusste Richtung für Machine Learning im MDE hin.

# Abstract

In the Model-Driven Engineering (MDE) domain, Machine Learning (ML)-based recommender systems can assist developers during conceptual modeling by suggesting plausible modeling elements based on the current state of a model. However, the conceptual model datasets commonly used for such machine learning research, often mined from public software repositories, frequently suffer from significant quality issues, including duplicate models, trivial examples, and multilingual model element names.

Beyond data quality, the multimodal nature of conceptual models, where information is conveyed through both semantic descriptions and structural relationships, poses an additional difficulty for the development of robust model completion tools. In particular, the transformation of Unified Modeling Language (UML) models into machine-readable representations often introduces a loss of information: approaches based on textual linearization disregard structural dependencies, whereas graph-based representations may lose the semantic richness of the model element names.

This study introduces a comprehensive data cleansing pipeline specifically designed for UML model datasets, combining heuristics filtering for dummy models, similarity-based clone detection, and language filtering. In addition, a recommender system for UML class diagram completion is developed by fine-tuning a hybrid architecture that integrates Graph Neural Networks (GNNs) and Large Language Models (LLMs). This resulting model can recommend missing modeling elements, including class names, attributes, operations, and relationship types.

The proposed approach is evaluated through reproducibility studies measuring the effect of cleansing on downstream ML tasks, as well as through experiments on UML class diagram completion tasks, comparing the performance of the fine-tuned model against a zero-shot baseline. The results demonstrate the potential of combining a UML-specific cleansing pipeline with a multimodal GNN-LLM auto-completion approach, pointing toward a more data-centric and structurally aware direction for machine learning in MDE.

# Rights and Permissions

# Contents

CHAPTER 1

# Introduction

MDE has emerged as a software engineering approach in which models serve as the primary artefacts throughout the software development process [1]. By abstracting away from low-level implementation details, models emphasize high-level domain concepts and allow developers to focus on system design while capturing only the relevant structural and behavioral aspects. Automation tools can then leverage these abstract representations to support various development activities, including the automatic generation of executable code.

A conceptual model must conform to a metamodel, which defines the modeling language through its allowed elements, relationships, and constraints. Numerous modeling languages are available today and can be divided into general-purpose and domain-specific languages, depending on how broad their intended scope is. UML is one of the most widely used general-purpose modeling languages across a variety of domains [2].

Conceptual models are typically represented graphically using diagrams, and UML defines 14 different diagram types. Each diagram type provides a different view of the system being modeled and is generally categorized as either structural or behavioral. Among these, class diagrams are one of the most widely adopted UML diagram types. Rooted in the concept of classes from object-oriented paradigms, they represent the static structure of a software system by modeling classes, their attributes and operations, as well as the relationships between them.

Despite the advantages of model-driven approaches, the construction of high-quality conceptual models remains a challenging activity. This is particularly true for inexperienced modelers, but it also affects experienced practitioners working in unfamiliar domains or collaborative environments where models evolve through contributions from multiple stakeholders [3]. In response to these challenges, the MDE community has increasingly explored ML-based techniques to support various modeling activities, including model classification, transformation, and completion [4]. One particularly promising direc-

tion is the use of ML-based recommender systems to support developers during model construction.

Recommender systems are intelligent tools designed to help users navigate an increasingly saturated information landscape by suggesting content tailored to their specific context and needs. In the context of software modeling, ML-based recommenders can assist users in completing conceptual models by proposing relevant elements based on the existing diagram content [5]. For UML class diagrams specifically, such elements may include classes, attributes, operations, or relationships.

The effectiveness of ML-based modeling assistance systems strongly depends on the availability of high-quality datasets for training and evaluation. In practice, the modeling community faces a shortage of large-scale, well-curated datasets. A common strategy for obtaining large datasets is to mine software models from platforms such as GitHub or from various modeling tools [6], [7], [8], [9], often resulting in datasets that are highly uncurated, heterogeneous, redundant, and noisy. Such noise can negatively affect the generalizability of ML models and may lead to inflated performance estimates, which makes dataset quality a critical issue that must be addressed before meaningful machine learning analysis can be carried out.

Identifying redundancy in conceptual model datasets is not as straightforward as in more conventional data types, such as numerical data. One reason is that conceptual models are often stored in XML Metadata Interchange (XMI)-based formats, which makes noisy or duplicated data more difficult to detect. For example, clones may be easy to overlook because the same model saved by two different modeling tools can contain tool-specific metadata, making the corresponding files appear different. At the same time, the preprocessing and cleansing steps required to ensure high-quality datasets are not standardized across the MDE domain, especially for UML datasets, which raises the question of how such preprocessing should be performed.

Beyond data quality issues, another challenge for ML research in MDE concerns how conceptual models are represented as machine learning input. Conceptual models encode information through both semantic and structural constructs, reflecting the meaning of model elements and the relationships between them. However, existing research often focuses on only one of these modalities, either transforming models into purely textual representations at the expense of structural information or modeling the structure while neglecting semantics. Consequently, there remains a need for approaches that can learn from both the linguistic and structural dimensions of conceptual models simultaneously.

These challenges highlight the need for a more data-centric research approach that improves the quality of the datasets used to train ML models while also exploring methods capable of leveraging the multimodal nature of conceptual models. This dual perspective forms the foundation for the research questions investigated in this thesis.

## 1.1 Research Questions

To develop an auto-completion approach for UML class diagrams, this thesis addresses two key challenges that hinder progress in applying machine learning to conceptual modeling: the data quality of available datasets and the inherent multimodality of UML class diagrams, which is still rarely explored in existing approaches.

To address the first challenge, this work proposes a cleansing pipeline for UML class diagram datasets that targets three major quality issues: model clones, dummy models, and the presence of multiple natural languages within UML datasets. The impact of the pipeline is evaluated through reproducibility studies of two previously published works, comparing the results obtained before and after the proposed cleansing steps are applied. While the pipeline and the accompanying reproducibility studies have already been presented in a publication [10], the present thesis reports full methodological details and provides a more in-depth analysis of the findings. The first research question addressed in this work is:

*RQ1: What is the effect of a data cleansing pipeline on downstream ML-enhanced MDE tasks (in terms of accuracy, precision, recall, and F1-score)?*

Once the available UML dataset has been cleansed, the focus shifts to the development of a recommender system based on an architecture capable of reasoning over both structural and semantic information contained in UML class diagrams. To this end, a recently proposed GNN-LLM architecture is employed and fine-tuned for four completion tasks: recommendation of class names, class attributes, class operations, and the relationship type between two selected classes. The second research question investigates whether fine-tuning improves the model's recommendation performance compared to its zero-shot capabilities:

*RQ2: To what extent does fine-tuning a GNN-LLM architecture improve recommendation quality for UML class diagram completion (in terms of accuracy, precision, recall, F1-score, MRR@5, SR@1, SR@5) compared to its zero-shot performance?*

## 1.2 Structure of the Work

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background information required to understand the concepts relevant to this work and outlines the state of the art in the related fields. Chapter 3 presents the methodology behind the proposed cleansing pipeline, as well as the design of the two reproducibility studies used to evaluate it. Chapter 4 describes the development of the recommender system for UML class diagram completion. Chapter 5 reports the main empirical findings and answers the two research questions. Finally, Chapter 6 discusses the main contributions, threats to validity, and directions for future work, while Chapter 7 concludes the thesis.

CHAPTER 2

# Background and Related Work

This chapter reviews the research areas most relevant for understanding the methodological and experimental choices made in the remainder of this work. Section 2.1 introduces ModelSet, the dataset used in the experiments. Section 2.2 discusses the main data quality issues affecting conceptual model datasets and reviews the existing cleansing approaches. Section 2.3 surveys the state of the art in recommender systems for MDE. Section 2.4 then introduces the GNN-LLM architecture employed in the proposed approach, explaining how it operates and how it enables reasoning over both semantic and structural information. Lastly, Section 2.5 briefly presents the evaluation metrics used in the experimental analysis.

## 2.1 ModelSet

ModelSet [9] was created as part of an initiative to facilitate the application of machine learning techniques in MDE. The dataset contains 5,466 Ecore metamodels and 5,120 UML models crawled from GitHub and GenMyModel repositories, respectively. To support large-scale annotation, the authors introduced a semi-automated labeling approach termed the *Greedy Methodology for Fast Labelling (GMFL)* [9]. This method involves grouping models by similarity in order to present the labeler with sequences of models that are likely to belong to the same category, thereby reducing the cognitive load associated with this otherwise labor-intensive task.

The dataset is available in multiple formats, including raw XMI files, textual files containing the names of model elements, and files in which the models are transformed into graph representations. Additionally, databases containing labels and metadata for each artefact are provided. The primary label is `category`, which describes the domain of the model, such as 'museum', 'restaurant', 'shopping-cart', or 'library'. Within this label, some models are assigned the value *dummy* to indicate trivial example models, and some are labeled *unknown* when their domain could not be inferred. Additional labels include

`tags`, which contain relevant keywords elaborating on the main category (e.g., for the category 'computer-videogame', the tag 'card-game' provides more context); `language`, indicating the natural language used for naming model elements; and `confidence`, expressing the annotator's level of certainty while assigning labels, among others.

The dataset also includes statistical metadata, such as counts of specific elements present in a model. For UML models, for example, the number of classes, packages, use cases, and related artefacts is reported for each given model.

## 2.2 Model Cleansing Techniques

Conceptual model datasets mined from public repositories often contain various forms of noise that can negatively affect machine learning experiments. Common issues include trivial example models, duplicated artefacts, and inconsistencies in the natural language used for naming model elements. Several techniques have been proposed in the literature to address these problems, and this section summarizes the most relevant approaches.

### 2.2.1 Dummy Model Detection

Dummy models are syntactically or semantically trivial models usually designed as examples for teaching purposes or automatically generated by modeling tools as placeholder models. When it comes to removing dummy models from MDE datasets used for machine learning, two main approaches are reported in the literature.

The first approach relies on defining heuristics that indicate typical characteristics of trivial models. In [11], the authors exclude Business Process Model and Notation (BPMN) models with fewer than four activities, models in which all activity names consist of fewer than three characters, and models composed of repeating activities named after generic workflow placeholders such as 'subprocess' and 'task', together with at most one other meaningful name. Another approach entails training a machine learning model for the binary classification task of distinguishing between trivial and non-trivial models, as demonstrated in [9].

### 2.2.2 Model Clone Detection

Model clones are (highly) similar (fragments of) software models with respect to a given definition of similarity. These clones are typically introduced through copy-and-paste reuse and pose both quality and maintainability issues [12]. From a machine learning perspective, identifying and removing duplicates within a dataset is particularly important. Without doing so, data leakage may occur, meaning that the same samples used for training the ML model also appear in the test set, which can lead to artificially inflated performance results.

Inspired by earlier work investigating the concept of code clones and given the parallels between software models and software code, Störrle [13] identified different UML model

clone types. Drawing similarities between metamodels and UML models, the authors of [14] later extended these definitions by incorporating concepts from Natural Language Processing (NLP), based on the observation that element names often play a crucial role in identifying similarities. Following the same reasoning for UML and metamodel similarity, this work adopts the model clone definitions proposed in [14]:

- **Type-A clones** are exact clones with possible differences in layout, formatting, or internal identifiers. Any minor cosmetic differences in naming (e.g., upper versus lower case) are also allowed.
- **Type-B clones** allow for some level of changes between the two (fragments of) models, such as differences in names or attributes. Additional elements may be present or missing, and small variations such as typos or synonyms are especially tolerated in these types of clones.
- **Type-C clones** are two (fragments of) models with a large proportion of differences in names, attributes, and presence of elements.
- **Type-D clones**, also referred to as semantic clones, are two (fragments of) models with the same behavior at the conceptual level; however, they differ significantly in both structure and naming.

In line with both [13] and [14], semantic clones are ignored due to the difficulty of reliably identifying them and are therefore considered out of scope in this work.

Within the MDE domain, numerous approaches have been proposed for clone detection across different modeling languages, typically operating on lexical, structural, or hybrid representations. Works such as [14] and [15] utilize a framework named SAMOS for detecting clones in metamodels and BPMN, respectively. The framework treats individual (meta-)models as documents in an information retrieval (IR) sense. Features such as unigram name-type pairs are extracted, and structural information may also be incorporated by extracting n-grams or sub-trees. The resulting textual representations are standardized using NLP techniques such as stemming, tokenization, and stop-word removal. These processed tokens are then converted into vector representations using a weighted term-frequency vector space model (VSM). Different distance measures can be computed between the vectors, and clustering algorithms (e.g., hierarchical clustering or DBSCAN) are applied over the distance matrix to identify groups of similar artefacts interpreted as clone clusters.

Several approaches instead transform models into graphs, where nodes represent model elements and edges represent relationships. Graph-matching and graph-isomorphism techniques are then applied to detect structural similarity. However, such techniques are known to be computationally expensive (NP-hard), requiring additional optimizations to make comparisons across large numbers of models feasible.

In practice, this problem is often addressed by restricting the analysis to smaller subsets of models, focusing on fragments within a single model, or applying heuristics to identify potential clone candidates before performing expensive comparisons. Notable examples include ConQAT [12], developed for Simulink models, and MQlone [16], which was adapted for UML models.

However, many of these approaches are computationally intensive and therefore difficult to apply to large datasets of conceptual models. Consequently, lightweight similarity measures are often preferred when preprocessing datasets for machine learning experiments, as illustrated by [17], where Jaccard similarity is applied to textual representations of conceptual models.

### 2.2.3 Language Filtering

Another common issue in open-access datasets of conceptual models is the presence of model element names written in different natural languages. When text-based machine learning models are used, underrepresented languages may unnecessarily increase the vocabulary by introducing additional out-of-vocabulary or rare tokens. This can introduce noise and negatively affect the quality of the learned embedding. In [11], the languages present in the dataset samples were identified using the Lingua language detection library [18], and non-English samples were filtered out as part of the preprocessing pipeline.

Although the literature proposes several techniques addressing individual data quality issues in conceptual model datasets, there remains a lack of comprehensive preprocessing pipelines specifically designed to prepare UML datasets for machine learning experiments. Existing approaches typically focus on a single aspect of the problem and are often tailored to specific modeling languages or artefact types. Moreover, many clone detection techniques rely on computationally expensive graph matching or structural comparison methods, which makes them infeasible to apply as a preprocessing step for large-scale datasets. These limitations highlight the need for lightweight and practical cleansing strategies that can address multiple dataset quality issues simultaneously.

## 2.3 Recommender Systems in MDE

Recommender systems have been increasingly explored in the MDE domain to assist developers during model construction. Similar to auto-completion tools prevalent in software development environments, recommender systems for conceptual modeling suggest relevant modeling elements based on the partially constructed model. This kind of support can help reduce modeling time, prevent common mistakes, and provide overall support to developers.

Early approaches to model recommendation were largely based on heuristics and similarity measures, while more recent research increasingly employs machine learning techniques to learn recommendation patterns. A comprehensive overview of recommender systems in the MDE domain is provided in [5], while [19] offers a perspective specifically focused

on UML modeling. Several recent works have begun investigating the potential of LLMs for conceptual model completion as well, given their rapid development [20], [21].

ModelMate [22] is a recommender system for textual modeling languages built by fine-tuning decoder-only pre-trained language models (PLMs). The system is meant to assist with three core tasks: identifier suggestion, line completion, and fragment (block) completion. The authors focus on three textual domain-specific languages (DSLs): Emfatic, Xtext, and a domain entities DSL. This demonstrates the generality of the approach across different modeling languages.

EcoreBert [23] presents a recommender system tailored to Ecore metamodels that provides a ranked list of relevant domain concepts, including classes, attributes, and associations. To capture both structural and textual information, the authors encode metamodels using a tree-based representation that is subsequently serialized into text. A RoBERTa architecture [24] is then trained using a masked language modeling objective. The model is evaluated on two renaming scenarios, using either a global or local metamodel context, as well as in an incremental metamodel construction scenario.

Chaaben et al. [25], [26] explore the use of LLMs, particularly GPT3, for model completion of UML class and activity diagrams in a few-shot prompting scenario. Given a partially constructed model, the authors extract a textual representation that is incorporated into a structured prompt together with several illustrative examples demonstrating the desired behavior and output. Multiple prompts are generated for each sample by considering different subsets of the partial model, producing several candidate recommendations. The suggested modeling elements are then aggregated and ranked based on their frequency of occurrence across the generated outputs, with the most frequent suggestions presented to the modeler.

RAMC [27] approaches the task of model completion using LLMs in a Retrieval-Augmented Generation (RAG) setting based on model histories rather than static model snapshots. The evaluation spans histories of public Ecore repositories, an industrial SysML dataset, and synthetic Ecore histories. Unlike earlier-mentioned approaches that predict the next element from a single model snapshot (typically focusing only on additions), this work models edit operations explicitly, including additions, deletions, and modifications. Concretely, two consecutive versions of a model are compared to compute their difference, from which a *simple change graph* is derived that captures the performed edit together with the surrounding structural context required to interpret it. This graph is serialized as an edge list, and the resulting representations are embedded to construct a vector database from which relevant examples are retrieved at prediction time. In the RAG setting, the retrieved examples are inserted into the prompt as in-context demonstrations.

MORGAN [28], [29] is a recommender system that supports model and metamodel completion by encoding artefacts into graph representations and retrieving similar examples using graph kernel functions. The artefacts extracted from (meta)models are first serialized into textual form and then encoded as graphs. Class nodes are connected to

their respective structural features, while relationships between different classes are not explicitly modeled. In its initial publication, MORGAN focuses on Ecore metamodels and MoDisco models, employing a Weisfeiler–Lehman Optimal Assignment (WLOA) kernel to compute similarity between graphs. In a later extension, the system is expanded to additional modeling artefacts, including JavaScript Object Notation (JSON) Schema and UML models, while replacing the kernel function with a Vertex Histogram kernel combined with an item-frequency matrix to improve computational efficiency.

While existing approaches demonstrate the potential of machine learning, and particularly large language models, for conceptual model completion, there remains a lack of approaches that jointly explore the semantic and structural modalities of UML models. This gap motivates the use of architectures capable of leveraging both modalities, such as the approach investigated in this work.

## 2.4 Graph Foundation Model GOFA

With LLMs achieving remarkable success in natural language processing, as well as other areas of Artificial Intelligence (AI) where different forms of data are linearized into text to enable the use of LLMs, there have been increasing efforts to develop similar Foundation Models for other types of data as well. Foundation Models are machine learning models pre-trained on vast amounts of data, capable of solving a wide range of tasks, reducing the need for task-specific architectures and repeated model retraining. Similar developments are taking place in the domain of graph-structured data, where Graph Foundation Models have become an important research topic. Existing approaches can broadly be categorized into GNN-based, LLM-based, and hybrid approaches that combine graph and language model architectures [30].

Generative One For All (GOFA) [31] is a graph foundation model that integrates the structural reasoning capabilities of GNNs with the generative abilities of LLMs. It is pre-trained on large-scale graph datasets with the goal of providing a universal framework that is not restricted to a single downstream task. To achieve this flexibility, the authors unify different graph tasks into a common format at a node, link, or whole-graph level, extending the concept of Text-Attributed Graphs (TAGs) introduced in [32]. TAGs attach free-form natural language text fields to nodes and edges, enabling ML models to reason jointly over graph structure and textual information.

To indicate where the model should begin generating output, a Node of Generation (NOG) is added to the graph. This node signals the generation point and can also contain task instructions in the form of a prompt. Since LLMs can only process a limited context, $k$-hop subgraphs are sampled around the target node(s). The model is then pre-trained on a corpus constructed from six different datasets.

Four specific tasks are used during pre-training to enable different reasoning capabilities. The first is a sentence completion task in which the text of a node is partially masked, the node is designated as a NOG, and the model is required to generate the missing text.

The second is a question-answering task, in which a chain of question-answer pairs is represented as nodes in a graph, and a final question is placed in a NOG prompt for the model to answer. The third task evaluates structural reasoning: a NOG is connected to two randomly selected nodes, prompting the model to compute the shortest path between them and output all the possible shortest paths. Finally, in the information-retrieval task the prompt instructs the model to output the textual content of a specified target node. An example of each task is illustrated in Figure 2.1.

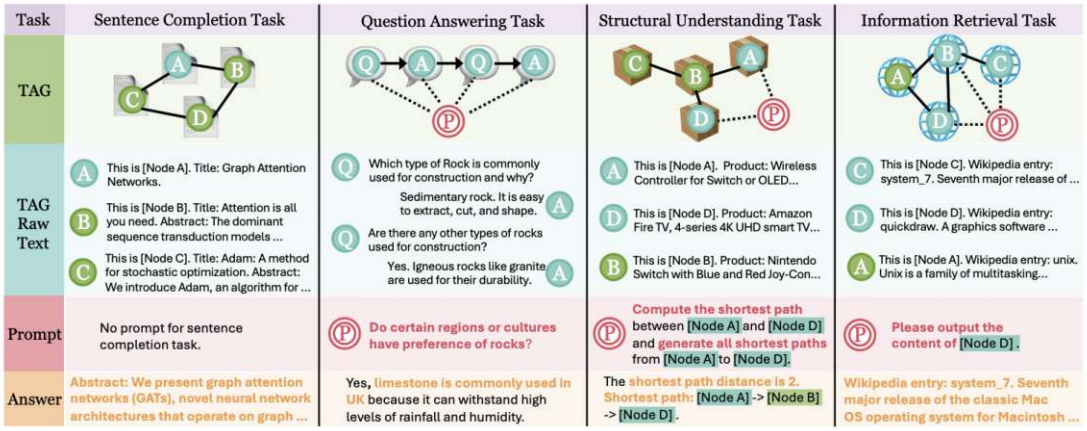



Figure 2.1: Overview of the four GOFA pre-training task formats: sentence completion, question answering, structural understanding, and information retrieval. Reprinted from Kong et al. [31] with permission.

GOFA consists of a Graph Language Encoder and an LLM Decoder, as illustrated in Figure 2.2. In the encoder, LLM layers based on In-Context Auto Encoders (ICAE) introduced in [33] are utilized in a sentence-compressor setup and are interleaved with GNN layers (modified Transformer Convolutional GNN layers [34]), with the first and last layers of the encoder always being LLM layers. For each node (and edge), $K$ memory tokens are appended to its text tokens, and the resulting sequence is fed into an LLM compressor layer. The LLM produces embeddings for both the text tokens and the memory tokens; the $K$ memory-token embeddings serve as a fixed-size compressed representation of the node or edge text.

Only these $K$ memory-token embeddings are passed to the GNN. The GNN operates token-wise: for each index $i$, the $i$-th memory token of a node is updated using the $i$-th memory tokens of its neighboring nodes (and the corresponding edges). The $K$ tokens within a single node do not interact with each other inside the GNN. In GOFA, only node memory tokens are updated by the GNN, while edge memory tokens are used only during message passing and are otherwise propagated unchanged.

The updated node memory tokens are then combined again with the node's text-token embeddings and fed into the next LLM compressor layer, where self-attention enables information exchange both among the $K$ memory tokens and between the memory tokens

and the text tokens. This alternating information exchange between the LLM and GNN layers is repeated throughout the encoder. Once encoding is complete, the final $K$ memory embeddings of the NOG are prepended to the target output text tokens and passed to the decoder. The decoder is trained using teacher forcing with a next-token prediction (NTP) objective. Concretely, given a target sequence $y = (y_1, \ldots, y_L)$, the model minimizes

$$\mathcal{L}_{\text{NTP}} = -\sum_{l=1}^{L} \log p_\theta(y_l^* \mid y_{<l}^*, v, G)\,, \tag{2.1}$$

where $p_\theta$ denotes the conditional probability distribution predicted by the decoder parameterized by $\theta$, $y_l^*$ is the ground-truth token at decoding step $l$, $y_{<l}^*$ are the preceding ground-truth tokens, $v$ denotes the NOG, and $G$ denotes the graph context containing the relevant structural and textual information.

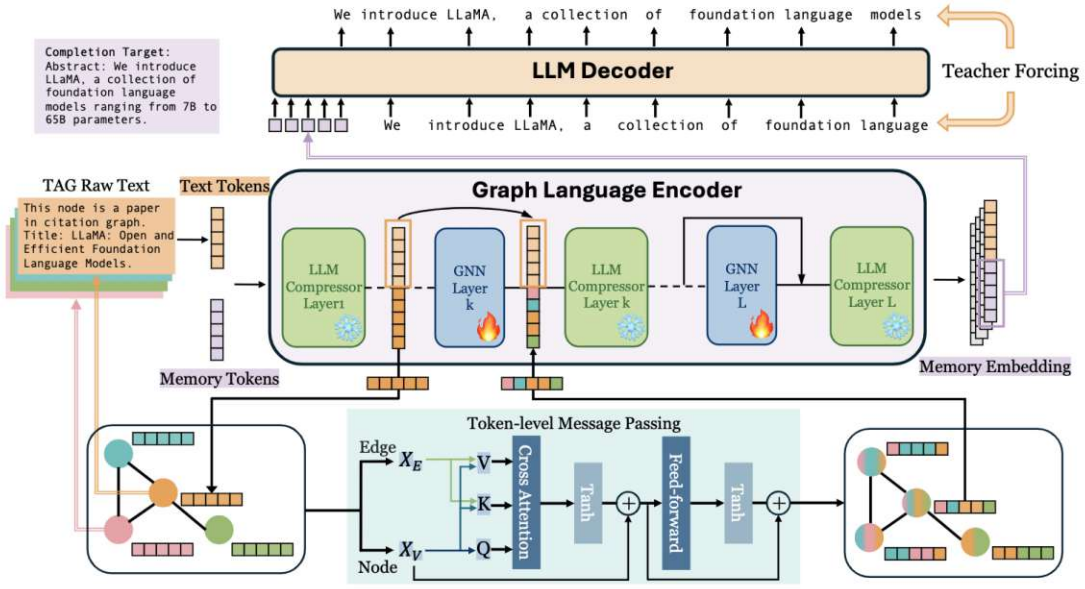



Figure 2.2: GOFA architecture. Reprinted from Kong et al. [31] with permission.

After passing through the Graph Language Encoder, each node obtains an embedding that captures information about the node's own text, the text of surrounding nodes, and the underlying graph structure. Such representations are particularly relevant for our setting, as UML diagrams contain both rich semantic information and explicit structural relationships. Furthermore, the instruction fine-tuning paradigm enables the model to support a wide range of downstream tasks, and its generative capabilities make this architecture especially well-suited for UML model completion.

## 2.5 Evaluation Metrics

Evaluation metrics are used in machine learning to objectively assess the performance of trained models. Depending on the type of task, different metrics can be employed to quantify how well a model generalizes to unseen data. As the tasks considered in this work include both classification and recommendation problems, different types of metrics are required. This section briefly introduces their definitions and interpretations.

### 2.5.1 Classification Metrics

In a binary classification setting, where only two classes are possible (typically denoted as 0 and 1), comparing model predictions with the ground truth yields four possible outcomes:

- **True Positive (TP):** the model correctly predicts the positive class (1).
- **True Negative (TN):** the model correctly predicts the negative class (0).
- **False Positive (FP):** the model incorrectly predicts the positive class for a sample that actually belongs to the negative class.
- **False Negative (FN):** the model incorrectly predicts the negative class for a sample that actually belongs to the positive class.

Based on these quantities, several commonly used evaluation metrics can be defined.

Accuracy measures the proportion of correctly classified samples among all predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

Precision measures how many of the samples predicted as positive actually belong to the positive class:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.3}$$

Recall measures how many samples belonging to the positive class were correctly identified by the model:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.4}$$

Since precision and recall often exhibit a trade-off, the F1-score is commonly used as a balanced measure combining the two. It is defined as the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.5}$$

In unbalanced datasets, accuracy alone may provide a misleading picture of model performance. For example, if 90% of the samples belong to the majority class, a classifier that always predicts this class would achieve 90% accuracy, while failing to correctly identify any samples from the minority class. To address this issue, balanced accuracy can be used.

Balanced accuracy is defined as the average of the True Positive Rate (TPR) and the True Negative Rate (TNR). The true positive rate, also referred to as sensitivity, measures the proportion of positive samples correctly predicted as positive. The true negative rate, or specificity, measures the proportion of negative samples correctly predicted as negative. Balanced accuracy is therefore computed as:

$$\text{Balanced Accuracy} = \frac{TPR + TNR}{2} \tag{2.6}$$

where

$$TPR = \frac{TP}{TP + FN}, \qquad TNR = \frac{TN}{TN + FP} \tag{2.7}$$

When classification involves more than two classes, macro-averaging is used. In this case, precision, recall, and F1-score are calculated separately for each class by treating that class as the positive class and all remaining classes as negative. The resulting values are then averaged across all classes, giving equal importance to each class regardless of its frequency.

### 2.5.2 Metrics for Recommendation Tasks

For recommendation and retrieval tasks, top-$k$ and ranking-sensitive evaluation metrics are commonly used. In this work, the quality of recommendations is evaluated using Success Rate (SR) and Mean Reciprocal Rank (MRR).

Success Rate at $k$ ($SR@k$) measures the proportion of cases in which the correct item appears among the top-$k$ recommendations produced by the model. Formally:

$$SR@k = \frac{1}{N} \sum_{i=1}^{N} rel_i@k \tag{2.8}$$

where $N$ denotes the total number of queries, and $rel_i@k$ is an indicator function that takes the value 1 if a relevant item appears within the top-$k$ results for query $i$, and 0 otherwise.

Mean Reciprocal Rank at $k$ ($MRR@k$) additionally considers the rank position of the first relevant item in the recommendation list. For query $i$, let $rank_i$ denote the rank of the first relevant item. If no relevant item appears within the top-$k$ results, the reciprocal rank is defined as 0. Otherwise, the reciprocal rank is computed as $1/rank_i$. The metric is then calculated as the average reciprocal rank across all queries:

$$MRR@k = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{rank_i} \tag{2.9}$$

# CHAPTER 3

# Cleansing Pipeline

To address identified data quality issues in datasets composed of conceptual models, an automatic data cleansing pipeline was developed. The pipeline consists of three main components: duplicate detection, dummy filtering, and automatic language detection. These methods are described in detail in the following sections.

The pipeline operates on raw XMI files of UML models. As a preliminary step, each XMI file (representing a single UML model) is transformed into a textual representation. A parser extracts the elements of each model, and two corresponding `.txt` files are generated. One file contains only the names of the elements (e.g., if a class is named 'patient', the retained token is 'patient'), while the other stores `type:name` pairs (e.g., 'class:patient'). Depending on the specific filtering step, either representation may be used.

All extracted names are normalized, so that naming conventions such as `camelCase`, `PascalCase`, and `snake_case` are split and converted to lowercase. For example, 'myClass', 'MyClass', and 'my_class' are all transformed into 'my class'. Figure 3.1 illustrates an example UML model in its original XMI format alongside its corresponding textual representations with and without type information.

Once the textual representations have been generated and normalized, the individual cleansing filters can be applied.

## 3.1 Cleansing Filters

### 3.1.1 Duplicate Detection

As discussed in Chapter 2, several categories of clones may be present in datasets of conceptual models. In light of this, the cleansing pipeline incorporates two complementary methodologies for clone detection.

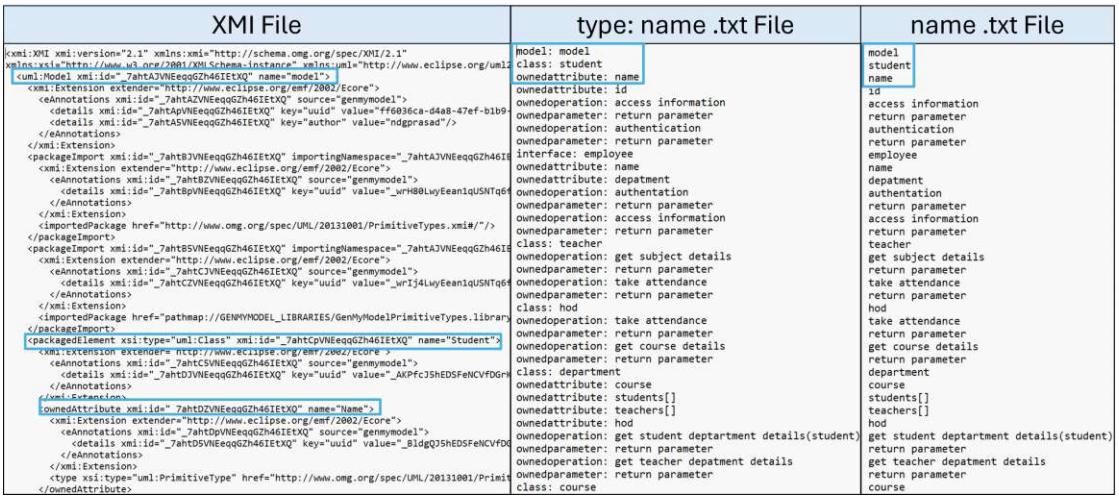



Figure 3.1: Illustration of the transformation from an XMI file to two textual representations. The blue boxes indicate example model elements in the original XMI source that are extracted and represented either in a `type:name` format or as plain names only.

1. **Exact Duplicate Detection** – This filter works on the textual files without type information, and the goal is to find exact clones. For each `.txt` file, a Secure Hash Algorithm (SHA)-256 hash is computed. Files producing identical hash values are grouped into the same hash bucket, representing an exact duplicate group.

2. **Near Duplicate Detection** – Here, each textual file is transformed into a Term Frequency-Inverse Document Frequency (TF-IDF) vector representation, and then cosine similarity is computed pairwise between files. TF-IDF reflects how often a certain word (or token) appears within a file while giving less weight to words that are common across many files. A similarity threshold of 0.8 is defined; whenever the cosine similarity between two files exceeds this value, the corresponding two files are considered near duplicates.

### 3.1.2 Dummy Detection

For the purpose of dummy detection, multiple heuristics indicative of toy or example UML models were devised. The process began by adapting similar rules originally proposed for BPMN models in [2] to the UML context, followed by manual inspection of UML models available in ModelSet to derive additional rules.

#### Heuristics on Textual Representation without Type Information

- Models with very few extracted elements (fewer than five), or with low vocabulary (number of unique words is less than four) are flagged as dummy, based on the rationale that such models are unlikely to be meaningful.

- The median length of all the element names within a model is calculated, and those models whose name length median is less than four are flagged. This heuristic captures models which mostly contain placeholder names (e.g., single letters or very short tokens).
- Models containing placeholders of the form ‘att’, ‘att1’, ‘attA’, ‘att A’ are flagged, as such patterns are indicative of tutorial, example, or unfinished models.
- Models in which at least 30% of names follow the pattern ‘letter followed by a digit’ (e.g., ‘a1’, ‘B2’), or ‘letter space letter’ (e.g., ‘a b’, ‘x y’) are flagged. The 30% threshold allows for some legitimate occurrences of such names depending on the domain (e.g., mathematical models may contain formulas where identifiers such as ‘c1’, ‘c2’ can be justifiable). The ‘letter space letter’ pattern often results from normalization, as relationships named after endpoint classes with an underscore in between (e.g., ‘a_b’), are transformed to ‘a b’. Additionally, cases such as ‘tV’ becoming ‘t v’ during normalization further justify the use of a threshold rather than strict filtering.
- It was observed that otherwise complex and meaningful models may still contain a limited number of unfinished placeholder tokens (e.g., ‘a’, ‘b2’). Therefore, a certain degree of incompleteness is tolerated provided that the proportion of short names (i.e., names shorter than three characters) remains below 30%. Models that exceed this threshold are classified as dummy. However, this threshold is lowered to 25% in cases where at least 40% of elements are named ‘control flow’. In activity diagrams, ‘control flow’ is a connector between nodes, and it is an element that does not have to be named. This means that modeling tools auto-generate names based on the artefact’s type. The high presence of such connectors in a model can artificially inflate the appearance that said model has many full-length tokens, when they in fact do not carry any domain-specific semantics.
- On a similar note, other UML elements across various diagram types are often not named by the creator of the model but rather get an auto-generated name based on their type. When many such elements are present in a model, they are often named sequentially (e.g., ‘control flow1’, ‘control flow2’). To address this, a non-exhaustive stopword list was compiled, including names such as ‘use case’, ‘actor’, ‘merge node’, among others. If 82% of elements are named after their UML type, or if 75% of elements follow a sequential naming scheme, the model is considered to lack sufficient domain-specific vocabulary and is excluded.

#### Heuristics on Textual Representation with Type Information

Here, element types are specifically checked to prevent relationship elements, which are frequently named after their endpoint classes, from inflating counts.

- Models containing more than one class type element named ‘my class’, optionally followed by a space and a number, are classified as dummy models.

- The ratio of names such as ‘class a’ or ‘class 1’ is computed, once again restricting the search to class-type elements only. Up to 13% of such names in a model are tolerated, but anything above is considered dummy. This threshold accommodates legitimate cases where the domain calls for such naming (e.g., school models might have classrooms called ‘class A’, ‘class B’).

### 3.1.3 Language Detection

We utilize the Python package *langdetect* [35] to automatically identify the natural languages in which model element names were written. For this purpose, the textual representation without type information is used to reduce noise, as element types are consistently in English. The langdetect library has prebuilt profiles for a total of 55 languages. Each language profile contains statistical information about character n-grams typical of the language in question. When detecting a language for input text, this method involves the initial extraction of n-grams, which are then compared against various language profiles. Through this process, probabilities are calculated, and the most probable language is selected. After language identification, all models whose detected language is not English are discarded.

### 3.1.4 Design Considerations

It is worth noting that, within the duplicate and dummy detection steps, several of the described filters partially overlap. For instance, TF-IDF and cosine similarity-based duplicate detection may also capture exact duplicates. Similarly, multiple dummy detection heuristics overlap in the models they flag, as they are designed to be complementary rather than mutually exclusive.

This was a deliberate design choice. The pipeline, released as an open-source Python library, was developed with extensibility and generalizability in mind. It is intended to be configurable: users may select any subset of filters depending on their use case, adjust threshold values as needed, or extend the pipeline with additional filters. Furthermore, since all filters operate on textual representations, the pipeline can be extended to other modeling languages, provided that the support for parsing specific elements for that language is included. The Python library, called Model Cleansing Pipeline for Conceptual Models (MCP4CM), was developed in collaboration with Syed Juned Ali. Interested readers may find more information about it in [10].

Another thing to consider is that, while duplicate and language detection are applicable to any other UML datasets (and potentially to datasets of other modelling languages), some dummy detection heuristics and their associated thresholds were largely tailored to ModelSet. For example, the stopword list reflects common UML element types. All threshold values were determined empirically through manual inspection of ModelSet, with the aim of balancing proper dummy detection and avoiding too many false positives.

## 3.2 Application in the Present Study

In the experiments described in the following chapters of this study, unless a certain step was omitted due to the specific use case, filters were applied in the following order: duplicate removal (including both exact and near duplicates), dummy removal (by applying all described heuristics), and finally language detection and removal of non-English models. Threshold values were kept as described above.

## 3.3 Reproducibility Studies

To evaluate the effect that dataset cleansing has on downstream ML tasks, two previously published works were selected based on their use of ModelSet. ModelSet contains separate collections of Ecore and UML models, and although both studies conduct experiments on each portion of the dataset, the focus of the present study is limited to the UML models. Therefore, only the results obtained for the UML portion of the dataset are replicated and compared with our own. The details of the individual studies are described in the following sections of this chapter.

### 3.3.1 Overview

The general procedure is as follows. First, the available replication packages of the original studies are executed to verify that the reported results can be reproduced. Once reproducibility is established, the original ML methodology and implementation are retained, and only the cleansing steps are modified where applicable. Modifications in cleansing steps may entail replacing some parts with our own equivalent procedures, or introducing additional steps not applied in the original study (e.g., duplicate removal where no such step was originally performed). This approach ensures that any differences in the results can be attributed to the cleansing procedures alone.

Both original studies perform hyperparameter tuning for each trained classifier. The authors report the results achieved with the best hyperparameters, and these are the results we aim to reproduce. We likewise perform hyperparameter tuning on our datasets, following the same methodology from the replication packages. To draw conclusions regarding the changes in performance between the original and our own setup, the experiments are repeated multiple times with the selected best hyperparameters fixed. The mean and standard deviation across repetitions are the ones reported, while the full distributions are subjected to statistical analysis.

For clarity, throughout the remainder of this work, we will refer to the datasets used in the original studies, with the authors' own cleansing steps, as the *baseline datasets*. The datasets obtained after our proposed cleansing pipeline was applied will be referred to as the *revised datasets*.

### 3.3.2 Statistical Analysis

Since the baseline datasets vary considerably from the revised datasets in size, the repetitions are not paired. More specifically, the run $i$ performed on the baseline dataset does not correspond to the run $i$ in the revised setup. Because the two datasets have test sets of different sizes, there is no meaningful mapping between corresponding repetitions. Therefore, we treat the repeated runs as independent.

The overall methodology for statistical testing can be seen in Figure 3.2. On both baseline and revised data, models are trained and performance metrics collected multiple times to account for different train-test splits. Then, the normality of the distribution of the population obtained by multiple runs is assessed using a normality test.

The Shapiro-Wilk test [36] is a statistical test which assumes a null hypothesis that the distribution of the population is normal. When a $p$-value is observed to be smaller than the designated $\alpha$, the null hypothesis is rejected, indicating that the given population is not normally distributed. Conversely, if the $p$-value is greater than the predetermined level of significance $\alpha$, the null hypothesis cannot be rejected, thereby indicating that the data may have a normal distribution (due to the absence of strong evidence contradicting this assumption).

If both distributions satisfy the normality assumption, Welch's two-sample t-test [37] is used to compare their means. This test is appropriate for two independent groups when equal variances are not assumed. A $p$-value smaller than the significance level $\alpha$ indicates that the difference in the two means is significant, whereas a $p$-value greater than or equal to $\alpha$ suggests that there is not enough evidence to claim a difference.

In all other cases, we do not assume normality, so a Mann-Whitney U statistical test [38] is applied. This non-parametric test determines whether two independent groups originate from the same distribution. The idea is to rank all observations from the combined samples, calculate the sum of ranks for each group, and then compare them to see if they differ significantly. If the $p$-value is smaller than $\alpha$, the null hypothesis is rejected, i.e., there is a significant difference in the distributions of the two groups. Otherwise, no statistically significant difference can be concluded.

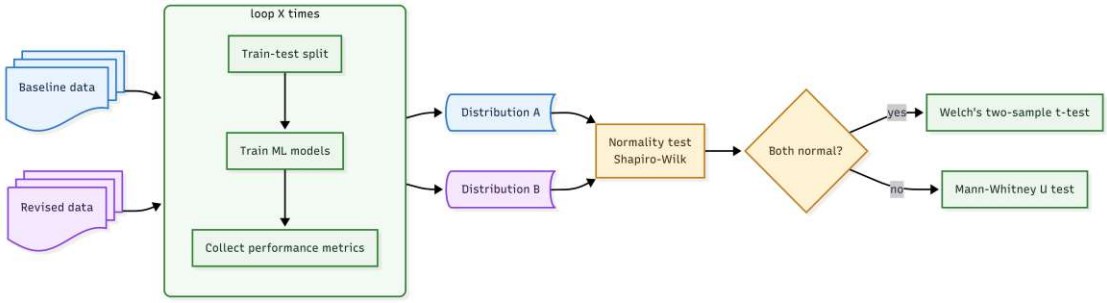



Figure 3.2: Workflow used for statistical testing of the performance distributions obtained across repeated runs.

### 3.3.3 Reproducibility Study One (RS1)

#### Original Study

As machine learning methods, often regarded as black-box approaches, are increasingly introduced in domains where trust and transparency are essential, Explainable Artificial Intelligence (XAI) has emerged as a branch of machine learning focused on providing interpretations of model decisions. Explanations can be generated at a global level, highlighting how individual features influence model behavior across the entire dataset, or at a local level, explaining decisions for specific instances.

In [39], the authors investigate global and local explainability methods for different ML tasks on UML and Ecore data. The ML tasks considered are: *dummy model detection*, a binary classification task aimed at correctly classifying trivial models as dummy (labeled 1) and proper models as non-dummy (labeled 0); *model classification*, a multiclass problem in which each conceptual model is categorized into one of the possible domain classes according to the `category` label; and *tag prediction*, a multilabel task in which each conceptual model may be assigned one or more relevant `tags`.

The classifiers used for the first two tasks are K-Nearest Neighbors (KNN), Support Vector Classification (SVC), and Random Forest Random Forest (RF), while for the third task only RF is trained. The numerical features provided in ModelSet are used to train the classifiers. The authors reserve 70% of the dataset for training and use the remaining 30% as a test set. To obtain optimal performance, they apply hyperparameter tuning via grid search with cross-validation, using F1-score as the selection metric. Each classifier is evaluated on the test set using precision, recall, and F1-score, reported in their macro versions for the multiclass and multilabel tasks. For dummy detection and the multiclass task, balanced accuracy is additionally reported. Once the classifiers are trained, global and local explanations are generated.

Permutation Feature Importance (PFI) [40], [41] is a global explanation method that is model-agnostic, meaning that it does not depend on knowledge of the internal workings of a specific ML model. Each feature is evaluated independently by shuffling its values across the dataset and observing how that affects the model's performance. The greater the drop in performance, the higher the impact (or importance) of the original distribution of this feature on the model's output. In this study, performance is measured using the F1-score.

For local explanations, the baseline study uses several established interpretability methods. LIME [42], SHAP [43], and Breakdown [44] are popular model-agnostic methods for assessing feature importance at the level of individual instances. LionForests [45] is a local explainability method specifically designed for Random Forest models and is among the few approaches applicable to multilabel tasks. However, direct comparison of local explanations between the baseline and revised setups may not be meaningful. The two datasets differ substantially in size due to the different cleansing procedures, meaning that some instances selected for local explanation in the original study are not present in the revised dataset. Furthermore, the baseline study applies specific filtering criteria

when selecting instances for explanation (e.g., restricting analysis to conceptual models belonging to the five most frequent categories), further reducing the overlap between the two setups. Consequently, the sets of instances for which local explanations are generated differ considerably. Since the comparison of global explanations is sufficient to support the conclusions of this study, local explainability is not further elaborated upon here.

Regarding data preparation, several general cleansing steps are applied across all three tasks, such as removing non-informative features (e.g., features containing only `NaN` or constant values) and excluding features such as `id` and `language`, which were considered irrelevant for the tasks.

For the dummy detection task, no additional steps were taken beyond the general ones described above. Dummy models are intentionally retained, as the objective is to automatically identify such models.

For the multiclassification task, on the other hand, dummy and unknown models are considered noise and are therefore removed. Due to the large number of categories and the resulting class imbalance, where many classes contain only a few samples, the authors filter out those models that belong to uncommon categories, keeping only the top ten frequent categories.

Similar to the multiclass problem, the multilabel classification task requires label reduction due to the sparsity of certain tag combinations. To start with, instances assigned only a single tag are removed. Labels that are infrequent or perfectly correlated (correlation equal to 1) are then eliminated. Subsequently, the most frequent labels are retained, and instances with fewer than two tags are again removed.

#### Experimental Setup under the Revised Dataset

As was already mentioned, once the reported results were successfully reproduced on the baseline dataset using the methodology provided in the replication package, we conducted additional experiments under the revised setup. In RS1, new cleansing procedures were introduced, and certain existing steps were replaced with equivalent methods from our proposed pipeline.

To be precise, duplicate detection, as defined in our cleansing pipeline, was applied to all three tasks. For the multiclassification and multilabel tasks, a dummy detection and removal step was also performed. Figure 3.3 illustrates the methodological workflow for each task. As can be seen, only the data cleansing stage is modified, with the ML pipeline remaining identical to that of the baseline study. The number of samples remaining after each cleansing stage can be observed for both the baseline study and the revised experiments. For instance, in the dummy detection task, while the baseline study trained ML models on 5,120 samples, only 1,355 samples remained after duplicate detection in the revised setup.

To assess whether differences between the baseline and revised setups are significant across all three tasks, each classifier was executed 100 times with hyperparameters fixed to the

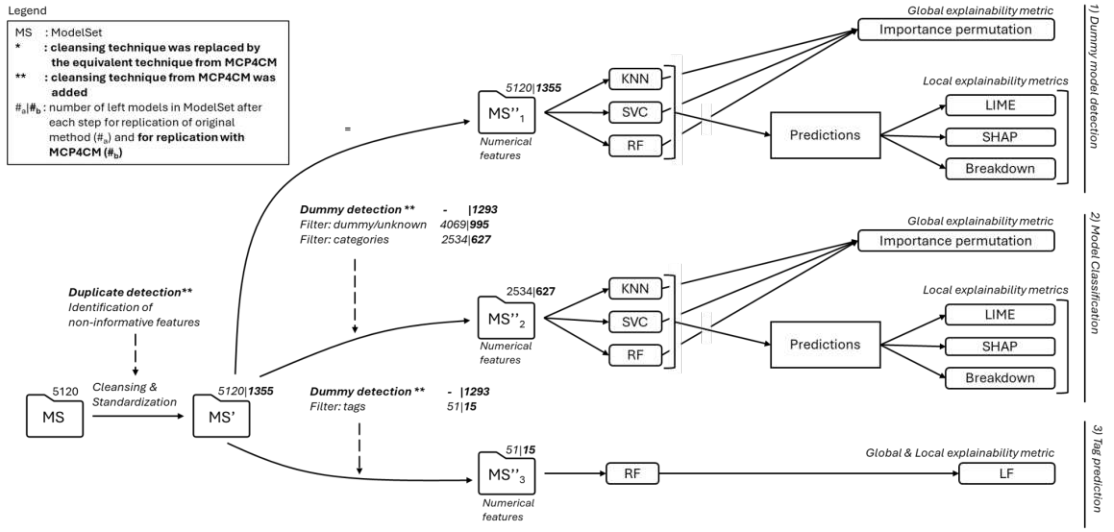



Figure 3.3: Workflow of RS1, showing the baseline and revised cleansing stage and the unchanged downstream ML pipeline. Reprinted from [10], ©2025 IEEE.

best values obtained during tuning. In each run, a random train/test split was generated while maintaining the 70/30 ratio. The evaluation metrics used correspond to those reported in the original study for the respective tasks. The performance distributions obtained were then subjected to statistical analysis using the methodology outlined earlier in this chapter and summarized in Figure 3.2.

### 3.3.4 Reproducibility Study Two (RS2)

#### Original Study

López et al. [17] developed a framework to systematically prepare data and benchmark different ML approaches for the conceptual model classification problem, i.e., the same multiclass classification task considered in [39]. ModelSet is again used; however, instead of training ML models solely on numerical features, the authors explore different ways of encoding the raw data. This includes extracting textual information (i.e., collecting the names of different artefacts present in each conceptual model) or transforming conceptual models into their graph representations. These representations are then further adapted into machine-readable formats depending on the specific ML model employed.

Encodings such as TF-IDF, word embeddings (GloVe), Bag-of-Paths (BoP) as adopted from [8], [46], 2D-TFIDF as proposed in [47], raw graph representation, and others are considered. The ML methods compared include published tools such as Lucene [48], MAR search engine [8], [46], AURORA [49], MEMOCNN [47], as well as traditional ML models like Support Vector Machine (SVM), KNN, and Naïve Bayes variants. Hyper-parameter tuning is performed with cross-validation, and balanced accuracy is reported to account for class imbalance.

Before training the ML models, several cleansing steps are applied, including the removal of non-English models based on the language tag, the exclusion of models labeled as dummy and unknown, and the filtering out of infrequent categories (i.e., those containing fewer than ten conceptual models). The authors also investigate the impact of duplicate retention versus removal on ML model performance. Duplicate detection is performed using an adaptation of an existing approach based on Jaccard similarity, as developed in [50] for detecting duplication in software code.

Due to dependency issues encountered when working with the provided replication package, we reproduced the results for the following ML models: Feed-Forward Neural Network (FFNN), SVM, KNN, and Naïve Bayes variants (Gaussian, Multinomial, Complement). Each of these is tested using both TF-IDF and 300-dimensional GloVe word embeddings, except for the Naïve Bayes models, which are only compatible with TF-IDF.

Furthermore, the part of the study focusing on the comparison between datasets with and without duplicates was not replicated. Instead, we concentrated on reproducing the results obtained on the duplicate-free dataset, as the effect of duplication had already been examined in RS1. RS2 puts focus on evaluating the effect of replacing baseline cleansing steps contingent on labels (e.g., filtering non-English, dummy, and unknown models) with our automated cleansing pipeline. Whilst it is possible to implement such label-dependent filtering for ModelSet, it may not generalize to other conceptual model datasets.

#### Experimental Setup under the Revised Dataset

Following the successful reproduction of the baseline results, a revised experiment was conducted. This involved replacing the baseline duplicate detection method with the one from our proposed pipeline, performing automatic dummy detection and removal, and applying automatic language detection to remove non-English models. Subsequently, any trailing models labeled as dummy or unknown were removed. In the original study, these two categories were excluded before training and were not part of the classification task. Although the automatic dummy detection applied in the revised setup removes most such models, not all of them are covered. To circumvent introducing rare or unintended categories, and to ensure that the baseline setup is adhered to with precision, these residual models were filtered out. Finally, UML models belonging to infrequent categories were filtered out, same as in the baseline study.

Figure 3.4 illustrates the methodology of both the baseline and revised setups. The encoding strategies and ML methods remain identical in both cases, and only the cleansing procedures are swapped to equivalent methods from our pipeline.

Due to the computational cost of training neural networks and limited available resources, the number of repetitions performed to obtain baseline and revised performance distributions was limited to 30, compared to the 100 repetitions conducted in RS1, where only traditional ML models were trained. The decision to perform 30 runs per configuration was informed by [51], which recommends at least 30 repetitions under time constraints

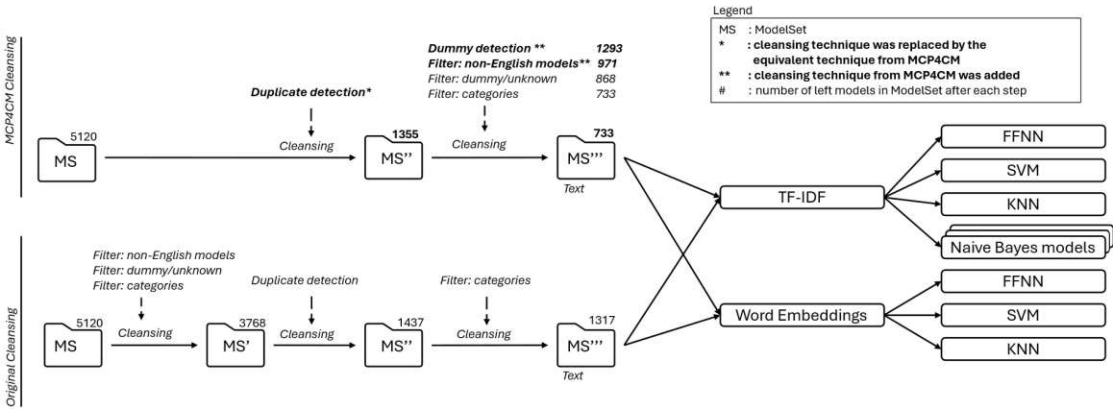



Figure 3.4: Workflow of RS2, showing the baseline and revised cleansing stage and the unchanged downstream ML pipeline. Reprinted from [10], ©2025 IEEE.

as a practical rule of thumb. Each classifier was trained using the best hyperparameters identified during tuning, for both the baseline and revised datasets, across 30 random 80/20 train-test splits. The statistical analysis follows the same methodology described earlier in this chapter and shown in Figure 3.2.

CHAPTER 4

# Recommender System

Bearing in mind that UML has 14 diagram types, each with very specific functionalities that are difficult to generalize, the scope of this part of the study was narrowed. At multiple stages, design decisions had to be made, from determining how to transform diagrams into graph representations to specifying individual tasks aligned with real-world use cases, which would otherwise have significantly increased the workload. Therefore, the decision was made to focus on class diagrams, as they are the most widely used type of UML diagram. To develop a recommender system for UML class diagram completion, ModelSet was once again used.

Since UML diagrams convey information through both semantic and structural signals, GOFA, a generative GNN-LLM architecture, was selected as a machine learning model. This graph foundation model is pre-trained to handle a variety of tasks, ranging from graph completion to question-answering style tasks. For our use case, the recommendation problem was reformulated into four generation tasks conditioned on the local graph context:

- **Node Name (NN):** plausible missing class names are generated.
- **Node Attributes (NA):** masked attributes of a target class are generated.
- **Node Operations (NO):** masked operations of a target class are generated.
- **Edge Type (ET):** the type of a relationship between two target classes is predicted.

## 4.1 Data Preparation and Graph Transformation

Starting from the XMI format of the UML models, multiple data preparation steps were applied, including (but not limited to) the developed cleansing pipeline. Figure 4.1

provides an overview of the data preparation and graph transformation stages used to convert the dataset into the intermediate JSON graph representations. In this section, we describe these two stages in detail.

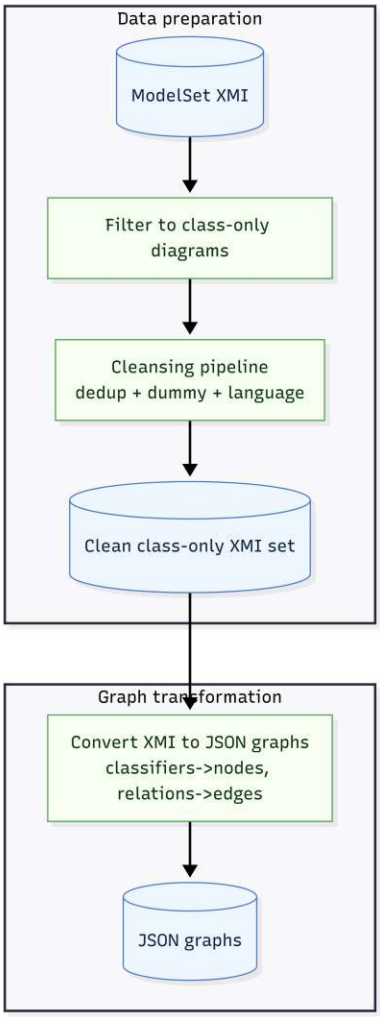



Figure 4.1: Data preparation and graph transformation from UML XMI to JSON graphs.

As previously discussed, ModelSet is not limited to class diagrams. Therefore, the first step was to filter out all other diagram types that were not relevant for this task. Lists of elements representative of different diagram types were devised, and each XMI file was checked for the presence of these elements. A total of 1,639 XMI files were identified as purely class diagrams. However, some XMI files contained both class diagrams and other diagram types. These files were further processed to remove the irrelevant elements and retain only the class-domain content.

To achieve this, elements from the corresponding devised list (e.g., classes, associations,

generalizations, attributes) were retained, together with their containers (e.g., model, package). Because XMI frequently encodes relationships through cross-references rather than Extensible Markup Language (XML) nesting, the references originating from the retained elements were then examined. Whenever a retained element referenced another element, the referenced target was also retained if it belonged to the class-domain. If a reference pointed to an element that had been removed as a non-class element, the reference was pruned to avoid dangling pointers. Associations were kept only when both their endpoints connected class-based elements.

This approach may not be entirely foolproof. It is possible that some class-type elements were not included in the devised list, particularly more obscure UML constructs, or that some generic types difficult to distinguish as non-class elements were retained. Nevertheless, the core elements required for the subsequent graph transformation were preserved. Any non-class elements that may have been inadvertently retained did not affect later stages, as the graph construction step only extracted specific class-domain elements. Overall, the resulting diagrams remained structurally valid UML class diagrams and provided a suitable foundation for the transformation into graph representations. An example of a UML file containing mixed diagram types is shown in Figure 4.2. After filtering the model to retain only class-diagram content, the resulting class-only version is shown in Figure 4.3.

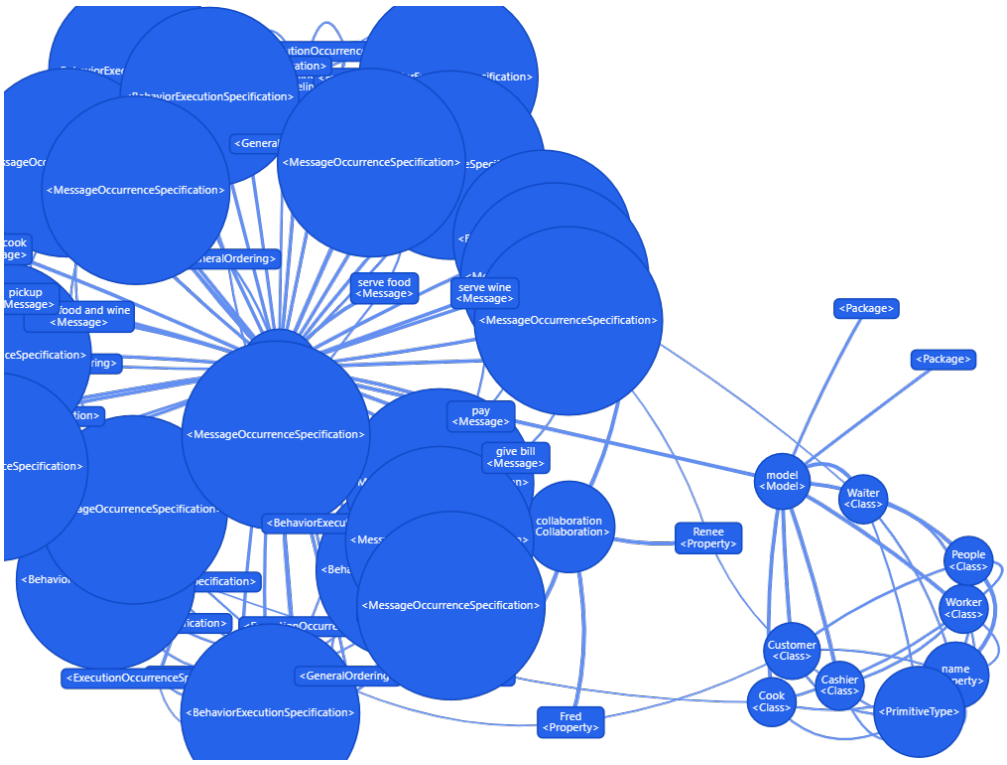



Figure 4.2: Example of a mixed UML model before filtering.

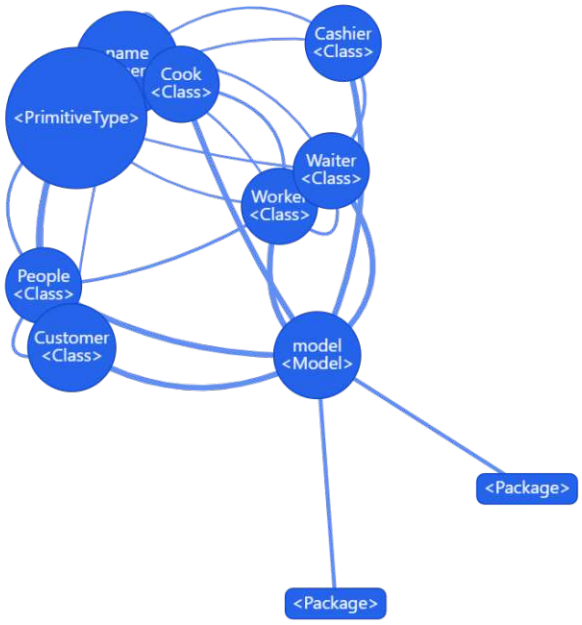



Figure 4.3: Retained class-diagram-only content after filtering.

After this step, 3,086 UML files remained. These were then subjected to preprocessing using the developed pipeline with all available filters and default thresholds applied. After deduplication and the removal of dummy and non-English models, 447 UML class diagrams remained in the dataset. These diagrams were subsequently transformed into JSON graphs whose format format is shown in Listing 4.1.

Listing 4.1: Structure of the JSON graph format used to represent the processed UML diagrams.

```
{
  "graph_id": "...",
  "nodes": [{"id": "...", "text": "...", ...}],
  "edges": [{"src": "...", "dst": "...", "type": "...", "text":
     "...", ...}],
  "meta": {"stats": {...}, "source_file": "..."}
}
```

At this stage, classifier elements (e.g., classes, enumerations, interfaces) were transformed into graph nodes, storing their `xmi:id` together with text describing their content. For example, in the case of classes, the text field contained a UML header (e.g., `UML Class: Name`) followed, when present, by inline sections describing attributes and operations. Attributes stored their names and types separated by semicolons, while operations stored their names together with input and return parameter types. An example is shown in Listing 4.2:

Listing 4.2: Example of the textual content stored for a UML class node.

```
UML Class: Card. Attributes: suit: Suit; value: Value;
MAX_VALUE_OF_ACE: Integer; BLACKJACK_VALUE: Integer. Operations: Card
(suit: Suit, value: Value); getSuit() -> Suit; getValue() -> Value;
numericalValue() -> Integer; toString() -> String.
```

Associations, generalizations, dependencies, and realizations were represented as edges. These stored the source and destination identifiers, the relationship type, and an associated textual field containing the relationship name, endpoint names, and multiplicities where applicable. An example is shown in Listing 4.3:

Listing 4.3: Example of the textual content stored for a UML relationship edge.

```
ASSOCIATION deck_DealerBot_Deck_6: DealerBot [role=dealerbot,
multiplicity=1] <-> Deck [role=deck, multiplicity=1]
```

The next step involved further processing the obtained JSON files to match the specific format required by the chosen ML architecture and the designed completion tasks. The details of this process are described in the following section.

## 4.2 Task-Specific Preparation

GOFA takes TAGs with rich textual information as input. Since the model can only process a limited amount of context, individual class diagrams had to be broken into smaller samples containing only local neighborhood information. The task-specific preparation stage, therefore, began by splitting the JSON graph dataset and sampling $k$-hop subgraphs from the available JSON graphs, as illustrated in Figure 4.4.

Although precautionary measures had already been taken to mitigate leakage risks by deduplicating the dataset, sampling smaller contexts from different models introduced the possibility of clones appearing again. Two models might not have been duplicates when considered as a whole, yet their fragments could be identical, particularly when the models originated from the same domain. Sampling multiple times from a single model could also introduce highly overlapping contexts that could eventually appear in different splits. For this reason, before sampling and further preprocessing the data to fit different tasks, the dataset was first split at the diagram level into training, validation, and test sets using an 80/10/10 ratio.

Afterwards, $k$-hop subgraphs were sampled within each split using $k = 3$, with a maximum of five randomly selected nodes per hop. Each available diagram was sampled multiple times to form samples for the four tasks. To avoid oversampling small diagrams or undersampling richer diagrams containing many different patterns, the following scheme was devised for determining the number of samples per diagram based on its size. Diagrams containing only three nodes were skipped. Diagrams with three or four nodes produced a single sample (preferring the NN task, with NA, NO, and ET used as fallbacks if necessary). Diagrams with five to fifteen nodes were sampled with a base quota of two

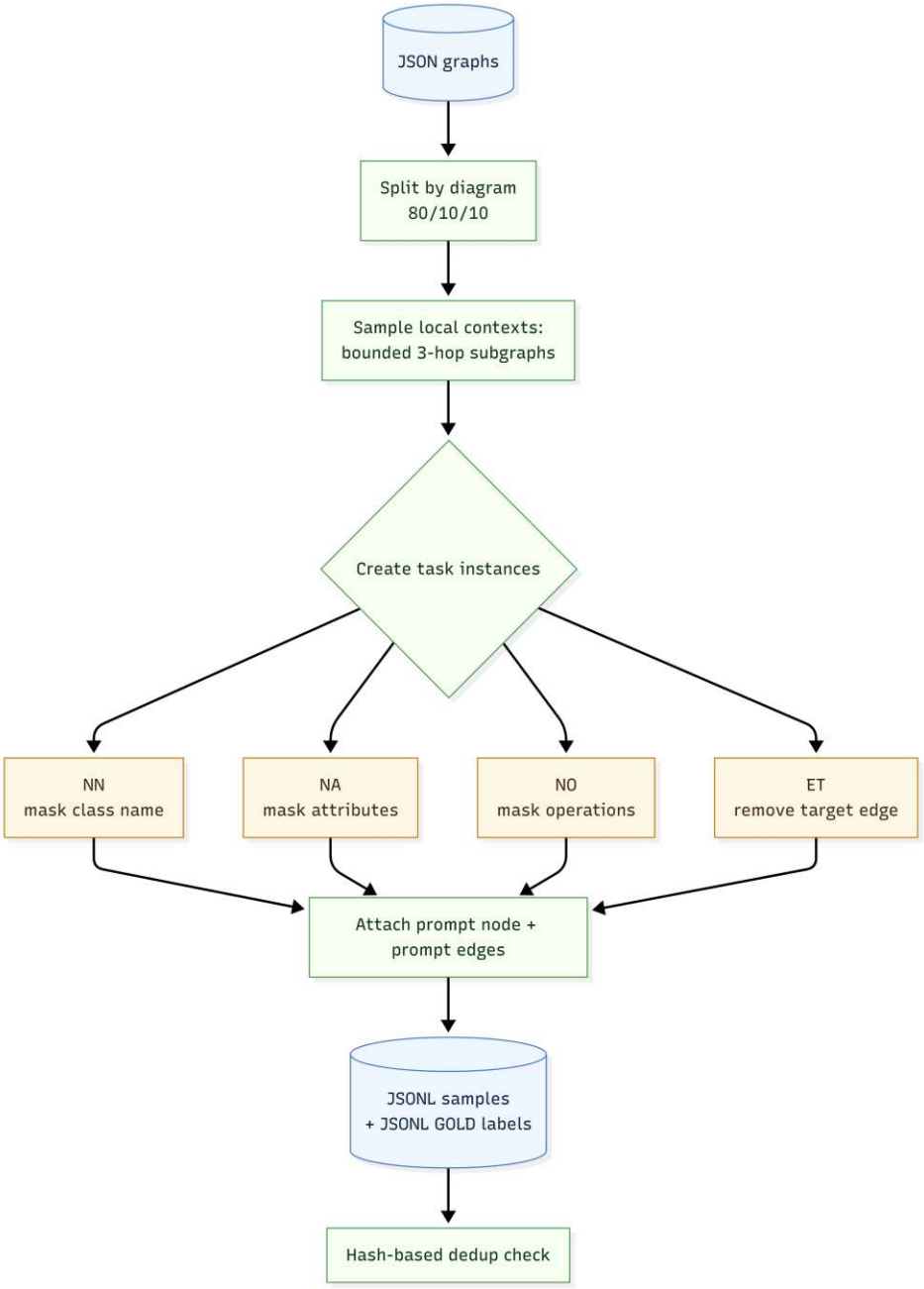



Figure 4.4: Task-specific preparation and sampling from JSON graphs to JSON Lines (JSONL) task instances and gold labels.

samples per task. Diagrams containing 16-50 nodes produced double the base quota, while diagrams with more than 50 nodes produced triple the base quota.

When sampling from diagrams for the NN, NA, and NO tasks, a seed node was randomly selected and declared the *target* node. The node was chosen randomly but subject to several constraints. Isolated nodes without neighbors were skipped. Additionally, if multiple samples were drawn from the same diagram, the seed nodes had to be at least two hops apart to avoid repeatedly sampling from the same local region. Nodes that were not of class type (e.g., enumerations or interfaces) were also skipped, as they did not contain class names, attributes, or operations that could be masked. From the chosen seed node, a 3-hop neighborhood was computed while retaining at most five randomly selected neighbors per hop.

For the ET task, the target was not a node but rather a relationship between two nodes. A list of unique source-destination edge pairs was first computed, after which a number of edges was sampled according to the required quota, with rarer relationship types being prioritized. As association relationships overwhelmingly outnumbered generalizations, realizations, and dependencies, they were the last to get picked. Subgraphs were then constructed from both nodes participating in the selected relationship, and the union of their neighborhoods was taken.

Once the subgraphs had been sampled, node identifiers were relabeled to integers ranging from 0 to $N - 1$. All bidirectional edges were converted into two directed edges (one in each direction), as required by GOFA. Within each node, a maximum of ten attributes and ten operations were retained to prevent excessively long textual descriptions. Each target node additionally received the tag `(TARGET)` appended to the end of its text block.

To simulate auto-completion scenarios in which a partial class diagram was available as context and certain elements were missing, four masking strategies were applied. For the NN task, the entire text line of the target node was replaced with the placeholder `UML Class: [MASKED]. (TARGET)`. If the name of the masked class appeared within relationship endpoints or roles, it was also replaced with the placeholder `[MASKED]`. For the NA task, the entire attributes block was replaced with `Attributes: [MASKED]`, while for the NO task, the operations block was replaced with `Operations: [MASKED]`. Finally, for the ET task, the relationship under consideration between the two target nodes was removed. The masked or removed content was stored separately in gold-label JSONL files to be used later for evaluation of the generated output.

In addition to the nodes obtained through 3-hop sampling, each sample contained an additional prompt NOG node. Five types of prompt nodes were attached to subgraphs depending on the task type, as shown in Table 4.1. These prompts provided instructions specifying the task to be performed by the model. For the NN task, the prompt differed depending on whether the sample belonged to the training/validation sets or to the test set. This distinction was necessary because the evaluation of this task required the model to generate five class-name recommendations for the test set, enabling the use of

recommender-system metrics. During training and validation, however, the model was optimized using a next-token prediction objective with teacher forcing, which assumed a single target output sequence. Consequently, the test prompt asked the model to produce five recommendations, whereas the training and validation prompts requested only a single class name.

Table 4.1: Prompt formulations used for the four GOFA task types.

| **Task** | **Prompt** |
|---|---|
| Node Name – Train and Validation | Suggest the UML class name for the masked `TARGET` node from the diagram context. Output exactly one word (no placeholders). |
| Node Name – Test | Suggest 5 plausible UML class names for the masked `TARGET` node, ordered from most to least appropriate. Return only one line in the form: `Name1, Name2, Name3, Name4, Name5` (each a single identifier, no generic placeholders). |
| Node Attributes | You complete masked attributes for a `TARGET` UML class. Output exactly one line: `Attributes: name1: Type1; name2: Type2; ...`. Use any count (0+). Types from the diagram or UML primitives. |
| Node Operations | You complete masked operations for a `TARGET` UML class. Output one line: `Operations: op1; op2; ...`. Each op: `name(param: Type, ...) -> ReturnType`; use `-> void` if none. |
| Edge Type | What is the UML relationship type between the two `TARGET` classes? Reply with only one word out of: `GENERALIZATION`, `ASSOCIATION`, `REALIZATION`, or `DEPENDENCY`. |

Once prompt nodes were included in the samples, additional edges were introduced to connect them to the required context. In the NA, NO, and ET tasks, the prompt node was connected only to the target node(s). However, in the NN task, the prompt node had to be connected to all nodes in the sampled subgraph. Experimental observations indicated that connecting the prompt node only to the target node caused the model to repeatedly generate 'TARGET' as the output rather than predicting a meaningful class name. This behavior was likely related to one of GOFA's pre-training tasks, namely an information-retrieval task in which the model was required to retrieve the content of a specific node. In the present setup, the model would therefore simply retrieve the appended tag `(TARGET)`, as all other node information in the NN task was masked.

Each relationship between the prompt node and any other node was recorded twice:

once as a directed edge from the prompt node to the graph node with the following text: 'This edge connects the prompt node to a node in the graph.'; and once in the opposite direction with text: 'This edge connects the nodes in the graph to a prompt node." This duplication was a design requirement of the GOFA architecture.

Another GOFA-specific requirement concerned two additional fields: `question` and `complete`, which indicated the task type according to GOFA's pre-training formulation. In this work, all tasks were treated as question-answering tasks. Hence, for each sample, the `question` field stored the identifier of the prompt node, while the `complete` field remained empty. If the task was instead marked as a completion task, the model tended to produce incoherent outputs.

Once the samples had been generated, SHA-256 hashes were computed again to ensure that no exact duplicates existed either within the individual splits or across different splits. Table 4.2 presents the number of samples in each split for every task.

Table 4.2: Number of generated samples in the training, validation, and test splits for each task.

| Split | Node Name | Node Attributes | Node Operations | Edge Type |
|---|---|---|---|---|
| Train | 704 | 603 | 489 | 673 |
| Validation | 87 | 72 | 60 | 83 |
| Test | 87 | 76 | 66 | 84 |

An additional task formulation was initially explored, in which the model was asked to predict whether there is a relationship between two target classes. This was formulated as a binary classification problem by sampling both positive pairs (where a relationship exists) and negative pairs (where no relationship exists in the original model). Although balanced sampling was employed (50% positive and 50% negative pairs), preliminary experiments showed that the model converged to predicting the positive class for all inputs. This behavior is plausible in the present setting because link-existence prediction is inherently noisy. Namely, the absence of an explicit relationship in a UML model does not necessarily imply that the two classes are semantically unrelated, and different modelers may reasonably introduce relationships that are not present in the original diagram. As this task did not yield any meaningful results and would have introduced additional noise into the multi-task training setup (all tasks were used to train a single model), it was not included in the final experimental design.

## 4.3 Experimental Set-Up

Although several auto-completion and recommendation approaches have been proposed in the MDE domain, direct empirical comparison was outside of the scope of this study. Existing works differ substantially with respect to the modeling languages they target,

task formulation, data representation, and experimental setups, which makes direct benchmarking difficult. Moreover, some methods rely on paid API access or private industry datasets, further limiting reproducibility and comparability.

Therefore, to establish a baseline for our experiments, we first evaluated the pre-trained GOFA model in a zero-shot setting by prompting it directly with the test set prepared as described above. The generated outputs were then compared to the gold labels, and the results were evaluated. The task-specific evaluation scheme included ranking metrics for name recommendations, classification metrics for the edge types, and soft matching for the attributes and operations tasks. The evaluation procedure is described in more detail in Section 4.4.

To further assess the model's potential for the task, we then performed instruction fine-tuning of the GOFA model and compared the resulting performance with the zero-shot baseline. Since the released GOFA training pipeline relies on TAGLAS [52], a dataset interface that unifies multiple graph datasets, we first implemented a lightweight dataset wrapper that converted our JSONL files into the expected `TAGData` format. The GOFA implementation included filters that approximate Graphics Processing Unit (GPU) memory consumption based on the number of nodes and edges; samples that were estimated to exceed memory limits were automatically discarded. In addition, we summed the number of characters across all node and edge text fields per sample and discarded samples exceeding 3,000 characters.

Fine-tuning was performed on an NVIDIA A100 GPU with 80 GB of memory. The model was initialized using the pre-trained GOFA checkpoints, and only the GNN layers and Low-Rank Adaptation (LoRA) adapters [53] were updated, while the backbone LLM (Mistral-7B) weights remained frozen. The original training objective was retained, namely, teacher-forced supervised next token prediction, and the model was trained jointly on all training samples across the four tasks.

Because the lengths of individual samples varied greatly, training was performed with a batch size of one sample, while gradient accumulation over 16 steps was used to simulate a larger effective batch size. Training was performed for a maximum of 10 epochs, with early stopping applied if no improvement was observed for three consecutive epochs; the best-performing checkpoint was saved. The maximum sequence length was set to 256 to avoid memory limitations. Optimization used the AdamW optimizer with a learning rate of $1 \times 10^{-4}$, weight decay of 0.1, and a cosine annealing learning-rate scheduler. All experiments were conducted with a fixed random seed.

## 4.4 Task Specific Evaluation

The NN task is evaluated as a top-5 ranked recommendation problem. The model is instructed to generate five candidate names for the masked class, and the metrics MRR@5, SR@1, and SR@5 are computed. Evaluating the correct name for a class can be ambiguous, however, as many synonyms could be acceptable, even if they do not match

the exact gold label. Nevertheless, the goal is to see whether the gold name appears among the top candidate predictions.

For the ET task, the model predicts the type of relationship between two nodes from the four possible types. This can therefore be treated as a multiclass classification setup. The recorded metrics include accuracy and macro-F1 across the four classes, as well as per-class precision, recall, and F1-score.

For the NA task, the predicted and gold strings for each sample are parsed into lists of `(attribute_name, attribute_type)` pairs, and matching is performed using fuzzy string similarity. For each predicted pair, an attempt is made to match it to one of the unmatched gold pairs. A full match is first attempted, where both the attribute name and type correspond. Similarity is computed using the `token_sort_ratio` function from the *RapidFuzz* library [54], which calculates a Levenshtein edit-distance-based similarity score between the two pairs, while allowing for minor formatting or spelling differences. If the similarity exceeds 85%, full credit of 1.0 is assigned. Otherwise, a match between only the attribute names is attempted, in which case partial credit is awarded if successful. For each sample, weighted true positives are computed, from which precision, recall, and F1-scores are derived. The macro-averaged precision, recall, and F1-score are then calculated across all samples.

Evaluation of the NO task follows the same procedure as the NA task. However, in this case, each sample is parsed into a list of operation strings of the form `name(params) [-> returnType]`. Because both input and return parameters are optional, achieving an exact string match between predictions and gold labels is difficult. Full credit is therefore assigned if the strings exceed 80% similarity, while partial credit is given when at least the operation name matches with over 85% similarity. The same evaluation metrics as in the NA task are used.

# CHAPTER 5

# Results

This chapter presents the main empirical findings of the thesis and provides answers to the two research questions. We begin with an exploratory analysis of ModelSet, examining the effect of each filtering step in the devised cleansing pipeline on the dataset's size and composition. We then report on the results for the two reproducibility studies, highlighting the effect of the cleansing procedures on downstream performance and assessing the extent to which the original findings extend to the revised setup. Finally, the chapter concludes with the evaluation of the GNN-LLM-based recommender system for UML class diagram completion, comparing the zero-shot and fine-tuned configurations across the four tasks.

## 5.1 Cleansing pipeline

In this section, we discuss the behavior of each component of the proposed pipeline on ModelSet and compare the outcomes, where possible, with the manually assigned labels provided in the dataset.

### 5.1.1 Duplicate Detection

With respect to exact duplicates in ModelSet, applying SHA-256 hashing revealed that, out of 5,120 UML models, 2,701 were unique files, while 2,419 files were grouped into 376 duplicate clusters. This corresponds to 47.2% of the models being exact duplicates.

To estimate the number of near clones, exact duplicates were first removed while retaining one representative file from each duplicate group. This resulted in 3,077 files, which were then processed using the TF-IDF and cosine similarity filter with a threshold of 0.8. Of these, 1,107 files were found to be unique, while the remaining 1,970 files were grouped into 248 near-duplicate groups, corresponding to 64% of near duplicates under the chosen threshold. It should be emphasized that the observed percentage is an estimate

contingent upon the specified similarity threshold and should not be interpreted as an inherent property of the dataset.

The presence of both exact and near duplicates in a dataset intended for machine learning should be a cause for concern. While it is evident that exact duplicates may be problematic in training, as they may compromise the independence of training and test sets, the impact of near duplicates is less straightforward. Determining when two models are sufficiently distinct is inherently ambiguous. This challenge is exacerbated when the dataset consists of models from a single domain (e.g., medical) where naming conventions may naturally overlap. Nevertheless, near duplicates can introduce highly similar structural and lexical patterns across splits and should therefore be carefully considered, with transparent reporting of similarity thresholds.

### 5.1.2 Dummy Detection

According to the available labels in ModelSet, 606 UML models are labeled as dummy, and 445 as unknown. As we have seen that many duplicate files are present, these counts may be inflated. On the other hand, using the proposed heuristics with default threshold values, 574 UML models were flagged as dummy. This results in 60 models labeled as dummy that were not detected by the automatic method (false negatives), and 28 models flagged by the heuristics that were not labeled as dummy (false positives), of which 13 were labeled unknown. As with near-duplicate detection, these numbers are sensitive to the selected thresholds.

Treating the provided labels as ground truth yields an accuracy of 98.28%, precision of 95.12%, recall of 90.10%, and F1-score of 92.54% for the dummy detection heuristics. These metrics reflect agreement with the provided labels and should be interpreted cautiously, as duplicate files inflate the sample size, and some models labeled as unknown may in fact represent dummy models as well.

Figures 5.1–5.4 present examples of models (in graph form) where the label and the heuristic classification disagree. Figures 5.1 and 5.2 show models flagged by the heuristics but not labeled as dummy, while Figures 5.3 and 5.4 show the opposite case. The model in Figure 5.1 contains numerous elements that appear meaningful at first glance; however, all names correspond directly to element types and lack domain-specific content. Conversely, Figure 5.2 depicts a segment of a model with many domain-specific names, albeit with a few placeholder classes. Figure 5.3 displays a model that, despite not being in English, seems to carry enough semantic signal to avoid being categorized as dummy. Finally, Figure 5.4 has names long enough to pass the heuristic thresholds, although these appear semantically weak.

Dummy detection is an intrinsically complex task. Many edge cases are open to interpretation, and defining a strict boundary between meaningful and trivial models is challenging for both automated methods and human annotators. This underscores the importance of careful heuristic design and threshold selection when preparing a dataset for subsequent ML tasks.


Figure 5.1: Heuristics flagged as dummy, but not labeled as dummy (example 1).

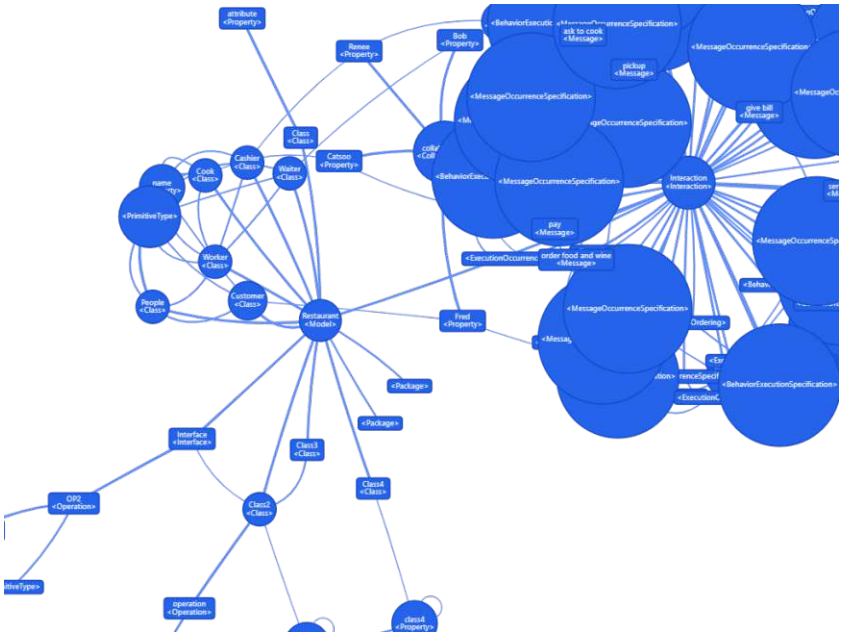


Figure 5.2: Heuristics flagged as dummy, but not labeled as dummy (example 2).


Figure 5.3: Labeled as dummy, but not flagged by the heuristics (example 1).

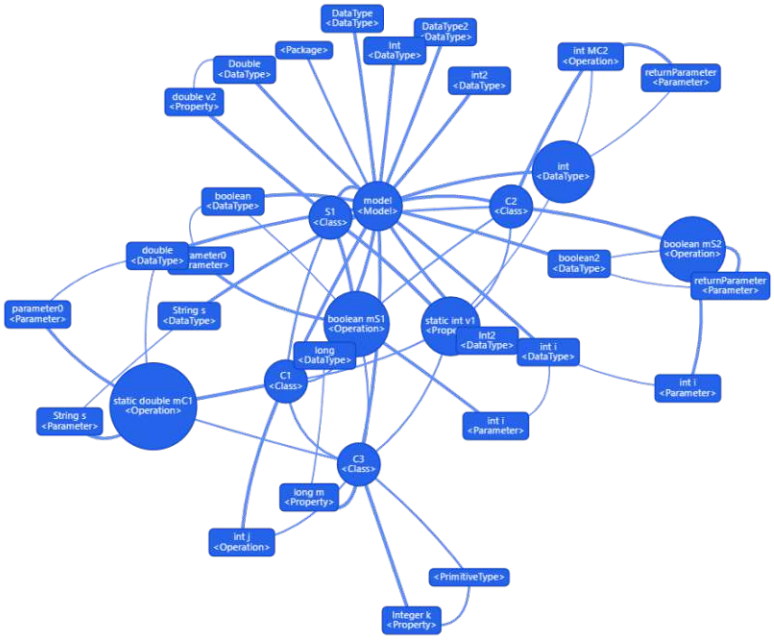



Figure 5.4: Labeled as dummy, but not flagged by the heuristics (example 2).

### 5.1.3 Language Detection

ModelSet contains 4,497 UML models labeled as English. The automatic language detection step identified 4,443 English models. Of these, 4,224 models were consistently classified as English by both methods. Additionally, 273 models labeled as English were not identified as such by the automatic method, and 219 models were detected as English by the pipeline but not labeled as English in ModelSet.

Manual inspection indicates that both approaches occasionally retain models that are not entirely in English. This may be attributed to the frequent use of standard English terminology in software design, especially in technical domains, with only a few words being in a language other than English. Moreover, as was already pointed out, many UML elements receive auto-generated names based on their type, which are always in English. These factors may lead to misclassification by both human annotators and automated language detection methods.

### 5.1.4 Summary

Overall, these findings highlight the substantial impact that preprocessing can have on dataset size and composition. Careful exploratory analysis is essential prior to applying data-driven methods, particularly for unconventional datasets such as collections of conceptual models. To further illustrate how dataset quality influences downstream performance, the results of two reproducibility studies of previously published works are presented in the following sections.

## 5.2 Reproducibility Study One

In the first reproducibility study, three classification tasks were considered: dummy detection, multiclass classification, and multilabel classification. Table 5.1 presents the averaged results (across 100 runs) for the dummy detection and multiclass classification tasks, together with best hyperparameters identified for each classifier. Results obtained on the original baseline dataset are highlighted in gray, while those obtained on the revised dataset (after cleansing with MCP4CM as described in Chapter 3) are shown alongside them. MCP4CM values are shown in bold when they differ significantly from their baseline counterparts at a significance level of $\alpha = 0.05$. Note that multiple hypothesis tests were performed across different tasks, classifiers, and metrics. Since no multiple-comparison correction was applied, boldfaced significant results should be interpreted cautiously.

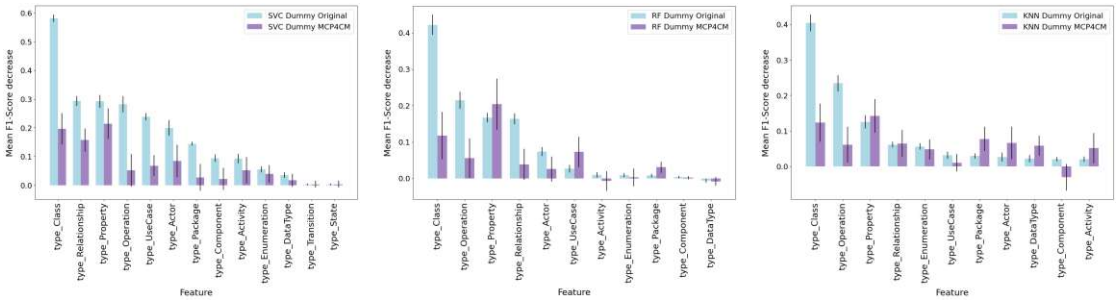
Performance across both tasks is significantly lower for the revised dataset compared to the baseline. The largest decrease (51%) is observed in precision for the multiclass task using the RF classifier. This result is consistent with earlier findings that 47.2% of the samples belong to exact duplicate clusters, which are retained in the baseline setup and may influence performance estimates. Additionally, in most cases, variance across splits

increases in the revised dataset (as indicated by higher standard deviation), suggesting reduced stability of the results.

Similar patterns can be observed in Table 5.2, which reports the results for the multilabel classification task. Again, baseline results are highlighted, and bold values show a significant difference. Performance is consistently lower in the revised setup, although recall does not exhibit a statistically significant difference. The performance drop, however, is less pronounced than in the other two tasks.

This may be explained by the substantially smaller dataset used for this task in the baseline study due to extensive filtering on tags. Specifically, only 51 samples were retained in the baseline setup (with 15 in the revised setup). This much smaller dataset may reduce the likelihood of duplicates appearing across train-test splits. Furthermore, given the limited number of samples and a considerable number of possible tags, the classifier may be prone to overfitting in both setups.

In this study, global explanations for all tasks and all classifiers have been generated in addition to performance evaluation. Figures 5.5, 5.6, and 5.7 compare the global importance of individual features, as measured by PFI, between the baseline and revised setups for dummy, multiclass, and multilabel tasks, respectively.




(a) PFI with SVC (b) PFI with RF (c) PFI with KNN

Figure 5.5: Comparison of global permutation feature importance results for dummy detection in RS1 using SVC, RF, and KNN. Reprinted from [10], © 2025 IEEE.

In general, individual scores given to each of the features are different across the board, with their relative ordering also changed between the two setups. In the dummy detection and multiclass classification tasks, absolute feature importances are generally lower in the revised setup, and the distributions tend to be flatter and more spread out. For example, the feature `type_class` dominates across all three classifiers in the baseline dummy detection task but shifts to the second position in the revised setup, with much lower importance. While in the baseline setup the classifiers relied heavily on the number of classes present in a model to determine whether it is dummy or not, in the revised setup other elements played a substantial role as well.

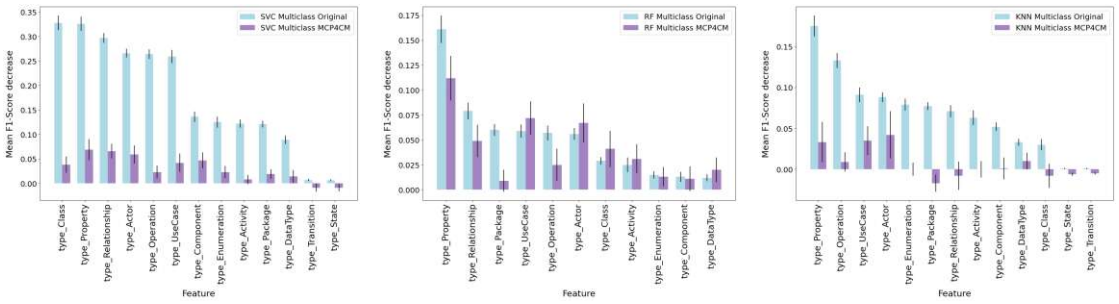
These findings indicate that, on top of affecting classifier performance, preprocessing also alters their decision-making logic. Features that dominate in one setup may become less

Table 5.1: Comparison of metrics for Dummy Detection and Model Classification from RS1 between the original and MCP4CM cleansed data, averaged over 100 runs. Bold values indicate statistical significance compared to the original at $\alpha = 0.05$. Reprinted from [10], © 2025 IEEE.

| Dataset | Task | Classifier | Best params | Balanced Accuracy (Avg ± SD) | Precision (Avg ± SD) | Recall (Avg ± SD) | F1-Score (Avg ± SD) |
|---|---|---|---|---|---|---|---|
| Original | Dummy | SVC | C: 300, cw: None, $\gamma$: 0.1, k: rbf | 0.88 ± 0.02 | 0.87 ± 0.03 | 0.77 ± 0.03 | 0.82 ± 0.02 |
| MCP4CM | Dummy | SVC | C: 1000, cw: None, $\gamma$: 1.0, k: rbf | **0.65 ± 0.04** | **0.44 ± 0.09** | **0.33 ± 0.08** | **0.37 ± 0.07** |
| Original | Dummy | KNN | ls: 5, nn: 3, p: 1, w: distance | 0.88 ± 0.01 | 0.89 ± 0.05 | 0.78 ± 0.03 | 0.83 ± 0.03 |
| MCP4CM | Dummy | KNN | ls: 5, nn: 2, p: 1, w: distance | **0.68 ± 0.04** | **0.49 ± 0.09** | **0.38 ± 0.08** | **0.43 ± 0.07** |
| Original | Dummy | RF | cw: None, md: None, msl: 1, mss: 5, ne: 300 | 0.88 ± 0.01 | 0.95 ± 0.02 | 0.78 ± 0.03 | 0.85 ± 0.02 |
| MCP4CM | Dummy | RF | cw: balanced, md: None, msl: 3, mss: 2, ne: 100 | **0.70 ± 0.04** | **0.41 ± 0.09** | **0.44 ± 0.09** | **0.41 ± 0.07** |
| Original | Multiclass | SVC | C: 500, cw: None, $\gamma$: 0.1, k: rbf | 0.76 ± 0.01 | 0.79 ± 0.01 | 0.76 ± 0.01 | 0.78 ± 0.01 |
| MCP4CM | Multiclass | SVC | C: 10.0, cw: balanced, $\gamma$: 0.1, k: rbf | **0.29 ± 0.03** | **0.30 ± 0.03** | **0.29 ± 0.03** | **0.28 ± 0.03** |
| Original | Multiclass | KNN | ls: 50, nn: 3, p: 1, w: distance | 0.79 ± 0.01 | 0.79 ± 0.02 | 0.79 ± 0.01 | 0.79 ± 0.02 |
| MCP4CM | Multiclass | KNN | ls: 50, nn: 10, p: 1, w: distance | **0.30 ± 0.02** | **0.29 ± 0.03** | **0.30 ± 0.02** | **0.29 ± 0.02** |
| Original | Multiclass | RF | cw: None, md: None, msl: 1, mss: 2, ne: 200 | 0.79 ± 0.01 | 0.83 ± 0.02 | 0.79 ± 0.01 | 0.81 ± 0.01 |
| MCP4CM | Multiclass | RF | cw: balanced, md: 10, msl: 1, mss: 3, ne: 100 | **0.33 ± 0.03** | **0.32 ± 0.03** | **0.33 ± 0.03** | **0.32 ± 0.03** |

Table 5.2: Comparison of the performance metrics for Multilabel Classification task from RS1 between the original and MCP4CM-cleansed data across different classifiers, averaged over 100 runs. Bold values indicate statistical significance compared to the original at $\alpha = 0.05$. Reprinted from [10], © 2025 IEEE.

| Dataset | Best params | Precision Macro (Avg ± SD) | Recall Macro (Avg ± SD) | F1-Score Macro (Avg ± SD) |
|---|---|---|---|---|
| Original | md: 7, mf: sqrt, b: True, msl: 1, ne: 500 | 0.43 ± 0.06 | 0.42 ± 0.08 | 0.41 ± 0.07 |
| MCP4CM | md: 5, mf: None, b: True, msl: 1, ne: 10 | **0.37 ± 0.11** | 0.41 ± 0.11 | **0.38 ± 0.10** |




(a) PFI with SVC (b) PFI with RF (c) PFI with KNN

Figure 5.6: Comparison of global permutation feature importance results for multiclass classification in RS1 using SVC, RF, and KNN. Reprinted from [10], © 2025 IEEE.

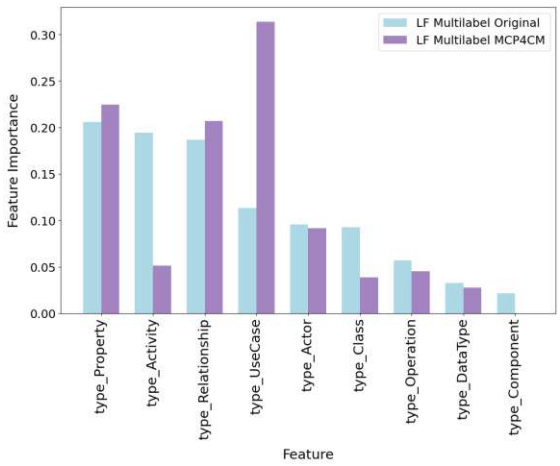



Figure 5.7: Comparison of global feature importance results for multilabel classification task in RS1. Reprinted from [10], © 2025 IEEE.

relevant in another. Consequently, interpretability conclusions derived from the baseline study may not directly generalize to the cleansed data.

## 5.3 Reproducibility Study Two

In the second reproducibility study, the focus was on multiclass classification using different encoding strategies and a range of classifiers, from traditional ML methods to FFNNs. Table 5.3 reports the average balanced accuracy (across 30 runs) for the revised and baseline setups, as well as the best identified hyperparameters, and the resulting ranking of the benchmarked classifiers. Values for the revised dataset that are statistically significantly different from the corresponding baseline results at a significance level of $\alpha = 0.05$ are in bold. As above, given the number of hypothesis tests conducted across classifier–encoding pairs, the resulting significance statements should be interpreted with caution.

Balanced accuracy increased for all the classifier-encoding pairs in the revised setup, with most differences being significant. The observed improvement reflects the strong dependency of the encoding and learning procedures on the underlying preprocessing decisions. In addition, the relative ranks of the individual classifiers changed across the two settings, indicating that data quality and distribution can influence comparative benchmarking outcomes. This highlights the importance of transparent and standardized preprocessing in benchmarking studies, as even small variations in data preparation may lead to substantial differences in reported model performance and ranking.

## 5.4 Answer to Research Question One - Summary

The results of the two studies differ substantially. While performance in the first study decreases after cleansing, performance in the second study increases. Several factors may explain these contrasting outcomes.

First, the baseline setup in the first study does not include duplicate detection and removal, which introduces the risk of data leakage. Once duplicates are removed, the number of available samples is considerably lower. Given that the classifiers in this study rely on numerical features representing counts of element types, which carry limited semantic information, the observed reduction in performance is consistent with the reduced dataset size and removal of noisy samples. Although lower, the results achieved in the revised setup likely provide a more conservative estimate of how well the classifiers generalize to unseen data.

In contrast, the second baseline study already accounts for duplicate removal, and the reported baseline results reflect this. Furthermore, whereas the first study uses numerical count-based features, the second study relies on textual representations. Text-based encoding, such as word embeddings, captures richer semantic information and is particularly sensitive to dataset composition. Reducing noise and redundant models may

Table 5.3: Comparison of the performance metrics for RS2 between the original and MCP4CM-cleansed data across different classifiers, averaged over 30 runs. Bold MCP4CM values indicate a statistically significant difference compared to the original at $\alpha = 0.05$. Adapted from [10], © 2025 IEEE.

| Model | Encoding | Original Acc. | MCP4CM Acc. | Original Best Hyper. | MCP4CM Best Hyper. | Original Rank | MCP4CM Rank |
|---|---|---|---|---|---|---|---|
| FFNN | TF-IDF | 0.758409 ± 0.034741 | **0.782893 ± 0.031025** | hidden layer = 50 | hidden layer = 200 | 1 | 3 |
| FFNN | Word2Vec | 0.750704 ± 0.025532 | **0.791098 ± 0.034357** | hidden layer = 150 | hidden layer = 200 | 2 | 1 |
| SVM | Word2Vec | 0.738369 ± 0.032936 | **0.790480 ± 0.038238** | kernel = rbf, C = 100 | kernel = rbf, C = 10 | 3 | 2 |
| SVM | TF-IDF | 0.730103 ± 0.030521 | **0.772314 ± 0.038073** | kernel = linear, C = 10 | kernel = linear, C = 10 | 4 | 4 |
| CNB | TF-IDF | 0.692885 ± 0.031471 | **0.747527 ± 0.035903** | alpha = 0.1 | alpha = 0.1 | 5 | 5 |
| KNN | TF-IDF | 0.674414 ± 0.030267 | **0.721247 ± 0.035361** | k = 1 | k = 5 | 6 | 6 |
| KNN | Word2Vec | 0.667139 ± 0.037451 | 0.684065 ± 0.045898 | k = 1 | k = 4 | 7 | 8 |
| GNB | TF-IDF | 0.618875 ± 0.038091 | 0.636232 ± 0.039672 | – | – | 8 | 9 |
| MNB | TF-IDF | 0.616596 ± 0.021059 | **0.713169 ± 0.041369** | alpha = 0.1 | alpha = 0.1 | 9 | 7 |

decrease vocabulary inflation and strengthen the signal available to the model, which aligns with the observed improvement in performance after cleansing.

Overall, these findings indicate that data cleansing is not merely a preliminary step in data science workflows. It influences dataset composition and distribution and can affect ML model performance, stability, interpretability, and comparative benchmarking outcomes.

## 5.5 Recommender System

The overall results for the zero-shot and fine-tuned GOFA across all tasks are presented in Table 5.4. Fine-tuning consistently improves performance compared to the zero-shot setup across all four tasks; however, the gains remain moderate. The NN and NA tasks benefit the most from fine-tuning. A more detailed analysis of individual tasks follows.

Table 5.4: Zero-shot and fine-tuned GOFA results across the four UML completion tasks.

| **Task** | **Metric** | **Zero-shot** | **Fine-tuned** | **Δ (Fine − Zero)** |
|---|---|---|---|---|
| Node Name | MRR@5 | 0.041 | 0.095 | +0.055 |
| | SR@1 | 0.011 | 0.067 | +0.056 |
| | SR@5 | 0.100 | 0.144 | +0.044 |
| Node Attributes | Precision | 0.124 | 0.236 | +0.112 |
| | Recall | 0.111 | 0.217 | +0.106 |
| | F1 | 0.107 | 0.210 | +0.103 |
| Node Operations | Precision | 0.052 | 0.072 | +0.020 |
| | Recall | 0.056 | 0.060 | +0.004 |
| | F1 | 0.051 | 0.061 | +0.011 |
| Edge Type | Accuracy | 0.451 | 0.488 | +0.037 |
| | Macro-F1 | 0.196 | 0.226 | +0.029 |

### 5.5.1 Node Name Task

An exact gold class name appeared more frequently in the recommendations in the fine-tuned setup, with SR@5 increasing from 0.100 to 0.144, and particularly appeared more often in the first position of the recommendation list, with SR@1 increasing from 0.011 to 0.067. These numbers remained very low, however, and it is important to note

that the evaluation setup did not account for synonyms. Exact-match evaluation is further complicated by the fact that class naming is inherently open-ended, as any given concept may have many semantically plausible alternatives. Nevertheless, exact matching provides an objective, reproducible benchmark and avoids introducing subjectivity into the evaluation process.

Manual inspection of predictions and gold labels in the test set revealed several patterns. Despite explicit instructions not to retrieve placeholder names, the zero-shot model showed a strong tendency to output lists containing names such as 'Target', 'Entity', 'Object', 'Node', 'EntityNode', 'MaskedNode', 'AnonymizedTarget'. This suggests that the model was often copying prompt artefacts rather than inferring names from the UML context. In addition, the model frequently generated short generic names such as 'User', 'Customer', 'Employee', 'Person', 'Doctor', 'Product', while compound names present among the gold labels, such as 'UserwithRole', 'ShoppingCart', 'GameOverPanel', were seldom produced.

The fine-tuned model, on the other hand, no longer generated placeholder names, indicating a modest improvement. However, issues remained regarding the length of the generated recommendation lists, with the model sometimes generating only one or two candidates or, conversely, producing lists much longer than the requested five. The fine-tuned model also tended to over-predict a small set of frequent class names, such as 'User', 'Customer', 'Card', among others. Compound names remained a challenge, as the model often predicted the semantic neighborhood but not the specific compound name. An illustrative example is shown in Listing 5.1.

Listing 5.1: Example of a semantically related but not exact node-name prediction.

```
{
  "candidates": ["User", "Product", "Order", "Payment", "Shipping"],
  "gold_raw": "ShippingInfo"
}
```

In some cases, predictions were marked as incorrect due to minor lexical variations, such as shortened forms or singular–plural differences. Two illustrative examples are shown in Listing 5.2.

Listing 5.2: Examples of predictions affected by minor lexical variation.

```
{
  "candidates": ["Coords", "Turn", "Board", "Player", "Strategy"],
  "gold_raw": "Coordinate"
}

{
  "candidates": ["Book", "Author", "Publisher", "Librarian"],
  "gold_raw": "Books"
}
```

### 5.5.2 Node Attributes Task

For the NA task, the F1-score roughly doubled, increasing from 0.107 to 0.210, which indicates a meaningful improvement. The evaluation setup remained restrictive: although it tolerated minor spelling errors, synonyms were not considered correct matches.

It is also important to note that the predicted and gold attribute sets did not necessarily have the same cardinality. The model had no prior knowledge of how many attributes were present in the gold case. Consequently, even when additional attributes predicted by the model were semantically plausible, they were treated as incorrect by the evaluation procedure.

Examination of predictions revealed that the zero-shot model frequently generated placeholder attribute names in a format such as 'name: string' or 'name: integer', often with optional numeric suffixes. Among the 20 full matches and 10 partial matches recorded for the zero-shot setup, several appear to be accidental: 9 of the 20 full matches and 6 of the 10 partial matches resulted from matching the frequent gold attribute 'name' with placeholder variants such as 'name1'.

Fine-tuning substantially reduced the occurrence of these placeholder predictions. In a fine-tuned setup, 44 full matches and 14 partial matches were observed. Typical errors involved predicting the correct name but an incorrect type (e.g., predicting 'string' instead of 'integer' for an identifier, or vice versa). Another common pattern was that the model predicted too few attributes, often generating only one attribute per sample, which negatively affected recall. In other cases, the model generated semantically reasonable attributes that were not present in the gold set, which reduced precision despite the predictions being plausible.

In the example shown in Listing 5.3, `amount` and `paymenttotal` could reasonably be considered semantically related, but the evaluation scheme did not account for such relationships.

Listing 5.3: Example of a semantically related but non-matching attribute prediction.

```
{
  "pred_pairs": [["paymentid", "string"], ["customerid", "string"], ["
  amount", "double"]],
  "gold_pairs": [["paymentid", "integer"], ["paymenttotal", "float"]]
}
```

### 5.5.3 Node Operations Task

Many of the observations for the NA task also apply to the NO tasks, although evaluation is even more challenging because operations are longer strings than attributes and exhibit greater variability in parameters, return types, and formatting. Hence, the relatively small improvement observed in this task is not overly surprising.

In the zero-shot setup, the model often generated placeholder operations such as 'op1' or 'op2'. This pattern disappeared in the fine-tuned setup. However, a common problem

was once again that the model would generate fewer operations than were present in the gold labels. The fine-tuned model often predicted the correct operation name but failed to reproduce the full signature. An example is shown in Listing 5.4.

Listing 5.4: Example of a partially correct operation prediction with an incomplete signature.

```
{
  "pred_ops": ["gameoverpanel() -> gameoverpanel"],
  "gold_ops": ["gameoverpanel(gui: connect4gui, winner: string)"]
}
```

The issue of synonyms and shorthand forms was also common. For instance, `requirements` appeared as `req` in the predictions but as `requi` in the gold label, and `remove` was not recognized as equivalent to `delete`. This is illustrated in Listing 5.5.

Listing 5.5: Example of operation prediction affected by shorthand forms and lexical variation.

```
{
  "pred_ops": ["get_req()", "update_req()", "delete_req()"],
  "gold_ops": ["add_requirements()", "remove_requirements()", "
  update_requi()"]
}
```

### 5.5.4 Edge Type Task

For the ET task, fine-tuning slightly improves both the accuracy and macro-F1. However, the substantially lower macro-F1 compared to the accuracy in both cases indicates that class imbalance remains a significant issue, despite the sampling scheme designed to increase the representation of rare relationship types.

The per-class precision, recall, and F1-scores for both the zero-shot and fine-tuned setups, shown in Table 5.5, reveal that the model struggled with minority relationship types as expected. It is important to note that although the realization type was present in the training set, it did not appear at all in the test set. This occurred because the dataset was split randomly at the diagram level before sample formation, resulting in no test diagrams containing this type of relationship. As a result, the reported macro-F1 score is somewhat affected.

The dependency relation type, on the other hand, was present in the test set yet still exhibits zero performance, indicating that the model failed to learn a distinguishable signal for this relation.

The observed improvement in macro-F1 for the ET task primarily stems from better performance on the generalization relationship type rather than improvements in the majority association class. Specifically, recall for generalization increased, while precision improved only slightly. This suggests that the model predicted generalization more often, but many of these predictions are false positives.

Table 5.5: Per-class precision, recall, and F1-scores for the ET task in the zero-shot and fine-tuned settings.

| | Zero-shot | | | Fine-tuned | | |
|---|---|---|---|---|---|---|
| **Edge type** | **Precision** | **Recall** | **F1** | **Precision** | **Recall** | **F1** |
| Association | 0.733 | 0.524 | 0.611 | 0.767 | 0.524 | 0.623 |
| Dependency | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| Generalization | 0.121 | 0.308 | 0.174 | 0.189 | 0.538 | 0.280 |
| Realization | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

## 5.6 Answer to Research Question Two - Summary

Overall, fine-tuning GOFA improved completion performance across all four tasks compared to the zero-shot setup, although the magnitude of the improvement remained moderate. The largest gains were observed in the NN and NA tasks, where the target text follows a relatively stable lexical pattern. The NO task proved more challenging due to the greater variability of the output structure. The ET task also showed only modest improvement, with the model continuing to struggle with class imbalance.

Manual inspection indicated that fine-tuning was largely successful in reducing placeholder outputs and improving compliance with the expected task format; however, the model still struggled with UML-specific completion. This may be partially explained by the heterogeneity of the four tasks and the limited amount of data available for each of them. The results are also consistent with the reports in [31], namely that the primary outcome of fine-tuning GOFA is task learning rather than the acquisition of domain-specific knowledge from the dataset.

Thus, fine-tuning improved GOFA's performance for UML class diagram completion, but the gains suggest that adaptation to the task format was stronger than the adaptation to the UML domain itself.

# CHAPTER 6

# Discussion

## 6.1 Contribution

Recent advances in machine learning have led to increasing interest in automating modeling tasks within the MDE domain. Among these, model completion has emerged as a particularly promising direction, with the potential to significantly improve both the efficiency and quality of software modeling.

Despite the progress achieved so far, AI-enhanced MDE remains a relatively young field constrained by a fundamental limitation: the lack of large-scale, high-quality datasets suitable for training and evaluating machine learning models. In practice, researchers rely on datasets mined from public repositories, which are often affected by substantial noise, including duplicated models, trivial examples, and inconsistencies arising from multilingual content. While prior work has acknowledged these issues, existing solutions either address individual problems in isolation or rely on computationally expensive techniques that do not scale well to large datasets. Furthermore, data preprocessing practices are frequently underreported, resulting in a lack of standardization across studies.

In parallel, the representation of conceptual models poses an additional challenge. Existing research typically relies on either textual or structural representations, thereby neglecting the inherently multimodal nature of conceptual models, where meaning arises from the interaction between semantics and structure. This gap suggests the need for approaches that jointly leverage both modalities.

This study addresses these challenges through a data-centric perspective. First, we introduce a comprehensive and scalable data cleansing pipeline tailored to UML datasets, combining custom heuristic filtering for trivial models, lightweight hash- and similarity-based clone detection, and language filtering. The impact of these preprocessing steps is systematically evaluated through two reproducibility studies, which demonstrate that

dataset quality substantially affects both ML model performance and the validity of experimental conclusions.

In particular, the results show that ML models trained on uncleansed data may achieve seemingly strong performance that does not generalize once the duplicates and noise are removed. In addition, the interpretability of ML models does not transfer consistently to the cleansed setting. Furthermore, changes in data distribution introduced by cleansing were shown to influence benchmarking outcomes, emphasizing that preprocessing is not merely a preliminary step but a critical component of the experimental pipeline.

Building on the cleansed dataset, we then investigate the potential of a hybrid architecture for UML class diagram completion. Specifically, we fine-tune a GNN-LLM model to jointly capture structural and semantic information and evaluate its performance across four completion tasks, including the recommendation of class names, attributes, operations, and relationship types.

While fine-tuning yields consistent improvements over a zero-shot baseline, particularly for class name and attribute prediction, the overall gains remain moderate. In general, this is partially explained by the fact that semantically plausible but non-matching outputs are penalized by the strict evaluation schema. A closer analysis of the results reveals several important patterns. Improvements are primarily related to better adherence to task-specific output formats, rather than to a robust attainment of UML-specific domain knowledge. Tasks with relatively stable lexical structures see the biggest gains, whereas tasks involving higher variability, such as operation prediction, remain challenging.

At the same time, the model shows a tendency to over-predict generic elements and struggles with more specific and compound concepts, pointing to limitations in both the training data and the learning capacity of the current setup. These findings suggest that, while the chosen hybrid architecture may be capable of leveraging multimodal information to some extent, it remains insufficient to fully capture the complexity of conceptual modeling tasks.

Overall, this work highlights two key insights. First, data quality has a decisive impact on the validity and interpretability of machine learning conclusions in MDE and should be treated as a central concern in future research. Second, while hybrid approaches present a promising direction, further advances are needed to enable the learning of domain-specific knowledge. This includes both architectural improvements and the availability of larger, high-quality datasets suited to the demands of modern machine learning architectures.

## 6.2 Limitations and Threats to Validity

While the findings of this study provide useful insights, several limitations should be considered when interpreting the results.

First, the analysis of the dataset quality is specific to the ModelSet dataset and may not generalize to other conceptual model repositories, particularly industrial datasets. The

proposed cleansing pipeline was designed and validated primarily on ModelSet and may therefore not fully capture the characteristics of real-world modeling environments. In addition, the selected thresholds used to identify dummy models, duplicates, and other forms of noise have a direct impact on the reported statistics.

The revised dataset used in the reproducibility studies relies on automatic filtering procedures, which may introduce both false positives and false negatives. As a result, valid models may have been removed, while some noisy or duplicate models may remain. In particular, near-duplicates may not be fully eliminated, meaning that the risk of data leakage cannot be entirely ruled out.

Furthermore, the evaluation of the cleansing pipeline considers the combined effect of multiple preprocessing steps. The individual contribution of each component (i.e., dummy model detection, clone detection, and language filtering) is not assessed in isolation, making it difficult to attribute observed performance changes to specific steps.

Regarding statistical evaluation, multiple hypothesis tests were conducted across different tasks, classifiers, and evaluation metrics. This increases the likelihood of obtaining at least one significant result by chance. Although such risks can be mitigated through multiple comparison correction, no such correction was applied in this study, and therefore, the statistical results should be interpreted with caution.

In addition, the proposed auto-completion model is restricted to UML class diagrams, while other diagram types have not been explored. The procedure used to identify and extract class diagrams from ModelSet may not be entirely accurate, particularly for models containing mixed diagram types. Consequently, some relevant information may have been inadvertently removed or retained. Similarly, the transformation of class diagrams into graph representations may result in a loss of information.

The adopted architecture operates on $k$-hop subgraphs, which limits the available context and may prevent the model from capturing global structures and long-range dependencies. Moreover, conceptual modeling tasks often admit multiple valid solutions; however, the evaluation metrics used in this study rely on exact matching and therefore may underestimate the quality of semantically correct predictions. In addition, LLMs are inherently non-deterministic, which may introduce variability in the results. Finally, the size of the fine-tuning dataset is relatively small compared to the capacity of the model, increasing the risk of overfitting and limiting the model's ability to generalize beyond the observed data.

## 6.3 Future Work

Building on the findings and limitations of this study, several directions for future work emerge.

First, the proposed data cleansing pipeline could be extended with more advanced techniques for clone detection, dummy model identification, and language filtering. While

this work adopts lightweight and scalable approaches, future work may explore the integration of clustering-based methods for duplicate detection, or the use of lightweight ML models for detection of trivial models, provided that computational efficiency can be maintained. Furthermore, the current pipeline is tailored to UML diagrams; however, it could be generalized to support other modeling languages such as BPMN, ArchiMate, or Ecore, thereby enabling broader applicability in MDE domain.

Future work could also investigate extending the recommender system to other UML diagram types. Moreover, one of the main insights of this study was that further research is needed to improve the ability of hybrid architectures to capture domain-specific knowledge. This may involve exploring alternative architectures, using larger training datasets better aligned with conceptual modeling tasks, or designing training objectives that are more suitable for recommendation settings. Finally, while this study relies on zero-shot performance as a baseline, future work should aim to compare the proposed approach against existing state-of-the-art methods.

# CHAPTER 7

# Conclusion

In this thesis, an auto-completion approach for UML class diagrams was investigated from both a data-centric and an architectural perspective. First, common quality issues present in conceptual model datasets mined from public repositories were identified, including duplicates, trivial models, and multilingual noise. To address these issues, a UML specific data cleansing pipeline was developed, combining heuristics for detecting trivial models, a lightweight approach for identifying potential clones, and a language filtering step aimed at reducing noise introduced by multilingual model element names.

The effect of these preprocessing steps on downstream machine learning tasks was evaluated through two reproducibility studies. The results showed substantial changes in model performance, interpretability, and benchmarking outcomes before and after the application of the proposed pipeline. These findings highlight the importance of further research on the preparation and standardization of UML datasets for machine learning applications.

The proposed preprocessing framework was then used to support the fine-tuning of a hybrid GNN -LLM architecture capable of reasoning jointly over structural and semantic information, two modalities of conceptual models that are often explored separately in existing research. The fine-tuned model showed consistent improvements over the zero-shot baseline across the tasks of recommending class names, attributes, operations, and relationship types. At the same time, the results indicate that current gains are still moderate and that important challenges remain, particularly with respect to domain-specific knowledge acquisition and the availability of large, high-quality datasets.

Overall, this study shows that progress in ML-assisted conceptual modeling is contingent not only on advances in model architectures, but equally on the quality of the datasets used to train and evaluate them.

# Overview of Generative AI Tools Used

The AI-assisted tools used in this study include Grammarly, DeepL Write, and ChatGPT variants. These tools were employed exclusively for grammar and style checking to improve clarity and ensure an appropriate academic writing standard.

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**AI** Artificial Intelligence. 10, 57

**BPMN** Business Process Model and Notation. 6, 7, 18

**ET** Edge Type. 29, 33, 35, 36, 39, 54, 55, 67

**FFNN** Feed-Forward Neural Network. 26, 49

**GNN** Graph Neural Network. ix, xi, 3, 5, 10–12, 29, 38, 41, 58, 61

**GOFA** Generative One For All. 10, 11, 29, 33, 35–38, 51, 55, 67

**GPU** Graphics Processing Unit. 38

**JSON** JavaScript Object Notation. 10, 30, 32–34, 65, 69

**JSONL** JSON Lines. 34, 35, 38, 65

**KNN** K-Nearest Neighbors. 23, 25, 26, 46, 48, 65

**LLM** Large Language Model. ix, xi, 3, 5, 9–12, 29, 38, 41, 58, 59, 61

**LoRA** Low-Rank Adaptation. 38

**MCP4CM** Model Cleansing Pipeline for Conceptual Models. 20, 45, 47, 48, 50, 67

**MDE** Model-Driven Engineering. ix, xi, xiv, 1–3, 5–9, 37, 57, 58, 60

**ML** Machine Learning. ix, xi, 1–3, 6, 10, 21, 23–27, 33, 42, 49, 51, 58, 60, 61, 65

**NA** Node Attributes. 29, 33, 35, 36, 39, 51, 53, 55

**NLP** Natural Language Processing. 7

**NN** Node Name. 29, 33, 35, 36, 38, 51, 55

**NO** Node Operations. 29, 33, 35, 36, 39, 53, 55

**NOG** Node of Generation. 10–12, 35

**PFI** Permutation Feature Importance. 23, 46, 48

**RF** Random Forest. 23, 45, 46, 48, 65

**RS1** Reproducibility Study One. 23–26, 46–48, 65, 67

**RS2** Reproducibility Study Two. 25–27, 50, 65, 67

**SHA** Secure Hash Algorithm. 18, 37, 41

**SVC** Support Vector Classification. 23, 46, 48, 65

**SVM** Support Vector Machine. 25, 26

**TAG** Text-Attributed Graph. 10, 33

**TF-IDF** Term Frequency-Inverse Document Frequency. 18, 20, 25, 26, 41

**UML** Unified Modeling Language. ix, xi, 1–3, 5–10, 12, 17–21, 23, 26, 29–33, 36, 37, 41, 42, 45, 51, 52, 55, 58–61, 65, 67, 69

**XAI** Explainable Artificial Intelligence. 23

**XMI** XML Metadata Interchange. 2, 5, 17, 18, 29–31, 65

**XML** Extensible Markup Language. 31

# Bibliography


[1] M. Brambilla, J. Cabot, and M. Wimmer, Model-Driven Software Engineering in Practice. in Synthesis Lectures on Software Engineering. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-031-02549-5.

[2] "UML @ Classroom: An Introduction to Object-Oriented Modeling | Springer Nature Link." Accessed: Mar. 10, 2026. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-12742-2

[3] P. Pourali and J. M. Atlee, "An Empirical Investigation to Understand the Difficulties and Challenges of Software Modellers When Using Modelling Tools," in Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, in MODELS '18. New York, NY, USA: Association for Computing Machinery, Oktober 2018, pp. 224–234. doi: 10.1145/3239372.3239400.

[4] A. C. Marcén, A. Iglesias, R. Lapeña, F. Pérez, and C. Cetina, "A Systematic Literature Review of Model-Driven Engineering Using Machine Learning," IEEE Transactions on Software Engineering, vol. 50, no. 9, pp. 2269–2293, Sep. 2024, doi: 10.1109/TSE.2024.3430514.

[5] L. Almonte, E. Guerra, I. Cantador, and J. de Lara, "Recommender systems in model-driven engineering," Softw Syst Model, vol. 21, no. 1, pp. 249–280, Feb. 2022, doi: 10.1007/s10270-021-00905-x.

[6] G. Robles, T. Ho-Quang, R. Hebig, M. R. V. Chaudron, and M. A. Fernandez, "An Extensive Dataset of UML Models in GitHub," in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), May 2017, pp. 519–522. doi: 10.1109/MSR.2017.48.

[7] Ö. Babur, "A labeled Ecore metamodel dataset for domain clustering." Zenodo, Mar. 06, 2019. Accessed: Jun. 09, 2025. [Online]. Available: https://zenodo.org/records/2585456

[8] J. A. H. López and J. S. Cuadrado, "An efficient and scalable search engine for models," Softw Syst Model, vol. 21, no. 5, pp. 1715–1737, Oct. 2022, doi: 10.1007/s10270-021-00960-4.

[9] J. A. H. López, J. L. Cánovas Izquierdo, and J. S. Cuadrado, "ModelSet: a dataset for machine learning in model-driven engineering," Softw Syst Model, vol. 21, no. 3, pp. 967–986, Jun. 2022, doi: 10.1007/s10270-021-00929-3.

[10] A. Delić, S. J. Ali, C. Verbruggen, J. Neidhardt, and D. Bork, "A Model Cleansing Pipeline for Model-Driven Engineering: Mitigating the Garbage In, Garbage Out Problem for Open Model Repositories," in 2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS), Oct. 2025, pp. 60–71. doi: 10.1109/MODELS67397.2025.00012.

[11] M. Saeedi Nikoo, S. Kochanthara, Ö. Babur, and M. van den Brand, "An empirical study of business process models and model clones on GitHub," Empir Software Eng, vol. 30, no. 2, p. 48, Dec. 2024, doi: 10.1007/s10664-024-10584-z.

[12] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model clone detection in practice," in Proceedings of the 4th International Workshop on Software Clones, in IWSC '10. New York, NY, USA: Association for Computing Machinery, Mai 2010, pp. 57–64. doi: 10.1145/1808901.1808909.

[13] H. Störrle, "Towards clone detection in UML domain models," Softw Syst Model, vol. 12, no. 2, pp. 307–329, May 2013, doi: 10.1007/s10270-011-0217-9.

[14] Ö. Babur, L. Cleophas, and M. van den Brand, "Metamodel clone detection with SAMOS," Journal of Computer Languages, vol. 51, pp. 57–74, Apr. 2019, doi: 10.1016/j.cola.2018.12.002.

[15] M. Saeedi Nikoo, Ö. Babur, and M. van den Brand, "Clone detection for business process models," PeerJ Comput Sci, vol. 8, p. e1046, Aug. 2022, doi: 10.7717/peerj-cs.1046.

[16] H. Störrle, "Effective and Efficient Model Clone Detection," in Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, R. De Nicola and R. Hennicker, Eds., Cham: Springer International Publishing, 2015, pp. 440–457. doi: 10.1007/978-3-319-15545-6_25.

[17] J. A. H. López, R. Rubei, J. S. Cuadrado, and D. di Ruscio, "Machine learning methods for model classification: a comparative study," in Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, in MODELS '22. New York, NY, USA: Association for Computing Machinery, Oktober 2022, pp. 165–175. doi: 10.1145/3550355.3552461.

[18] Y.-T. Lin et al., "Selective In-Context Data Augmentation for Intent Detection using Pointwise V-Information," Feb. 10, 2023, arXiv: arXiv:2302.05096. doi: 10.48550/arXiv.2302.05096.

[19] E. A. Marand, A. Sheikhahmadi, M. Challenger, P. Moradi, and A. Khalilipour, "Recommender Systems for Unified Modeling Language and Vice Versa—A Systematic Literature Review," IEEE Access, vol. 13, pp. 23426–23460, 2025, doi: 10.1109/ACCESS.2025.3535527.

[20] J. D. Rocco, D. D. Ruscio, C. D. Sipio, P. T. Nguyen, and R. Rubei, "On the use of Large Language Models in Model-Driven Engineering," Oct. 22, 2024, arXiv: arXiv:2410.17370. doi: 10.48550/arXiv.2410.17370.

[21] L. Burgueño, D. Di Ruscio, H. Sahraoui, and M. Wimmer, "Automation in Model-Driven Engineering: A look back, and ahead," ACM Trans. Softw. Eng. Methodol., p. 3712008, Jan. 2025, doi: 10.1145/3712008.

[22] C. D. Costa, J. A. H. López, and J. S. Cuadrado, "ModelMate: A recommender for textual modeling languages based on pre-trained language models," in Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, Linz Austria: ACM, Sep. 2024, pp. 183–194. doi: 10.1145/3640310.3674089.

[23] M. Weyssow, H. Sahraoui, and E. Syriani, "Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models," Feb. 21, 2022, arXiv: arXiv:2104.01642. Accessed: Nov. 21, 2024. [Online]. Available: `http://arxiv.org/abs/2104.01642`

[24] Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach," Jul. 26, 2019, arXiv: arXiv:1907.11692. doi: 10.48550/arXiv.1907.11692.

[25] M. B. Chaaben, L. Burgueño, and H. Sahraoui, "Towards using Few-Shot Prompt Learning for Automating Model Completion," in 2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), May 2023, pp. 7–12. doi: 10.1109/ICSE-NIER58687.2023.00008.

[26] M. B. Chaaben, L. Burgueño, I. David, and H. Sahraoui, "On the Utility of Domain Modeling Assistance with Large Language Models," Oct. 16, 2024, arXiv: arXiv:2410.12577. doi: 10.48550/arXiv.2410.12577.

[27] C. Tinnes, A. Welter, and S. Apel, "Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets," Dec. 10, 2024, arXiv: arXiv:2406.17651. doi: 10.48550/arXiv.2406.17651.

[28] J. Di Rocco, C. Di Sipio, D. Di Ruscio, and P. T. Nguyen, "A GNN-based Recommender System to Assist the Specification of Metamodels and Models," in 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), Oct. 2021, pp. 70–81. doi: 10.1109/MODELS50736.2021.00016.

[29] C. Di Sipio, J. Di Rocco, D. Di Ruscio, and P. T. Nguyen, "MORGAN: a modeling recommender system based on graph kernel," Softw Syst Model, vol. 22, no. 5, pp. 1427–1449, Oct. 2023, doi: 10.1007/s10270-023-01102-8.

[30] Z. Wang, C. Zhang, J. Li, N. Chawla, and Y. Ye, "Graph Foundation Models: Challenges, Methods, and Open Questions," in Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2, in KDD '25. New York, NY, USA: Association for Computing Machinery, Aug. 2025, pp. 6184–6194. doi: 10.1145/3711896.3736568.

[31] L. Kong et al., "GOFA: A GENERATIVE ONE-FOR-ALL MODEL FOR JOINT GRAPH LANGUAGE MODELING," in 13th International Conference on Learning Representations, ICLR 2025, International Conference on Learning Representations, ICLR, 2025, pp. 58209–58240. Accessed: Mar. 06, 2026. [Online]. Available: https://profiles.wustl.edu/en/publications/gofa-a-generative-one-for-all-model-for-joint-graph-language-mode/

[32] H. Liu et al., "One for All: Towards Training One Graph Model for All Classification Tasks," Jul. 12, 2024, arXiv: arXiv:2310.00149. doi: 10.48550/arXiv.2310.00149.

[33] T. Ge, J. Hu, L. Wang, X. Wang, S.-Q. Chen, and F. Wei, "In-context Autoencoder for Context Compression in a Large Language Model," May 08, 2024, arXiv: arXiv:2307.06945. doi: 10.48550/arXiv.2307.06945.

[34] Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun, "Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification," May 10, 2021, arXiv: arXiv:2009.03509. doi: 10.48550/arXiv.2009.03509.

[35] langdetect: Language detection library ported from Google's language-detection. Python. Accessed: Mar. 17, 2026. [OS Independent]. Available: https://github.com/Mimino666/langdetect

[36] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," Biometrika, vol. 52, no. 3–4, pp. 591–611, Dec. 1965, doi: 10.1093/biomet/52.3-4.591.

[37] B. L. WELCH, "THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED," Biometrika, vol. 34, no. 1–2, pp. 28–35, Jan. 1947, doi: 10.1093/biomet/34.1-2.28.

[38] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," The Annals of Mathematical Statistics, vol. 18, no. 1, pp. 50–60, Mar. 1947, doi: 10.1214/aoms/1177730491.

[39] F. J. Alcaide, J. R. Romero, and A. Ramírez, "Can explainable artificial intelligence support software modelers in model comprehension?," Softw Syst Model, Jan. 2025, doi: 10.1007/s10270-024-01251-4.

[40] L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, Oct. 2001, doi: 10.1023/A:1010933404324.

[41] A. Fisher, C. Rudin, and F. Dominici, “All Models are Wrong, but Many are Useful: Learning a Variable’s Importance by Studying an Entire Class of Prediction Models Simultaneously,” J Mach Learn Res, vol. 20, p. 177, 2019.

[42] M. T. Ribeiro, S. Singh, and C. Guestrin, “‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier,” Aug. 09, 2016, arXiv: arXiv:1602.04938. doi: 10.48550/arXiv.1602.04938.

[43] S. Lundberg and S.-I. Lee, “A Unified Approach to Interpreting Model Predictions,” Nov. 25, 2017, arXiv: arXiv:1705.07874. doi: 10.48550/arXiv.1705.07874.

[44] M. Staniak and P. Biecek, “Explanations of model predictions with live and breakDown packages,” The R Journal, vol. 10, no. 2, p. 395, 2019, doi: 10.32614/RJ-2018-072.

[45] I. Mollas, N. Bassiliades, and G. Tsoumakas, “Conclusive local interpretation rules for random forests,” Data Min Knowl Disc, vol. 36, no. 4, pp. 1521–1574, Jul. 2022, doi: 10.1007/s10618-022-00839-y.

[46] J. A. H. López and J. S. Cuadrado, “MAR: A structure-based search engine for models,” Aug. 26, 2020, arXiv: arXiv:2008.11858. doi: 10.48550/arXiv.2008.11858.

[47] P. T. Nguyen, D. Di Ruscio, A. Pierantonio, J. Di Rocco, and L. Iovino, “Convolutional neural networks for enhanced classification mechanisms of metamodels,” Journal of Systems and Software, vol. 172, p. 110860, Feb. 2021, doi: 10.1016/j.jss.2020.110860.

[48] R. Rubei, J. D. Rocco, D. D. Ruscio, P. T. Nguyen, and A. Pierantonio, “A Lightweight Approach for the Automated Classification and Clustering of Metamodels,” in 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Oct. 2021, pp. 477–482. doi: 10.1109/MODELS-C53483.2021.00074.

[49] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino, “Automated Classification of Metamodel Repositories: A Machine Learning Approach,” in 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), Sep. 2019, pp. 272–282. doi: 10.1109/MODELS.2019.00011.

[50] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, in Onward! 2019. New York, NY, USA: Association for Computing Machinery, Oktober 2019, pp. 143–153. doi: 10.1145/3359591.3359735.

[51] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in Proceedings of the 33rd International Conference on Software Engineering, in ICSE ’11. New York, NY, USA: Association for Computing Machinery, Mai 2011, pp. 1–10. doi: 10.1145/1985793.1985795.

[52] J. Feng, H. Liu, L. Kong, M. Zhu, Y. Chen, and M. Zhang, “TAGLAS: An atlas of text-attributed graph datasets in the era of large graph and language models,” Oct. 19, 2024, arXiv: arXiv:2406.14683. doi: 10.48550/arXiv.2406.14683.

[53] E. J. Hu et al., “LoRA: Low-Rank Adaptation of Large Language Models,” Oct. 16, 2021, arXiv: arXiv:2106.09685. doi: 10.48550/arXiv.2106.09685.

[54] RapidFuzz: rapid fuzzy string matching. Python. Accessed: Mar. 17, 2026. [Online]. Available: `https://github.com/rapidfuzz/RapidFuzz`