Master's Thesis

TOWARDS MODEL EDITING PATTERN DETECTION VIA GRAPH VARIATIONAL AUTOENCODERS

ALISA WELTER

September 13, 2023

Advisor:
Christof Tinnes Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Jilles Vreeken Exploratory Data Analysis

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 13.09.2023 A. Weiter

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Model Transformations are a primary artifact in Model Driven Engineering that ensure the consistency between models by automating model changes. But their usage is not limited to model evolution. They further can be used for model creation, model merging, model repair, semantic lifting and auto-completion. Previously, several methods have been suggested for automating the specification of model transformations, for example, from meta-models or labeled examples. These approaches are limited to rather simple transformations or require time consuming pre- or post-processing steps. We would like to overcome these limitations by proposing a different, unsupervised machine learning approach that learns model transformations from model histories, which can be derived from repositories. We propose to use a Graph Variational Autoencoder (GVAE), which combines graph theory and variational inference to generate new graph structures that should be preferably similar to the input data. We provide a simplified version of model differences as input to this framework, aiming to generate new realistic edit operations. To the best of our knowledge, this work is the first of its kind and serves as a proof of concept. The thesis conducts an initial investigation into whether GVAE can be applied to the specialized task of discovering new/unseen edit operations. We evaluate the method across various experiments targeting specific architectural decisions in machine learning. Our findings confirm that basic language concepts can indeed be learned realistically. Additionally, there are indications that the GVAE can identify previously applied edit operations. In summary, our preliminary results suggest that using GVAE for deriving model transformations is worth further investigation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

Acronyms

GVAE Graph Variational Autoencoder

VAE Variational Autoencoder

GNN Graph neural network

GAT Graph Attention

TRF Transformer

GIN Graph Isomorphism

GAT-A Graph Attention Layer combined with Augmented Input Graph Setting

GAT-U Graph Attention Layer combined with Unchanged Input Graph Setting

GIN-A Graph Isomorphism Encoder Layer combined with Augmented Input Graph Setting

GIN-U Graph Isomorphism Encoder Layer combined with Unchanged Input Graph Setting
TRF-A Transformer Layer Combined with Augmented Input Graph Setting
TRF-U Transformer Layer Combined with Unchanged Input Graph Setting
SCG Simple Change Graph
MTBE Model Transformation by Example
TP True Positives
FP False Positives
TN True Negatives
FN False Negatives
MDA Model-Driven Architecture
MDE Model-Driven Engineering

1 INTRODUCTION

1.1 GOAL OF THIS THESIS

In recent years, software and software development has become increasingly complex. Often so-called Model-Driven Engineering (MDE) is therefore used to efficiently create software products [18]. MDE utilizes models as key documentation artifacts, which should provide a common understanding of the architecture and the design of the software system, such that certain design choices can be discussed more easily [18]. This not only increases the efficiency and success of the software development process but can also help in achieving certain non-functional requirements, for example, maintainability and traceability [14]. In addition, it also opens the possibility to automatically create or execute the software systems based on these models, which is known as Model-Driven Architecture (MDA) [18, 46]. This approach can also be used to overcome limitations of the underlying programming languages, which, for example, lack features like classes and inheritance, making the code quickly become unmaintainable. Partial code generation improves the maintainability of such systems [12]. During the life cycle of the software systems, these models need to be modified regularly. Main ingredients in MDE are therefore model transformations [10], specifically, in our case, edit operations, which ensure the consistency between models and the software product by automating model changes and updates [56]. But the usage of edit operations is not only limited to the evolution of models but they can also be used for model repair [62], semantic lifting [26, 42] and auto-completion [47].
However, the creation of these edit operations is a non-trivial task. Meta models usually do not contain all domain-specific knowledge needed to construct these model transformations, such that often understanding of implicit concepts and the semantics are required. Thus, expert knowledge or even some kind of tacit knowledge of the language's meta model and the underlying paradigm of the transformation language is needed to construct model transformations [7, 16, 73]. As a consequence, crafting these model transformations, particularly in non-trivial scenarios and when domain-specific languages come into play, becomes a challenging and error-prone task, contributing to significant overhead for numerous projects [16, 41, 57, 73].
To extend the generation of model transformations to more complex and domain-specific tasks, several approaches have been proposed [16, 41]. However, the vast majority of studies concentrate on supervised approaches. These often necessitate manual pre-processing steps where experts are required to provide transformation examples, which is error-susceptible and labor-intensive, resulting in a limited amount of available data [73]. This restriction notably undermines the practical use of supervised machine learning techniques. As a consequence, supervised approaches are mostly restricted to mining edit operations that were seen in the labeled data and are limited to rather simple cases. Unsupervised methods do not require any special labeling. Edit operations are extracted out of a given meta model [43] or extracted from model histories [73]. While meta models do not contain

enough information to extract more sophisticated edit operations, meanwhile, other strategies excel on smaller graphs but face scalability challenges on larger ones, making them suboptimal [73]. In general, these methods struggle with achieving generalizability, their limitations becoming even more pronounced when domain-specific languages are used [7, 16, 57, 73]. As far as our knowledge extends, the academic literature has yet to present a comprehensive and universally effective solution. This highlights the ongoing disconnect between academic research and practical needs, as no approach has yet proven to be highly effective in real-world applications.

Intending to surmount these limitations, we propose a new approach. Instead, this master thesis utilizes a machine learning model as a step towards mining edit operations in an unsupervised manner. We investigate to what extend GVAE are in general suitable to detect edit operations and discover yet unseen, new edit operations which can be useful to domain experts [73]. This approach focuses on edit operations as an important class of model transformations and is inspired by the fact that models can ideally be visualized and mathematically formulated by graphs. As we will see, edit operations can be seen as recurring patterns between two different model versions [73]. Our work is also inspired by the recent progress in learning and generating graph structures, for example molecules, and by the fact that Variational Autoencoders are well known for their ideal abstraction and compression capabilities [32], which we hope we can benefit from. Further, using this approach circumvents the need for data labeling and extensive pre-processing, such that more data can be made available, for example, by crawling data from software repositories.

1.2 OVERVIEW

Starting with Chapter 2, the thesis offers essential background information, crucial for understanding the subsequent chapters. The chapter delves into the process of how edit operations can be modeled as graphs in general and how the differences between subsequent models are fed into the GVAE. A foundational overview of the machine learning context is also provided, with a particular focus on GVAEs. Moving on, Chapter 3 delves into the current state of research concerning mining model transformations and their limitations. Further, it provides a review of related works on GVAE, which assisted us in determining an appropriate framework for our application. The rationale and reasoning behind our choice of GVAE is elaborated in Chapter 4. Further, Chapter 4 provides detailed information on our implementation. In Chapter 5, our primary research questions are introduced, followed by a description of the datasets employed for both training and the subsequent evaluation of our approach. Detailed insights into the specific experiments conducted to answer the research questions are provided, further specifying our exact implementation. Chapter 5 also introduces our choice of metrics. Chapter 6 culminates with an analysis and interpretation of our experimental results. Limitations along with our future research directions are outlined in Chapter 7. Lastly, Chapter 8 serves as the capstone of this thesis, providing a concise summary of our findings and reflections on the research journey.

2 BACKGROUND

In the context of this thesis, which both addresses machine learning and software engineering, we encounter two distinct usages of the term "model". To ensure clarity and readability throughout this document, we adopt the following convention. When referring to machine learning models, we will explicitly state so. This disambiguation aims to facilitate a clearer understanding of the respective domains and their overlap within the scope of this thesis.

Definition 1 ***Software Models*** *are abstractions of a software system and its environment. Software models are ways of expressing a software design, including, for example, Entities, Attributes, and Interfaces.*

Definition 2 ***Machine learning models*** *are programs that learn from data to make predictions or decisions for previously unseen data [58].*

2.1 MODEL TRANSFORMATIONS

During the lifecycle of a software system, software models need to be adjusted on a regular basis. Model transformations are fundamental to this process, serving as an automated means to construct and transform models. They address key tasks including model creation, co-evolution management, and model merging within the realm of model-driven software development [66].

Definition 3 ***Model transformations*** *automate changes between models, which take one or more source models as input and produce one or more target models as output, following a set of transformation rules [66].*

As previously mentioned, the process of crafting these model transformations is not a straightforward task. To shed a light on the intricacies involved in creating such model transformations, one can refer to the illustrative example provided by Mokaddem et al. to get the basic concept [57]. Model transformations can either be endogenous or exogenous. Endogenous transformations are transformations between models expressed in the same language and exogenous transformations are transformations between models expressed using different languages and different meta models [56]. In literature, edit operations are specified as endogenous, in-place model transformations [56, 73]. Edit operations include model refactorings, recurring bug fixes, and evolutionary changes [73]. They can include very large model changes, for example, setting up the entire model architecture and very small ones where only one item is changed. We are not interested in either of these cases, but rather interested in the most useful ones (Definition 14).

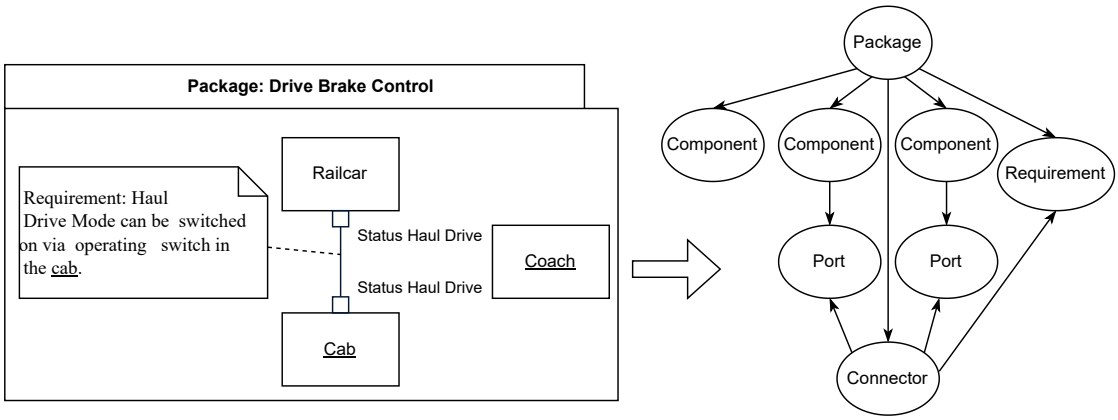

Figure 2.1: Example of deriving an Abstract Syntax Graph from a concrete model, example is from Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining by Tinnes et al. [73]

2.1.1 *Modelling of Edit Operations and Model States as Graphs*

Models can be ideally represented by graphs [10, 25]. In the first step, Abstract Syntax Graphs are created according to the types given in its meta model. A meta model $\mathcal{M}$ defines the abstract syntax and static semantics of the language in use. A model is, therefore, an instance of its meta model. In this thesis, we assume that the models are already correctly typed.

Definition 4 *An **Abstract Syntax Graph** of a model m is a typed attributed graph, typed over an attributed type graph that is given by the meta model $\mathcal{M}$.*

An example of transforming a model to an Abstract Syntax Graph according to a given meta model is given in Figure 2.1. The according meta model is given in Figure 5.1. We continue to use a more simplified graph representation of the models. In this representation, the Abstract Syntax Graph is simply portrayed as a labeled directed graph. The labels for the nodes correspond to the node type names, while the edge labels represent the edge type names within the Abstract Syntax Graph [73].

Definition 5 *A **labeled directed graph** G is a tuple $(\mathcal{V}, \mathcal{E}, \lambda)$, where $\mathcal{V}$ is a finite set of nodes, $\mathcal{E}$ is a subset of $\mathcal{V} \times \mathcal{V}$, called the edge set, and $\lambda : \mathcal{V} \cup \mathcal{E} \to \mathcal{L}$ is the labeling function, which assigns a label to nodes and edges according to the label alphabet $\mathcal{L}$ [73].*

To further process these models represented as graphs, model differences between successive model m_n, m_{n+1} versions are computed. Therefore, first, deleted elements and newly

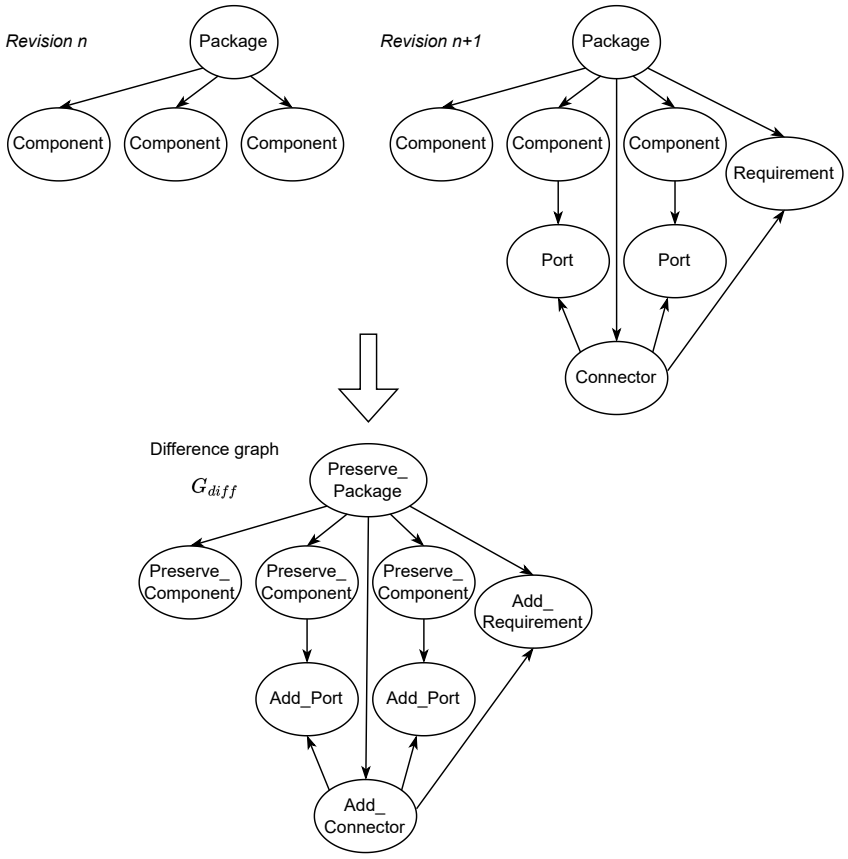

Figure 2.2: Example of deriving the difference graph $G_{diff} = G_{n,n+1}$ from two model versions m_n, m_{n+1}, example is taken from Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining by Tinnes et al. [73]

created ones are marked with a prefix as such in the corresponding model version, all elements that were not changed are marked as preserved. After merging these models by matching corresponding elements these adjustions lead to so called difference graphs, here $G_{diff} = G_{n,n+1}$, an example can be found in Figure 2.2 [73].

Definition 6 *A **difference graph** G_{diff} is a labeled directed graph where a prefix is added to each individual node and label element, indicating whether the component was newly created, deleted, or preserved.*

Usually, only a small amount of model elements are subject to change within a single evolution step. That is why these difference graphs are reduced to *Simple Change Graphs* , which only include elements that are newly created, deleted, or nodes that are directly

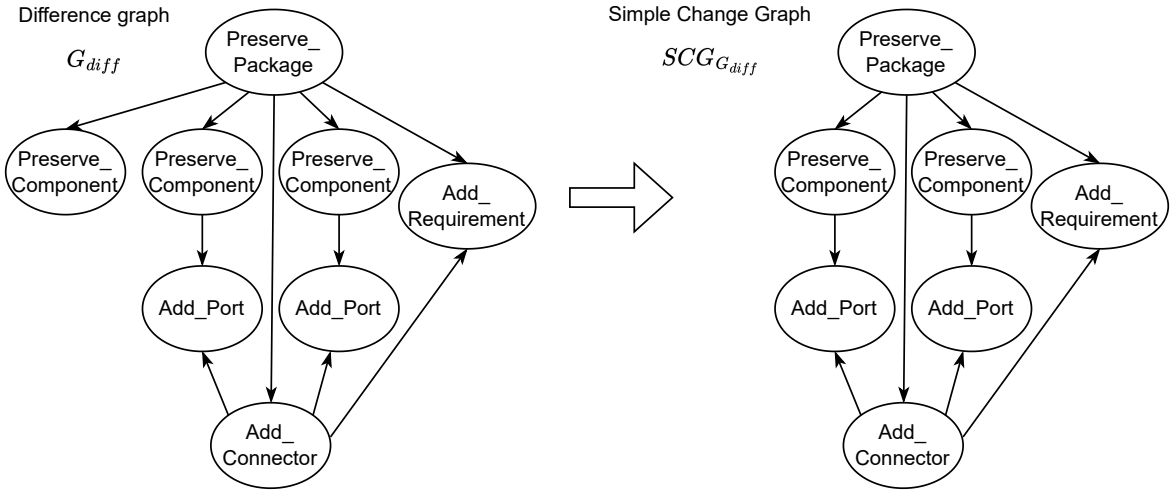

Figure 2.3: Example of deriving the Simple Change Graph Graph $SCG_{Gn,n+1}$ from a difference graph $G_{diff} = G_{n,n+1}$. The $SCG_{Gn,n+1}$ includes all elements that were newly created (prefix "Add") or deleted and all elements that are directly connected to such elements. The example is taken from Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining by Tinnes et al. [73]

connected to such an element. An example is again provided in Figure 2.3. An edit operation can be seen as a collection of model transformations that correspond to the same Simple Change Graph.

Definition 7 *A **subgraph of a graph** $G = (V, E)$ is another graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. A subgraph is formed by selecting a subset of vertices and a subset of edges from the original graph. Each edge in the subgraph (with possibly a specific type) links two nodes (with possibly also having specific types). Such an edge (with this specific type) connecting nodes (of the defined types) must also exist in the original graph with the correct direction [22].*

Definition 8 *The subgraph, **Simple Change Graph** ($SCG_{G_{n,n+1}}$) is derived from $G_{n,n+1}$ by selecting all the elements in $G_{n,n+1}$ representing a change (added, removed nodes and edges) and adding preserved nodes that are adjacent to a changed edge.*

2.2 MACHINE LEARNING

The thesis utilizes machine learning , specifically a GVAE to mine edit operations. That is why we will continue with a short, informal summary of the basic machine learning concepts.

Machine learning, in general, is the development and use of computer systems that are able to recognize various patterns or regularities from large data sets or their experience [1]. The collected information is then used to solve specific tasks [1]. Machine Learning is divided into supervised, unsupervised, and reinforcement learning. In supervised learning the training data needs to be labeled, meaning that each data point is mapped to its belonging group. Later on, the machine learning tool should be able to map yet unseen data points to their belonging group [1]. In contrast to supervised learning, unsupervised learning does not require annotated data [1]. In this case, the algorithm learns the underlying characteristics and connections of the data without them being labeled beforehand [1]. Another way to learn regularities is reinforcement learning. In this case, the learning process is based on a reward function, where the agent being trained tries to maximize its reward [1]. Consequently, it only learns from its experience, so no input data is necessary [1].

The creation of a machine learning model requires careful consideration of several important aspects.
Developing robust machine learning models requires appropriate validation strategies to assess their generalizability. A common practice is to divide the dataset into training, validation, and testing subsets. The machine learning model is trained on the training set, the hyperparameter tuned on the validation set, and finally evaluated on the test set, which consists of previously unseen data. The selection of evaluation metrics is another crucial aspect that hinges on the nature of the task. For instance, regression tasks may call for metrics such as Mean Squared Error or R-squared, while classification tasks could require measuring Accuracy, Precision, and Recall. Each of these metrics provides a different perspective on the machine learning model's performance, making the choice dependent on the specific objectives and constraints of the problem at hand [63].
It is crucial to pay attention to the optimization of hyperparameters, which can be a time-consuming and complex process. This is primarily done to avoid pitfalls such as overfitting (see Definition 9) [32, 79]. Hyperparameters, unlike other machine learning parameters such as node weights, are not adjusted during the training process but rather prior to it [79]. Therefore, they must be optimally determined, often posing a significant challenge, given that even minor adjustments can drastically affect the learning behavior [79].

Definition 9 ***Overfitting*** *is a term in statistics that describes a function that is too closely aligned with the training dataset. As a result, the machine learning model is only useful for this training data, generalizes poorly, and therefore provides poor predictions on other, unknown data [72].*

Choosing the right loss function to optimize is a key factor in machine learning model training. A loss function quantifies how well the machine learning model's predictions align with the true values, and the goal during training is to minimize this discrepancy. Depending on the task and method selected, different loss functions such as mean squared error, and cross-entropy loss can be selected [11].
Further, one must ensure that the datasets used are representative, preventing any potential bias or systematic errors relating to certain data groups.
The decision about the learning paradigm and the algorithm or machine learning model to use should be guided by the end goal of the task. To name some examples of supervised

approaches, if the aim is to assign each input to one of a finite number of categories, we are talking about classification problems [11]. In classification, the goal is to predict the categorical class C_k, where $k = [1, ..., K]$ of an input element x, for example, an image or graph [11]. This decision is based on past observations, which contain for each element a certain label. If $K = 2$ we are talking about a binary classification. Unlike other classification problems where an instance is tied to a single class, in the multi-label problem, the classes are nonexclusive, meaning that an instance can be assigned to several classes without any restrictions [54]. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. If the task involves predicting continuous outcomes instead, regression machine learning models might be the right choice. Example algorithms for both include Support Vector Machines, Decision Trees, and Random Forest [11].
In unsupervised learning tasks, one possible aim could be to uncover clusters of analogous instances within the dataset (clustering). K-Means or Hierarchical Clustering are possible approaches for this [11]. If the task involves reducing the dimensionality of data for visualization or feature selection, techniques such as Principal Component Analysis (PCA) can be used [11]. For tasks like teaching an agent to win a game by learning from its actions and their consequences, reinforcement learning methods might be the best fit, including, for example, Q-Learning or Actor-Critic algorithms [71].

2.2.1 *Graph Neural Networks*

In our exploration of machine learning techniques, we now turn our attention towards neural networks, specifically graph neural networks. This choice is motivated by the particular needs of this thesis, as we employ a Graph Variational Autoencoder, for mining edit operations. Neural networks are machine learning models, which are able to capture non-linear, complex relationships between data elements. They are widely used in many different domains, such as image recognition, natural language processing, and predictive modeling, to only name a few [32]. Neural networks are composed of several layers of interconnected neurons that process data [60]. Often, one visualizes the layers of a network as listed from left to right, where the input x is introduced on the left and the output y emerges from the right (see Figure Figure 2.4). The leftmost layer in the network is called the input layer and the rightmost is called the output layer. The middle layers are called hidden layers [60].

Graph neural networks are a specific form of neural networks, that accept and process graph-structured data as their input. Since graphs in general are usually built very differently, for example, have different sizes and structures with regard to their edges, they somehow need to be transformed to a comparable representation that can be understood by neural networks. Therefore different approaches were introduced in the past, Probably the most commonly used is to present graphs through their adjacency matrix, feature matrix, and edge feature matrix [76]. Let $G = (\mathcal{V}, \mathcal{E}, \lambda)$ be a directed, labeled graph with n nodes and e edges (Definition 5) and λ a labeling function $\lambda : \mathcal{V} \cup \mathcal{E} \to \mathcal{L}$. We further define a sentence encoding function:

Definition 10 *A **sentence encoding function** maps the node and edge labels or labels in general to an integer value. $\xi : \mathcal{L} \to \Xi$, where $\Xi \subseteq \mathbb{R}$*

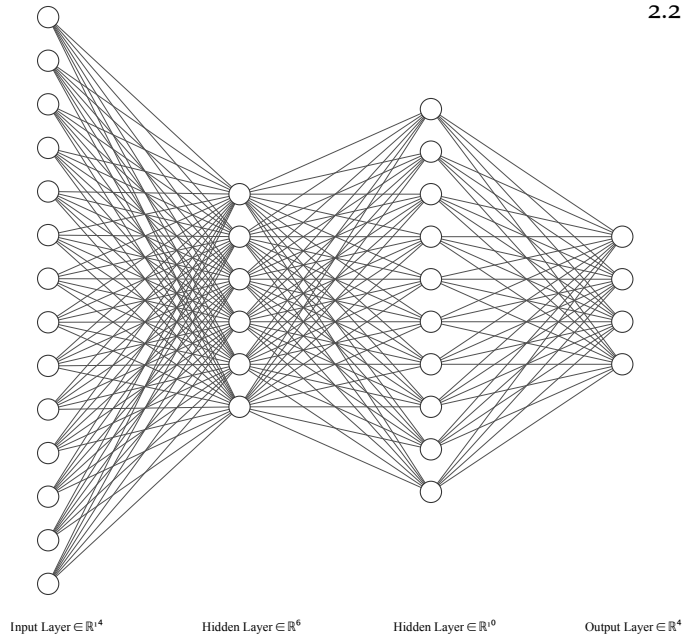

Figure 2.4: Introductory example of a neural network architecture

The adjacency matrix A of G is an $n \times n$ matrix defined as follows:

$$A_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{E} \\ 0, & \text{otherwise} \end{cases}$$

where $A_{i,j}$ is the (i,j) entry of the matrix A.

The feature matrix V of G, with d_n features is defined as follows. Note that in our case $d_n = 1$ because we currently consider only a single node feature, therefore $j = 0$. This feature encapsulates both the node type, as defined by the meta model, and an additional prefix. Each node i therefore has its feature f_i, such that each row i represents the feature vector of node i. In our specific case $f_i \in \Xi$.

$$V_{i,j} = \begin{cases} f_{i,j=0} & \text{if } i \in \mathcal{V} \\ 0, & \text{otherwise} \end{cases}$$

where $V_{i,j}$ is the (i,j) th entry of the matrix V and $j = 0$ since we currently consider only a single node feature.

The edge feature matrix E of G of size $n * n \times d_e$ is defined as follows. Again, we only have one label per edge, so $d_e = 1$, therefore $j = 0$ and $h_i \in \Xi$. .

$$E_{i,j} = \begin{cases} h_{i,j=0} & \text{if } i \in \mathcal{E}_{index} \\ 0, & \text{otherwise} \end{cases}$$

where $E_{i,j}$ is the (i,j) th entry of the matrix E, and $h_{i,j}$ is the j-th feature of edge i.

The edges in the matrix are sorted according to an additional edge index matrix $\mathcal{E}_{index}$, mapping an edge with source s and destination p to the corresponding edge index i of the matrix.

2.2.2 *Graph Variational Autoencoders*

To further dive into the Definition of Graph Variational Autoencoders, it is necessary to understand the basic concept of Autoencoders and Variational Autoencoders. An Autoencoder is a neural network framework that is used for dimensionality reduction and representation learning in an unsupervised manner [32]. It consists of an encoder and a decoder [32]. The encoder maps an input data point, represented by a vector x, to a lower-dimensional latent representation encoding z. The decoder then maps the latent representation back, trying to reconstruct the original input data [32]. So the goal is to learn to minimize the difference between the input data x and the reconstructed data x', which is measured through the loss function [32, 77]. Since the latent vector z has a restricted, smaller size than the input and output vector, the autoencoder usually needs to prioritize which aspects of the input are of most importance to minimize the loss between input and output data [32]. This has the consequence that often useful properties of the input data are derived [32].
Variational Autoencoder (VAE) are similarly built but extend Autoencoders with an additional probabilistic element [45]. Instead of solely learning a representation, Variational Autoencoders are generative models, where the main difference lies in the latent space. During the following, we will solely focus on Graph Variational Autoencoders, which learn representations of graph-structured data and are devoted to generating similar graphs to the input graphs.
In Variational Autoencoders, the encoder $q_\varphi(z \mid x)$ maps the input data to a latent space parameterized as a multivariate Gaussian distribution. The encoder is essentially learning to approximate the posterior distribution of the latent variables given the input data. This multivariate Gaussian is characterized by its mean and covariance, both of which are outputs of the encoder. To simplify the calculations, often the assumption is made that the latent variables are independent, such that only the variances for each dimension in the latent space are required [23, 45]. The encoder takes a data point x, in our case a graph, and predicts for each latent variable the mean and variance of a distribution [49, 77]. This is in contrast to traditional autoencoders, which map inputs to a fixed point in the latent space [77]. To dive deeper into understanding GVAE, we will compare the latent space of GVAE with the one of a basic autoencoder. Figure 2.6 and Figure 2.7 and shows how different data points (in our case, graphs) are mapped to a 2D embedding space.

In Figure 2.6 we see that similar graphs do not necessarily need to be mapped closely to each other, such if sampling from the latent space can result in unrealistic graphs that do not look similar to any of the input graphs used for training. While in Figure 2.7, points that are close in the latent space are also similar graphs when decoded. This opens the possibility to sample new, realistic graphs.
After the input data points are mapped to the latent space distribution, the decoder then generates new graphs by sampling random points from the distribution, the parameters of

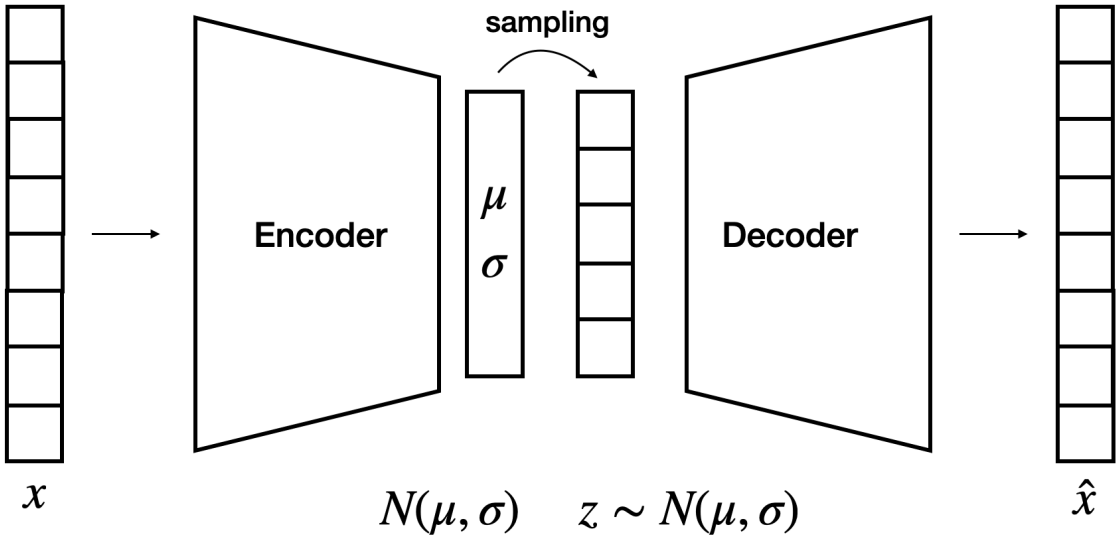

Figure 2.5: Basic concept of a Variational Autoencoder

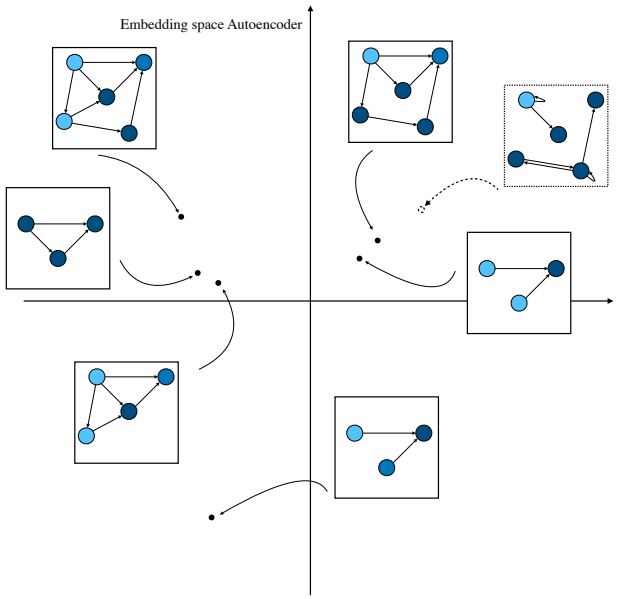

Figure 2.6: Latent space of an autoencoder with its arbitrary structure

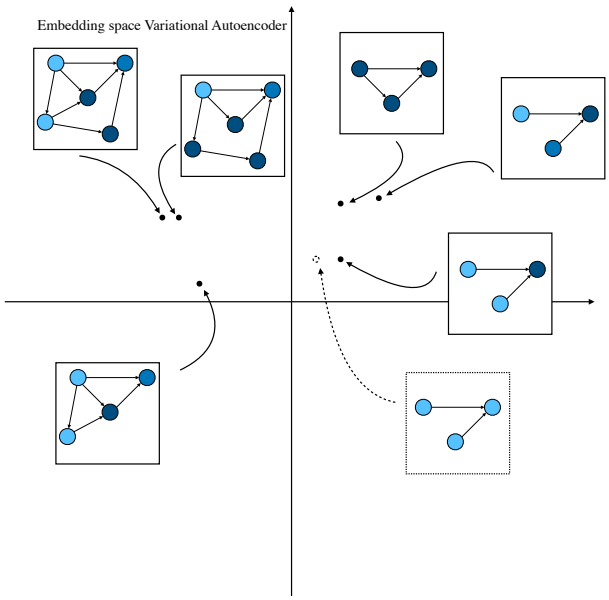

Figure 2.7: Latent space of an GVAE, where graphs are not arbitrarily spread over the latent space due to the regularization term of the loss function

which were calculated by the encoder. However, the process of sampling from a distribution that is parameterized by our machine learning model is not differentiable, which is a problem for gradient-based approaches [45]. As a solution to this problem, the reparameterization trick was introduced. The reparameterization trick is a technique used in variational inference to allow gradient-based optimization of stochastic variables by transforming them into a different form that is differentiable with respect to the machine learning model parameters [45].

The reparameterization trick involves representing the sampling process as noise, specifically in this case as a standard normal distribution (i.e., $\mu = 0$ and $\sigma = 1$) [45]. This noise term is independent of the machine learning model [45]. From this standard normal distribution, a sample is randomly taken [45]. Then the scaling and shifting properties of the Gaussian distribution are used, which includes changing the mean and variance of a sample by simply adding the mean and multiplying by the variance [45]. So the sample can consequently be scaled by the predicted mean and variance. So overall, for the decoder input, random points z are sampled via $z = z_\mu + z_\sigma * \epsilon$ from the encoder distribution, where $\epsilon \sim N(0, 1)$ is a sample from the standard normal distribution [45].
In the simple case of using autoencoders, usually, loss functions such as the mean squared error or binary cross-entropy are usually used. As a consequence, the data points are arbitrarily spread over the latent space, sometimes leading to discontinuities or gaps in the latent space [28]. Thus if the decoder samples new objects from these regions, the output can be unrealistic [28, 45]. That is why the loss function of VAE consists of two main parts, the reconstruction loss, and regularization loss [49]. The reconstruction loss measures how

well the network reconstructs the data. Often, mean squared error or binary cross-entropy are used, for example, for image data. One main problem of using Graph Variational Autoencoders arises in the reconstruction loss, which necessitates a graph matching procedure to compare the input graph and its probabilistic reconstruction. Unlike sequence generation graphs can have arbitrary connectivity and there is no clear best way how to linearize their construction in a sequence of steps [68]. Checking whether two graphs are isomorphic is in general is NP, but it is not yet clear whether it belongs to either P or NP-complete. Its generalization, the subgraph isomorphism problem, is known to be NP-complete and therefore tied to high computational costs [6, 29]. Therefore, in practice, approximations depending on the task at hand need to be specified.
Since there are not yet any constraints on μ and σ the encoder can learn to generate very different μ for different graphs, clustering them apart and also minimizing σ such that the new graphs do not vary much from the input graphs. To avoid this, the loss function does not only include the reconstruction loss but also a regularization term. The Kullback–Leibler divergence(KL divergence) provides a measure of how much one probability distribution differs from another [28, 45]. In the case of Variational Autoencoders, we aim to assess the difference between our learned normal distribution, parameterized by μ and σ, and the standard normal distribution.

Definition 11 ***KL Divergence Variational Autoencoder.***

$$D_{KL}\left[N(\mu,\sigma\|N(0,1)\right] = \frac{1}{2}\sum\left(1+\log\left(\sigma^2\right)-\mu^2-\sigma^2\right) \quad [3,28] \tag{2.1}$$

Ultimately, the KL divergence serves to harmonize the latent space, encouraging an even distribution around the center of the latent space [28, 45]. It also encourages a smooth and continuous mapping in the latent space, where similar data points are mapped more closely together and not arbitrarily spread over the latent space. To conclude the loss function of a GVAE is defined as follows. In the specific case of using a GVAE, x are graphs, specified via their adjacency matrix A, and feature matrices V and E.

Definition 12 ***Loss Function Graph Variational Autoencoder [3, 32].***

$$L(\theta,\phi) = -\underbrace{E_{z\sim q_\theta(z|x)}\left[\log p_\phi\left(x\mid z\right)\right]}_{\textit{Reconstruction Error}} + \underbrace{D_{KL}\left(q_\theta\left(z\mid x\right)||p(z)\right)}_{\textit{KL Divergence}} \tag{2.2}$$

A rigorous mathematical foundation for VAEs is presented in the original work [45]. This approach demonstrates that, under certain assumptions, the network can accurately learn the original distribution [45]. However, the numerous assumptions and approximations involved when dealing with graphs may compromise its accuracy in real-world scenarios.

2.2.3 *Probing*

In the context of machine learning probing refers to the examination of machine learning models to understand what kind of information they have captured [8]. Specifically focusing on GVAE, we can analyze and examine the latent representation of the GVAE to find out how different graph structures, properties, and features are embedded in the latent space.

3 RELATED WORK

This study resides at the intersection of machine learning and software engineering, focusing particularly on model transformations and the automation of specifying edit operations with the help of GVAEs. Given this interdisciplinary approach, the related work is organized into two main sections, discussing related work on model transformations and approaches from the VAE and GVAE domain.

3.1 MINING MODEL TRANSFORMATIONS

Several approaches for mining model transformations have been proposed in the last few years. Recent work varies in focus, with some studies concentrating exclusively on exogenous model transformations [7, 9, 65], and others, including this Master thesis, focusing on endogenous model transformations [73]. There are many different approaches including, for example, syntax-based support [5] [38] approaches and Model Transformation by Example (MTBE) approaches [44].
MTBE simplifies the creation of exogenous model transformations by making use of the knowledge embedded in practical examples, allowing users to design more complex, domain-specific model transformations with standard model editors [16]. In MTBE approaches usually one or more examples of model transformations are provided [2] [44]. For example, in the form of a concrete example [15], a pre-model and post-model [16] [44] or a set of positive and negative examples is given [30]. Then either edit operations should be derived from these examples, or it should be learned how one model is transformed into another [15][16][44].
To name a concrete example, Saada et al. propose an approach that is supposed to assist in writing declarative exogenous model transformations [65]. In the approach, models are also represented by labeled graphs and both the fragmented source models and target models are given to their system as input [65]. Furthermore matching links between fragments established by experts or by an automatic matching technique are given to the system. These links show which fragments of the source are mapped to which target fragments and are often a combination of some simple transformations [65]. These are then classified using the so-called GRAAL approach, which is a graph mining approach, to get useful common subgraphs. Afterward, new transformation patterns are then found from the classification of the matching links [65].
In opposite to several MTBE methods that haven't yet incorporated machine learning in their frameworks [15] [70] [57] [41], the following approaches utilize machine learning to infer model transformations bypassing the extremely long runtime. They introduce a supervised learning approach that automatically infers heterogeneous model transformations from sets of input-output model pairs with the help of Long Short-Term Memory (LSTM) neural networks [16, 17]. The authors state, that differing from previous works, their method learns purely from input/output model pairs without needing any user-provided correspondence

or traceing information. Making the method also usable for non-experts. As an overall goal, the system should be able to transform input models into their corresponding output models. They combine two LSTMs to an Encoder-Decoder architecture and choose a tree-like structure as an input that represents the models [16]. The tree representation also introduces a problem similar to the isomorphism problem with graphs [16]. Different ordering of the model elements in the tree would be considered as different input models [16]. Burgueno et al. address this problem by sorting the elements in alphabetical order [16]. Before their input trees are given to the first LSTM, they map variable names and attribute names to a closed set of words [16]. This additional step needs to be done since these variable and attribute names can be arbitrarily, so no finite dictionary of words would be given [16]. This would lead to the problem that only names that are used in the train set could be recognized by the Encoder-Decoder framework. When predicting new models, the inverse operation is performed as a postprocessing step [16]. In our first step of using graph neural networks, i.e. in this Master thesis, we similarly limit the dictionary of available words. The authors evaluate their approach to model-to-model transformations, for example, Class-to-Relational and code generation where they limited the size of the input model to 30 model elements [16]. The accuracy of their approach is tested by comparing the output model by the network with the corresponding expected output model [16]. In their second study, the network was trained to use a UML class model as an input and generate its corresponding structural Java code [16]. One problem stated in the paper is that for this supervised learning approach enough data is required to have adequate results [16]. Further, the authors state that the quality of the training depends a lot on the diversity of the dataset and state that the network is limited to only predicting models according to a pattern is has seen before [16], so the approach does not perform well in generalizing [16]. One main problem is that they do not consider mapping isomorphic AST-trees to the same representation. We first, address their problems by using graphs instead of AST representations. Further, our work stands out against the work of Burgueno et al., which focuses on exogenous model transformations and provides only a supervised setting.
To conclude, typically, these approaches also require some manual processing steps, for example, the user might need to provide a set of examples, which then need further manual adjustments to be turned into general specifications [73]. This process is not only time-consuming but also limits the data quality and quantity, affecting the overall quality, especially if machine learning is used [73]. For general-purpose languages like UML, there's a large user and researcher base to create, test, and validate refactoring rules. However, this isn't the case for Domain-Specific Languages (DSLs) [57]. The lack of a substantial user base and the need for specific expertise make defining edit rules for DSLs challenging [57] [73] [41]. Consequently, performance is often suboptimal, as most methods are only able to handle patterns seen during training and fail to generalize well [7, 16]. Especially if domain-specific languages are used.

Unsupervised methods don't require any special labeling and include creating model transformations out of a given meta model [43]. Meta models, beyond basic consistency and well-formulated rules, don't always incorporate domain-specific concepts and semantics. Consequently, generating model transformation from a meta model is usually limited to primitive tasks [73].

Tinnes et al. propose an unsupervised approach, called OCKHAM, which can learn useful edit operations from model histories in model repositories. Of course, which edit operation can be considered to be useful is highly task dependent. Due to the fact that the paper focuses on the evolution of models, in this context useful edit operations could be the ones helping to understand the model evolution [73]. Their work follows the idea of the Occam's Razor principle, which can also be applied to several topics in the field of software engineering [73].

Definition 13 *The **Occam's Razor principle** states that entities should not be multiplied unnecessarily which is interpreted as requiring that the simplest of competing theories be preferred to the more complex or that explanations of unknown phenomena be sought first in terms of known quantities [27].*

In the specific context of model transformations, this implies that the most important and therefore useful edit operations are the ones that compress the model differences [73].

Definition 14 *In the context of this paper, drawing upon the work of Tinnes et al., **useful edit operations** to domain experts are defined as those that facilitate an understanding of model evolution and most effectively compress the modeling history [73].*

As an illustration, a singular modification might recur numerous times, however, it is typically not interesting [73]. On the contrary, the total transformation in a model might involve a multitude of changes, yet with a frequency of just one occurrence, it is typically also not an interesting edit operation [73]. This idea of simplicity is also applied in this thesis with the help of Graph Variational Autoencoders, which, by design, reduce the input graph to its most important features [73]. The paper focuses on in-place edit operations and uses labeled graphs instead of typed graphs [73]. OCKHAM consists of several steps, starting with computing the difference graphs between subsequent models. Afterward, these difference graphs are reduced to Simple Change Graph (SCG) (more in Definition 8)[73]. Afterward, recurring change patterns are discovered using frequent subgraph mining [73]. Frequent subgraph mining takes a threshold t as an input and a set of graphs. Its output consists of all the subgraphs with, at least, t occurrences in the graph set [21]. This hyperparameter, t, needs to be carefully chosen [73]. A small support threshold might lead to a huge number of frequent subgraphs and computational effort and would it make difficult to find relevant subgraphs. As a tool for this process, they use the GASTON miner [73][61]. Their idea by Nijssen et al. is based on an iterative approach where the complexity of the graphs increases step by step. First, their approach finds frequent paths, then frequent free trees, and finally, cyclic graphs, which should make frequent subgraph mining more efficient[61]. Afterward, these frequent subgraphs are ranged according to their usefulness defined earlier.
One of the significant challenges that this Master thesis aims to tackle is the substantial computational effort required by this approach. Moreover, the method fails to consider important attribute information such as component names and generally does not account for inheritance. As a consequence, the approach by Tinnes et al. lacks abstraction capabilities. For instance, it perceives two edit operations that are identical except for one type, even if they share a parent type in the meta model, as different operations. This limitation can potentially lead to some operations being overlooked. These deficiencies could potentially

be addressed using a Graph Variational Autoencoder with the help of feature vectors. There are also minor issues that need attention. The first is the dataset's high variance, while some difference graphs are quite small, others are significantly larger. The second problem is that the method can only identify edit operations if all components are linked by one hop in the model graph, a limitation that usually poses a challenge in subgraph mining. The so-called "locality relaxation" problem limits the discovery of patterns that spread across multiple connected components of the SCG. By using Graph Variational Autoencoders we circumvent this issue. Still, this thesis integrates several aspects of the work by Tinnes et al. [73], for instance, using the dataset and computing the SCG and is, to our knowledge the only work mining endogenous model transformations in an unsupervised way [73].

Comparatively, the volume of academic papers focusing on models is notably less than those zeroing in on code patterns and completion [55, 59]. These fields are closely related, which suggests that some ideas from the realm of code patterns and completion could also be insightful for our purposes. To give a small illustration of some work, Nguyen et al. propose CPATMINER, which does not work on models but on code [59]. Their proposed approach detects previously unknown repetitive changes from a large set of code repositories [59]. Nguyen et al. also model code changes as graphs to better capture program dependencies. These change graphs are then given to their mining algorithm, which produces frequent subgraphs [59]. They made some adjustments to their recursive mining algorithm, which first constructs small subgraphs, expanding them in the course of the algorithm [59]. These adjustments include clustering isomorphic graph extensions into groups of isomorphic graphs using a heuristic. Further, not all possible extensions of a frequent subgraph are taken into account, but only the most frequent ones [59]. Their approach seems to be promising with regard to effectiveness performing better than AST-based techniques, supporting our assumption that graphs are in general better suitable for mining model transformations [59]. Revisar is a technique aimed at discovering quick fixes in large code repositories without explicit suggestions from users. Further, they are prioritizing them based on actual usage patterns [64]. It leverages collective programmer wisdom, learning from revision histories to detect frequently applied modifications. The process involves the use of Abstract Syntax Trees (AST) edits and d-caps to mine edit patterns [64]. Further, Yu et al. present a technique to automatically detect only those changes in software programs that are relevant to a specific development task, addressing the problem of traditional tools that often notify developers of more changes than necessary [81].

Overall, methods for model transformation either have the drawback of being computationally expensive due to the subgraph isomorphism problem [6, 73] or utilize (semi-)supervised machine learning algorithms that often need manual processing [73]. In conclusion, the current state of model transformation methodologies faces significant challenges, calling for further innovation and advancements in this field. In our work with the Graph Variational Autoencoder, we encounter a similar challenge, dealing with the graph isomorphism problem. In recent years, various neural network architectures have been developed to efficiently represent graphs. The goal is to accurately capture graph similarities while maintaining reasonable processing times.

We will have a closer look at these methods in the following chapter.

3.2 DEEP LEARNING APPROACHES FOR GRAPH-LIKE STRUCTURES

Several related works provide insight into how GVAEs can be effectively utilized and tailored to our setting. Graph Variational Autoencoders are used in several research fields for different purposes, including Malware detection [34], identifying of graph diffusion sources which find applications in the identification of rumor propagation, computer viruses, or smart grid failures [51] and in the generation of Dynamic Topic Models for text corpora [33]. The most significant success of GVAE was achieved in the application field of molecule generation. While most work concerning GVAE focuses on molecule generation, only a small amount of research in the field of software engineering utilizes GVAE or similar concepts like Variational autoencoders [36] or graph autoencoders.

In an approach by Sfax et al., a semantic-based approach using a Variational autoencoder is used to detect code smells. Instead of using graphs as input, they transform the code to an Abstract Syntax tree, which is transformed into a vector representation before being given to the autoencoder [36]. Once this step is completed, the Variational autoencoder will be used to construct representations that ideally embed the required semantic features. Afterward, these learned semantic features will be fed into a Logistic Regression to determine whether something is a code smell or not [36]. Their method focuses only on detecting the code smells Blob, which are large classes that implement most and therefore too many functionalities, Feature Envy and Long Method [36]. Feature Envy is a method that is more interested in using the data in classes other than its own [36].
Further, Sukur et al. present a proof-of-concept system for automatic program optimization [69]. They use Abstract Syntax Trees to represent the programs, Graph Autoencoders to embed the programs, and a Deep Reinforcement Learning Agent to learn which program transformations are to be applied to various types of programs in order to improve it with respect to the McCabe Complexity Metric [69]. They compare their approach with FermaT which relies on exhaustive search algorithms [69]. The system was found to significantly outperform the traditional FermaT method, reducing program complexity by an additional 15% on average, while requiring fewer transformations [69]. A notable drawback of the proposed system is its runtime, attributable to the computational complexity of running a deep learning machine learning model [69]. However, once trained, the embedding machine learning model and Reinforcement Learning agent can be used to transform new, previously unseen programs, demonstrating its efficacy over exhaustive search methods [69]. In conclusion, this approach offers a promising advancement in program optimization, with the potential for broad real-world applicability and performance improvement in software engineering. [69].

Paper focusing on molecule generation appear to hold considerable potential. There is a broad spectrum of research concentrating on Graph Variational Autoencoders, with many of them geared toward highly specific properties. A comprehensive review of all these works would be beyond the scope of this thesis. Below, we selected a few that guided us in our decision-making process.
Research can be divided based on numerous distinguishing factors. One of these factors is how the input graphs are being processed. Liu et al. introduce a constrained Graph

Variational Autoencoder (CGVAE) to generate molecules [52]. In the paper, they address the problem of huge graphs being intractable by regarding graphs sequentially. Meaning that their decoder extends a certain graph step by step. One downside of this approach is that arbitrary graph linearizations have to be considered, immensely increasing the amount of training data needed [52]. Moreover, by adopting such an approach, we risk losing the inherent advantages of possessing graph structures. We aim to effectively utilize and preserve the properties these structures offer, hence, linearization strategies do not present a viable solution for our requirements.

Liu et al. address this linearization problem by designing the learning objective in a way, that it only depends on the current state of generation and not on the arbitrarily chosen path to that state [52]. Their generative process interleaves between selecting a new node, edge selection, edge labeling, and node updates [52]. Even though their decoder has a completely different structure than ours, especially interesting is that they also optimize their CGVAE by exploiting hard domain-specific constraints, like molecule stability and enforce these constraints by masking out graphs that do not correspond to these constraints [52]. Further, such a sequential graph generation could become handy when dealing with model completion.

Simonovsky et al. further present a non-autoregressive VAE that generates molecular graphs, named GraphVAE [68]. Instead of generating output graphs step by step, the GraphVAE approach outputs graphs at once with a predefined maximum size. Considering that the objective of this study is to conduct an initial exploration into the applicability of Graph Variational Autoencoders, we have opted for the simpler approach of One-shot Graph Generation. This decision allows us to maintain a clear focus on assessing the general utility of GVAEs in a straightforward context.

Addressing isomorphism is a significant challenge when working with graph data due to its computational expensiveness [6, 29, 68]. Moreover, dealing with isomorphism can be particularly challenging. Overall, isomorphism presents both theoretical and practical challenges in the generation of graph data and is a critical issue to address when designing algorithms [6, 29]. An approximate graph matching strategy should be chosen carefully and aligned with the desired objectives. Simonovsky et al. address the problem of graph isomorphism, meaning that isomorphic graphs may have completely different matrix representations by using an approximate graph matching algorithm, which is tied to expensive computation [49, 68].

Winter et al. address the problem of graph isomorphism with respect to Graph Variational Autoencoders differently [75]. Alongside the GVAE, they train a permuter machine learning model that assigns to each input graph a permutation matrix to align the input graph node order with the node order of the constructed graph. Their approach does not impose particular node order nor is expensive graph matching needed [75].

Xu et al. present a theoretical framework to analyze different graph neural networks with regard to the success in capturing and differentiating different graph structures [78]. Further, they introduce a new neural network architecture, Graph Isomorphism Network (GIN), which they state outperforms other known architectures with regard to how well graph isomorphisms can be detected [78]. GINs follow the idea of the Weisfeiler-Lehman graph isomorphism test, which is an effective and computationally efficient test that distinguishes a broad class of graphs [78].

Overall recent work showed us, that one of the main bottlenecks for performance is the computation of mapping isomorphic graphs to the same matrix representation [75].
This work is partly based on the work of Kwon et al. [49]. In their work, they utilize a Graph Variational Autoencoder in a non-autoregressive manner to generate new molecules [49]. They further improved their approach by adding three additional learning objectives. Approximate graph matching, reinforcement learning, and auxiliary property prediction. They utilize reinforcement learning to improve the generation performance [49]. The reward for the action is the chemical validity of the probabilistic graph, which returns 1 if the graph can be decoded into a chemically valid molecule and 0 otherwise [49]. Concerning the machine learning architecture, they use a message-passing neural network (MPNN) as an encoder, which is able to work on graphs of different sizes and is invariant to graph isomorphism. For the decoder, a fully connected neural network is used. The problem of graph isomorphism is addressed by approximate graph matching to make training more efficient [49]. We use their implementation as a base for ours, which also includes their graph matching objective. Still, their formula seems to be a bit arbitrary and may be too simple. Anyway, we use it as a starting point for further improvement. While their approach performs well in generating smaller molecular graphs, nonetheless it still has some difficulties in machine learning model training and thus in generating larger graphs (low validity) [49]. Kwon et al. state that one downside of their approach still is the high complexity with regard to space and time [49].
Liu et al. also propose a learning framework that compares different approaches of representations and networks with regard to how they are performing in the approximation of the subgraph isomorphism counting problem [53]. Given a pattern and a graph, the method aims to determine how often the pattern occurs in the given graph [53]. While these methods may be inexact, Liu et al. introduce a method that generalizes well such that errors are kept at an acceptable low amount while the computational performance is tremendously increased (linear time instead of exponential) [53]. Regarding the representation of graphs and patterns, the authors compare sequence inputs and graph inputs. For sequence inputs, they use CNNs, RNNs such as LSTM, or Transformer-XL to extract high-level features. For machine learning models which use adjacent matrices and node features, they test RGCN and GIN [53]. The authors conclude that graph representations in general perform better in capturing isomorphism and counting patterns. Also, the authors support the claim by Xu et al. that GINs perform considerably well with regard to capturing homogeneous graph structures and therefore outperform other existing approaches [53]. The work by Xu et al. and Liu et al. gives valuable insides into the performance of graph neural networks, which will also help in the remainder of this thesis in the architectural decision process.
Liu et al. additionally provide an algorithm to generate training data since not enough data is provided by existing datasets to train graph neural networks [53]. The algorithm generates graphs, and patterns and at the same time calculates the ground truth of how often the pattern is contained in the graph [53].
Unlike recent studies, we focus on edit operations instead of molecules. While molecular studies often rely on undirected graphs, dictated by the inherent nature of molecular structures, our work necessitates the use of directed multigraphs.

In our case of generating edit operations, these newly generated objects should be valid

according to our meta model. Even with an excessive amount of training data, the validity can not always be guaranteed, leading to the need to filter out such invalid edit operations afterward. This problem also occurs in other research fields, since many structures have domain-specific constraints [19].
To address this problem Kusner et al. propose a Grammar Variational Autoencoder, which directly incorporates a check for syntactic validity of graph-like structures. In their work, they use the SMILES notation and its grammar to test their approach on molecules [48]. Although Kusner et al. propose a method for generating syntactic valid molecules, it still does not consider semantic validity [48].
Dai et al. extend their work, which embeds also the generation of only semantically valid objects in their Variational Autoencoder framework [19]. We propose guaranteeing the validity of the generated SCG as future work.

De Cao et al. propose MolGAN, which instead of Graph Variational Autocoders, uses a generative adversarial network (GANs) to generate molecules [20]. A GAN consists of two neural networks contesting with each other [20]. While the generative model learns to sample new data, the discriminative model tries to classify whether samples came from the original data or from the data generated by the generative model [20]. The approach should circumvent the need for expensive graph matching procedures or node ordering heuristics [20]. De Cao et al. also add a Reinforcement learning objective using a simplified version of DDPG to their GAN to support the generation of special desired molecule properties [20]. DDPG is an off-policy actor-critic algorithm that uses a deterministic policy to maximize an approximation of the expected future reward [20]. Additionally using Reinforcement Learning could also improve our results of generating valid edit operations according to the meta model. Therefore the GAN approach gives us a great inside on the performance of the interplay between this additional Reinforcement Learning factor and their main contribution, which could be beneficial in the future to decide which Reinforcement Learning approach should be used and gives us a starting point for hyperparameter tuning. On the other hand, GANs have a reputation for being difficult to train [4]. The interaction between the generator and the discriminator can often lead to non-convergence, due to players competing with each other [31]. Secondly, GANs can suffer from a problem known as mode collapse. The generator can discover a particular type of data that is exceptionally effective at deceiving the discriminator [31]. Once this happens, the generator starts producing limited or even identical outputs over time, resulting in a lack of diversity and uniqueness in the generated samples [31]. Conversely, Graph Variational Autoencoders present a more favorable approach for our work. GVAEs are generally easier to train allowing easier and more stable optimization [20]. GVAEs are less prone to mode collapse, thereby offering greater diversity and uniqueness in the generated molecules [20]. While I have elected to use GVAEs in our study, inspired by De Cao et al., I propose a comparison of GAN-generated results to ours in future work.

4 APPROACH

4.1 PROBLEM STATEMENT AND THE MOTIVATION BEHIND USING GRAPH VARIATIONAL AUTOENCODERS

Creating edit operations is far from simple. Meta models lack the specific domain knowledge needed to define these transformations. Further, understanding of implicit concepts and semantics are often required [7, 16, 73]. The complexity rises when dealing with domain-specific languages, adding significant overhead to many projects [16, 41, 57, 73]. Various methods have been proposed to tackle more complex and domain-specific tasks (see Section 3.1). Shortly summarizing, traditional methods often require laborious, error-prone manual pre-processing, indirectly also limiting the training data available [37]. As a consequence, they are limited in their ability to generalize [7, 16, 37]. Most methods can only identify edit operations present in the training data and/or are limited to simpler cases. We address these issues with an unsupervised approach aimed at automatically extracting model transformations from model histories. Unsupervised methods have the advantage of not needing any labeled data, model transformation could, for example, be derived from meta models only. Anyway, beyond ensuring basic consistency and well-defined rules, meta models often lack the inclusion of domain-specific knowledge and semantics. As a result, creating model transformations based on these meta models is generally restricted to simpler tasks [73]. Instead, we propose to mine model transformation from model histories making it directly possible to extract more data, for example, from online repositories. There is only limited existing work in this unsupervised setting, our method tackles their shortcomings, namely generalizability issues and ideally scalability in the future [73] through machine learning, hopefully benefiting from recent advancements in the field. Using machine learning opens a wide range of new possibilities, probably many of them suitable for our use case. For example, our chair is also currently working with language models. Still, using GVAE seems to be one of the most promising ways to go due to its many different advantages. First, models inherently exhibit a graph-like structure, and we aim to preserve this crucial information throughout our processing. Conversely, language models require serialization, which could account for their comparatively sub-optimal performance. Second, Variational Autoencoders are well known for their ideal abstraction and compression capabilities [32]. We hope that this factor can help us in finding the most relevant edit operations for domain experts [73].

By employing a GVAE, we directly have a generative model that can be directly applied to tasks such as completion and fuzzing, for example, in the context of tools like GitHub CoPilot. This setup allows us to, given a model, automatically suggest edit operations or generate new models. A minor advantage is that we can include additional information in the form of extra nodes and edge features. This could, for example, be used to include inheritance information, which was not yet possible in other approaches. While the former approach by Tinnes et al. is limited to recognizing edit operations that are connected, GVAE

open the possibility to extract edit operations spread over several non-connected components [73].
Specifically focusing on graph neural networks, one alternative could be Graph Generative Adversarial Networks (GANs). A GAN consists of two neural networks contesting with each other [20]. While the generative model learns to sample new data, the discriminative model tries to classify whether samples came from the original data or the data generated by the generative model [20]. Recent work stated that these methods often suffer from mode collapse problems (Section 3.2), are rather difficult to train, and tend to generate non-divers, non-unique graphs [4, 31]. Graph Variational Autoencoders seem to allow easier and more stable optimization, also stated by several papers, such that, as a first step we decided against using GANs [31]. While we focused on GVAE, it's worth noting that alternative approaches like diffusion models or GNNs with a reinforcement learning setting may also be viable but were not closely examined. Though GVAE appears to be a suitable choice, future research may reveal other effective methods. Though we utilized GVAE for this study, it shouldn't be interpreted as asserting it's the ultimate or only solution for this problem. Other methods could prove equally or more effective in future research. Still, to our knowledge, it is the first of its kind, so using GNNs for mining model transformations. Later on, we would like to compare the presented approach to other methods, for example, Kernel methods and language models.
The use of Graph Variational Autoencoders is a relatively new area. Its theory is still developing, making it hard to find straightforward solutions. There are many ideas and methods in this field, but most are designed for very specific situations, adding complexity to the exploration of this new area (Section 3.2). Navigating the landscape of existing systems, it quickly becomes apparent that a straightforward adoption of a pre-existing GVAE is unfortunately not possible. Significant modifications and adjustments are required to adapt a GVAE to our unique needs, particularly in our pursuit of mining edit operations. A broad survey of readily available implementations showed that none of these implementations could directly be adapted to our setting. These include a lack of clarity in their design, the presence of bugs, or the absence of essential elements crucial for optimal performance. Furthermore, nearly all of the existing work focuses on the generation of molecules, leading to the necessity to modify the given implementation to our setting of generating edit operations. A key distinction in our study involves the usage of labeled edges and nodes in directed graphs, setting it apart from other studies that primarily focus on undirected graphs, such as those related to molecules. To formally specify, we have the following problem.

Graph Characteristics In the domain of model transformations, we have directed graphs with (possibly multiple) node and edge labels, where the labels can be arbitrary. The labels can range from basic natural language to domain-specific extensions, which may only be understood by domain experts. The graph sizes can vary significantly and graphs may be extremely large.

In this thesis, our primary aim is to create a pipeline that can be potentially expanded later on. Initially, we will concentrate on handling directed graphs with node and edge labels from a restricted dictionary.

4.2 REALIZATION

4.2.1 *Specific Technologies*

As a starting point for the GVAE implementation, we used an implementation based on the work of Kwon et al. [49][1]. We chose that implementation due to its rather simple nature, which does not require complex, additional features such as an auxiliary network for mapping similar adjacency matrices to each other. For the purpose of our study, we have chosen to implement our machine learning model using PyTorch rather than TensorFlow. Our decision was driven by a few key considerations. PyTorch, known for its straightforward and user-friendly nature, allows for intuitive programming, both of which are highly advantageous in our context. With PyTorch, we can focus more on the implementation of our machine learning model and less on managing the intricacies of the library itself. This leads to an efficient and more streamlined development process, helping us to remain focused on the core of our research. Our study serves as existence proof, a solid foundation justifying further investigation into the usage of Graph Variational Autoencoders for mining edit operations. We further use the torch-geometric extension for graph-based layers and architectures. For the graph representations the NetworkX, library is used. Additionally, Optuna is used for hyperparameter optimization and matplotlib for data visualization.

4.2.2 *Conceptual Approaches*

Graph-based representations provide an ideal framework for capturing edit operations, serving as the foundation for our approach. Starting from model histories, we convert these into Abstract Syntax Graphs based on their meta model types (Definition 4). These are then represented as labeled directed graphs, laying the groundwork for further analysis (Definition 5). Building upon these graph representations, we move on to identifying changes between successive model versions. In this phase, elements that are newly created or deleted are marked with specific prefixes, while unchanged elements are labeled as preserved. This results in the generation of difference graphs, which map identical elements between model versions. Since only a small subset of elements typically changes a single evolution step, we simplify these difference graphs into Simple Change Graphs (Definition 8). These specifically include only newly created, deleted elements, or nodes directly connected to such elements. These Simple Change Graphs then serve as the inputs to the Graph Variational Autoencoder. In essence, an edit operation can be viewed as a recurring pattern in Simple Change Graphs. These graphs were then saved in files, see Figure 4.1 on the very left-hand side.
During the actual process, we parse these files to construct Digraphs for the respective dataset (Section 5.2.1). Each dataset is partitioned into training, testing, and validation sets using a specified seed for shuffling. Node and edge labels are then converted to integers based on a sentence encoding function (Equation 15), which we specifically define in this thesis as follows.

1 https://github.com/deepfindr/gvae

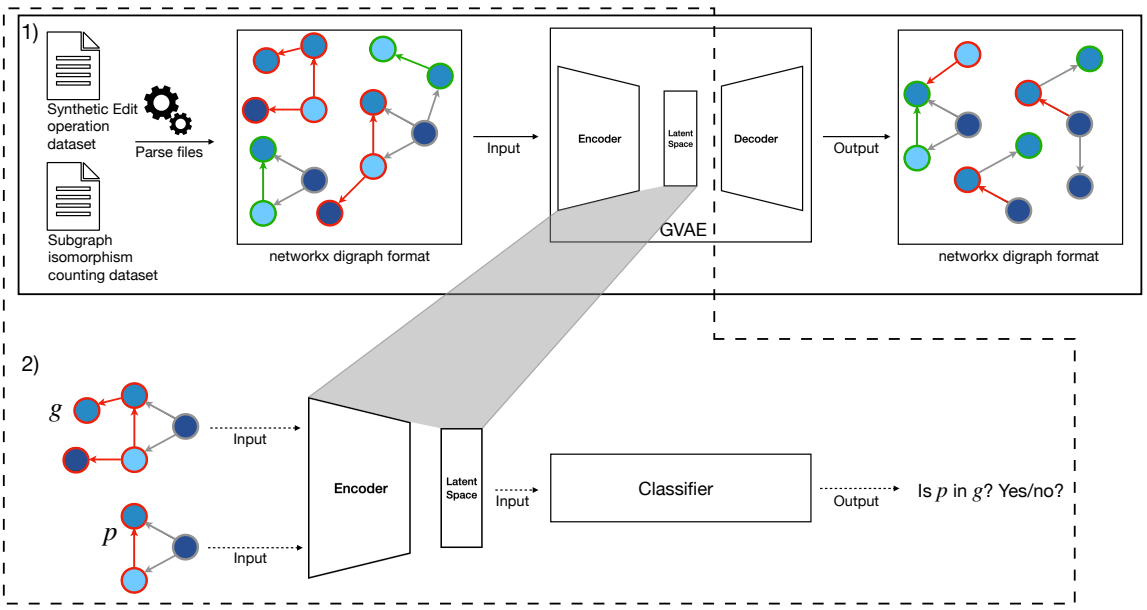

Figure 4.1: Overview of our pipeline, 1) specifically focuses on RQ1, while 2) demonstrate our setting for answering RQ2, parsing the files is similar to 1), we train our encoder on the train set without the target values (Yes/No), afterward the the classifier is trained on this specific set, only getting the latent representation of graphs and patterns as input.

Definition 15 *Sentence Encoding Function* ξ_{GVAE}*.*

$$\xi_{GVAE} : \mathcal{L} \to \{0, 1, \ldots, |\mathcal{L}|\} \tag{4.1}$$

$$\xi_{GVAE}(x) = i \tag{4.2}$$

$$\xi^{-1}_{GVAE}(i) = x \tag{4.3}$$

ξ is a bijection between $\mathcal{L}$ and $\{0, 1, \ldots, |\mathcal{L}|\}$.

As a consequence, we have Digraphs with integer labels. The layers from the torch_geometric.nn library are directly able to process these Digraphs, each one having their own specific internal representation[2]. Afterward, the information is given in the adjacency matrix A, feature matrix V, and edge feature matrix E (Definition 2.2.1). The whole process is again depicted in Figure 4.1.
In our study, we have kept certain variables constant, despite their impact on performance. These include factors such as the aggregation function and readout function. While we recognize the necessity of a thorough exploration of these variables for a more detailed understanding, we take into account existing studies that provide a general direction on how these factors influence performance. Despite the importance of these variables, an in-depth study of them exceeds the boundaries of our current work.

2 for a closer look at the exact implementation details, have a look at https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

The aggregation function combines the information from a node's neighbors in order to compute the node's representation. Some common aggregation functions include simple averaging, weighted averaging, and taking the maximum or sum. There is some evidence that sum aggregation is performing best in most cases [35, 78]. In our case, the used aggregation function is already defined by the layers we selected. For the convolutional layer, we use TransformerConv that includes weighted sum aggregation [67]. GINEConv is based on Graph Isomorphism (GIN) but extends its implementation by incorporating edge features, and also uses sum aggregation [39]. GATConv uses an attention-weighted sum [74]. We use the torch_geometric.nn library for all layer implementations.
While the aggregation function is responsible for summarizing the local neighborhood information around each node, pooling is used to compute a representation of an entire graph that can then be obtained through pooling, for example, by summing the representation vectors of all nodes in the graph [78]. One requirement is here that the function used should be permutation-independent with regard to the node order. This is essential because otherwise, similar graphs with different node orderings would yield different representations, undermining the consistency of the model. Alternatives include mean, maximum, or even combinations of these. For global pooling, we use a function that returns graph-level-outputs by adding node features across the node dimension, so that for a single graph g its output is computed by

Definition 16 *The **pooling layer computation** for graph g is defined as:*

$$r_g = \sum x_n, \tag{4.4}$$

where x_n is the result of each node n.

So the edge features are only indirectly cooperated in the global representation of the graph via the pooling layers input, which is equivalent to the encoder's output. We leverage not only the output of the last layer but also the outputs from all encoder layers. By concatenating these layer-wise graph representations, our graph neural network can benefit from the multilevel information, while early layers tend to focus on local features, later layers capture more abstract, global information [35, 40, 80].
For training, we opt for the Adam optimizer. As a starting point for further research we started by using an adjusted form of the loss by Kwon et al [49]. To avoid expensive computation, we propose to use approximate graph matching to be able to further focus on the question of whether GVAE can be used for mining edit operations.
Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the original graph and $\widetilde{\mathcal{G}} = (\widetilde{\mathcal{V}}, \widetilde{\mathcal{E}})$ its probabilistic reconstruction.

The paper's main idea is to approximate the distance between $\mathcal{G}$ and $\widetilde{\mathcal{G}}$ by comparing the numbers of node types, edge types, node-edge pair types, and node-edge-node pair types. We define the reconstruction loss as, where v_i are the one hot encoded feature vectors of node i, in our case each node has only one feature f_i, which has a maximum value in can reach f_{max}, such that so $v_i = (v_{i,1}, ..., v_{i,f_{max}})$. Similarly, the edge vector $e_{i,j} = (e_{i,j,1}, ..., v_{i,j,d_{max}})$

Definition 17 ***Reconstruction Loss of the GVAE.***

$$\mathbb{E}_{\mathbf{z}\sim q_\theta(\mathbf{z}|\mathcal{G})}\left[-\log p_\phi(\mathcal{G}\mid\mathbf{z})\right] \simeq \left\|\left(\sum_i \mathbf{v}_i\right)-\left(\sum_i \widetilde{\mathbf{v}}_i\right)\right\|^2 \tag{4.5}$$

$$+\left\|\left(\sum_{i,j}\mathbf{e}^{i,j}\right)-\left(\sum_{i,j}\tilde{\mathbf{e}}^{i,j}\right)\right\|^2 \tag{4.6}$$

$$+\sum_{b\in\{1,\dots,d_{max}\}}\left\|\left(\sum_{i,j}e^{i,j,b}\mathbf{v}^i\right)-\left(\sum_{i,j}\tilde{e}^{i,j,b}\widetilde{\mathbf{v}}^i\right)\right\|^2+\left\|\left(\sum_{i,j}e^{j,i,b}\mathbf{v}^i\right)-\left(\sum_{i,j}\tilde{e}^{j,i,b}\widetilde{\mathbf{v}}^i\right)\right\|^2 \tag{4.7}$$

$$+\sum_{b\in\{1,\dots,d_{max}\}}\left\|\left(\sum_{i,j}e^{i,j,b}\mathbf{v}^i\mathbf{v}^{jT}\right)-\left(\sum_{i,j}\tilde{e}^{i,j,b}\widetilde{\mathbf{v}}^i\widetilde{\mathbf{v}}^{jT}\right)\right\|^2 \tag{4.8}$$

$$\tag{4.9}$$

As a reminder, edit operations can be seen as reoccurring patterns in the Simple Change Graphs. One aim is to determine if the GVAE can recognize these pre-existing previously applied edit operations (Section 5.1). As a starting point, our primary focus is therefore to explore whether the GVAE is capable of retaining information about frequently occurring subgraphs or patterns within these Simple Change Graphs. Meaning, that we would like to find out whether information about these edit operation patterns or patterns is already present in the latent space. The additional regularization term in the loss function of Variational Autoencoders should encourage a smooth and continuous mapping in the latent space, where similar data points are mapped close together. This enables interpolation between different data points in the latent space. Anyway, the question remains, whether this only includes rather primitive features like graph size or also advanced features like different patterns. In the thesis, we introduced an additional classifier for this probing task. We trained the classifier in a supervised manner, to analyze the latent space of the GVAE to predict the presence and frequency of specific patterns within an input graph (Figure 4.1, Part 2). Alternatively, we could have also used Visualization Techniques such as t-SNE, PCA, but since representations are high-dimensional we postponed that idea, especially with respect to the pattern inclusion task. Anyway, we used it shortly to analyze rather trivial features, such as whether graphs with, for example, the same sizes are closely mapped together or not. An alternative choice that came to our mind was the use of a regressor. Anyway since we have clear categorical true values, namely whether a pattern is in a graph or not, a classifier seems to be a natural fit for a first investigation. Notably, the GVAE is not provided with the true values indicating the presence or frequency of patterns in the input graphs. It solely receives the graphs themselves as input data. After training the GVAE, we leveraged the latent space obtained from the GVAE to further train the classifier. To be precise, the classifier, which is a neural network, is fed only the latent representations of each graph during the training process to predict how often a certain pattern is contained in the given graph. In order to ensure a more robust training process for the classifier, we addressed the issue of having imbalanced datasets, by using oversampling. In the course of our research, we evaluated a range of alternative strategies, including undersampling, which proved infeasible due to the exceedingly low frequencies of certain classes. Additionally,

we considered the creation of synthetic instances of the minority class. However, each of these techniques comes with its own set of limitations such that we first employ the more straightforward method. Additionally, to mitigate the impact of the imbalanced data distribution, we explored the implementation of a cost-sensitive learning approach. This method assigns greater significance to classes with fewer occurrences, empowering the classifier to make more accurate predictions and effectively handle the challenges posed by the imbalanced dataset. Anyway, in the specific case of our initial study, we already balanced the data, making the cost-sensitive learning redundant. During implementation, we decided on Cross Entropy Loss as a loss function.

Definition 18 ***Undersampling*** *removes samples from the majority classes to balance them with the minority classes.*

Definition 19 ***Oversampling*** *adds samples to the minority classes to balance it with the majority classes by randomly copying samples from the minority classes.*

5 EVALUATION

5.1 RESEARCH QUESTIONS

To address the overall problem of mining edit operations with the help of Graph Variational Autoencoders, this work is divided into several research questions, which will be addressed in the remainder of this thesis.

We are first interested if the basic concepts of the input data can be learned.

RQ 1: ***Can realistic SCG be generated via a GVAE?*** When we say "realistic" graphs, we are referring to graphs that adhere closely to the structural information found in real-world scenarios, in our case, given in the form of the input data. Due to the fact that graph matching is tied to high computation costs and therefore infeasible in our case, we restrict our Definition of "realistic" graphs to two points [6, 29]. First, "realistic" graphs should follow the meta model's rules, so be correct according to the meta model. Second, the graphs should replicate the structural intricacies of real-world data. One assumption we take here is that the input dataset is sufficiently large facilitating a higher occurrence of frequently used snippets over those that are less common. For example, in the context of code, realistic code would not only use the correct syntax but also follow common coding conventions, like a higher frequency of variable assignments compared to class definitions. It means that the distributions of attributes such as node types and edge types in the generated graphs should mirror those in the original data. For example, indicating that the network can learn the most often used element. Coming back to the code example, we hope that the GVAE would produce "realistic" code, including lots of variable assignments and fewer class definitions and not just defining class after class.
RQ 1.1: ***What influence do certain factors have on the generation of realistic SCG?*** We want to identify the factors that influence the performance of our GVAE in generating realistic SCG. We specifically test three different commonly known graph neural network layers as encoder layers.

In the second part, we extend our inquiry to assess the ability of the GVAE from RQ1 to identify previously applied edit operations.

RQ 2: ***Can the GVAE from RQ1 recognize previously applied edit operations?*** As a quick reminder, the edit operations can be seen as reoccurring patterns between model changes, given in the form of SCG. As part of our exploration into mining edit operations with the help of GVAE, our initial focus is to determine if, given an edit operation and a SCG, we can identify whether and to what extent this edit operation has been applied. This crucial step will set the foundation for more sophisticated analyses. Drawing inspiration from the work presented by Liu et al., we aim to investigate whether it can be identified if a certain pattern

is included in a given graph or not [53]. It's worth noting that while Liu et al. focused on Graph neural networks, they did not incorporate dimensionality reduction, which sets our work apart [53]. This exploratory step is critical as it lays the foundation for the subsequent generation of edit operations and gives us valuable insights.

RQ 2.1: ***Does the latent space of the GVAE capture information on whether or not a certain pattern is present within a given graph? (Binary Pattern Inclusion Task)*** Our method should be able to generally identify a specific pattern whether it is included in a certain graph, given its latent representation only. We utilize an additional classifier to answer this question.

RQ 2.2: ***What are the main factors that influence the performance of the Binary Pattern Inclusion Task?*** We investigate the impact of certain elements (neural network architecture of GVAE, neural network architecture of classifier, Hyperparameters).

RQ 2.3: ***Does the latent space of the GVAE contain information regarding the frequency of a specific graph pattern within a given graph?*** We are further interested if the compressed representation retains any information about how many times a particular graph pattern appears in the graph.

5.2 AVAILABLE DATA

For our experiments, we have two available datasets. One is provided by related work from Liu et al. [53], and the other by Tinnes et al. [73].

5.2.1 *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining [73]*

Tinnes et al. provide a synthetic dataset, representing model instances, differences, and edit operations as graphs [73]. The dataset is based on the component meta model given in Figure 5.1. Each dataset contains model differences in the form of a chain/history of model instances reaching from $m_0, m_1 ..., m_{d-1}$. They conduct model histories, with either 10 or 20 model instances [73]. In each instance, m_i, a certain edit operation is randomly applied $e \in \{1, ..., 100\}$ times to the model, leading to the next version $m_i + 1$ [73]. Additionally, a specific probability $p \in \{0.1, 0.2, ..., 1.0\}$ of applying a second edit operation is given [73].

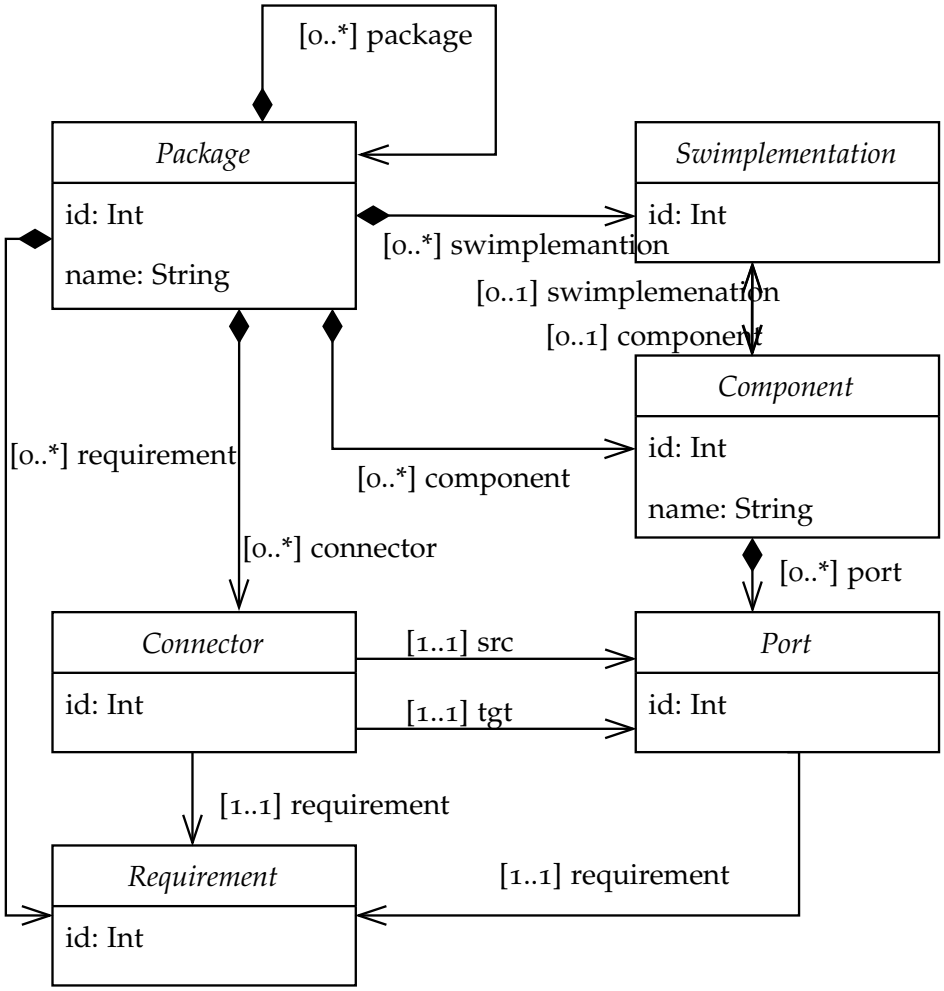

Figure 5.1: Meta Model: Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining [73]

If applied, the second edit operation should overlap with the first one. Overall, the data provided by Tinnes et al. includes 2000 $(= 2 \times 100 \times 10)$ datasets [73]. For the purpose of this master thesis, we divided the data into several sub-datasets given in Table 5.1.

Table 5.1: Details of the used datasets from Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining

Dataset name	Max size	Description	# Graphs	Pattern Characteristics
Dataset Small 1	10	Random selection of graphs	780	832
Dataset Small 2	10	Random selection of graphs	1583	1991
Dataset Small 3	10	Random selection of graphs	2048	2353
Dataset Small Small-E-values	10	Focus on model histories with small e values ($e \in \{1, 11\}$)	952	1140
Dataset Small Small-P-values	10	Focus on histories with small e values ($p = 0.0$)	2451	2609
Dataset Small Large-E-values	10	Focus on histories with large e values ($e = 81$)	4807	5897
Dataset Small Large-P-values	10	Focus on histories with large p values ($p = 1.0$)	3918	5385
Dataset Large	10	Large data amount ($e \in \{21, 31, 41, 51, 61, 71\}$)	7801	9427

5.2.2 *Neural Subgraph Isomorphism Counting [53]*

Besides investigating how different approaches perform in approximating the subgraph isomorphism counting problem, Liu et al. also introduce an algorithm for creating graph datasets [53]. The resulting datasets consist of graphs and patterns and the algorithm directly tracks how often a certain pattern is contained in a certain graph while generating the dataset [53]. Liu et al. additionally provide an algorithm to generate training data since not enough data is provided by existing datasets to train graph neural networks [53]. The algorithm generates graphs, and patterns and at the same time calculates the ground truth of how often the pattern is contained in the graph [53]. The generator is divided into different components. The pattern generator produces any connected multigraph without

identical edges. The number of subgraph isomorphisms instead needs to be tractable. The graph generator first generates multiple disconnected components, which can contain some subgraph isomorphisms [53]. Then the generator merges these components into a larger graph and ensures that there is no more subgraph isomorphism created during the merging process [53]. Although their work broadly focuses on graphs, their approach proves extremely beneficial for our purposes. This is because various parameters and constraints can be individually tailored for each dataset generation [53]. For example, one can control the density of subisomorphisms, the number of graph edges and pattern edges, and several other characteristics [53]. So overall, we can add edit operation-specific constraints to the dataset and at the same time have control over how often and which patterns are contained in the datasets. The work by Tinnes et al., in contrast, provides only datasets where each model graph nearly always includes a certain pattern [73], which could hinder the performance and the evaluation of the Graph Variational Autoencoder. Additionally, no extra implementation is needed to count the true values of the subgraph isomorphisms contained in a certain model graph. For our project, we have generated graphs mimicking the characteristics of the synthetic dataset by Tinnes et al., including patterns having similar characteristics to the edit operation patterns. When we say "similar," we refer to aspects such as the patterns possessing an approximately equal number of nodes and edges, with $+/-$ 1-2 elements, and matching the average count of incoming and outgoing edges. The main difference lies in the labels for nodes and edges, which are integers rather than attribute names. Despite this difference, the number of potential labels remains comparable. Furthermore, we have adhered to similar restrictions on the size of graphs as outlined in the dataset by Tinnes et al. Although we can control several factors, there are still uncontrollable elements such as the distribution of graph sizes. The generated datasets and their main characteristics are given in Table 5.1.

Table 5.2: Details of the used datasets from Learning Domain-Specific Edit Operations from model repositories with Frequent Subgraph Mining

Dataset name	**Max size**	**Description**	**# Graphs <= 10**	**# Graphs**
Dataset Subgraphcount filtered 1	10	Dataset created by the graph generator from Liu et al. [53]. Generator parameters are aligned with Tinnes et al.'s synthetic dataset [73] (see Table A.3). These datasets focus on one pattern only.	2000	$\sim$ 50% do not include the pattern, 35% include the pattern once
Dataset Subgraphcount filtered 2-4	10	Same as above	2000	> 90% of graphs don't include the pattern
Dataset Subgraphcount more frequent 5-8	10	Same as above	2000	> $\sim$ 48% of graphs don't include the pattern, $\sim$ $35 - 38$% include the pattern once

5.3 OPERATIONALIZATION

5.3.1 *Experiment 1—Pilot Study*

As an initial stage of our research, we conducted a pilot study, which served multiple purposes, including debugging code, adjusting to our domain, and to overall get a basic idea and tendency of what was working well and what was not. During the initial experiment, we also did a small manual examination of the generated graphs to see if they seemed realistic at first sight. In the initial investigation, the primary objective was to get a basic understanding of the impacts of various components with respect to the learning process in general and also the generation of SCG (dataset by Tinnes et al. Section 5.2.1). This experiment focuses on answering RQ1. We, for example, changed the implementation's readout function to Softmax.
Readout layers produce the final output of the neural network from the internal representation of the previous layers. There are different types to choose from, for example, linear layers pass through the values unchanged, which was used in the implementation we based our implementation on. We exchanged the readout layer with softmax, which is a generalization of the sigmoid activation function and well known to be appropriate for categorical outcomes. As a result, the softmax, in our case, returns the probability distribution of the edge types and the node types. Softmax further guarantees that these probabilities sum to one. Then, to produce the final graph, for each edge and node, the class with the highest probability is selected. Other functions, like ReLU, sigmoid, and tanh are not considered since these do not seem suitable in our case. The ReLU activates a node only if the input is above a certain quantity. When the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable [32]. The sigmoid activation function converts variables into probabilities between 0 and 1, most of its output will be very close to 0 or 1 [32]. It can reduce extreme values in data, but it may get "stuck" with outputs close to 0 or 1, hindering the learning process. The tanh function maps inputs to [-1, 1], similar to sigmoid. It's used in hidden layers and less prone to getting "stuck" as its outputs are between -1 and 1 [32]. We also changed the machine learning architecture of the system (replaced encoder layers). Further, we shortly compared the performance we regard to the number of epochs, and the size of the datasets and tried out different hyperparameters. However, these results may not be generalizable, due to the specificity of the chosen hyperparameters, the initial seed, and the dataset chosen. Still, this was a necessary step to help reduce the search space for further investigations and consequently necessary to answer both RQ1 and RQ2.

5.3.2 *Experiment 2*

To answer research question RQ1, in this experiment, we study the influence of certain factors of the system with regard to the GVAE's ability to reconstruct valid SCG. We compare three different layer architectures Graph Attention (GAT), Transformer (TRF), and GIN, optimize them with regard to their number of layers and hyperparameters, and try out different graph formats. These factors are further explained in paragraph Section 5.3.2.2. For evaluation purposes, we use the synthetic data generated by Tinnes et al. [73]. We assume

that the data mirrors a real-world dataset. For each of the three layers, GAT, TRF, and GIN in combination with each graph format (further explained in Section 5.3.2.2), we perform hyperparameter optimization, resulting in six combinations. We conduct hyperparameter tuning with the help of Optuna on a small dataset (Dataset Small 1, Table 5.1). We perform 50 trials with a maximum number of epochs of 1000 (in each epoch the complete train dataset is included). We implemented an early stopping mechanism, which prunes a trial if its performance does not seem promising with regard to its validation loss. For pruning purposes, we use the Median Pruner that stops a trial at a certain step if the trial's best intermediate result is worse than the median of intermediate results of previous trials at the same step. The scope of hyperparameters we tested is given in Table 5.3.

Table 5.3: Preselection of Hyperparameters and their suggested values

Hyperparameter	**Suggested Values**
Learning Rate	1e-5, 5e-4, 1e-4, 5e-3, 1e-3
Batch Size	32, 64, 128
KL Beta	0.2, 0.3, 0.4, 0.5, 0.6
Encoder Size	64, 128, 256, 512
Latent Size	64, 128, 256, 512
Decoder Size	32, 64, 128, 256
Number of Layers	1, 2, 3
Dropout	0.0, 0.2, 0.4, 0.6
Regularization	0.0, 0.2, 0.4, 0.6

Subsequently, employing the optimal hyperparameters identified for each combination, we assessed the methods across various datasets (refer to Table 5.1). The evaluations were conducted using random seeds $s \in \{0, 1, 42\}$ to enhance internal validity and also ensure reproducibility of our experiments. We fixed the random seed for the PyTorch, NumPy, and PyTorch Geometric libraries. By setting the seed, we made sure that any stochastic operations or random choices made by these libraries would be the same across different runs of the experiments. This includes the initialization of neural network weights, the shuffling of datasets, the random partitioning of data into batches, and any other operations that rely on random number generation Each dataset was partitioned into train, test, and validation sets, maintaining a ratio of 75%-12.5%-12.5%, achieved by shuffling the graphs using the specified seed. We investigate the system's performance after 100 train epochs by generating new graphs (sampling 100 graphs each time). We compare the system performance with respect to the following metrics.

5.3.2.1 *Metrics*

To address question RQ 1, we compare the number of correct edges in the generated graphs. In general, meta models can also include additional constraints besides having correct edges, like quantifiers, that we will not consider yet.

Definition 20 ***Meta Model Valid Edge.*** *An edge is defined as being correct according to the meta model if the edge is also present in the meta model. This includes that there is an edge with the same label in the meta model and source node label and the target node label of the corresponding meta model edge agrees with the labels of the source and target node.*

To better account for a different number of edges in different graphs, we consider the portion of correctly identified edges. If there are no edges, the validity is zero.
Further, we will have a look at the overall distribution of the node types, edge types, graph size, and number of incoming/outgoing edges. Those distributions are then compared to the distribution of the original and test dataset, the closer these are to each other, the better. First, we will have a look at visual representations of the generated graphs and distributions of the factors named above, and then we will continue comparing the distributions' total variation distance. Total variation distance is a measure of the difference between two probability distributions. The value ranges from 0 to 1, a value of 0 means that the distributions are identical.

Definition 21 ***Total Variation Distance [50].***

$$\delta(P, Q) = \frac{1}{2} \sum |P(x) - Q(x)| \tag{5.1}$$

Low values indicate a stronger resemblance between the original distribution and the distribution with respect to a graph invariant. Therefore, lower values are considered preferable.

5.3.2.2 *Independent Variable*

To address RQ1, we had a closer look at several factors and their influence on the overall performance. We investigate the performance with regard to different layers. Similar to other work using convolutional layers [68], we investigate their performance in our work. Since many related papers suggested the usage of Graph Isomorphism Networks (GIN), we also replace the encoder layers with GIN convolutional layers [35, 53, 78]. GIN layers were introduced by Xu et al. [78] and are based on the Weisfeiler Leman test. The Weisfeiler Leman test is a graph isomorphism test that compares the structural similarities between graphs [24]. It iteratively refines the label of the nodes based on their current label, the labels of their neighbors, and the labels of the edges connected to this node [24]. The test terminates either at a fixed point or after a certain number of iterations [24]. Following the relabeling process, a frequency count of the node labels for each graph is generated. The Weisfeiler-Lehman test determines the isomorphism of the two graphs by comparing these histograms. If the histograms are identical, the graphs might be isomorphic. Conversely, if the histograms differ, it indicates that the graphs are not isomorphic. The Weisfeiler-Lehman test is not infallible and may sometimes fail to accurately differentiate between all non-isomorphic graphs falsely labeling them as isomorphic. Theoretically, GIN layers can differentiate various graph structures as effectively as the Weisfeiler-Lehman test, making them ideal for tasks sensitive to minor structural variations in graphs [78]. We further investigate Graph Attention Networks, which were introduced by Veličković et al. [74]. To

summarize, using GAT layers has the key advantage of using attention mechanisms, allowing for weighted node influence during neighborhood information aggregation. This enhances performance on tasks involving complex graph structures or tasks where certain nodes bear greater relevance [74]. Finally, we also use transformer convolutional layers, which similar to GAT utilize self-attention to prioritize different sections of the input during individual element processing. The transformer convolutional layer for graph data is a combination using the principle of Graph neural network (GNN)s for feature propagation and LPA for label propagation. In summary, while both use attention mechanisms, GAT focuses on local neighborhood attention. Transformer convolutional layers are more complex architecture aiming to capture more complex relationships in graph data [67]. We use TransformerConv, GINEConv, and GATConv implementation from the torch_geometric.nn library fromt the corresponding papers [39, 67, 74]. In all settings, we specifically used implementations that incorporated edge features.
Due to the fact that each graph has a variable size of unordered nodes and each node in a graph has a different number of neighbors, a consistent way to provide these graphs as input to the GVAE needs to be given. Existing research has indeed proposed a variety of innovative solutions to accommodate varying graph sizes and structures [68, 75]. Regrettably, it is not within our means to overcome this particular constraint. The necessity for the adjacency matrix to have a fixed size is well-documented in the literature [20, 68, 75]. To the best of our knowledge, adhering to this requirement is indispensable, especially if our objective is to make one-shot predictions of graphs. In this part of research question RQ1, we investigate whether the format of the input graphs has a significant impact on the overall performance. Therefore we consider several formats and limit the size of the considered graphs up to a certain value $n_{max} \in \{10\}$.

Definition 22 ***Complete Graph Format.*** *Each input graph with $n < n_{max}$ is filled up such that $n = n_{max}$, where the node labels are defined with an additional type "None". Further, the graph is filled up with edges such that each pair of graph vertices is connected by an edge of type "None".*

Later, after the generation of new graphs, this step is reversed. Using the complete-graph format is compared to just giving the graphs as input to the GVAE. In the experiments, we often abbreviate using the complete-graph format with "A" (Augmented) and the non filled up graphs with "U" (Unchanged). By "unchanged," we refer to Digraphs that are not modified, thereby representing labeled graphs that align with the original Simple Change Graphs. For both formats, we then apply the Sentence Encoding Function for the GVAE, as detailed in Equation 15. These Digraphs are then given to the GVAE as input (more in Equation 4.2.2). Finally, for the loss function, both formats are transformed into the matrices A, V, and E in accordance with the guidelines specified in Definition 2.2.1.

5.3.2.3 *Baselines for RQ1*

We define the baseline RandomGraphBaseline as follows. We create random graphs by selecting their size randomly and selecting for each node a label from all valid node types randomly. Then for each pair of nodes, it is randomly decided whether to add a directed edge from the first node to the second node. If so the edge type is randomly chosen from all available edge types. The exact code for the baseline is given in Listing A.1.

Further, we also specify ValidGraphBaseline, which only includes valid graphs according to the meta model. This baseline differs from the first one in the creation of edges. For each possible valid edge of a certain type between two nodes, it is randomly decided if the edge should be added or not. More detail on this baseline can be found in Listing A.2.

5.3.3 *Experiment 3, 4, 5*

To answer RQ2.1, RQ2.2, and RQ2.3, we solely focus on the latent space of the GVAE. We train a classifier in a supervised manner, that given the latent space, predicts whether/ how often a certain pattern is present in an input graph. One disadvantage of the dataset by Tinnes et al. is the over-application of the edit operation in model differences, such that in nearly every model change every edit operation is applied, making the dataset unsuitable for RQ2 [73]. Considering the challenge of accessing the true values of how frequently an edit operation is applied in a model change in general, which is also present in the work by Tinnes et al. [73] our focus shifts to a dataset derived from Graph Theory, specifically counting subgraph isomorphisms [53]. As a training set for the GVAE, we use a dataset that we create by the graph generator provided by the work of Liu et al. [53] (Section 5.2.2). The dataset already includes the true values, so whether and how often a certain pattern is contained in a graph. We opted for this particular setting because it allows us to better observe the structure of the dataset. For example, it ensures the presence of multiple graphs that do not contain a specific pattern, thereby enhancing our ability to train the classifier more effectively. Further, the generator by Liu et al. can be used to adjust the training dataset such that the distributions like edge and node number per graph are similar to the dataset of the SCG. It should be highlighted that our work sets itself apart from Liu et al. who focused on Graph Neural Networks but did not incorporate dimensionality reduction [53]. We use the best performing machine learning architectures from RQ1 with their fixed hyperparameters and train the GVAE on the dataset explained above.
For Experiment 3, we use a binary classifier that should tell whether or not a certain pattern is included in a given graph. We choose the three best performing approaches from RQ1 and compare them with regard to their RQ2 performance. We consider not only the best approach but also two additional approaches to check whether they may perform better with regard to the RQ2 even though the results of RQ1 were less promising. For each approach, we tune the classifiers using a selection of hyperparameters on a randomly selected dataset for simplicity, specifically, Dataset Subgraphcount filtered 1 from Section 5.2.2. The scope of hyperparameters we tested is given in Table 5.4. We test the classifier on 3 seeds again and 4 datasets from the generator of Section 5.2.2, leading to a total number of 12 runs per approach. We specifically chose the setting of performing hyperparameter tuning first to save computational resources and better generalisation across different seeds and therefore initializations. The goal of this experiment is to find out whether in general RQ2.1 can be answered with "yes". Further, with this experiment, we also want to answer part of RQ2.2, specifically what impact the encoder layers of the Graph Variational Autoencoder have on

the performance of the classifier.
For Experiment 4, we fixed the approach used (best performing of Experiment 3) and investigated how different factors influence the performance of the classifier with respect to the specified metrics (Section 5.3.2.1). In the Experiment, we extended the dataset collection with Dataset Subgaphcount morefrequent 5-8 (Section 5.2.2) (E4.1). We further choose other datasets for the initial hyperparameter tuning (E4.2). For Experiment 3 and Experiment E4.1 and E4.2, we added the pattern latent representation to the input of the classifier, hoping to improve its performance with the additional information. In E4.3, we explore the significance of this factor in determining the classifiers' high performance. With Experiment 4 we would like to mainly answer RQ2.2. But this experiment gives also more specific insight into answering RQ.2.1. In Experiment 5, we investigate if we can extract how often a pattern is included in a graph. We fixed the output of the classifier to a maximum of five classes, since the true values for all other classes are too low. With this experiment, we would like to answer RQ2.3. Again we tune the non-binary classifiers with respect to a selection of (hyper-) parameters on a randomly selected dataset. Afterward, we check the performance with respect to certain metrics.

Table 5.4: Hyperparameters and their suggested values for classification

Hyperparameter	**Suggested Values**
learning_rate	1e-5, 5e-4, 1e-4, 5e-3, 1e-3
batch_size	[1, 2, 4, 8, 16,] 32, 64, 128, 256
hidden_size	[2, 4, 8, 32,] 64, 128, 256, 512
dropout	0.0, 0.2, 0.4, 0.6
probe_type	linear, mlp1, mlp2
regularization	0, 1e-5, 1e-4, 1e-3, 1e-2

5.3.3.1 *Metrics*

For classification, we take the following metrics into account.

Definition 23 ***Accuracy*** *measures the proportion of correct predictions over the total number of predictions [82].*

$$Accuracy = \frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions} \tag{5.2}$$

Accuracy only gives a limited view of the performance of the machine learning model, especially in the case of an imbalanced dataset, a high accuracy may falsely indicate a good performance. Further, it does not differentiate between false positives and false negatives. To further specify the metrics Precision and Recall, the following Definitions are required.

Definition 24 ***True Positives (TP)*** *for a given class, refers to the number of samples where the classifier correctly predicts this specific class [82].*

Definition 25 ***False Positives (FP)*** *for a given class, refers to the number of samples incorrectly classified as being of this class [82].*

Definition 26 ***True Negatives (TN)*** *refers to the number of samples correctly classified as being not of this class [82].*

Definition 27 ***False Negatives (FN)*** *for a given class, refers to the number of samples incorrectly classified as not being of this class. So the samples were actually of the given class but were not predicted as such [82].*

Definition 28 ***Precision*** *or* ***Specificity*** *measures the proportion of true positives among all positive predictions.*

$$Precision = \frac{TP}{(TP + FP)} \tag{5.3}$$

Definition 29 ***Recall, Sensitivity*** *or* ***True Positive Rate*** *measures the proportion of actual positive instances that the classifier correctly predicts as positive*

$$Recall = \frac{TP}{(TP + FN)} \tag{5.4}$$

Due to the fact that a non-binary classifier is used in RQ 2.2, the values per class are averaged to receive one final value. The F1-score is a combination of Precision and Recall ranging from 0 to 1. Higher values indicate better machine learning model performance. F1-score is usually better suitable than accuracy especially if classes are unevenly distributed.

Definition 30 ***F1-score.***

$$F_1 = 2 \cdot \frac{Precision \times Recall}{Precision + Recall} \tag{5.5}$$

[82]

The Receiver Operating Characteristic (ROC) is a visual representation of the tradeoff between sensitivity and specificity. The ROC curve starts at (0,0) and ends at (1,1). The better the curve passes the top left corner, the better the classifier performs. The AUROC metric is based on the ROC curve, measuring the area under the curve. It ranges from 0 to 1, where higher values indicate better classification performance [82], a closer definition is given in the work by Bardley et al. [13].

Definition 31 ***Matthews Correlation.***

$$MCC = \frac{(TP \times TN - FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.6}$$

It has the advantage that all four categories of the confusion matrix are taken into account and can also deal with imbalanced datasets.

For RQ 2.3, we again use the same metrics as in RQ 2.1 and RQ 2.2. Still, using these metrics is not sufficient due to the fact that firstly, our dataset is imbalanced and secondly, we only consider a limited amount of data in the form of some samples for evaluation. Consequently, it can not be guaranteed that our samples are sufficiently representative to infer general results about our method. We address these problems in Definition 19 and Section 5.4.

5.3.3.2 *Baselines for RQ2*

As a first baseline, we choose GraphSizeBaseline, as a baseline that given the graph size and pattern size should determine whether/ how often a pattern is included in the graph. Hewitt et al. [37] further provides great insights on the shortcomings of pruning tasks, for example, that classifiers tend to memorize the task at hand. Therefore they introduce Selectivity as the difference between the control task accuracy and the task accuracy. As a control task, they randomly assign classes of the test set such that only memorization could perform above random guessing. With this information, we introduce MemoryBaseline, which given the test and validation set, randomly shuffles the true values.

5.4 RESULTS

5.4.1 *Experiment 1*

As discussed in Section 5.3.2.1, we will investigate the distribution of node types, edge types, incoming/outgoing edges, and graph size. To compare these distributions, the total variation distance of probability measures is used [50]. Low values suggest a closer match between the original distribution and the distribution relating to a graph invariant. Hence, smaller values are more desirable.
Comparing Graph Isomorphism Encoder Layer combined with Augmented Input Graph Setting (GIN-A) and Graph Isomorphism Encoder Layer combined with Unchanged Input Graph Setting (GIN-U) (Table 5.5), the total variation distance between the original graph sizes and generated graph sizes for GIN-A was significantly lower, along with slightly reduced values for the edge type invariant. However, GIN-U performed better in terms of node types, while GIN-A slightly outperformed GIN-U in incoming/outgoing metrics. A similar pattern was observed when comparing the softmax counterparts with regard to the graph layout used. In the softmax setting, TRF-U were better in each category than the Transformer Layer Combined with Augmented Input Graph Setting (TRF-A). With the linear setup, TRF-U were only better with regard to node types, incoming and outgoing edge distribution.
Moreover, the use of GIN-U, GIN-A, TRF-U, and TRF-A with a linear readout layer performed remarkably worse than softmax with regard to nearly all invariants. Moreover, the use of GIN-U, GIN-A, TRF-U, and TRF-A with a linear readout layer performed remarkably worse than softmax with regard to nearly all invariants. Upon examining Table 5.6, it becomes evident that methods employing the linear setup do not necessarily exhibit consistent learning behavior. While some values slightly improve, others deteriorate. In contrast, methods utilizing softmax consistently show better values after 1000 epochs.
When comparing Table 5.5 and Table 5.6, we see that GIN initially learns faster and outperforms TRF. However, TRF eventually overtakes GIN, performing slightly better as more epochs pass. Additionally, when increasing the dataset size, we see that the learning process significantly improves, demonstrating the positive effect of a larger dataset on model learning and performance (Table 5.7).
In general, the graph size invariant emerged as the most challenging measure to optimize. This first investigation helped us fix bugs, which are not explicitly mentioned here and has helped us narrow down the search space. Due to the fact the methods using softmax performed much better, we focused on using it as a readout function.

5.4.2 *Experiment 2*

For each remaining combination from Section 5.4.1 of the network architecture and the graph format, we perform the evaluation on 3 seeds and 8 datasets, leading to a total number of 24 runs per approach.

Table 5.5: Results of the initial experiment after 100 epochs and with a small dataset (∼ 800 graphs), we used the total variation distance (Definition 21) as a metric

Method/Invariant	graph size	node types	edge types	incoming edges	outgoing edges
GIN-U L.	0.87	0.45726	0.64107	0.68745	0.66411
GIN-A L.	0.59513	0.61569	0.5	0.69118	0.67157
TRF-U L.	0.99880	0.37815	0.65177	0.48487	0.36892
TRF-A L.	0.95513	0.65490	0.5	0.69118	0.671569
GIN-U S.	0.85513	0.13781	0.09593	**0.23448**	**0.15468**
GIN-A S.	**0.63513**	**0.12127**	**0.03696**	0.30601	0.17250
TRF-U S.	0.89513	0.20162	0.18935	0.27713	0.16608
TRF-A S.	0.95513	0.44412	0.64107	0.35784	0.33578

Table 5.6: Results of the initial experiment after 1000 epochs and with a small dataset (∼800 graphs), , we used the total variation distance (Definition 21)

Method/Invariant	graph size	node types	edge types	incoming edges	outgoing edges
GIN-U L.	0.51513	0.58642	0.89983	0.48333	0.51693
GIN-A L.	0.63	0.68021	0.80467	0.48333	0.38585
TRF-U L.	0.7	0.57148	0.89983	0.48333	0.35339
TRF-A L.	0.92	0.64333	0.5	0.69118	0.67157
GIN-U S.	0.19949	0.06201	0.05442	**0.06595**	0.16833
GIN-A S.	0.48000	0.05782	0.03270	0.16562	0.11838
TRF-U S.	**0.10513**	**0.04162**	0.03306	0.16268	**0.02609**
TRF-A S.	0.32513	0.05108	**0.026482**	0.20006	0.09883

Table 5.7: Results of the initial experiment after 100 epochs and a larger dataset than the initial one (∼ 2000 graphs), we used the total variation distance (Definition 21)

Method/Invariant	graph size	node types	edge types	incoming edges	outgoing edges
GIN-U S.	0.36371	**0.07429**	0.07849	**0.14464**	0.07512
GIN-A S.	0.64277	0.11085	**0.06434**	0.18915	0.08644
TRF-U S.	**0.29628**	0.10311	0.10610	0.18731	**0.05713**
TRF-A S.	0.55679	0.08178	0.08570	0.21505	0.10795

5.4.2.1 *Generalizability with Respect to Seeds and Datasets*

We investigate first, whether the approaches with fixed hyperparameters are able to run across different seeds and datasets. The unsuccessful runs yielded NaN values either in the initial stages or after a few epochs. Potential issues leading to this phenomenon include overly high learning rates leading to exploding gradients, or an unsuitable batch size contributing to model instability.

Table 5.8: Generalizability with respect to seeds and datasets (Figure 2.2.1)

Method	**Number of failed runs**
TRF-U	0
TRF-A	8 (+3 early termination)
GIN-U	0
GIN-A	4
GAT-U	1
GAT-A	0 (+1 early termination)

Approaches that are mainly stable across different datasets and seeds are TRF-U, GIN-U and Graph Attention Layer combined with Augmented Input Graph Setting (GAT-A), and Graph Attention Layer combined with Unchanged Input Graph Setting (GAT-U). Upon closer analysis of the failed runs, no patterns have emerged regarding the seed and dataset.

5.4.2.2 *Meta Model Validity*

The average meta model validity overall seeds and datasets for each approach is depicted in Table 5.9. For each run, we consider the best performing epoch as the final result. We again, for each approach have 24 (3 seeds x 8 datasets) runs. Due to the failed runs ($run_{validity} = 0$), the overall validity of these approaches is lower. To check independently of the approaches' generalizability whether the basic meta model concepts were learned, the meta model validity without these failed runs are added in brackets.

Table 5.9: Meta Model Validity of generated graphs, the meta model validity without failed runs are added in brackets

Method	**Meta Model Validity mean**	**Meta Model Validity standard deviation**
TRF-U	0.9982	0.0025
TRF-A	0.6560 (0.9840)	0.4648 (0.0358)
GIN-U	0.9971	0.0038
GIN-A	0.8261 (**1.0**)	0.3790 (0.0)
GAT-U	0.9560 (0.9995)	0.2038 (0.0014)
GAT-A	0.9932	0.0172 (0.0172)
RandomGraph Baseline	0.0468	0.0021

We see that overall, all approaches perform very well in producing valid graphs only. For each approach a, we consider the following Null hypothesis.

Null hypothesis for approach a: H_0
The generated graphs by the approach a do not demonstrate a greater meta model validity than randomly generating graphs (RandomGraphBaseline from Section 5.3.2.3) on average.

We set the significance level in all tests $\alpha = 0.01$ and perform a right tailed t-test. We can see that the mean meta model validity for all approaches a is significantly higher than the mean of our baseline (for t-statistic and p-values see Table A.1) and therefore can reject the Null hypotheses for $a \in \{\text{TRF-U}, \text{TRF-A}, \text{GIN-U}, \text{GIN-A}, \text{GAT-U}, \text{GAT-A}\}$.
Due to the fact that nearly all approaches had a meta model validity of 1.0, we did a closer manual analysis of the graphs that did not have a validity of 1.0, to possibly check for patterns and the reason behind this. Through the analysis, we found out that indeed sometimes edges were not correct according to the meta model, but the exact combination of labels seems random.

5.4.2.3 *Distributions*

We compare the mean values of the total variation distance of each graph invariant and approach (Table 5.10, Table 5.11). As discussed in Section 5.3.2.1, we will investigate the distribution of node types, edge types, incoming and outgoing edges, and graph size. To compare these distributions, the total variation distance is used. The lowest values per invariant are highlighted.

When examining the results based purely on performance, excluding the failed runs referenced in Section 5.4.2.1, it's evident that TRF-U exhibits relatively strong performance. It ranks as the best with respect to edge types, node types, and graph size, and secures the second-best position for incoming and outgoing edges. While GAT-U underperforms in comparison, all other approaches still manage to deliver commendable performance. To conclude the overall performance of each method, we also included the failed runs in the calculation (Table 5.10), highlighting that TRF-U achieves the best overall performance (see also Table 5.9). Followed by GIN-U and GAT-A. TRF and GIN perform better with the "unchanged" graphs setting, while GAT performs better with the "augmented" setting with respect to the overall performance, if including their generalizability. With regard to the invariants, the edge type distribution (mean value: 0.102464), node type distribution (0.095145), and number of outgoing edges (0.118611) are better learned, followed by the number of incoming edges (0.169378). The graph size distribution exhibits the least similarity to the original distribution (0.363405).

The box plots in Figure 5.7 offer a nuanced view of the total variation distance values. Wider sections indicate a range of values, while narrower sections suggest less variability. TRF-U, GIN-U, GIN-A, and GAT-A exhibit small values with a dense distribution, especially for edge types, where values are mainly below 0.1. In contrast, GAT-U and TRF-A display rather skewed distributions. For the graph size invariant, the best methods are TRF-U and GIN-A. All approaches show suboptimal results for the graph size distribution. The incoming edge invariant box plots are more on the left compared to the outgoing edge invariant. The whiskers in the box plots indicate the range where most values fall. The caps at the ends signify the minimum and maximum data values.

Null hypothesis for approach a, invariant i and baseline b: $H_0^{a,i,b}$
The total variation distance with regard to the graph invariant distribution i of approach a is not significantly less than the total variation distance of the baseline b.

We formulate the Alternative hypothesis $H_1^{a,i,b}$ for approach a and invariant i as the total variation distance of the graph invariant distribution i for approach a is significantly less than that of the baseline b.

We set the significance level in all tests $\alpha = 0.01$ and perform left-tailed (one-sided) t-test. For invariant $i = edge\_types$, we can reject the Null hypothesis for all approaches and both baselines. For the number of outgoing edges and $b = RandomGraphBaseline$, all $H_0^{a,outgoing\_edges,RandomGraphBaseline}$ are rejected. For $b = ValidGraphBaseline$, the Null hypothesis was not rejected for GAT-U, TRF-A, and GIN-A. For the number of incoming edges, all Null hypotheses $H_0^{a,ingoing\_edges,RandomGraphBaseline}$ could be rejected. For $b = ValidGraphBaseline$ all Null hypotheses but TRF-A could be rejected.

Table 5.10: Mean value of total variation distance per graph invariant and approach, lowest values are highlighted. The last row is the mean of all approaches with respect to a certain category (baselines are excluded)

Method/Invariant	edge types	node types	graph size	incoming edges	outgoing edges
TRF-U	**0.027917**	**0.036250**	**0.189583**	**0.102500**	0.052083
TRF-A	0.197500	0.200000	0.480833	0.255417	0.212917
GIN-U	0.053750	0.073333	0.433333	0.167500	0.098750
GIN-A	0.106957	0.108261	0.267391	0.157826	0.123913
GAT-U	0.187826	0.112609	0.583043	0.202609	0.176087
GAT-A	0.040833	0.040417	0.226250	0.130417	**0.047917**
RandomGraph Baseline	0.401304	0.212609	0.560435	0.786522	0.647826
ValidGraph Baseline	0.481739	0.208261	0.533043	0.281739	0.166522
mean of approaches	0.102464	0.095145	0.363405	0.169378	0.118611

Table 5.11: Mean value of total variation distance per graph invariant and approach, lowest values are highlighted. The last row is the mean of all approaches with respect to a certain category (baselines are excluded)

Method/Invariant	edge types	node types	graph size	incoming edges	outgoing edges
TRF-U	**0.027917**	**0.03625**	**0.189583**	0.1025	0.052083
TRF-A	0.04625	0.1	0.480833	0.133125	0.117222
GIN-U	0.05375	0.073333	0.433333	0.1675	0.09875
GIN-A	0.069524	0.070952	0.267391	**0.085789**	0.106818
GAT-U	0.173636	0.095	0.583043	0.189091	0.161364
GAT-A	0.040833	0.040417	0.22625	0.130417	**0.047917**
ValidGraph Baseline	0.482917	0.208333	0.53625	0.282083	0.16625
RandomGraph Baseline	0.400833	0.2125	0.559167	0.786667	0.647917

5.4.3 *Experiment 3*

For RQ 2, we only consider the 3 best performing approaches from RQ1 (GIN-U, GAT-A, TRF-U). The best performance was achieved by approach TRF-U. We test the classifier on 3 seeds again and 4 datasets from the generator of Section 5.2.2, leading to a total number of 12 runs per approach. We balanced the dataset equally among the two different classes, such that the accuracy metric is a good first indicator for the performance of the classifier. The performance of the binary classifier is given in Table 5.12. The highest accuracy is again achieved by the TRF-U approach (91,2 % on the validation set and 85,7 % on the test set). Also Precision, Recall, and F1 (harmonic mean between these two values) are similarly high. The corresponding baselines are shown in Table 5.13.

We formulate the following Null hypotheses.
$H_0^{\text{GIN-U}}$: The classifier that is based on the GIN-U Graph Variational Autoencoder does not demonstrate a greater accuracy than the GraphSize Baseline on average.
$H_0^{\text{GAT-A}}$: The classifier that is based on the GAT-A Graph Variational Autoencoder does not perform better than the GraphSize Baseline with respect to its accuracy.
$H_0^{\text{TRF-U}}$: The classifier that is based on the TRF-U Graph Variational Autoencoder does not perform better than the GraphSize Baseline with respect to its accuracy.
We have chosen accuracy here as an indicator since we have a balanced dataset and again performed one-sided t-test.
We choose $\alpha = 0.01$. We can reject all H_0^a, where $a \in \{\text{TRF-U}, \text{GAT-A}, \text{GIN-U}\}$ such that they are significantly better than the baseline with respect to its accuracy.

We further formulate $H_0^{\text{TRF-U, GAT-A}}$ as there is no significant difference in the accuracy between the two approaches TRF-U and GAT-A. Specifically, TRF-U is not significantly better than GAT-A.

We further formulate $H_0^{\text{TRF-U, GIN-U}}$ as there is no significant difference in the accuracy between the two approaches TRF-U and GIN-U. Specifically, TRF-U is not significantly better

than GIN-U.

When using $\alpha = 0.01$, we can reject $H_0^{\text{TRF-U, GAT-A}}$ but not $H_0^{\text{TRF-U, GIN-U}}$. But by changing $\alpha = 0.05$, both hypotheses can be rejected.

Table 5.12: Performance of the different approaches with regard to different architectures (using CrossEntropyLoss with ReLU and including the pattern as input)

Method	Data	Accuracy	Precision	Recall	F1	MCC	AUROC
TRF-U	test	**0.857400**	**0.861558**	**0.864575**	**0.855108**	**0.726058**	**0.935550**
	val	**0.911992**	**0.910108**	**0.913217**	**0.909467**	**0.823250**	**0.955600**
GAT-A	test	0.768567	0.774350	0.765842	0.761175	0.539808	0.842342
	val	0.793125	0.799625	0.785892	0.784500	0.584842	0.859850
GIN-U	test	0.775592	0.804008	0.775817	0.764608	0.576358	0.865525
	val	0.852392	0.867225	0.852342	0.848725	0.718533	0.902558

Table 5.13: Performance of the different approaches on the GraphSizeBaseline (using CrossEntropyLoss with ReLU and including the pattern as input)

Method	Data	Accuracy	Precision	Recall	F1	MCC	AUROC
TRF-U	test	0.549850	0.566300	0.549850	0.519600	0.114950	0.560350
	val	0.643310	0.677320	0.643310	0.628460	0.315690	0.659630
GAT-A	test	0.584467	0.598600	0.584467	0.562600	0.182233	0.579000
	val	0.647522	0.665978	0.647522	0.639111	0.312633	0.670033
GIN-U	test	0.594700	0.605900	0.594700	0.575700	0.200050	0.582650
	val	0.620000	0.664930	0.620000	0.598070	0.277500	0.658010

5.4.4 *Experiment 4*

For the second experiment of RQ2, we fixed the approach used and investigated, how different factors influence the performance of the classifier with respect to the Accuracy and other metrics.

5.4.4.1 *Experiment E4.1-E.4.2 Dataset and Hyperparametertuning*

In the second step of this research question, we further investigated the performance of our model on different data. Due to the evidence that the TRF-U approach seemed to perform best, we focused on this approach in the remainder of RQ2.1 and RQ2 in general. We extended the number of datasets to eight (Table 5.15). In these datasets, the test and validation accuracy were much lower. Due to the acquisition of further knowledge, we further extended the choice of hyperparameters to the ones given in Table 5.4 in brackets [37] and did the hyperparametertuning again. The chosen model was a linear classifier instead

of an MLP1 classifier. The resulting latent space was also much smaller and reduced from 64 to 4 (Table 5.14).

Table 5.14: Comparison of Hyperparameters (linear classifier and MLP1 classifier)

Hyperparameter	Linear classifier	MLP1 classifier
learning_rate	0.0005	0.001
batch_size_new	1	4
hidden_size	4	64
dropout	0.4	0.4
probe_type	linear	mlp1
regular	0	0.01

Table 5.15: Performance of TRF-U approach on Dataset Subgraphcount more frequent 5-8 (using CrossEntropyLoss with ReLU and including the pattern as input)

Method	**Data**	**Accuracy**	**Precision**	**Recall**	**F1**	**MCC**	**AUROC**
TRF-U	val	0.805725	0.800900	0.791250	0.795000	0.592025	0.887650
	test	0.780400	0.776312	0.760775	0.765625	0.536763	0.839087

Table 5.16 shows the metric values of the new classifier with its new hyperparameters on all datasets.

We formulate $H_0^{\text{TRF-U},linear}$ as the TRF-U approach does not perform significantly better than its corresponding baseline (Table 5.16) and see that this selection of hyperparameters also performs significantly better than the baseline ($\alpha = 0.01$).
By comparing Table 5.16, Table 5.15 and Table 5.12, we can already see this new selection of hyperparameters (linear classifier) in general performs better than the previous selection (MPL1 classifier).
Still, we formulate $H_0^{\text{linear, MLP1}}$ as the linear classifier is not significantly better than the MLP1 on average. We choose $\alpha = 0.01$ for the right-tailored t-test and can reject the Null hypothesis $H_0^{\text{linear, MLP1}}$.

Table 5.16: Performance of TRF-U approach on all datasets using a linear classifier and a smaller hidden size (CrossEntropyLoss with ReLU and including the pattern as input).

Method	**Data**	**Accuracy**	**Precision**	**Recall**	**F1**	**MCC**	**AUROC**
TRF-U	val	0.918232	0.891195	0.893711	0.89090	0.784374	0.958526
	test	0.902480	0.874540	0.865740	0.86942	0.740020	0.950900
GraphSize Baseline	val	0.600000	0.618181	0.600000	0.583419	0.217148	0.605529
	test	0.584467	0.598600	0.584467	0.562600	0.182233	0.579000

5.4.4.2 *Experiment E4.3 Pattern inclusion*

In Experiment E4.3, we check whether or not including the pattern latent presentation in the input data has a significant impact on performance. In Table 5.17, the pattern latent representation is not given to the classifier. We can see that the performance is less good than the implementation that includes the pattern (Table 5.16).
We formulate the Null hypothesis as including the pattern as an input to the classifier has no significant effect on the classifier's performance. We choose $\alpha = 0.01$ and can reject the Null hypothesis. Not including the pattern performs significantly worse than including it.

Table 5.17: Performance of TRF-U approach on all datasets using linear classifier but the pattern is not included in the input(CrossEntropyLoss with ReLU)

Method	Data	Accuracy	Precision	Recall	F1	MCC	AUROC
TRF-U	val	0.772767	0.805783	0.797200	0.753367	0.602717	0.895650
	test	0.689861	0.739561	0.741800	0.675783	0.481033	0.791350

5.4.5 *Experiment 5*

We investigate if we can extract how often a pattern is included in a given graph. Therefore we performed hyperparameter tuning again and evaluated our approach on different datasets. The results are given in Table 5.18. We see that the accuracy dropped to 69.57% in the test set. The average precision and recall are 36.91% and 39.39% respectively. Our results of Experiment 5 showed values for Precision and Recall under 50% indicating that the classifier may be having difficulty distinguishing between some of the five classes. The F1 score, which balances precision and recall, is approximately 37.02 %, further reinforcing the previous point. The AUROC is 74.85%, suggesting a reasonably good ability to differentiate between classes. Comparing the values indicates potential difficulties in distinguishing among some of the five classes.

Table 5.18: Performance of TRF-U approach on all datasets using a non-binary linear classifier with a maximum of 5 classes

Method	Data	Accuracy	Precision	Recall	F1	MCC	AUROC
TRF	test	0.695738	0.369137	0.393858	0.370229	0.462825	0.748513
	val	0.728950	0.433342	0.453725	0.429767	0.531067	0.781821
GraphSize Baseline	test	0.192788	0.104667	0.163163	0.119654	0.003962	0.431250
	val	0.257013	0.184675	0.229050	0.174638	0.114742	0.499062

We formulate $H_{TRF-U,linear,non-binary}$ as the TRF-U method using a linear classifier performs worse on average than its GraphSize Baseline. With $\alpha = 0.01$, we can reject the Null hypothesis, TRF is significantly better than the baseline.

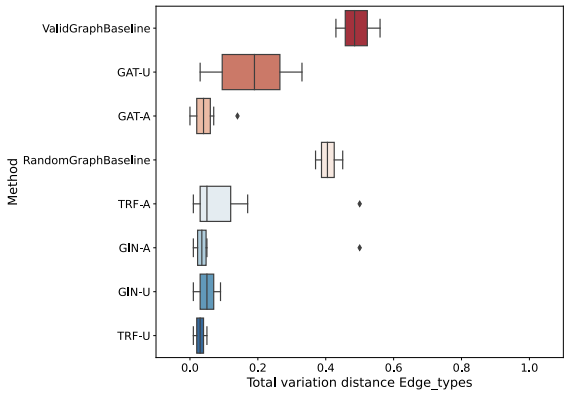

Figure 5.2: Box plot of Edge type total variation distance

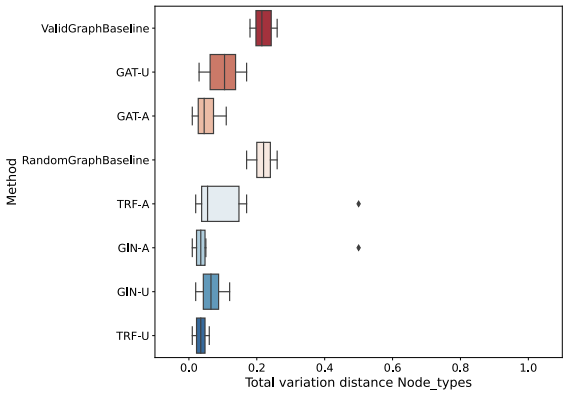

Figure 5.3: Box plot of node type total variation distance

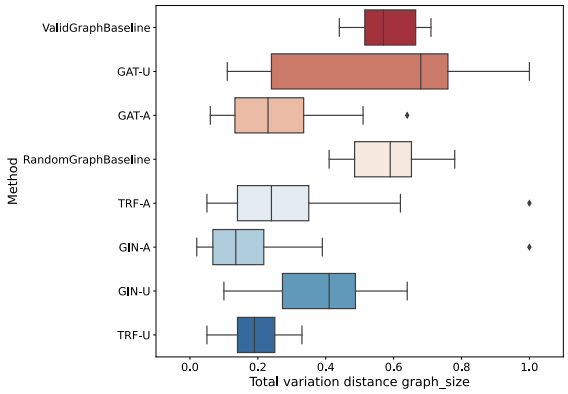

Figure 5.4: Box plot of graph size total variation distance

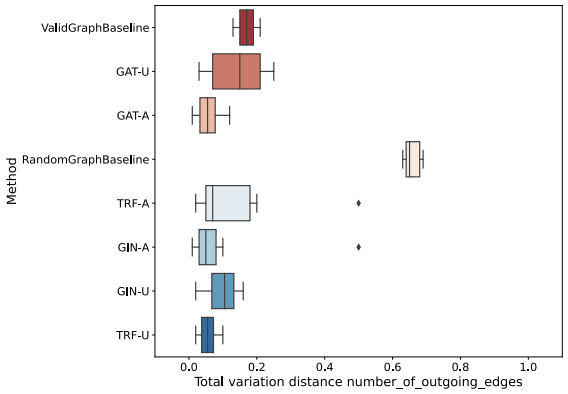

Figure 5.5: Box plot of total variation distance of outgoing edge distribution

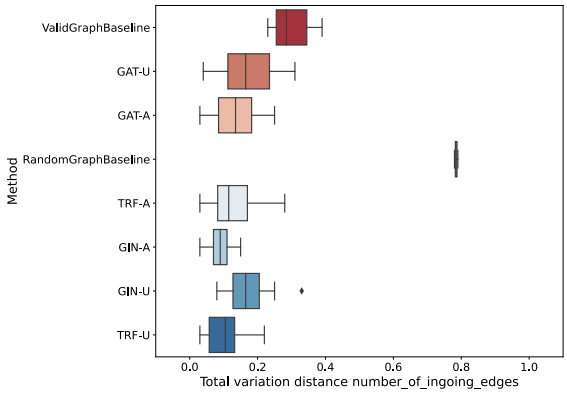

Figure 5.6: Box plot of total variation distance of incoming edge distribution

Figure 5.7: Box plots of different methods with regard to different graph invariants, failed runs are not included

6 DISCUSSION

6.1 RQ 1: CAN REALISTIC SIMPLE CHANGE GRAPHS BE GENERATED VIA A GRAPH VARIATIONAL AUTOENCODER?

Our results indicate that it is indeed possible to generate realistic SCG using a Graph Variational Autoencoder. Our evaluation across multiple seeds and datasets showed robust performance for several approaches (TRF-U, GIN-U). Additionally, the approaches were successful in producing graphs that are correct according to the meta model, with GIN-A achieving the highest average meta model validity of 100%. Furthermore, it was observed that these approaches are capable of generating graphs whose node types, edge types, and other graph invariants showed similar distributions to the original input graphs. Interestingly the graph size invariant showed the least promising results, which is not directly a bad sign indicating that the graph size was not learned. Instead, our input data had very discrete values with respect to the graph size, namely $n \in \{4, 7, 10\}$. Such that due to the smoothing properties of the GVAE the total variation distance values were rather high. Instead when comparing the mean values, we can see that they are very alike (Table 6.1).

Table 6.1: Mean values of the graph size

Method	Mean values original dataset	Mean value generated graphs
TRF-U	5.277784	5.440722
TRF-A	5.262864	4.839896
GIN-U	5.273744	5.774000
GIN-A	5.299557	5.084841
GAT-U	5.277105	6.141190
GAT-A	5.271678	5.643011

Relating back to the definition of RQ1, these findings support the potential of GVAEs in generating SCG that are "realistic". This implies that the networks have indeed learned to replicate commonly used elements in coherence with the implicit rules given by the meta model.

Answer to RQ1: Yes, our findings confirm the viability of using a Graph Variational Autoencoder for generating realistic Simple Change Graphs. The robust performance observed across various seeds and datasets underscores the generalisability of our approaches in generating type correct graphs with similar properties like the edit operation dataset.

6.1.1 *RQ 1.1: How does the neural networks architecture and input format affect the generation of SimpelChangeGraphs?*

The results of RQ1 indicate that the architecture of the neural network impacts the generation of SCG, but also depends highly on the graph input format chosen. By comparing different architectures, including TRF layers, GIN layers, and GAT layers in the encoder, we found out that the machine learning model's performance varies across these different setups. For instance, the TRF-U and GIN-A were particularly stable across different seeds and datasets. This suggests that these architectures generalize better across different datasets and random initializations. While the GIN-A method is effective at learning the meta model validity, it, unfortunately, has the downside of being somewhat unstable with respect to the dataset and initialization of the seeds. Regarding meta model validity, all approaches perform very well. Regarding the sampled graphs and their invariant distributions, the overall performance of the different approaches is also noteworthy. None of these perform much worse than all others. To conclude, all approaches perform comparably. The main difference, therefore, lies in their generalisability with respect to the input data. For a clearer overview of which approaches perform well overall, we set the meta model validity of the failed runs to zero. Also, we included the failed runs in the distributions. So, looking at everything TRF-U performs best. Anyway while TRF-U seems to work best, it could be also the case that the pre-selection of possible hyperparameters and the tuning itself was not that suitable for the GIN and GAT layers, such that we can not conclude that is definitely is better than the others, especially because they perform very similar with respect to validity and the total variation distance. We can only talk about a tendency here (more in Section 6.3).

Answer to RQ1.1 In summary, all approaches show comparable performance in generating Simple Change Graphs, with TRF-U slightly outperforming others. Especially in the stability across different datasets and seeds, we see noteworthy differences. However, this may be due to suboptimal hyperparameter tuning for GIN and GAT layers. We can see trends on which factors lead to better performance, however further investigation for a final statement would be required.

6.2 RQ2: CAN THE GVAE FROM RQ1 RECOGNIZE PREVIOUSLY APPLIED EDIT OPERATIONS?

6.2.1 *RQ2.1: Does the latent space of the GVAE capture information on whether or not a certain pattern is present within a given graph? (Binary Pattern Inclusion Task)*

In addressing RQ2, the three best-performing approaches from RQ1 were selected, namely GIN-U, GAT-A, and TRF-U. A series of experiments were then conducted under RQ2, where the impact of varying factors on classifier performance was investigated. The TRF method seemed to be the most effective, further reinforcing the initial findings. Table 5.16 shows an accuracy of over 91 % strongly indicating that this research question can also be answered with yes. Initially, we had a bigger hidden size in combination with an MP1 classifier being more prone to overfitting and memorization of the data. Especially the lower accuracy

values for the Dataset "Subgraphcount more frequent 5-8" (Table 5.15) support this claim. Later on, we found a more suitable combination, namely a linear classifier with a hidden size of four. We hope that this machine learning model is less prone to overfitting and memorization due to its less capacity to memorize. We have implemented a method to assess potential memorization, which is closely inspired by the work of Hewitt et al. [37]. Further evaluation of this would exceed the scope of the thesis though.

Answer to RQ2 and RQ2.1 Yes, our results compellingly suggest that the GVAE's latent space effectively captures the presence of specific patterns within a given graph. This capability suggests that the GVAE from RQ1 includes information about previously applied edit operations.

6.2.2 *RQ2.2: What are the main factors that influence the performance of the Binary Pattern Inclusion Task?*

Additionally, the experiments delved into dataset and hyperparameter tuning, which showed that the dataset chosen for hyperparameter tuning had an impact on the overall generalisability of the classifier on different datasets. This suggests that the machine learning model's chosen hyperparameters may be overfitting to the specific dataset they were tuned on. Later on, we found a more generalizable combination of hyperparameters performing well on all datasets. We also observe that adding the pattern's latent representation to the input of the classifier had a positive effect on the average accuracy.

Answer to RQ2.2 In summary, for RQ2.2, the key determinants affecting the Binary Pattern Inclusion Task's performance include the classifier's hyperparameter tuning, its hidden size, and the type of classifier used. Incorporating the pattern's latent representation into the classifier's input has been observed to enhance average accuracy.

6.2.3 *RQ 2.3: Does the latent space of the GVAE contain information regarding the frequency of a specific graph pattern within a given graph?*

Unfortunately, even though our approach performs significantly better than the baseline, we can not directly answer this question with yes. The findings from Experiment 5 revealed suboptimal scores for both Precision and Recall, suggesting challenges in the classifier's ability to differentiate among the five distinct classes. The AUROC instead is comparably high. In a multi-class setting, AUROC is the macro-average of the one-vs-all binary classification tasks for each class. There could be scenarios where the machine learning model is excellent at distinguishing some classes (leading to high AUROC) but poor for others, resulting in low average precision and recall. Given the characteristics of our dataset, there's a potential risk of memorization. While we were able for RQ 2.1 and RQ 2.2 to construct a very balanced dataset, where approximately the same number of graphs included and did not include the pattern, this was not possible for the non-binary tasks. It was much more

difficult to construct a dataset of graphs where a pattern occurred exactly x times in the graph.

> **Answer to RQ2.3** In response to whether the GVAE's latent space contains information on the frequency of specific graph patterns, our findings are inconclusive. The suboptimal Precision and Recall scores indicate the classifier's difficulty in distinguishing different classes. Further research is needed.

Given the affirmative response to RQ1 and RQ2 partly, we have successfully laid the groundwork for employing GVAE in mining edit operations and utilizing them for model completion. This work demonstrates that the usage of GVAE for mining edit operations is worth further investigation and exploration. It is also, as far as we know, the first of its kind.

6.3 THREATS TO VALIDITY

Even though Experiment 2 gives us valuable insights into which factors affect the performance of GVAE as a starting point, one has to consider some potential threats to validity. Due to limited computational resources, hyperparameter tuning was performed only on one dataset. This may have led to an overfitting scenario, where the machine learning models are excessively tailored to the specific characteristics of the tuning dataset, thus compromising their ability to generalize well to other datasets. Interestingly, despite these constraints, no significant difference was observed between the accuracies achieved on the dataset used for hyperparameter tuning and those achieved on the other datasets. This suggests that the tuned machine learning models, while potentially overfitting, are still managing to provide reasonably generalized performance across multiple datasets. Further studies would be required to explore the generalizability of these machine learning models in a more comprehensive manner.

The potential suboptimal performance of the Graph Isomorphism Network (GIN) and Graph Attention Network (GAT) could be attributed to the non-ideal selection of hyperparameters. Perhaps the scope of chosen hyperparameters was not expansive enough to encapsulate the best values for these machine learning models. Anyway, Shi et al. support our finding by directly comparing the transformer convolutional layer with GAT [67]. Still, further research needs to be done to understand under which conditions these architectures perform optimally.
Concerning the classifier task, varying latent sizes used as input pose a threat to validity. Larger latent sizes may capture more information, potentially explaining better performance. Relying on synthetic data by Tinnes et al. as a real-world representation poses a threat to validity. We had a limited dictionary, which could be problematic in real-world scenarios. If the data does not accurately reflect real-world conditions, the study's results may have limited generalizability. However, conducting experiments in a controlled setting, such as using a restricted label alphabet, was essential.

Probing by adding a classifier to the GVAE opens the problem that the classifier rather memorizes the input data points and their corresponding true values than truly learning to correctly extract whether/how often a pattern is given in a graph. Due to final linear classifier chosen, which has a small hidden size given in Section 5.4.4.1, we hope to have obtained preliminary evidence that suggests against the occurrence of memorization. Still, we will also investigate this point in the near future.

7 LIMITATIONS AND FUTURE WORK

As we reflect on the outcomes of our research, it is clear that the utilization of GVAEs for mining edit operations and model completion shows promise. We affirm the overall response to RQ1 and RQ2, suggesting using GVAE for mining edit operations warrants further investigation. Nevertheless, we also recognize several limitations in our current research that provide potential directions for future work.

A key constraint of our work is the lack of real-world datasets. Given that our data may not fully capture the complexities of real-world environments, for example, arbitrary natural language labels, the generalizability is limited.

The results of RQ1 of our study provide an indication that the architecture of the neural network impacts the generalization of the generation of SCG. By comparing different architectures, we conclude that the machine learning model's generalizability varies across these different setups. In future research, a more detailed investigation into the elements of each machine learning architecture that contribute to their performance could be beneficial. Understanding what makes certain machine learning architectures more stable or better at producing meta model adhering graphs could guide the design of even more effective architectures for this task. Moreover, further studies could explore additional network architectures that were not included in this study. Testing a wider range of architectures might reveal others that perform even better on these metrics.

A similar limitation occurs in the research question RQ2, where the performance of the classifier depends on the selection of hyperparameters and the dataset they were tuned on. In the future, one could extend the work and increase generalizability by using, for example, methods like Cross-dataset validation (Chapter "Hyperparameters and Validation Sets" [32]), considering other hyperparameter optimization methods or Transfer Learning (Chapter "Transfer Learning and Domain Adaptation" [32]). Cross-dataset validation involves splitting multiple datasets into training and validation sets and tuning the hyperparameters based on performance across all validation sets [32]. In future work, we aim to investigate alternative algorithms for hyperparameter tuning beyond Optuna and use visual tools for a more insightful view of the exploration of the search space. Despite the promising findings, several aspects need to be further explored. For instance, if the Transformer approach further outperforms the other methods in different settings, for example, in larger datasets. More so, the accuracy of the Transformer method is dependent on various factors, and additional research could help identify these dependencies and examine how they influence the final results. In the future trying Multilabel classification or regression could be more suitable for a more comprehensive investigation of RQ2.3. As for the topic of machine learning mentioned in the paragraph above, while further research remains captivating, I would like to emphasize that our aim was to provide proof of concept. We showed that it is indeed

interesting to further investigate the usage of Graph Variational Autoencoders for mining edit operations and model completion in general. Therefore, for our specific application and future research, the subsequent points appear to be of higher priority first.
Our study was restricted by the maximum graph size of 10. This size is not representative of typical difference graphs of model changes. In realistic scenarios, these graphs can be much larger. Further research could address this by adapting the approach to accommodate larger graphs, possibly by developing more efficient methods. Related to this issue is the variability in the sizes of the graphs we encounter. These range from small model changes, such as moving a single method, to larger refactorings all done in one step.
Secondly, we had a limited dictionary, which could be problematic in real-world scenarios. Real-world models and meta models could include arbitrary labels of natural language. Going forward, addressing this challenge requires the development of an effective representation for node and edge labels, so a suitable Sentence Encoding (GPT, Bert, Word2Vec).
Finally, we used synthetic data in our experiment, which might not represent real-world scenarios fully. It would be a valuable step forward to conduct similar experiments using real-world data in future research. After these points, it would be appropriate to perform a real-world study, asking domain experts whether the edit operations produced by GVAE are useful to them and represent typical edit scenarios [73].
An alternative approach for evaluation could involve the use of a surrogate task. Specifically, we could generate edit operations and then employ them in a model completion procedure to assess their usefulness. Also, a rather smaller problem is that the Graph Variational Autoencoder cannot guarantee the generation of valid graphs according to the meta model. This problem is commonly known in the research field of GVAE, several related works, therefore, include mechanisms to check for the validity of graphs according to some domain-specific constraints. Some of these approaches include Reinforcement Learning to improve these results, which we would also try in the future. Inspired by this adjustment, I also propose the usage of reinforcement learning as an additional validity check for edit operations [49]. Similar to the limitations of the work of Tinnes et al., we do not yet consider transient effects, which means that one applied edit operation can invalidate the pre- or post-conditions of another edit operation [73]. This will remain future work. Further, even though not considered yet, we can easily extend our dataset and therefore also our approach to edit operation patterns scattered across more than one connected component.
One of the main motivations for using Graph Variational Autoencoders is the lack of an efficient way to find edit operations. That is why, in the future, we should also check the performance of our approach and compare it to recent work, for example, LLMs. We further could shift our focus to other, different GNN approaches, for example, diffusion models.
To further proceed with research in the field of mining edit operations, one needs to explore how to specifically extract edit operations from the Graph Variational Autoencoder. We hope that the GVAE can already generate edit operations that could be useful to domain experts (Definition 14). In the future, one could therefore focus on metrics like Uniqueness, Novelty, and Diversity commonly used in previous work on Graph Variational Autoencoders to generate relevant graphs [49, 68].

During the thesis, our goal shifted from mining edit operations to model completions. Our initial approach will involve smaller code adjustments to adapt it for model completion

tasks. While much of the subsequent work remains consistent, one potential avenue is to incrementally generate graphs, rather than creating them in a singular step. This iterative method may be more intuitive, allowing for selective expansion of models or modifications to smaller model components, ultimately enhancing user experience. Such an approach is sophisticated and may yield more complex graph structures. However, a thorough exploration of this methodology remains a topic for future research [52].

8 CONCLUSION

In conclusion, our research provides substantial evidence supporting the use of GVAEs for generating realistic SCG, particularly in maintaining the distributional characteristics of the original graphs such as nearly generating type correct graphs all the time. Further, we provided a step towards answering whether GVAE can recognize previously applied edit operations, which can be seen as reoccurring patterns in SCG. We found out that indeed the latent space of the GVAE is sorted in such a way, that it also includes information about patterns contained in graphs and not only encodes high level features like graph sizes and node types. Looking ahead, we believe that the embeddings generated by the GVAE could have broader applications, serving as a versatile tool for various future tasks. The effectiveness of GVAE in automatically generating new edit operations is a matter of keen interest. If successful, we could provide a robust and adaptive mechanism for automatically extracting edit operations from model histories or even find new similar ideas for a specification of an edit operation. Further, we could apply complicated transformations within models, possibly tailored to specific domain languages. At this early stage of research, numerous challenges and open problems persist. Not only from a software engineering standpoint but also from a machine learning point of view. To the best of our knowledge, this is the only application where very large graphs, featuring both node and edge labels, which originate from natural language, are required. However, it's worth noting that Knowledge Graphs could be similar in scope. While they also handle large, labeled graphs, they generally do not focus on graph evolution.

APPENDIX

Listing A.1: RandomGraphBaseline: Create random graphs by selecting its size randomly, selecting a label from node_types for each node. Node_types includes all types that are found in the dataset by Tinnes et al. (Section 5.2.1). Randomly decides whether to add a directed edge from the first node to the second node. Whether an edge of type x should be created between node i and j is decided randomly. If the edge should be created, a label for the edge is randomly selected from edge labels. Edge_labels are again all edge types that are found in the dataset by Tinnes et al. (Section 5.2.1).

```
def baseline_randomgraphs(number_of_graphs_to_generate, s):
    random.seed(s)
    graphs = []

    for _ in range(number_of_graphs_to_generate):
        # define graph size
        n = randint(4, 10)
        G = nx.DiGraph()

        # Add nodes
        for i in range(0, n ):
            node_label = choice(node_types)
            G.add_node(i, label=node_label)

        # add edges with labels to graph
        for i in range(0, n):
            for j in range(0 , n ):
                #direction  ->
                if choice([True, False]):
                    edge_label = choice(edge_types)
                    G.add_edge(i, j, label=edge_label)

        graphs.append(G)
    return graphs
```

Here, I've structured the parameters as rows in a table, with the parameter name in the left column and its value in the right column.

Table A.1: p-values and t-statistics from RQ1: Meta Model Validity

Method	t-statistics	p-value
GAT-U	1763.1646	6.2320×10^{-109}
GAT-A	262.2689	2.7585×10^{-75}
TRF-A	124.8288	6.9736×10^{-52}
GIN-A	1964.5509	3.3808×10^{-104}
GIN-U	1054.7638	4.4058×10^{-103}
TRF-U	1414.2263	6.1021×10^{-109}

Table A.2: Summary of the best combination of hyperparameters for each approach

Method	lr	batch size	kl beta	encoder	latent	decoder	layers	dropout	regularization
TRF-A	0.001	32	0.2	256	128	64	1	0.0	0.0
TRF-U	0.0005	32	0.2	128	128	64	2	0.0	0.0
GAT-A	0.0005	32	0.3	512	64	64	3	0.0	0.0
GAT-U	0.005	32	0.3	128	512	32	2	0.0	0.0
GIN-A	0.0005	32	0.3	128	64	256	3	0.2	0.0
GIN-U	0.001	64	0.3	512	64	128	2	0.2	0.2

Table A.3: Parameters chosen for the generator by Liu et al. [53]

Parameter	**Value**
max_subgraph	512
alphas	[0.1, 0.3, 0.5, 0.7]
number_of_patterns	1
number_of_pattern_vertices	[4]
number_of_pattern_edges	[5]
number_of_pattern_vertex_labels	[2, 3, 4]
number_of_pattern_edge_labels	[2, 3, 4]
number_of_graphs	10
number_of_graph_vertices	[10, 7, 4]
number_of_graph_edges	[6, 8, 12, 15, 17, 20, 25, 30]
number_of_graph_vertex_labels	[4, 8]
number_of_graph_edge_labels	[4, 8]
max_ratio_of_edges_vertices	5
max_pattern_counts	512

Listing A.2: ValidGraphBaseline: Create random graphs by selecting its size randomly, selecting a label from node_types for each node. Node_types includes all types that are found in the dataset by Section 5.2.1. For each possible valid edge of type x between node i and j it is randomly decided if that edge should be added or not.

```
def baseline_validgraphs(number_of_graphs_to_generate, seed):
    random.seed(seed)
    graphs = []
    for _ in range(number_of_graphs_to_generate):
        # define graph size
        n = randint(4, 10)
        G = nx.DiGraph()

        # Add nodes
        for i in range(0, n):
            #node_types: includes all possilbe node types
            node_label = choice(node_types)
            G.add_node(i, label=node_label)

        # add edges with labels to graph
        for i in range(0, n):
            for j in range(0, n):
                # Get the types of nodes i and j
                i_type_ori = G.nodes[i]['label']
                j_type_ori = G.nodes[j]['label']

                # Ignore prefixes
                for prefix in ignore:
                    if i_type_ori.startswith(prefix):
                        i_type = i_type_ori[len(prefix):]
                    if j_type_ori.startswith(prefix):
                        j_type = j_type_ori[len(prefix):]

                #valid edges includes only correct edges
                valid_edges_dir_left_right = [edge[1] for edge in valid_edges if edge
                    [0] == i_type and edge[2] == j_type]

                for e in valid_edges_dir_left_right:
                    for pre in ignore:
                        label = pre + e
                        if ((label in edge_types) and (random.choice([True, False]))):
                            G.add_edge(i, j , label= label)
        graphs.append(G)
    return graphs
```

Listing A.3: __init__ function of the GVAE using the transformer convolutional layers as encoder layers

```
class GVAE(nn.Module):
    def __init__(self, feature_size, encoder_size, latent_size, decoder_size,
        num_layers, dropout):
        super(GVAE, self).__init__()

        self.encoder_embedding_size = encoder_size
        self.edge_dim = 1
        self.latent_embedding_size = latent_size
        self.num_edge_types = config.SUPPORTED_EDGES
        self.num_node_types = config.SUPPORTED_NODES
        self.max_num_nodes = config.MAX_GRAPH_SIZE
        self.decoder_hidden_neurons = decoder_size
        self.dropout=dropout
        self.num_layers = num_layers
        self.convs = nn.ModuleList()
        self.bns = nn.ModuleList()

        for i in range(self.num_layers):
            conv = TransformerConv(self.encoder_embedding_size if i > 0 else
                feature_size, self.encoder_embedding_size, heads=4, concat=False,beta=
                True, edge_dim=self.edge_dim)
            bn = BatchNorm(self.encoder_embedding_size)

            self.convs.append(conv)
            self.bns.append(bn)

        # Latent transform layers
        self.lin1 = Linear(self.encoder_embedding_size * (num_layers),
                           self.encoder_embedding_size * (num_layers))


        self.mu_transform = Linear(self.encoder_embedding_size * (num_layers ),
                                   self.latent_embedding_size)
        self.logvar_transform = Linear(self.encoder_embedding_size * (num_layers ),
                                       self.latent_embedding_size)

        # Decoder layers
        self.linear_1 = Linear(self.latent_embedding_size, self.decoder_hidden_neurons
            )
        self.linear_2 = Linear(self.decoder_hidden_neurons, self.
            decoder_hidden_neurons)

        node_output_dim = self.max_num_nodes * (self.num_node_types + 1)
        self.node_decode = Linear(self.decoder_hidden_neurons, node_output_dim)

        edge_output_dim = int((self.max_num_nodes * self.max_num_nodes) * (self.
            num_edge_types + 1))
        self.edge_decode = Linear(self.decoder_hidden_neurons, edge_output_dim)
```

BIBLIOGRAPHY

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[2] Abdullah M Alshanqiti, Reiko Heckel, and Tamim Khan. "Learning minimal and maximal rules from observations of graph transformations." In: *Electronic Communications of the EASST* 58 (2013).

[3] Jaan Altosaar. *Tutorial - What is a Variational Autoencoder?* Aug. 2016. DOI: 10.5281/zenodo.4462916. URL: https://doi.org/10.5281/zenodo.4462916.

[4] Martin Arjovsky and Léon Bottou. "Towards principled methods for training generative adversarial networks." In: *arXiv preprint arXiv:1701.04862* (2017).

[5] Iman Avazpour, John Grundy, and Lars Grunske. "Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations." In: *Journal of Visual Languages & Computing* 28 (2015), pp. 195–211.

[6] László Babai. "Graph isomorphism in quasipolynomial time." In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 684–697.

[7] Islem Baki and Houari Sahraoui. "Multi-step learning and adaptive search for learning complex model transformations from examples." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.3 (2016), pp. 1–37.

[8] Yonatan Belinkov. "Probing classifiers: Promises, shortcomings, and advances." In: *Computational Linguistics* 48.1 (2022), pp. 207–219.

[9] Karima Berramla, El Abbassia Deba, Jiechen Wu, Houari A Sahraoui, and Abou El Hassan Benyamina. "Model Transformation by Example with Statistical Machine Translation." In: *MODELSWARD*. 2020, pp. 76–83.

[10] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Formal foundation of consistent EMF model transformations by algebraic graph transformation." In: *Software & Systems Modeling* 11 (2012), pp. 227–250.

[11] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[12] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. "Classifying edits to variability in source code." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 196–208.

[13] Andrew P Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms." In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[15] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. "An example is worth a thousand words: Composite operation modeling by-example." In: *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings 12*. Springer. 2009, pp. 271–285.

[16] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. "An LSTM-based neural network architecture for model transformations." In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2019, pp. 294–299.

[17] Loli Burgueño, Jordi Cabot, Shuai Li, and Sébastien Gérard. "A generic LSTM neural network architecture to infer heterogeneous model transformations." In: *Software and Systems Modeling* 21.1 (2022), pp. 139–156.

[18] Alberto Rodrigues Da Silva. "Model-driven engineering: A survey supported by the unified conceptual model." In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155.

[19] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. "Syntax-directed variational autoencoder for structured data." In: *arXiv preprint arXiv:1802.08786* (2018).

[20] Nicola De Cao and Thomas Kipf. "MolGAN: An implicit generative model for small molecular graphs." In: *arXiv preprint arXiv:1805.11973* (2018).

[21] Aarzoo Dhiman and SK Jain. "Optimizing frequent subgraph mining for single large graph." In: *Procedia Computer Science* 89 (2016), pp. 378–385.

[22] Reinhard Diestel. *Graph theory*. 2000.

[23] Carl Doersch. "Tutorial on variational autoencoders." In: *arXiv preprint arXiv:1606.05908* (2016).

[24] Brendan L Douglas. "The weisfeiler-lehman method and graph isomorphism testing." In: *arXiv preprint arXiv:1101.5211* (2011).

[25] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. "Fundamental theory for typed attributed graph transformation." In: *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings 2*. Springer. 2004, pp. 161–177.

[26] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. "Search-based detection of high-level model changes." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 212–221.

[27] Lewis S Feuer. "The principle of simplicity." In: *Philosophy of Science* 24.2 (1957), pp. 109–122.

[28] David Foster. *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[29] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[30] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El Boussaidi. "Model refactoring by example: A multi-objective search based software engineering approach." In: *Journal of Software: Evolution and Process* 30.4 (2018), e1916.

[31] Ian Goodfellow. "Nips 2016 tutorial: Generative adversarial networks." In: *arXiv preprint arXiv:1701.00160* (2016).

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[33] Zhinan Gou, Lixin Han, Ling Sun, Jun Zhu, and Hong Yan. "Constructing dynamic topic models based on variational autoencoder and factor graph." In: *IEEE Access* 6 (2018), pp. 53102–53111.

[34] Hakan Gunduz. "Malware detection framework based on graph variational autoencoder extracted embeddings from API-call graphs." In: *PeerJ Computer Science* 8 (2022), e988.

[35] Maroun Haddad and Mohamed Bouguessa. "Exploring the representational power of graph autoencoder." In: *Neurocomputing* 457 (2021), pp. 225–241.

[36] Mouna Hadj-Kacem and Nadia Bouassida. "Deep representation learning for code smells detection using variational auto-encoder." In: *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019, pp. 1–8.

[37] John Hewitt and Percy Liang. "Designing and interpreting probes with control tasks." In: *arXiv preprint arXiv:1909.03368* (2019).

[38] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. "Systematically deriving domain-specific transformation languages." In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2015, pp. 136–145.

[39] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. "Strategies for pre-training graph neural networks." In: *arXiv preprint arXiv:1905.12265* (2019).

[40] Takeshi D Itoh, Takatomi Kubo, and Kazushi Ikeda. "Multi-level attention pooling for graph neural networks: Unifying graph representations with multiple localities." In: *Neural Networks* 145 (2022), pp. 356–373.

[41] Timo Kehrer, Abdullah Alshanqiti, and Reiko Heckel. "Automatic inference of rule-based specifications of complex in-place model transformations." In: *Theory and Practice of Model Transformation: 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings 10*. Springer. 2017, pp. 92–107.

[42] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. "Understanding model evolution through semantically lifting model differences with SiLift." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 638–641.

[43] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. "Automatically deriving the specification of model editing operations from meta-models." In: *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings 9*. Springer. 2016, pp. 173–188.

[44] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. "Search-based model transformation by example." In: *Software & Systems Modeling* 11 (2012), pp. 209–226.

[45] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes." In: *arXiv preprint arXiv:1312.6114* (2013).

[46] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[47] Stefan Kögel, Raffaela Groner, and Matthias Tichy. "Automatic Change Recommendation of Models and Meta Models Based on Change Histories." In: *ME@ MoDELS*. 2016, pp. 14–19.

[48] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. "Grammar variational autoencoder." In: *International conference on machine learning*. PMLR. 2017, pp. 1945–1954.

[49] Youngchun Kwon, Jiho Yoo, Youn-Suk Choi, Won-Joon Son, Dongseon Lee, and Seokho Kang. "Efficient learning of non-autoregressive graph variational autoencoders for molecular graph generation." In: *Journal of Cheminformatics* 11.1 (2019), pp. 1–10.

[50] David A Levin and Yuval Peres. *Markov chains and mixing times*. Vol. 107. American Mathematical Soc., 2017.

[51] Chen Ling, Junji Jiang, Junxiang Wang, and Zhao Liang. "Source Localization of Graph Diffusion via Variational Autoencoders for Graph Inverse Problems." In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 1010–1020.

[52] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. "Constrained graph variational autoencoders for molecule design." In: *Advances in neural information processing systems* 31 (2018).

[53] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. "Neural subgraph isomorphism counting." In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 1959–1969.

[54] Oded Maimon and Lior Rokach. *Data mining and knowledge discovery handbook*. Vol. 2. 2005. Springer, 2005.

[55] Matias Martinez, Laurence Duchien, and Martin Monperrus. "Automatically extracting instances of code change patterns with ast analysis." In: *2013 IEEE international conference on software maintenance*. IEEE. 2013, pp. 388–391.

[56] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation." In: *Electronic notes in theoretical computer science* 152 (2006), pp. 125–142.

[57] Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. "Recommending model refactoring rules from refactoring examples." In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2018, pp. 257–266.

[58] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[59] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 819–830.

[60] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[61] Siegfried Nijssen and Joost N Kok. "The gaston tool for frequent subgraph mining." In: *Electronic Notes in Theoretical Computer Science* 127.1 (2005), pp. 77–87.

[62] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. "Revision: a tool for history-based model repair recommendations." In: *Proceedings of the 40th International conference on software engineering: companion proceeedings*. 2018, pp. 105–108.

[63] Foster Provost and Tom Fawcett. *Data Science for Business: What you need to know about data mining and data-analytic thinking*. " O'Reilly Media, Inc.", 2013.

[64] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. "Learning quick fixes from code repositories." In: *arXiv preprint arXiv:1803.03806* (2018).

[65] Hajer Saada, Marianne Huchard, Michel Liquière, and Clémentine Nebut. "Learning Model Transformation Patterns using Graph Generalization." In: *CLA*. Citeseer. 2014, pp. 11–22.

[66] Shane Sendall and Wojtek Kozaczynski. "Model transformation: The heart and soul of model-driven software development." In: *IEEE software* 20.5 (2003), pp. 42–45.

[67] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. "Masked label prediction: Unified message passing model for semi-supervised classification." In: *arXiv preprint arXiv:2009.03509* (2020).

[68] Martin Simonovsky and Nikos Komodakis. "Graphvae: Towards generation of small graphs using variational autoencoders." In: *International conference on artificial neural networks*. Springer. 2018, pp. 412–422.

[69] Nataša Sukur, Nemanja Milošević, Doni Pracner, and Zoran Budimac. "Automated program improvement with reinforcement learning and graph neural networks." In: *Soft Computing* (2023), pp. 1–12.

[70] Yu Sun, Jeff Gray, and Jules White. "MT-Scribe: an end-user approach to automate software model evolution." In: *Proceedings of the 33rd international conference on software engineering*. 2011, pp. 980–982.

[71] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[72] Igor V Tetko, David J Livingstone, and Alexander I Luik. "Neural network studies. 1. Comparison of overfitting and overtraining." In: *Journal of chemical information and computer sciences* 35.5 (1995), pp. 826–833.

[73] Christof Tinnes, Timo Kehrer, Mitchell Joblin, Uwe Hohenstein, Andreas Biesdorf, and Sven Apel. "Learning domain-specific edit operations from model repositories with frequent subgraph mining." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 930–942.

[74] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." In: *arXiv preprint arXiv:1710.10903* (2017).

[75] Robin Winter, Frank Noé, and Djork-Arné Clevert. "Permutation-invariant variational autoencoder for graph-level representation learning." In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 9559–9573.

[76] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. "A comprehensive survey on graph neural networks." In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[77] Yihui Xiong, Renguang Zuo, Zijing Luo, and Xueqiu Wang. "A physically constrained variational autoencoder for geochemical pattern recognition." In: *Mathematical Geosciences* 54.4 (2022), pp. 783–806.

[78] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[79] Li Yang and Abdallah Shami. "On hyperparameter optimization of machine learning algorithms: Theory and practice." In: *Neurocomputing* 415 (2020), pp. 295–316.

[80] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. "Hierarchical graph representation learning with differentiable pooling." In: *Advances in neural information processing systems* 31 (2018).

[81] Yijun Yu, Thein Than Tun, and Bashar Nuseibeh. "Specifying and detecting meaningful changes in programs." In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 273–282.

[82] Alice Zheng. *Evaluating machine learning models: a beginner's guide to key concepts and pitfalls*. O'Reilly Media, 2015.