Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

SSA Data Flow Information for Semantic Code Tasks

Bachelor's Thesis

Tobias Stocker

tstocker@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Peter Belcák, Florian Grötschla
Prof. Dr. Roger Wattenhofer

July 31, 2022

Abstract

Capturing semantic information in representations of code is an ongoing challenge in machine learning for code-related tasks such as code search, completion, classification, explanation, or clone identification. Using Semantized Abstract Syntax Trees, we will explore the capabilities of graph neural networks on code-based graphs. After introducing the concept of Semantized Abstract Syntax Tree (SAST), we develop a graph neural network (GNN) model featuring different GNN types, residual connections, and jumping knowledge. We evaluate our model by solving two graph tasks, namely the task of node type prediction and link prediction, which form the basis of more complex models that might have the potential to solve code-related machine learning tasks in the future. We get promising results that are reflected when predicting the use of variables in code examples.

Contents

Chapter 1

Introduction

Machine learning has gained a lot of attention in computer science over the past decade. While terms like "machine intelligence" or "artificial intelligence" have quickly become buzzwords used for advertising, machine learning has advanced many fields in computer science such as speech recognition, natural language translation and recommendation algorithms just to name a few. With the steady growth of the software industry over the last years, there has also been a constant increase in the amount of public and private source code repositories. This large amount of available code data gives rise to many code related machine learning tasks such as code search, completion, classification, explanation or clone identification.

It is natural to use some sort of natural language processing (NLP) models to solve code-related machine learning tasks as one might understand source code as some special kind of natural language. In fact, quite some work has gone into solving code-related machine learning task using NLP models [1], [2]. However, working with source code can impose some additional challenges compared to natural languages. Some code-related tasks require zero error rate. In the task of completing code, for example, a single misprediction can lead to the entire code not compiling anymore. Further, with code we can write programs that do the exact same thing yet they are written in a completely different way. It can be very hard for a natural language model to understand this. As a result, researchers have enhanced NLP models for example by extending the model's input with data flow information [3].

Similarly, Liu et al. very recently used graph neural networks on graphs based on the abstract syntax tree of some source code combined with multi-head attention applied on the source code to solve the problem of code search [4]. Inspired by this work, we want to explore the capabilities of graph neural networks (GNNs) to solve code-related machine learning tasks only using graphs generated from code. GNNs have only recently gained a lot of attention by the machine learning community. While they have been used to solve many graph related tasks in biology and chemistry, there has only been little research conducted in graphs related to code.

The goal of this work is to generate graphs from source code using the tree structure that is naturally given by its abstract syntax tree and extend them with more semantic information. We then build a GNN based model architecture and investigate its capabilities by solving simple graph tasks that form the basis for more complex tasks that might once be used to solve code-related machine learning tasks.

Chapter 2

Background

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, where $\mathcal{V}$ is the set of vertices of $\mathcal{G}$ and $\mathcal{E}$ is the set of directed edges (u, v) for $u, v \in \mathcal{V}$. Further $\mathcal{N}(u)$ denotes the neighborhood of some node $u \in \mathcal{V}$, i.e. $\mathcal{N}(u) = \{v \mid v \in \mathcal{V} \cup (v, u) \in \mathcal{E}\}$.

2.1 Machine Learning on Graphs

The challenge of applying machine learning on graph-structured data is that the traditional deep learning approaches do not apply. For example, convolutional neural networks (CNNs) are only applicable to grid-structured inputs and recurrent neural networks (RNNs) are only applicable to sequential data. Further, simply flattening the adjacency matrix of the graph and feeding it through a multi-layer perceptron (MLP) is not a good approach either as the result of the model would depend on the arbitrary order of the nodes in the adjacency matrix. That is why we will follow the approach proposed by [5] and demand that our methods satisfy the following properties:

First, as pointed out by the approach from above, we want our machine learning models to be independent of the order in which the graph is stored, i.e. we want the models to be *permutation-invariant*. In mathematical terms:

$$f(\mathbf{PAP}^\top) = f(\mathbf{A}) \tag{2.1}$$

where f is any function that takes an adjacency matrix $\mathbf{A}$ and $\mathbf{P}$ is a permutation matrix. Further, the goal of our machine learning model is to take an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ along with a set of node features $\mathbf{X} \in \mathbb{R}^{d_\mathcal{V} \times |\mathcal{V}|}$ as well as a set of edge features $\mathbf{Y} \in \mathbb{R}^{d_\mathcal{E} \times |\mathcal{E}|}$ and use them to generate node embeddings $\mathbf{z}_u, \forall u \in \mathcal{V}$.

The general framework that we will be using is called a Graph Neural Network (GNN). The defining feature of a GNN is that it uses a form of neural message passing in which vector embeddings are passed along edges and updated using neural networks. To be more specific, we start with some initial embedding $\mathbf{h}_u^{(0)}$ for each node $u \in \mathcal{V}$. During each message passing iteration the *hidden embedding*

$\mathbf{h}_u^{(k)}$ of each node $u \in \mathcal{V}$ is then updated based on information aggregated from the neighbors of u denoted as $\mathcal{N}(u)$. One such update step can be expressed as [5]:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u))) \\ &= \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})\end{aligned} \tag{2.2}$$

where UPDATE and AGGREGATE are arbitrary differentiable functions (i.e. neural networks) and superscripts denote the message passing iteration. After all message passing iterations the final hidden embeddings are the resulting node embeddings, i.e.

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{2.3}$$

One message passing step is commonly implemented as a *layer* of a GNN. With one layer, a node may only learn from its direct neighbors, and with each additional layer a node may learn from the nodes that are an additional edge further away. On the contrary with too many layers GNNs often suffer from *over-smoothing* [6], which means that after too many update iterations the node representations can become very similar. This means that choosing the right amount of layers is important in a GNN.

By specifying the UPDATE and AGGREGATE functions used in the message passing process we fully define concrete GNN types. In the following subsections we introduce some common GNN types.

2.1.1 Graph Convolutional Network (GCN)

Graph Convoluational Networks (GCNs) were first proposed by Kipf and Welling in 2016 [7] and are one of the most popular baseline GNN types.

The most basic neighborhood aggregation operation would be to simply sum up all hidden embeddings of the neighborhood, i.e.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v. \tag{2.4}$$

However this approach is highly sensitive to the node degree as $||\mathbf{m}_{\mathcal{N}(u)}|| >> ||\mathbf{m}_{\mathcal{N}(v)}||$ if $\deg(u) >> \deg(v)$. A simple solution to this issue is to just take the average instead of the sum, i.e.

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|}. \tag{2.5}$$

A Graph Convolutional Network (GCN) employs a similar approach called symmetric normalization:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{2.6}$$

It also uses *self-loops* which means that it omits the UPDATE function and instead adds the own hidden embedding to the hidden embeddings received by the neighbors, i.e.

$$\mathbf{h}_u^{(k+1)} = \text{AGGREGATE}^{(k)}(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u) \cup u) \tag{2.7}$$

Finally, after adding an activation function we get the definition of a GCN:

$$\mathbf{h}_u^{(k+1)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right) \tag{2.8}$$

where σ is an activation function such as ReLU or tanh and $\mathbf{W}^{(k)}$ is a trainable parameter matrix.

2.1.2 Graph Isomorphism Network (GIN)

Graph Isomorphism Networks (GINs) are inspired by the *Weisfeiler-Lehman (WL) graph isomorphism test* [8]. The WL test is sometimes able to tell that two graphs are non-isomorphic (i.e that the two graphs do not have the same structure). However, if it does not decide that two graphs are non-isomorphic it does not imply the two graphs are isomorphic. As the WL test works very similar to how GNNs learn, Xu et al. [9] designed a new GNN architecture (GIN) for which they proved it to be as expressive as the WL test. One message passing iteration works as follows:

$$\mathbf{h}_u^{(k+1)} = \text{MLP}^{(k)}\left((1+\epsilon) \cdot \mathbf{h}_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)}\right) \tag{2.9}$$

where $\epsilon \in \mathbb{R}$ can be learnable but may also be fixed and MLP stands for multi-layer perceptron which can be any kind of fully connected neural network.

2.1.3 Graph Attention Network (GAT)

Graph Attention Networks (GATs) were first proposed by Veličković et al. in 2017 [10]. The basic idea of GATs is to use attention to assign an importance to each neighbor and itself:

$$\mathbf{h}_u^{(k+1)} = \alpha_{u,u}\mathbf{h}_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \alpha_{u,v}\mathbf{h}_v^(k) \tag{2.10}$$

where $\alpha_{u,v}$ denotes attention coefficients computed as:

$$\alpha_{u,v} = \frac{\exp(\sigma(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]))}{\sum_{v' \in \mathcal{N}(u) \cup \{u\}} \exp(\sigma(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]))} \tag{2.11}$$

where $\mathbf{a}$ is a trainable attention vector, $\mathbf{W}$ is a trainable parameter matrix, $\oplus$ is the concatenation operation and σ denotes a LeakyReLU activation function.

2.2 Abstract Syntax Trees (ASTs)

An abstract syntax tree (AST) is a tree representation of some source code that describes the structure the code. When converting a piece of code to the AST representation only structural and content related information is kept and any additional information is discarded.

Parsing code into an AST usually happens in two stages: First, during lexical analysis, the source code is converted into a list of tokens that describe the code. During the second step, called the syntactical analysis or parsing, that list of tokens is then turned into a tree.

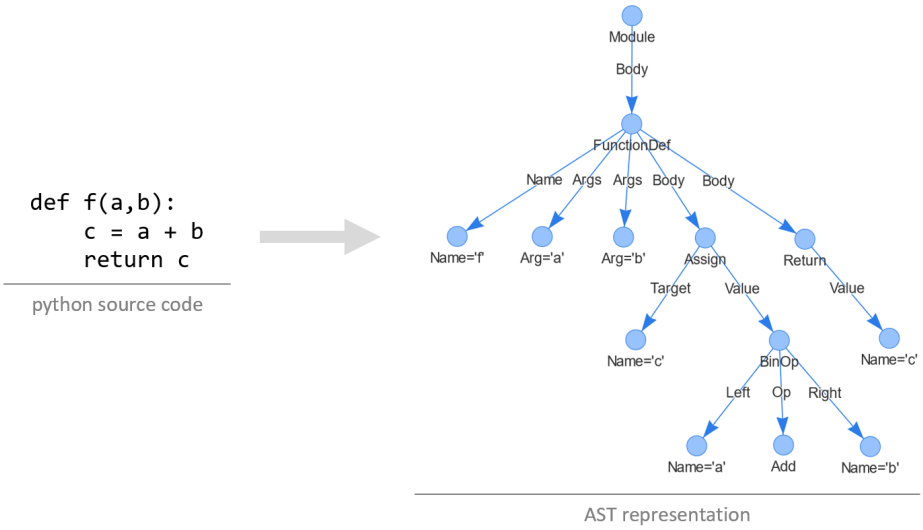

Figure 2.1: An example showing the AST representation of some python source code. Tokens generated from the code are turned into nodes and linked together according to the syntax of python.

Chapter 3

Creating Graphs from Code

3.1 Semantized Abstract Syntax Trees (SASTs)

Parsing source code into its abstract syntax tree representation makes for a simple way of generating a graph out of code. However, while this approach nicely represents the syntactic structure of the source code, it lacks semantic information. For example, it is hard to find out where the value of a variable assignment is used and where a used variable is defined.

Semantized ASTs (SASTs) try to combat the lack of semantic information by replacing variables with invariables in the static single assignment (SSA) form [11]. As in the SSA form, a new invariable is created for each assigned variable in the code (green nodes in figure 3.1) and thus an invariable is only assigned once. Each invariable is further connected to all variable uses that are influenced by that invariable using a new type of edge (orange edges in figure 3.1).

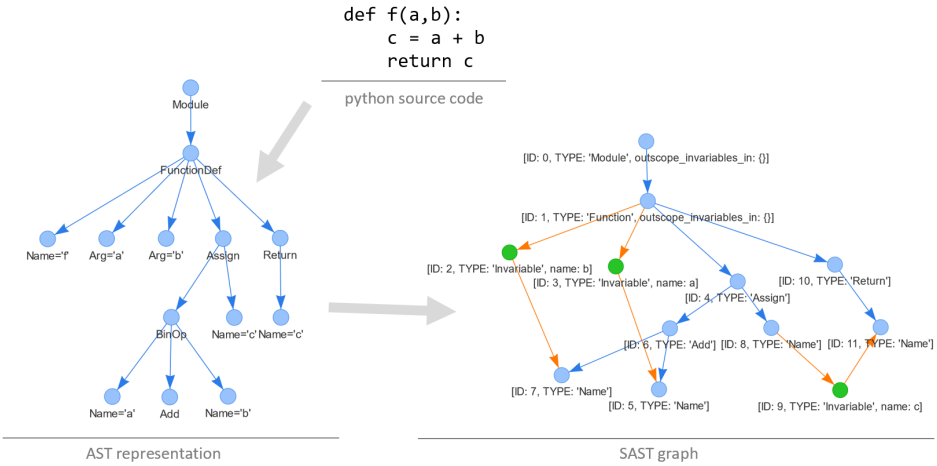

Figure 3.1: Example showing the conversion of python source code into its AST and then SAST representation.

As not every variable assignment is certainly executed, an invariable might be defined by multiple different invariables. In SSA this is solved using ϕ-functions. In SAST graphs we can simply connect all invariable nodes to the node representing the variable use (see figure 3.2).

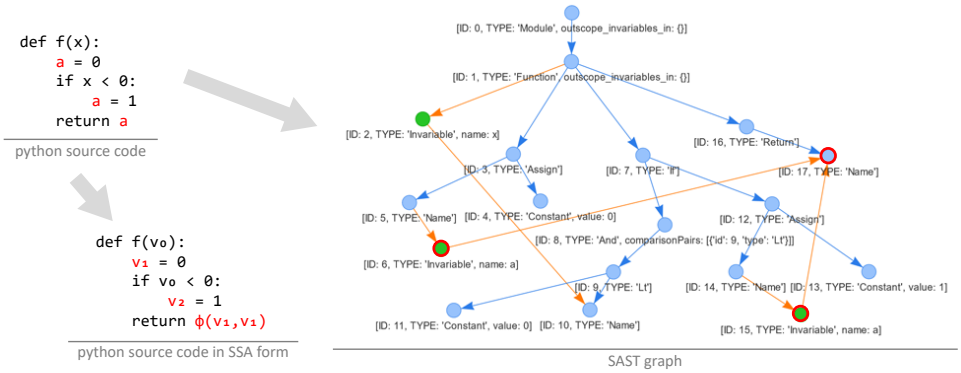

Figure 3.2: Example: SAST graph when multiple variable assignments effect its use. The two corresponding invariables (marked in red) both share an edge to the node representing the variable use.

In the subsequent section, we explain how our approach to generating SAST graphs from source code. We will use the implementation from previous work conducted on SAST graphs by Jakob Flunger [12] and adapt it to the form in which we use it.

3.2 Dataset Generation

We will use python source code in this work for a number of reasons. First, python already provides a module that converts python source code into its AST representation and also provides methods to traverse the AST. Furthermore, there is a big amount of python source code freely available on different online sources such as GitHub or StackOverflow.

To collect the required source code, we use a GitHub crawler [12] that collects all python files from the most visited GitHub repositories. With this tool we collected 242 repositories from which we extracted around 400,000 python functions.

3.2.1 Tracking of Variable Assignments

In order to be able to convert an AST into an SAST, we need to know for every variable which assignments might have modified it at any given point in the program. The basic idea is that we register every variable assignment with its

name and invariable in a class while also tracking the scope. We call this class the CompassManager. Tracking the scope is important as not every assignment is certainly executed and thus does not always change the value of the assigned variable. An assignment inside an if-clause, for example, might not always be executed.

The CompassManager is a tree like data structure consisting of Compass and CompassContainer instances. Compass instances store all registered variable assignments in their scope with the corresponding invariable. CompassContainer instances may hold multiple Compass or CompassContainer instances. We refer to them as *subcompasses*. We further differentiate between *parallel* and *sequential* CompassContainers depending on their context. The CompassManager tree initially contains only a Compass instance as its root. Any new variable assignments are always registered to the Compass representing the current context. We will call the Compass representing the current context *active*.

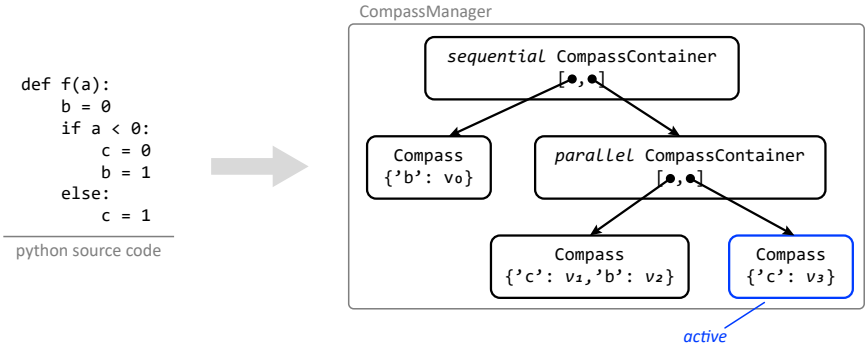

Figure 3.3: Example CompassManager of some python source code. For any further diagrams we will mark the active Compass in blue.

When initiating a branch, the *active* Compass is added to a sequential CompassContainer. Further, a parallel CompassContainer containing a new Compass is also added to the sequential CompassContainer. The new Compass now represents the current context and thus is *active* (see figure 3.4).

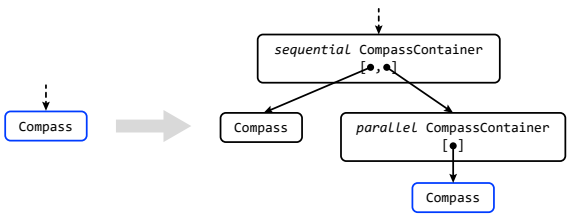

Figure 3.4: CompassManager: Initiating a branch.

When initiating an alternative branch, we add a new Compass to the parallel CompassContainer, which is the parent of the *active* Compass, and set the new Compass as the *active* one (see figure 3.5).

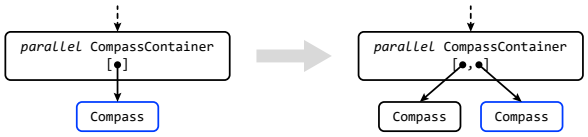

Figure 3.5: CompassManager: Initiating an alternative branch.

Finally, when we conclude a branch, we simply add a new Compass to the sequential CompassContainer, which is the parent of the parallel CompassContainer that is holding the *active* Compass (see figure 3.6).

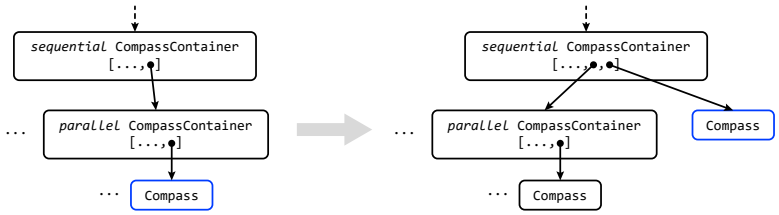

Figure 3.6: CompassManager: Concluding a branch.

Querying the invariables that might have modified a variable from the CompassManager is done by recursively querying the tree. Upon query, a Compass returns whether a variable has been registered in their scope, i.e. whether it has certainly modified the variable. If a variable has certainly been modified, the Compass also returns the corresponding invariable. Upon query, a CompassContainer returns if a variable has certainly been modified based on its scope and the subcompasses it is holding. It also returns all invariables that might have modified the variable. A parallel CompassContainer, representing a branch context, only returns that a variable certainly has been modified if all subcompasses return that the variable has certainly been modified. The invariables returned are all invariables returned by the subcompasses. A sequential CompassContainer, representing a sequential context, returns that a variable certainly has been modified if any subcompasses returns that it has certainly modified the variable. The invariables returned are the ones returned by the subcompasses up to the first one that certainly has modified the variable in reverse sequential order.

If a variable has not been assigned for certain by the invariables registered

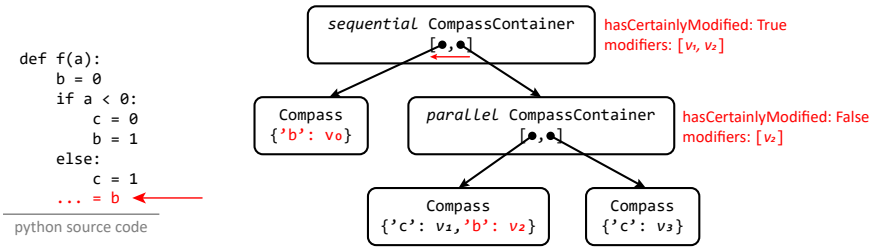

Figure 3.7: CompassManager Example: Querying all invariables that might have modified variable 'b'. First, when queried the parallel CompassContainer returns that it does not have certainly modified variable 'b' together with all potential modifiers. As variable 'b' has not yet been certainly modified, the Compass directly connected to the sequential CompassContainer is also queried. It returns that it has certainly modified variable 'b' together with the modifier. Finally the CompassManager returns that variable 'b' has certainly been modified together with all potential modifiers.

in the CompassManager, we add an *outscope invariable* and add it to the list of potential modifiers. *Outscope invariables* imply that the variable might be defined outside the function definition for which we are creating the graph. They are not further interesting for our work.

3.2.2 Converting an AST to a SAST

Now that we have a way of retrieving all invariables that might have modified a variable, we have all the tools needed to convert the source code of a python function into an SAST graph. Reading the source code and parsing it into its AST representation can easily be done using python's *pathlib* and *ast* modules. After generating the AST, we iterate through it starting from the root node while traversing nodes in the sequential order of the code.

We leave most of the tree untouched, except for renaming some types, and only process the following nodes. For each variable that is assigned, we replace the node representing the variable with a new Invariable node and register the assignment and invariable in the CompassManager. For each variable that is loaded, we replace the variable node with a Name node and query from the CompassManager all invariables that might have modified the variable. All potential modifiers are then connected via an edge to the Name node. For each If-, While-, For-, and Try-nodes we will initiate and conclude branches in the CompassManager.

3.2.3 Data Export

We export each SAST graph as a dictionary that contains the following keys:

- *num_ nodes* and *num_ edges* describe the number of nodes and edges respectively.
- *edge_ index* is a $2 \times |\mathcal{E}|$ matrix E, where the matrix elements $e_{1,i}$ and $e_{2,i}$ for $i \in 1, ..., |\mathcal{E}|$ describe a directed edge from $m_{1,i}$ to $m_{2,i}$.
- *edge_ type* is a list of length $|\mathcal{E}|$ that describes the type of each edge in the graph encoded as an integer.
- *edge_ ast* is a list of length $|\mathcal{E}|$ that describes if an edge is part of the original AST.
- *node_ type* is a list of length $|\mathcal{N}|$ that describes the type of each node in the graph encoded as an integer.
- *invar_ idcs* is a list that contains all indices of invariable type nodes.
- *invar_ neighbor_ idcs* is a list that contains all potential neighbors indices of invariable type nodes.

We randomly split the set of all generated graphs into a training data set and a validation data set. All types are encoded as an integer using a mapping. Finally, any other information such as the syntactic name of the variables is discarded.

3.3 Limitations of SAST

SAST's biggest limitation currently lies in the approach used for tracking the variables. Tracking variable assignments while traversing the AST makes it hard to find invariables that are only created after a variable is loaded. A simple example is shown in figure 3.8 where the current approach misses an edge. It might be possible to extend the current approach using some form of jumping back after scopes that have the potential of producing invariables that potentially effect variable loads happening earlier in the program (e.g. loops). Another potential approach would be to make use of the *dominance frontier* algorithm [13] that is commonly used in compilers to convert code into SSA form.

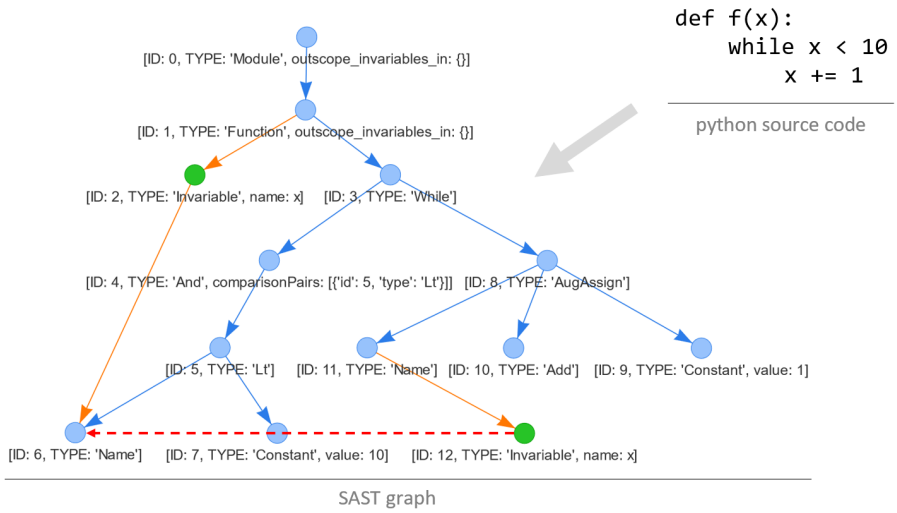

Figure 3.8: Code example where the SAST graph generation approach fails to connect the *Invariable* node with index 12 to to the *Name* node with index 6 (red dashed edge). The mentioned *Invariable* and *Name* nodes represent the assignment inside the while-loop (line 3) and the use of variable x in the loop-header (line 2) respectively. The error occurs because the assignment of variable x happens after its use. Thus at the time when the variable use is processed the mentioned invariable has not yet been created and therefore also not been added to the CompassManager even though it potentially modifies the use.

Chapter 4

Semantic Node Type Prediction

4.1 Task Description

The goal of this task is to predict the type of any node in a SAST graph. To be more precise, we are given a SAST graph in which we do not know the type of a node. We want our model to predict the type of that node based on the rest of the graph.

4.2 Data, Training and Evaluation

We use python functions that are crawled from GitHub repositories as described in section 3.2. From each function a SAST graph is generated and stored in the form explained in section 3.2.3.

We create a small data set that contains 10,000 graphs and a larger data set that contains 100,000 graphs. It is split into into 95% (9,500) training graphs and 5% (500) validation graphs randomly and we use it to initially try out new ideas. The large data set is split into 99% (99,000) training graphs and 5% (1,000) validation graphs randomly. We use it to evaluate and fine-tune our models.

Before we can use the graphs of a data set to train or evaluate our model we have to turn them into *PyTorch Geometric Data* [14] objects. In this step, we might also pre-transform the graphs depending on how we are planning to use them during training or evaluation. Creating basic *PyTorch Geometric Data* objects is easy as the edge index is already in the correct form and so are the node attributes. We only combine the two edge features we have collected, i.e. edge type and is-AST attribute, because they are commonly stored together as edge attributes. Finally, we use the DataLoader from PyTorch Geometric [15] which automatically splits the data objects into batches.

4.2.1 Training the Model

We train the model by masking a random node in every graph before every epoch. Masking a node is performed by setting the node type as *masked* (a new additional node type) while remembering the original node type. As the masked node is a different one in every epoch, we create the training graphs without pre-transforming them, i.e. we create a basic data object for every graph in our training data set. Masking is then performed right before passing the graph into the model. After a graph is passed through the model, we apply the loss function on the predicted type and the original type.

4.2.2 Evaluating the Model

In contrast to training, when evaluating our model we do not want to randomly select a masked node as this would make the results less comparable between different evaluations due to the randomness involved. Thus we create one graph data object for every node in the original graph and mask the corresponding node while creating the graph. This way we get a more consistent result that tells us how the model is performing. The downside of this approach is that we create around 20 to 100 graphs out of a single validation graph enlarging the total number of validation graphs. A simple solution to this issue would be to pre-select some nodes and thus reduce the number of nodes we test while keeping the results consistent.

We use accuracy as the metric to evaluate the performance of our models. Accuracy is defined as:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.1}$$

Further to get an idea about where the model needs improvement we also make use of confusion matrices. A confusion matrix compares the model's prediction with every original node type. Let n be the number of node types. In a confusion matrix we have n rows and n columns each representing a node type. The value at the i-th row and j-th column is the percentage the model has predicted type j when the original type was i compared to all predictions made for type i (see figure 4.1).

4.3 Model Architecture

The model architecture is heavily inspired by the example code provided for the ogbg-code2 [16] data set from the Open Graph Benchmark (OGB) [17]. We use PyTorch Geometric (PyG) [18] which is build upon PyTorch [19] to implement

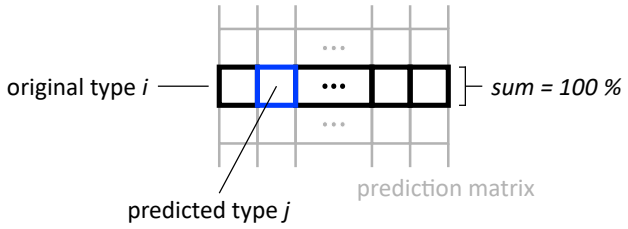

Figure 4.1: One row of a confusion matrix.

our models in Python. A diagram of the complete model architecture can be found at the end of this section.

4.3.1 Node Encoder

The node encoder is used to generate an initial embedding for each node in the graph. Ideally, it creates the initial embedding based on the information available for each node. Such information may be the node type (including the *unknown/masked* type), any additional node attributes, the structural information about the node in the graph, etc.

In this work we use a simple PyTorch Embedding [20] layer as our node encoder. Embedding layers are no more than a lookup table for a fixed size. In our case the size is the number of different types and the resulting embedding is based only on the node type of the SAST graph.

Generally, the node encoder may also be a more sophisticated algorithm or a neural network. What is important is only that it generates an embedding of the correct size for each node.

4.3.2 Graph Neural Network

The GNN is the core of our model. In this part the initial node embeddings are passed through the GNN layers resulting in the final node embeddings.

As we have collected edge information (i.e. edge type and the is-AST edge attribute) during the SAST graph generation, we want our model to make use of this information. The basic idea is to generate an embedding for each edge and include it during the message passing process. In this work we use two PyTorch Embedding layers to encode both the type information and the is-AST edge information and sum them up. Once again, as for the node encoder, one might choose a more sophisticated approach to encoding the edge information such as a neural network.

The GNN consist of multiple GNN convolution layers. A GNN convolution layer may be of any type described in section 2.1, i.e. GCN, GIN or GAT,

though we need to adapt them slightly to incorporate our edge embedding. In the following paragraphs, we explain the changes made to each GNN type. We use $\mathbf{h}^*$ to denote the resulting hidden embeddings of the GNN convolution as we further pass them through a batch normalization layer, an activation layer as well as a dropout layer. Optionally, one can add residual connections to the model as shown in figure 4.2.

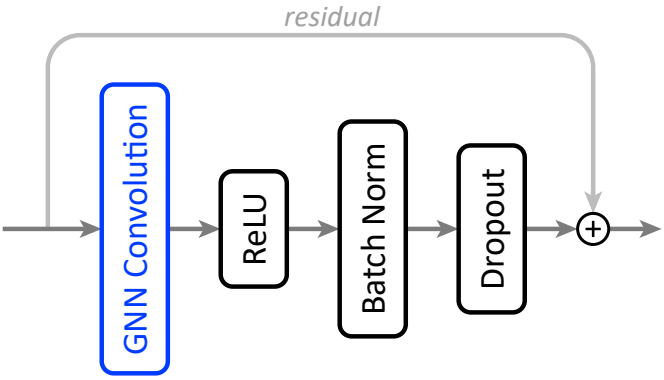

Figure 4.2: Diagram depicting a single GNN layer.

To GCN (see section 2.1.1) we apply the following changes: We first pass each hidden embedding trough a linear layer and then add the edge embedding before applying a ReLU activation function. The GCN convolution can mathematically described by

$$\mathbf{h}_u^{*(k+1)} = \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathrm{ReLU}(\mathbf{W}^{(k)} \cdot \mathbf{h}_v^{(k)} + \mathbf{e}_{vu}^{(k)})}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}} \tag{4.2}$$

where $\mathbf{h}$ is the hidden embedding and $\mathbf{e}$ is the edge embedding.

We adapt GIN (see section 2.1.2) by using GINE from [21] which is already implemented in PyTorch Geometric [22]. GINE is a variation of GIN that incorporates an edge embedding by summing up the hidden embedding and the edge embedding before applying a ReLU activation function. Mathematically we can describe the convolution as

$$\mathbf{h}_u^{*(k+1)} = \mathrm{MLP}^{(k)}\left((1+\epsilon) \cdot \mathbf{h}_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \mathrm{ReLU}(\mathbf{h}_v^{(k)} + \mathbf{e}_{vu}^{(k)})\right) \tag{4.3}$$

where $\mathbf{h}$ is the hidden embedding and $\mathbf{e}$ is the edge embedding.

Instead of the original GAT (see section 2.1.3) we use GATv2 [23] which improves the standard version. GATv2 is already implemented in PyTorch Geometric [24]. Mathematically we can describe the convolution as:

$$\mathbf{h}_u^{(k+1)} = \alpha_{u,u}\mathbf{h}_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \alpha_{u,v}\mathbf{h}_v^{(k)} \tag{4.4}$$

where $\alpha_{u,v}$ denotes attention coefficients computed as:

$$\alpha_{u,v} = \frac{\exp(\mathbf{a}^\top \text{LeakyReLU}(\mathbf{W}^{(k)}[\mathbf{h}_u^{(k)} \oplus \mathbf{h}_v^{(k)} \oplus \mathbf{e}_{vu}^{(k)}]))}{\sum_{v' \in \mathcal{N}(u) \cup \{u\}} \exp(\mathbf{a}^\top \text{LeakyReLU}(\mathbf{W}^{(k)}[\mathbf{h}_u^{(k)} \oplus \mathbf{h}_{v'}^{(k)} \oplus \mathbf{e}_{v'u}^{(k)}]))} \tag{4.5}$$

where $\mathbf{a}$ is a trainable attention vector, $\mathbf{W}$ is a trainable parameter matrix, $\oplus$ is the concatenation operation, $\mathbf{h}$ is the hidden embedding and $\mathbf{e}$ is the edge embedding.

Finally, we implement three different ways of computing the final node embeddings. The first option is to simply use the node embeddings after the last layer. The other two options include *jumping knowledge* [25], i.e. the model combines the node embeddings resulting from every GNN layer. Combining them can be taking the mean embedding or summing all embeddings up (see figure 4.3).

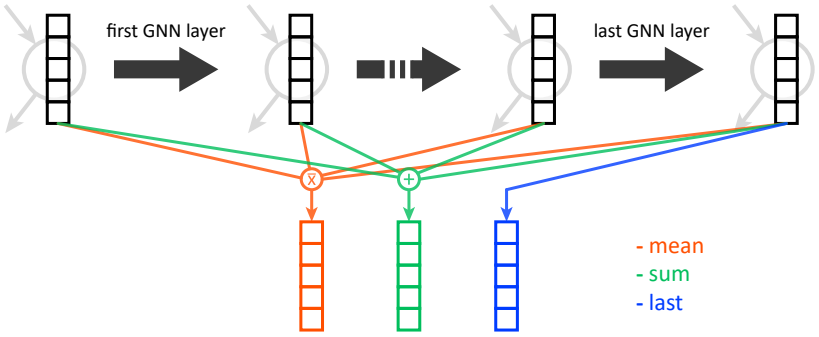

Figure 4.3: The three different jumping knowledge options shown on one node. *Sum* and *mean* aggregate the embedding of resulting from every GNN layer while *last* simply takes the embedding resulting after the last GNN layer.

4.3.3 Node Decoder

After computing the final node embeddings in the GNN part of our model, we select the final embedding of the masked node and pass it through a small neural network to predict the type of the node. We call this neural network the *decoder*.

Let the final node embedding be of size *emd_dim*. Our decoder consists of one hidden layer with size $2 \cdot$ *emb_dim* as well as a final layer of size *# of node types*. One might also choose to use a more complex neural network as the decoder. While this might improve the model we want to focus on the GNN in this work and thus keep the decoder simple.

4.3.4 Optimizer and Loss Function

We use the Adam optimizer provided by PyTorch [26] with the default parameters except for the learning rate which we set to $5 \cdot 10^{-5}$ (see section 4.4). As we

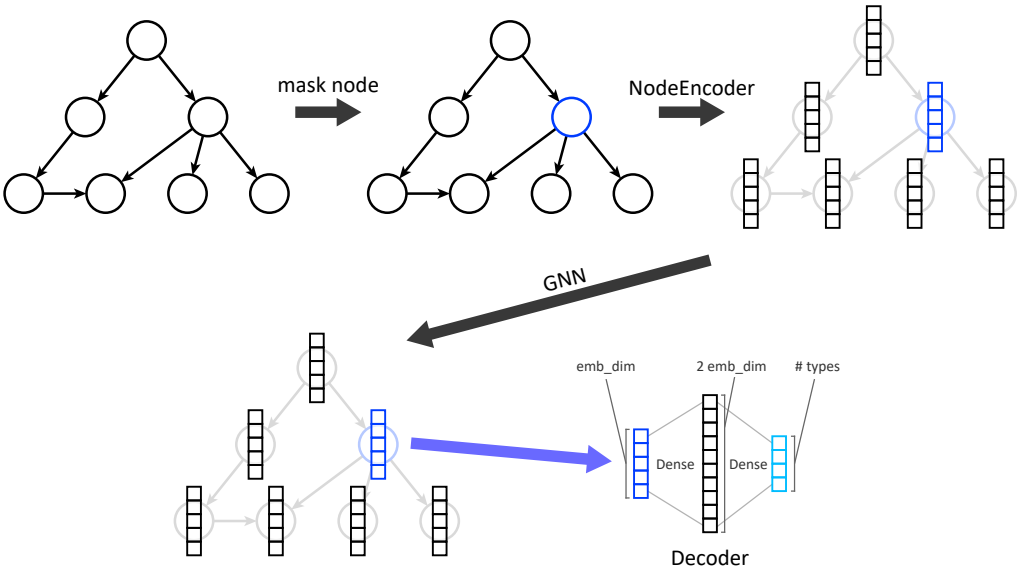

Figure 4.4: Diagram of the Semantic Node Type Prediction model. The model takes a graph with a masked node as the input. First, the NodeEncoder generates the initial node embeddings. Then, the GNN layers compute the final node embeddings. Finally, the decoder, a small neural network, predicts the type based on the final node embedding of the masked node.

are performing a classification task with our model, we use cross-entropy as our loss function.

4.3.5 Backward Edges

SAST graphs are directed graphs and thus nodes that are close to the root can only learn from the few nodes that are even closer to the root. All nodes further down the tree that do not have a path that leads back to some node further up the tree will never share any information with that node. A simple yet effective solution that we apply in this work is to add an edge (v, u) for every edge (u, v) in the graph. We call those edges *backward edges*. Backward edges allow a node to learn from nodes in the data flow direction as well as the opposite data flow direction. We also add another edge attribute that indicates if an edge is a backward edge or not. This ensures that the graph still contains the information about the original data flow direction.

4.4 Results

Backward Edges

Transforming the graph by adding backward edges as described in section 4.3.5 results in big performance increases for all GNN types. Thus all of the remaining results are computed using backward edges enabled. Table A.1 shows the difference between using backward edges or not.

Learning Parameters

It has proven to be beneficial to use a learning rate that is lower than the default parameter set by the Adam optimizer. Using the default learning rate, all models tend converge after a few epochs and then start to fluctuate slightly. A learning rate of $5 \cdot 10^{-5}$ works best for us. Figure A.1 shows convergence the different learning rates we tested.

Dropout does not improve our models. With or without dropout the models show no sign of overfitting. Adding dropout rather slows down the convergence of the models, especially the more complex ones such as GAT. However, we still add dropout to the MLPs used by the GIN layers. A comparison between different dropout ratios can be seen in table A.2.

Model Parameters

A larger embedding dimension increases the model's performance. However, with increasing size of each embedding the model's memory footprint increases as well. Thus we choose to use an embedding dimension of 300 (see table A.3).

As mentioned in the section about machine learning on graphs (see section 2.1) the number of GNN layers effects from how many different nodes a node can learn. For this task the impact of the number of layers is different for each GNN type. Using GCN the impact is very little even if only one layer is being used. Generally, more layers perform slightly better. For GIN the impact is again very little but the sweetspot seems to lay at around three layers. For GAT the benefit of having more layers is more noticeable especially between using one or two layers. It performs the best with five to six layers. For all models residual connections improve performance slightly. The effect of residual connections becomes more noticeable with more layers (see table A.4).

The jumping knowledge option does not influence the model's performance by a lot. Further, for each GNN type a different jumping knowledge approach performs best (see table A.5).

Performance Visualization

The confusion matrices show the different node types ordered in descending number of appearances. The top 13 most frequent types, which make up 70% of all node types, are all well predicted by the models. After that, we can see that the models have some small problems differentiating between the *Tuple* and *List* types. The two node types can have a very similar neighborhood in the SAST graph and thus it is hard for the model to differentiate the two types. This is a recurring theme throughout the confusion matrix. As an example the types *Add*, *Mult* and *Sub* are also mistaken often.

Overall the confusion matrices show us that the models are very confident about types that occur often. However, types that occur very little in the data set are often predicted as more common ones by the models.

Below, figure 4.5 shows the confusion matrix of the best performing model. The confusion matrix of the best performing models for the other GNN types can be found in the appendix - see figure A.2 for GIN and figure A.3 for GAT.

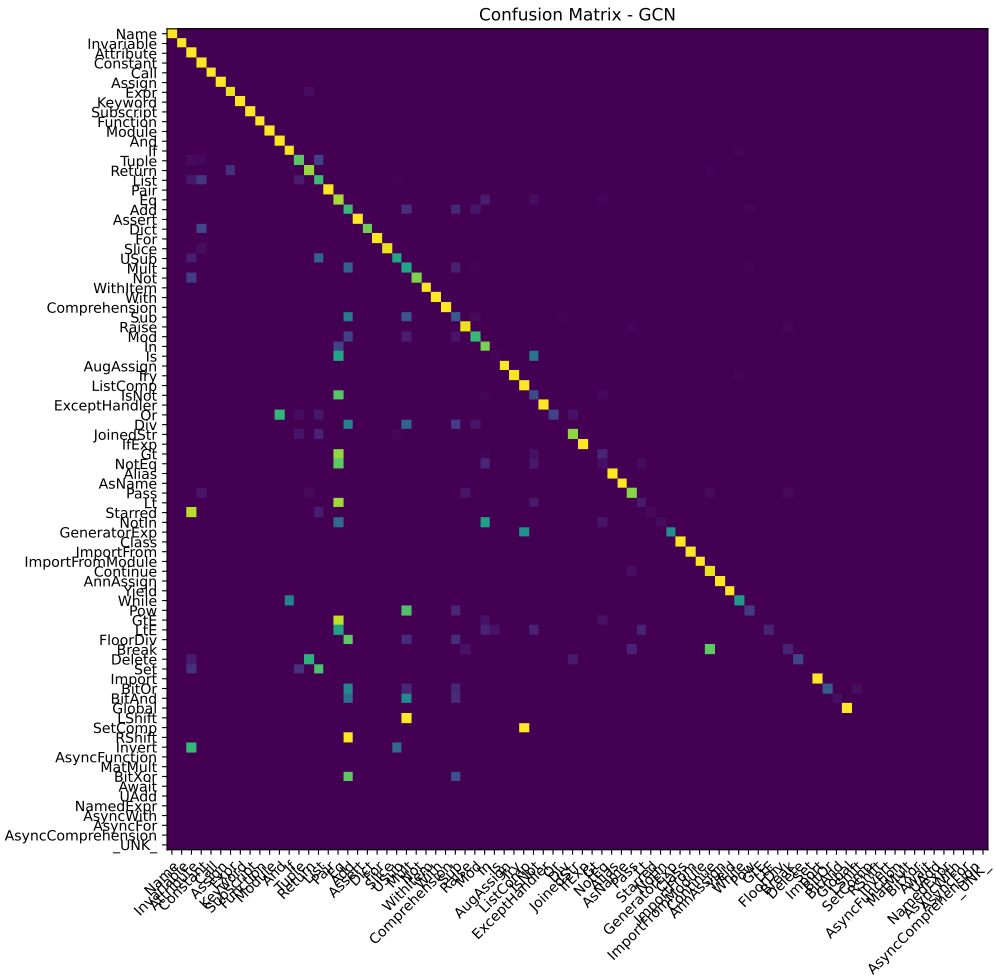

Figure 4.5: Confusion matrix, as described in figure 4.1, of the best performing model. Rows that do not have any entries indicate that the type never occurred in any validation graph. We see that the top 13 most common types are predicted very well. After that, types with similar graph neighborhoods (e.g. *Add*, *Mult* and *Sub* nodes) tend to be confused more often.

Chapter 5

Semantic Link Prediction

5.1 Task Description

The goal of this task is to reliably predict the edges a random invariable shares with other nodes in the SAST graph. To be more precise, we are given an invariable that is only connected to the SAST graph via a directed edge coming from the node that defines the invariable. We then want our model to predict for a given node if the invariable shares an edge to it or not.

5.2 Data, Training and Evaluation

The basic approach of creating the training and validation graphs is the same as in the Semantic Node Type Prediction task (see section 4.2). The main differences are in the pre-transformation of the training and validation graphs.

Once again we use python functions that are crawled from GitHub repositories as described in section 3.2. From each function a SAST graph is generated and stored in the form explained in section 3.2.3.

We create a data set that contains a total of 1000 graphs and split it into 70% (700) training graphs and 30% (300) validation graphs. This data set is a lot smaller than the one we use to test the Semantic Node Type Prediction models due to the training approach we choose to use (see section 5.2.1). We also create a bigger data set containing 5000 graphs and split it into 90% (4500) training graphs and 10% (500) graphs. We will only use this data set to test the effect of using more training graphs on the models.

5.2.1 Training the Model

To simplify the training we only train our model on invariable and node pairings that can be connected via an edge, i.e. we do not consider any node types that are

never connected to an invariable node. We call the node types that potentially share an edge with an invariable *potential neighbors* in this chapter.

Testing has shown that randomly selecting an invariable and a potential neighbor for each graph per epoch leads to far worse results (see section 5.4) than training every possible combination of invariable and potential neighbor per graph. Thus we create one graph (i.e. data object) for every combination of invariable and potential neighbor. All outgoing edges from the corresponding invariable are removed leaving only the ingoing edge. Further, the corresponding invariable and potential neighbor nodes are marked. This results in a significant enlargement in the number of training graphs which increases training time and limits the amount of different training graphs.

5.2.2 Evaluating the Model

To get an evaluation score that represents the performance of our model as consistently as possible we do not want any randomness involved. In contrast to the Semantic Node Type Prediction task, this is already not the case during training. Thus we can use the same strategy to create the evaluation graphs as in training.

As the metrics to evaluate the performance of our models we again use accuracy and confusion matrices. Additionally, as the model only decides between *edge* and *no edge*, we further compute the f1-score. The f1-score is defined as:

$$f1\text{-}score = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{5.1}$$

where: $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$.

5.3 Model Architecture

The architecture of the model used for this task is very similar to the model used for the Semantic Node Type Prediction task (see section 4.3). Thus we will only focus on the differences.

5.3.1 Node Encoder and Graph Neural Network

We adapt the node encoder slightly by adding another Embedding layer to encode the *marked* node attribute. The initial node embedding is now the sum of the type embedding and the mark embedding.

There is no need to change anything in the GNN and we thus use the same GNN architecture as in the Semantic Node Type Prediction task (see section 4.3.2).

5.3.2 Node Decoder

The biggest changes compared to the Semantic Node Type Prediction model are in the decoder. We implement two different approaches - one based on graph classification and the second one based on node classification.

The first approach using a graph classification approach creates a graph embedding from the final node embeddings using a pooling function. Simple pooling functions include summing up all final node embeddings, taking the maximum or mean final node embedding. A more complicated pooling method is to compute an attention score for each embedding and then summing up the embeddings after the attention score is applied. Another more complicated pooling method is the Set2Set approach [27]. After having computed the graph embedding, we pass it through a small neural network similar to the Sequential Node Prediction task as described in figure 5.1.

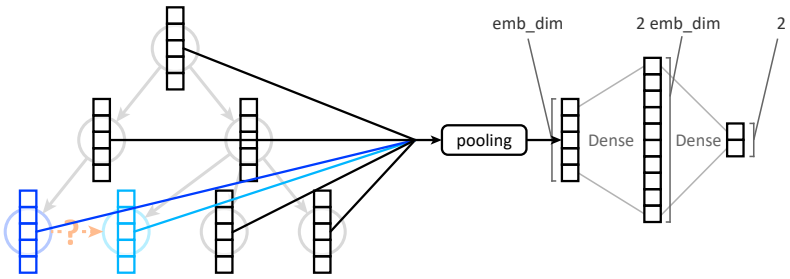

Figure 5.1: Diagram of the decoder using the graph classification approach.

The second approach using a node classification approach, selects the final node embedding from both the marked invariable and the marked neighbor. It then first passes both embeddings through the same small neural network resulting in two tensors of size 32. Finally the two tensors are concatenated and passed through a dense layer resulting in the final tensor (see figure 5.2).

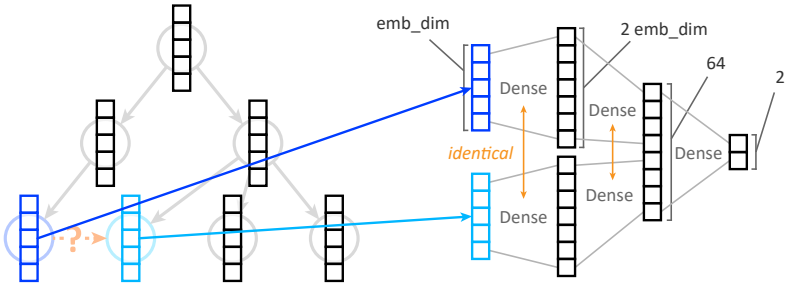

Figure 5.2: Diagram of the decoder using the node classification approach.

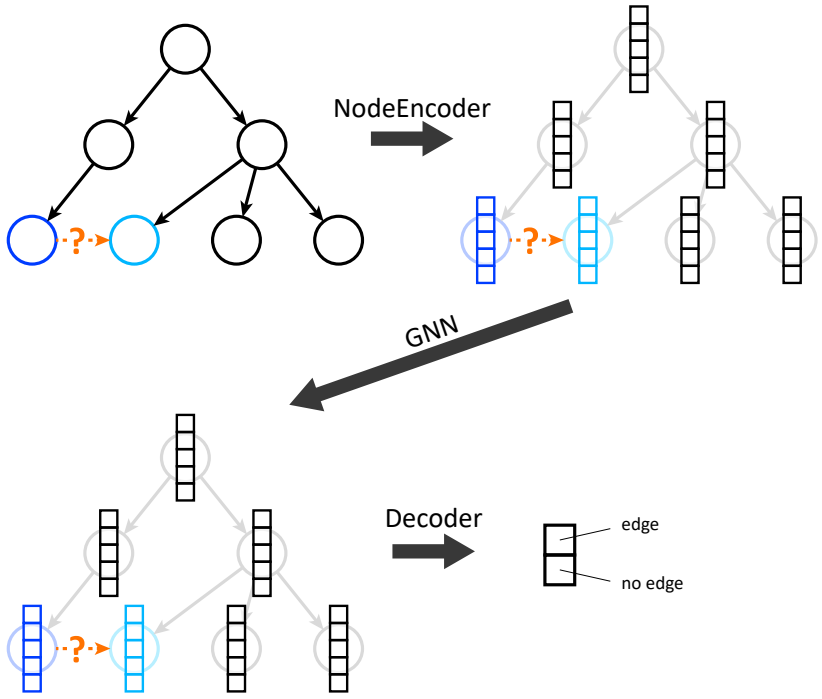

Figure 5.3: Diagram of the Semantic Link Prediction model. The model takes a graph with a marked invariable (without any outgoing edges) and a marked potential neighbor as the input. First, the NodeEncoder generates an embedding for each node. Then the GNN layers compute the final node embeddings. Finally, the decoder generates a tensor with two values from the final node embeddings depending on the approach. If the first value is bigger than the second one the model predicts an edge otherwise it predicts no edge.

5.3.3 Optimizer and Loss Function

As in the Semantic Node Type Prediction task, we use the Adam optimizer provided by PyTorch [26] with the default parameters except for the learning rate which we set to $5 \cdot 10^{-5}$ for the graph classification approach (see section 5.4.2) and $5 \cdot 10^{-6}$ for the node classification approach (see section 5.4.2). We will once again be using cross-entropy as our loss function because we are performing a classification task.

5.3.4 Master Node

As the invariable for which we want to predict the edges is only connected to the graph via the edge to the node that defines it, it is hard for the node to learn anything about the graph and it takes many GNN layers. A potential solution to this issue is to add an additional node that is connected to every node in the

graph. We will call this node the *master node*. The idea is that every node can learn from every other node via the master node. In order to tell the model that an edge is connecting a node in the graph to the master node we will set the type of all those edges to a new additional type.

5.4 Results

Training Approach

We first tried training the models using the same approach used in the Semantic Node Type Prediction task, i.e. we choose a random invariable and potential neighbor pairing per graph per epoch. However, this approach results in very poor results regarding the f1-score no matter the exact model type. Training the model on all pairings of an invariable and a potential neighbor, as described in section 5.2.1, results in far better results even though we cannot train it on as many different graphs as in the first approach. Table B.1 shows the performance differences using the two training approaches.

Generally, using a bigger data set does not change the performances of the models significantly while increasing the training time drastically. Thus, due to resource limitations we will use the smaller data set containing 1000 graphs for all further experiments.

Backward Edges and Master Node

As backward edges are crucial for the model's performance in the Semantic Node Type Prediction task, we will also use them in this task without further conducting any experiments.

When using the graph classification approach, adding a *master node* to the graph, as described in section 5.3.4, increases performance for GCN and GIN significantly but not GAT. Using the node classification approach, the performance increases only slightly for each GNN type. We will use the approach of adding *master nodes* for all further experiments as it almost always improves the model's performance.

5.4.1 Graph Classification Approach

Generally, while achieving good performance results, all models using the graph classification approach suffer from a lot of instability. Some models start very well only to get worse after one epoch until suddenly improving again. The initialization seems to play a big role on the overall performance of the models. This is different for the node classification approach (see section 5.4.2).

Training Parameters

We use a learning rate of $5 \cdot 10^{-5}$. There is not a big difference to using a slightly higher learning rate of $1 \cdot 10^{-4}$. Using a lower learning rate results in slow convergence without improving the final result (see figure B.1).

As already mentioned, the validation f1-scores for the graph classification approach are generally very unstable. This is also the case when testing different dropout ratios. The best f1-scores in table B.4 occur in the first or second epoch except for GAT with a dropout ratio of 0.3. We will use a dropout ratio of 0.3 for all further experiments as this ratio resulted in the most stable scores independent of a strong or weak start.

Model Parameters

For both the GIN and GAT models, as in the Semantic Node Type Prediction task, a bigger embedding dimension improves the model's performance. Again with bigger embeddings the model's memory footprint increases as well. What is different is that for GCN the models with smaller embedding dimensions seem to perform better (see table B.8.

Generally, the models perform well with four or more layers except for GAT which has its performance peak at three layers. Residuals almost always improve the models slightly (see table B.7).

For GCN every pooling method except for *mean* seems to perform fairly well with the *sum* pooling method resulting in the best performance. For GIN the *max* pooling method as well as the more complex ones result in good performance. Finally, for GAT only the more simple pooling methods work well.

5.4.2 Node Classification Approach

While the models using the node classification approach achieve equally good results as models using the graph classification approach, it has to be said that the node classification approach achieves far more consistent results.

Training Parameters

For this approach we chose a learning rate of $5 \cdot 10^{-6}$. Using this learning rate we get rather slow convergence compared to the other learning rates we tested. However, once almost converged the model performs much more stable than using other learning rates (see figure B.2).

Compared to the graph classification approach the best validation f1-scores usually occur at around 6 epochs. The best scores are achieved with a dropout

ratio of 0.3 and thus we will use this for all future experiments.

Model Parameters

A larger embedding dimension improves all models while also increasing their memory footprint. We choose to use an embedding size of 300 (see table B.9).

For all GNN types using three GNN layers works well. GCN achieves the same performance for all experiments using one up to five layers. For GIN and GAT the best performing models use around three GNN layers. Residuals improve the performance (see table B.10).

5.4.3 Concrete Code Examples

Visualizing the predictions on concrete code examples allows us to see in which cases the model is accurate and where it struggles. The model performs well when there are nodes in the graph that are missing a dataflow edge. An example for this can be seen in figure 5.4 where the model is predicting all edges originating from the invariable representing the argument variable *a*. We can see that the model accurately predicts the edge that is missing. The predictions also indicate that the model is sure that there are no further edges which is also correct.

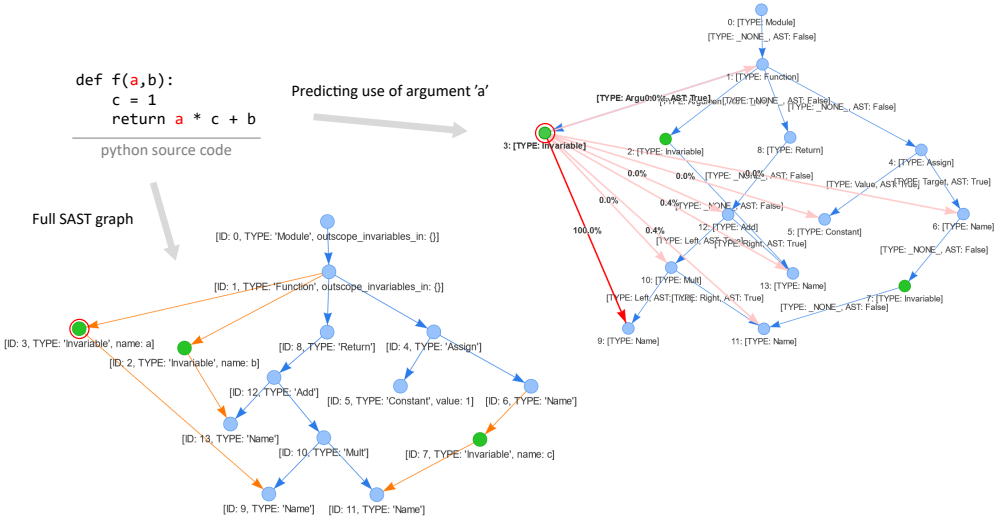

Figure 5.4: A concrete code example showing the original SAST graph and the model's edge predictions for the invariable with index 3 in the modified graph. The model correctly predicts all missing edges.

An example for which the model struggles to predict the correct edges can be seen in figure 5.5. Here the model has to predict the edges originating from the invariable representing the assignment of variable *b* inside the if-clause. Compared

to the previous example, there is no node in the graph that is missing a dataflow edge as the node with index 17 has a second ingoing edge from the invariable with index 6 in the original graph. While we see that our model anticipates the original edge with a 9.5% probability, it fails to predict the edge. This example nicely shows the general weakness of our models.

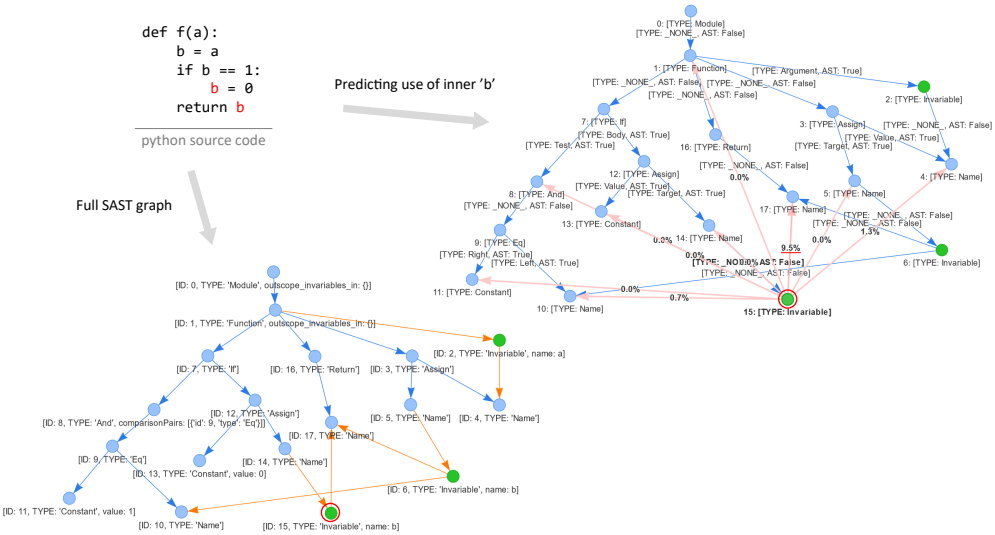

Figure 5.5: A concrete code example showing the original SAST graph and the model's edge predictions for the invariable with index 15 in the modified graph. The model fails to predicts the missing edge.

Chapter 6

Future Work

While we conducted experiments on all hyper-parameters and tested their effect on the model, we did not optimize every single aspect due to time limitations. Thus with more extensive testing and potentially bigger data sets one might still enhance performance.

Performance enhancements may also be achieved by changing the graph generation process. We have already mentioned some limitations of the SAST graph generation procedure, that we use including potential methods to fix them (see section 3.3). However, one might also choose to use a completely different graph generation approach. An example would be to generate control flow like graphs such as the ones used in compilers. Changes can also be more subtle ones such as adding or changing attributes for both the nodes and edges to supply the model with more information about the source code. One might include the value of constants in the data set, something that is definitely needed if we want to solve code classification tasks.

Further, the node- and edge-encoder as well as the decoder are all parts of the model we did not further investigate in this work. Using more powerful encoders may improve the model's performance significantly. One might train an auto encoder first and then use it as the encoders as an example. This might even be necessary if one were to use more attributes for both nodes and edges as our approach may no longer work

Finally, the results of this work may be extended to create models that solve more complex code-related tasks such as code completion or code classification. A code completion model might predict nodes and links for an incomplete SAST graph in an iterative way in order to complete the graph. A code classification model might use a graph classification approach similar to the one applied in the Semantic Link Prediction task. As the model is based on data flow graphs it could allow it to understand the code in a semantic way. To solve such tasks one might have to combine the GNN based models used in this work with other machine learning methods. Such methods include natural language based models that could be directly applied to the source code as well as path based approaches that can be applied to the generated graphs.

Chapter 7

Conclusion

In this work we have adapted the SAST graph generation approach to create graphs that represent both the syntactic structure as well as the semantic data flow information of some source code. We presented a GNN model architecture that solves the task of predicting a node type in a SAST graph and adapted this architecture to solve the task of predicting links for an invariable in a SAST graph. Our node type prediction models achieved 97.6% accuracy on a data set consisting of python functions crawled from GitHub. We showed the importance of adding backward edges and analyzed the effects of different hyperparameters on our models. Adapting the model architecture as well as the training approach, we achieved 94.5% F1-score on the Semantic Link Prediction task using two different decoder approaches. We analyzed the impact of adding a master node and found that it can help some models to reach the performance that other models attain without a master node. It remains to be seen what the full potential of the model architecture is when using superior encoders and decoders that encode more information in a more efficient way and interpret the results of the GNN in a more meaningful way respectively.

Bibliography

[1] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[2] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," 2020. [Online]. Available: https://arxiv.org/abs/2001.00059

[3] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[4] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," 2022. [Online]. Available: https://arxiv.org/pdf/2111.02671

[5] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 3, pp. 1–159.

[6] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," 2019. [Online]. Available: https://arxiv.org/abs/1909.03211

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016. [Online]. Available: https://arxiv.org/abs/1609.02907

[8] B. Weisfeiler and A. Leman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," 1968.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018. [Online]. Available: https://arxiv.org/abs/1810.00826

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017. [Online]. Available: https://arxiv.org/abs/1710.10903

[11] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 12–27. [Online]. Available: https://doi.org/10.1145/73560.73562

[12] J. Flunger, "Graph pattern mining in code," https://pub.tik.ee.ethz.ch/students/2021-HS/BA-2021-35.pdf, 2022.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 25–35. [Online]. Available: https://doi.org/10.1145/75277.75280

[14] "Pytorch geometric data documentation," https://pytorch-geometric.readthedocs.io/en/latest/modules/data.html#torch_geometric.data.Data, accessed: 2022-07-13.

[15] "Pytorch geometric dataloader documentation," https://pytorch-geometric.readthedocs.io/en/latest/modules/loader.html#torch_geometric.loader.DataLoader, accessed: 2022-07-13.

[16] "Ogb code2 example on github," https://github.com/snap-stanford/ogb/tree/master/examples/graphproppred/code2, accessed: 2022-07-11.

[17] "Open graph benchmark website," https://ogb.stanford.edu/, accessed: 2022-07-11.

[18] "Pytorch geometric website," https://www.pyg.org/, accessed: 2022-07-11.

[19] "Pytorch website," https://pytorch.org/, accessed: 2022-07-11.

[20] "Pytorch embedding documentation," https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html, accessed: 2022-07-22.

[21] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, "Strategies for pre-training graph neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1905.12265

[22] "Pytorch geometric gine documentation," https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GINEConv, accessed: 2022-07-12.

[23] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" 2021. [Online]. Available: https://arxiv.org/abs/2105.14491

[24] "Pytorch geometric gatv2 documentation," https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GATv2Conv, accessed: 2022-07-12.

[25] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "Graph neural network-based android malware classification with jumping knowledge," Jan. 2022. [Online]. Available: https://arxiv.org/pdf/2201.07537

[26] "Pytorch adam documentation," https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, accessed: 2022-07-13.

[27] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," 2015. [Online]. Available: https://arxiv.org/abs/1511.06391

Appendix A

Semantic Node Type Prediction Results

Note: All experiments were conducted on the data set containing 100,000 graphs described in section 4.2. All test scores come from a single test run due to time and resource limitations. Thus one has to interpret the results with some care.

GNN type	GCN	GIN	GAT
without backward edges	0.8285	0.7322	0.7204
with backward edges	**0.9764**	**0.9740**	**0.9708**

Table A.1: Comparing best validation accuracies with or without using backward edges.

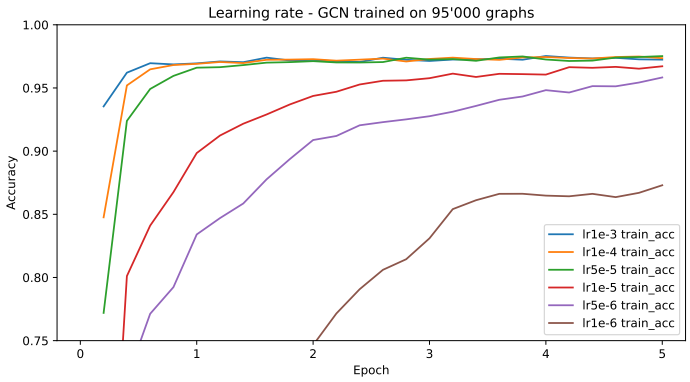

Figure A.1: Convergence of training accuracies using different learning rates and GCN as the GNN type.

Dropout	None	0.3	0.5
GCN	**0.9733**	0.9694	0.9649
GIN	**0.9688**	0.9661	0.9615
GAT	**0.9650**	0.8705	0.4535

Table A.2: Comparing the best validation accuracies for different dropout ratios and GNN types.

Embedding Dimension	50	100	300	600
GCN	0.9675	0.9732	**0.9764**	0.9760
GIN	0.9584	0.9693	0.9747	**0.9749**
GAT	0.9570	0.9677	**0.9732**	*oom*

Table A.3: Comparing the best validation accuracies for different embedding dimensions and GNN types. '*oom*' indicates that the model ran out of memory.

Layers	1	2	3	4	5	6
GCN	0.9703	0.9754	0.9764	0.9761	0.9760	0.9761
GCN *(res)*	0.9711	0.9749	0.9764	0.9757	**0.9768**	0.9766
GIN	0.9713	0.9738	0.9740	0.9730	0.9727	0.9709
GIN *(res)*	0.9704	0.9744	**0.9747**	0.9744	0.9738	0.9734
GAT	0.9571	0.9698	0.9708	0.9711	0.9733	0.9724
GAT *(res)*	0.9613	0.9720	0.9732	0.9748	**0.9754**	**0.9754**

Table A.4: Comparing the best validation accuracies for different numbers of GNN layers, with or without residual connections, and GNN types. '*(res)*' indicates that residual connections were used.

Jumping Knowledge	last	mean	sum
GCN	**0.9764**	0.9762	0.9743
GIN	0.9747	**0.9748**	0.9730
GAT	0.9732	0.9726	**0.9762**

Table A.5: Comparing the best validation accuracies for different jumping knowledge approaches and GNN types.

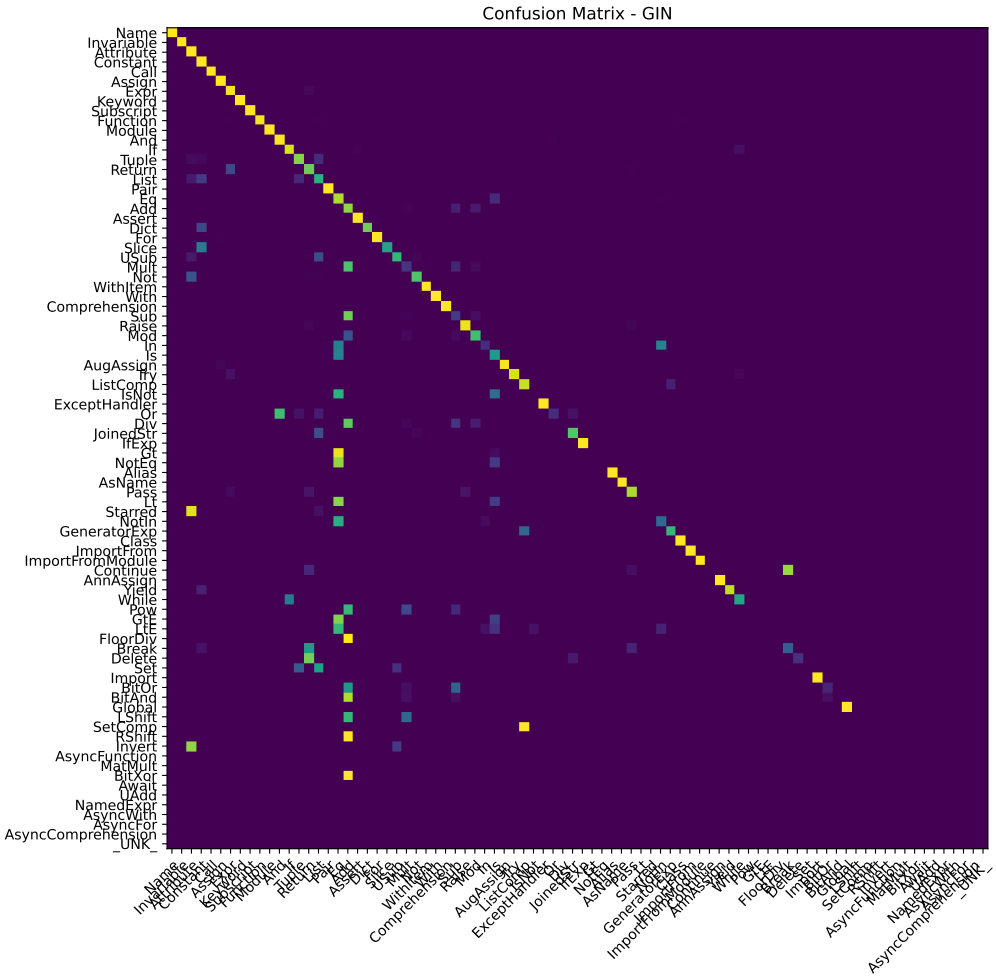

Figure A.2: Confusion matrix of the best performing GIN model. Confusion matrix as described in figure 4.1. Rows that do not have any entries indicate that the type never occurred in any validation graph.

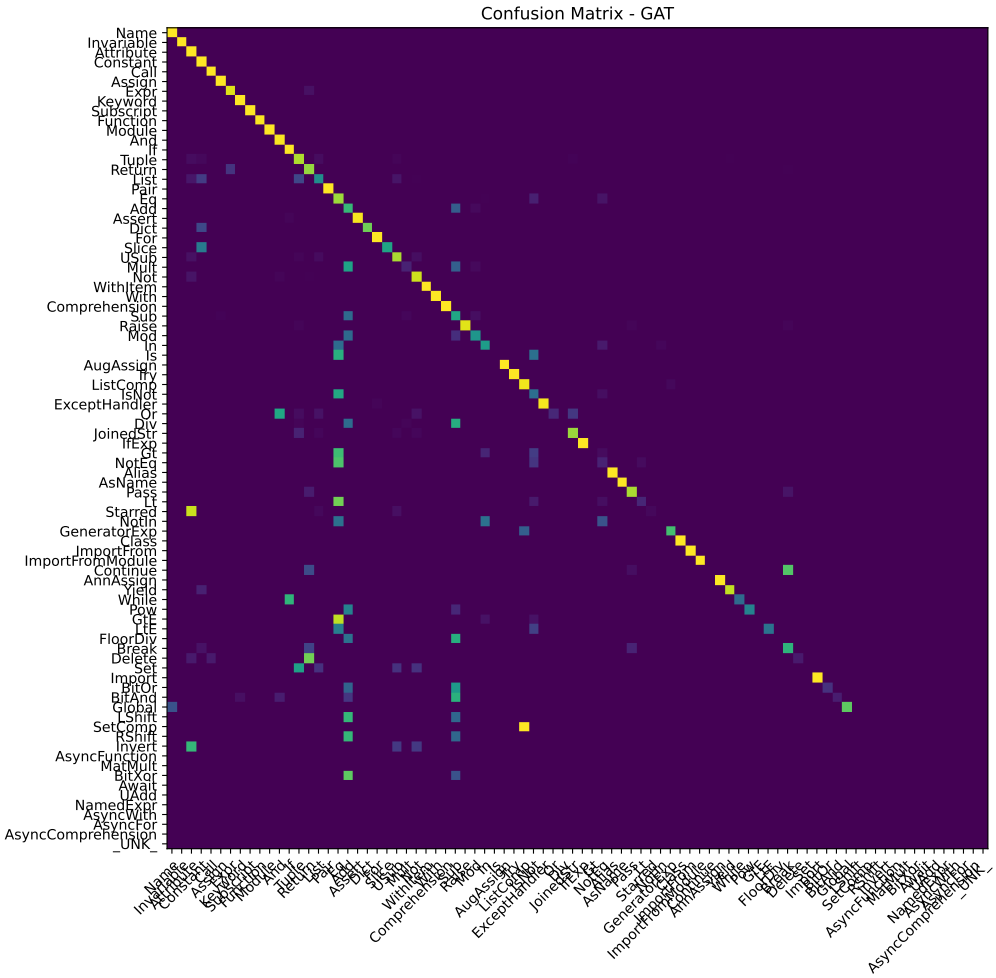

Figure A.3: Confusion matrix of the best performing GAT model. Confusion matrix as described in figure 4.1. Rows that do not have any entries indicate that the type never occurred in any validation graph.

Appendix B

Semantic Link Prediction Results

Note: All experiments were conducted on the data set containing 1000 graphs described in section 5.2.1. All test scores come from a single test run due to time and resource limitations. Thus one has to interpret the results with some care. Especially, the results for the graph classification approach as the models are less stable and more dependent on the initialization.

Graph classification approach

GNN type	GCN	GIN	GAT
first training approach	0.4562	0.4470	0.3964
second training approach	**0.9410**	**0.9315**	**0.9270**

Table B.1: Comparing best validation f1-scores using the first training approach, i.e. selecting a random invariable and potential neighbor pairing per graph per epoch, and the second approach, i.e. testing every possible pairing. For the first training approach we used the same data set as in the Semantic Node Type Prediction task.

Graph classification approach

GNN type	GCN	GIN	GAT
without master node	0.8801	0.8202	**0.9408**
with master node	**0.9410**	**0.9315**	0.9270

Table B.2: Comparing best validation f1-scores with or without using a master node.

Node classification approach

GNN type	GCN	GIN	GAT
without master node	**0.9451**	0.9427	0.9435
with master node	**0.9451**	**0.9451**	**0.9451**

Table B.3: Comparing best validation f1-scores with or without using a master node.

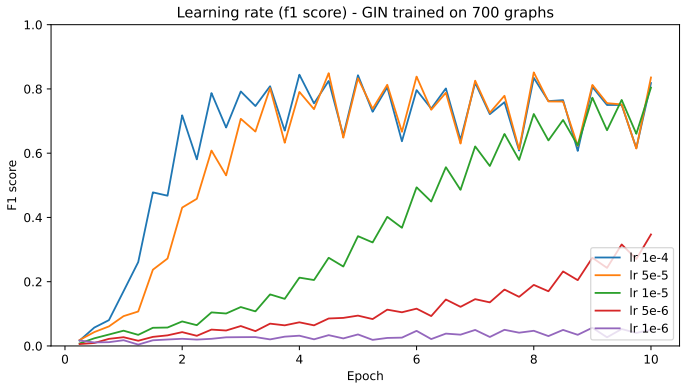

Figure B.1: Convergence of training f1-score using different learning rates, the *graph classification* approach and GIN as the GNN type.

Graph classification approach

Dropout	None	0.3	0.5
GCN	0.8925	**0.9400**	0.9341
GIN	**0.9333**	0.9089	0.9065
GAT	**0.9450**	0.9447	0.0817

Table B.4: Comparing the best validation f1-scores for different dropout ratios and GNN types using the graph classification approach.

Node classification approach

Dropout	None	0.3	0.5
GCN	0.9421	0.9445	**0.9451**
GIN	0.9419	**0.9451**	0.9433
GAT	0.9428	**0.9451**	0.4151

Table B.5: Comparing the best validation f1-scores for different dropout ratios and GNN types using the node classification approach.

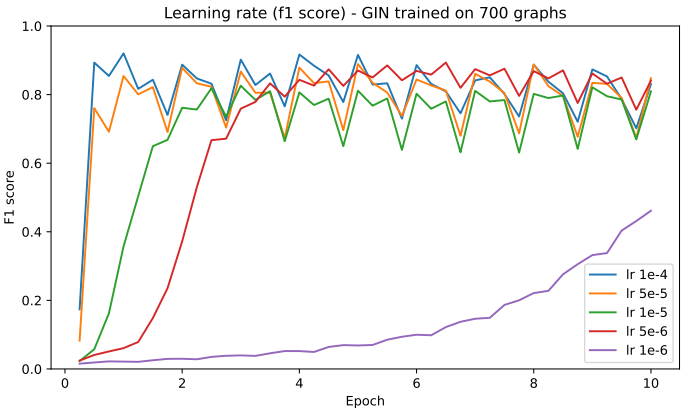

Figure B.2: Convergence of training f1-score using different learning rates, the *node classification* approach and GIN as the GNN type.

Graph classification approach

Embedding Dimension	50	100	300	600
GCN	**0.9451**	**0.9451**	0.8628	0.8001
GIN	0.9061	0.9265	0.9302	**0.9389**
GAT	0.5891	0.0000	0.9402	**0.9451**

Table B.6: Comparing the best validation f1-scores for different embedding dimensions and GNN types using the graph classification approach.

Graph classification approach

Layers	1	2	3	4	5
GCN	0.9408	0.8721	0.9078	0.9410	0.9410
GCN *(res)*	0.9410	0.8233	0.9092	0.9410	**0.9451**
GIN	0.8749	0.9354	0.9315	0.9175	0.9360
GIN *(res)*	0.8481	0.6602	0.8598	0.9315	**0.9402**
GAT	0.9449	0.9416	0.8558	0.7030	0.9365
GAT *(res)*	0.7877	0.7889	**0.9450**	0.9270	0.8365

Table B.7: Comparing the best validation f1-scores for different numbers of GNN layers, with or without residual connections, and GNN types using the graph classification approach. '*(res)*' indicates that residual connections were used.

Graph classification approach

Pooling	mean	sum	max	attention	set2set
GCN	0.9092	**0.9451**	0.9364	0.9395	0.9410
GIN	0.8598	0.9298	0.9410	0.9410	**0.9445**
GAT	**0.9450**	0.9442	0.9156	0.2373	0.8805

Table B.8: Comparing the best validation f1-scores for different pooling methods and GNN types using the graph classification approach.

Node classification approach

Embedding Dimension	50	100	300	600
GCN	0.9410	0.9410	**0.9451**	**0.9451**
GIN	0.8843	0.9410	**0.9451**	**0.9451**
GAT	0.6640	0.9442	**0.9451**	**0.9451**

Table B.9: Comparing the best validation f1-scores for different embedding dimensions and GNN types using the node classification approach.

Node classification approach

Layers	1	2	3	4	5
GCN	**0.9451**	0.9426	0.9444	0.9416	0.9448
GCN *(res)*	0.9446	**0.9451**	**0.9451**	**0.9451**	**0.9451**
GIN	0.9410	0.9410	0.9442	0.2078	0.2181
GIN *(res)*	0.9444	0.9448	**0.9451**	**0.9451**	0.9426
GAT	0.9406	0.9365	0.4558	0.0404	0.9419
GAT *(res)*	0.9410	0.7971	**0.9451**	0.9410	0.9400

Table B.10: Comparing the best validation f1-scores for different numbers of GNN layers, with or without residual connections, and GNN types using the node classification approach. '*(res)*' indicates that residual connections were used.