Spring 2020

Predicting Imports in Java Code with Graph Neural Networks

Aleksandr Fedchin
Bard College

Predicting Imports in Java Code with Graph Neural Networks

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Aleksandr Fedchin

Annandale-on-Hudson, New York
May, 2020

Abstract

Programmers tend to split their code into multiple files or sub-modules. When a program is executed, these sub-modules interact to produce the desired effect. One can, therefore, represent programs with graphs, where each node corresponds to some file and each edge corresponds to some relationship between files, such as two files being located in the same package or one file importing the content of another. This project trains Graph Neural Networks on such graphs to learn to predict future imports in Java programs and shows that Graph Neural Networks outperform various baseline methods by a wide margin.

Contents

Acknowledgments

One of the most fortunate things that has happened to me in the last four years was meeting Sven Anderson in my first semester at Bard. Thank you so much for all the things that I learned from you, for the wonderful two months of BSRI, for advising me on courses and graduate schools, for encouraging me to continue working on both of my Senior Projects when I most needed it, and for being always there to help with any problem!

The idea to write a Senior Project on import prediction was pitched to me by Vitaly Khudobakhshov, to whom I am immensely grateful for it. Thank you so much for the idea, for advising me on the project, and for the unforgettable internship experience at JetBrains!

Thank you so much:

to Jeff Foster, for the opportunity to continue type prediction research at Tufts.

to Keith O'Hara and Arseny Khakhalin, for reading this text.

to Olga Voronina, for helping me get through my four years at Bard.

to Rob Cioffi, for advising me to do two separate Senior Projects.

to my mom and dad, for helping me to believe in myself.

1
Introduction to the Import Prediction Task

Code tends to be less ambiguous than natural language and allows one to prove statements about it. For this reason, code analysis had originally been a proof-oriented discipline. In recent years, however, machine learning has become prominent in static code analysis research. Bavishi et al. (2018) used statistical methods to infer original variable names from code that was intentionally minified, that is in which all the variable identifiers were replaced with random meaningless strings. Allamanis et al. (2017) used machine learning to locate bugs in large open-source projects. These are but a few examples of machine learning being employed to solve problems that would previously have been approached from a purely analytical standpoint. For a thorough review of the literature on the matter, see a paper by Allamanis, Barr, Devanbu, and Sutton (2017).

In this project, I build on past machine learning and static code analysis research in an attempt to develop a novel approach to code completion in Java. Code completion aims to speed up the coding process by predicting what a programmer would want to type next. More specifically, I focus on predicting which local classes or interfaces a programmer might want to use among those not yet imported. Local classes are classes defined within the same project a programmer is currently working on.

Prediction of future imports in Java code is a problem that has been addressed before, but existing approaches are relatively inaccurate in predicting class-level imports and even less so in predicting local class-level imports. The most recent work on import prediction relies on gathering global information about import co-occurrence to then compute vector representations of each potential import (Theeten et al., 2019). This approach appears to be effective at predicting library-level imports, but it is unlikely to be feasible for class prediction: this would require one to compute and store vector representations of every single widely-used Java class. Moreover, this method can hardly be used to predict local imports because there is unlikely to be enough co-occurrence statistics to be gathered from a single project.

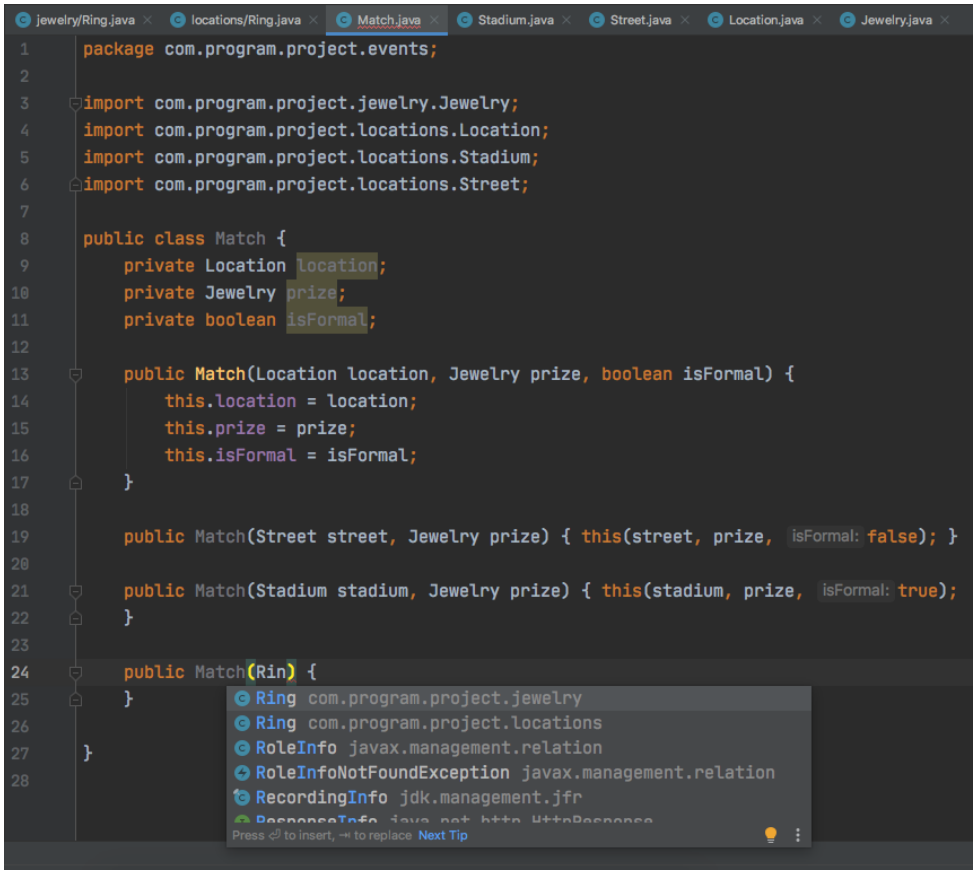


Figure 1.0.1. An example of a potentially inaccurate import prediction by Intellij IDEA (Version 2020.1). The IDE proposes to import class "Ring" from the "jewelry" package even though the context suggests that importing the identically named class from the "locations" package might be preferable. Note that the "locations" package is more extensively used in the code. Moreover, all written constructors of the "Match" class take an instance of a class extending "Location" as the first parameter.

Local imports can sometimes be successfully predicted by IDEs like IntelliJ IDEA. As far as I have been able to determine experimentally, IntelliJ IDEA relies on computationally inexpensive rule-guided heuristics to guide local import prediction. At times, these heuristics are not enough: Figure 1.0.1 shows one scenario, in which IntelliJ IDEA ranks a less probable import candidate above everything else.

This report presents a new method for predicting local imports. The core idea is to model relationships between classes and interfaces in a Java project with a graph and then use Graph Neural Networks to infer future imports from the graph. A node in such a graph corresponds to a particular file (more properly referred to as a compilation unit), in which some class or interface is defined. An edge between two nodes marks some relationship between the two corresponding files.

Figure 1.0.2 presents an example of such a graph built for one of the GitHub repositories in my dataset. Grey undirected edges connect compilation units defined within the same package. Black directed edges correspond to import statements. Blue edges mark class inheritance or interface implementation. I call graphs like this **repository graphs**.

In my approach, I gather a dataset of Java repositories from GitHub (Chapter 3), build corresponding repository graphs, and extract some relevant information about each class in the repository from the source code (Chapter 4). Graph Neural Networks (GNNs) are then trained to predict future imports from the geometry of a repository graph (Chapter 2). During testing, a network is presented with several potential imports that could be added to some file in a repository, one of the candidates corresponding to the actual import statement found in the code. The accuracy with which the network selects the correct import is used as the evaluation metric. The accuracy naturally varies with the number of candidates the network is presented with; it is at its highest when the network has to make a choice between just two candidates, which is the scenario referred to as the **binary import prediction task** throughout this text. As detailed in Chapter 5, the comparison between the baseline results achieved by other means suggests that GNNs are well suited for this task.

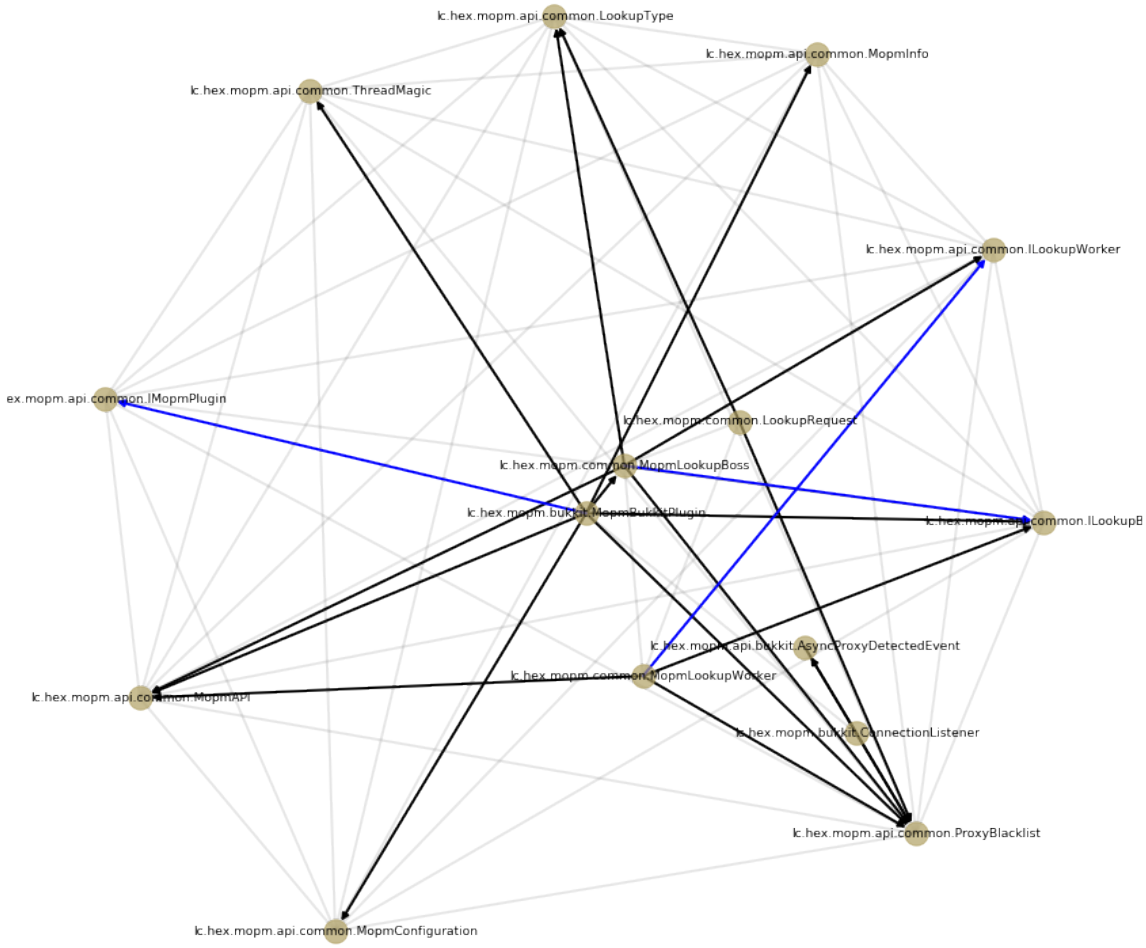


Figure 1.0.2. An example of a repository graph built for the GitHub Minecraft Open Proxy Monitor project (github.com/0x277F/mopm). Grey undirected edges connect compilation units defined within the same package. Black directed edges correspond to import statements. Blue edges mark class inheritance or interface implementation.

The notion of embedding is critical in this report: embedding is a vector that encodes the properties of a particular word, image, sentence, or some other object. Such vectors are commonly used as inputs to neural networks and some neural networks learn to compute embeddings. Unlike symbols, which are arbitrary, embeddings are a semantically rich form of representation. Unless an embedding is manually constructed according to some rule, it is usually very difficult to tell what exact property each particular dimension in a vector encodes. Most often, embeddings

are used to compare objects: if embeddings are constructed properly, similar embeddings will correspond to similar objects. For example, one can use an unsupervised learning algorithm to construct embeddings such that the embedding for word "king" minus that for word "man" plus that for word "woman" would be very close to the embedding for word "queen" (Mikolov et al., 2013).

All of my code is available on GitHub[1]. Most of the code is written in Python, although there are several files written in SQL and Java, and a few lines of JavaScript. The reader can reproduce all the steps described in this text by running the code on GitHub provided that they have the necessary libraries installed.

[1] github.com/Dargones/import_prediction

2

Graph Neural Networks: Motivation and Implementation

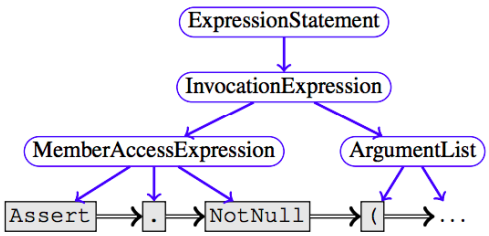
Graph Neural Networks (GNNs) are a type of neural network designed to make inferences about graphs and particular nodes in these graphs. One can think of GNNs as consisting of two separate components. During a forward pass, a GNN first computes node embeddings (also referred to as node representations), which are a set of vectors each describing a single node in the graph. Gated Graph Neural Networks (GGNNs) use Gated Recurrent Units to compute node representations. I outline the algorithm for computing node embeddings at the end of this chapter (Section 2.4), but I do not alter it in any way in my code and refer the reader to the original paper on the topic (Li et al., 2017).

After the node embeddings are computed, a GNN maps from node representations to the output, whatever form the output might take. This final step is highly problem-specific and there are many different ways it could be performed. In my implementation, I mirror the approach that Allamanis et al. (2017) take in solving the VarMisuse task. I describe the VarMisuse task below in order to compare it to the import prediction task and highlight the motivation behind using GNNs to solve both.

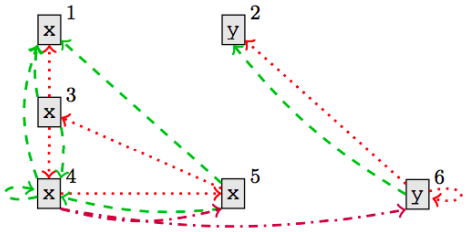
2.1 VarMisuse task: A similar GNN

VarMisuse stands for variable misuse, a situation in which a programmer types the name of an incorrect variable at a specific location in the code. Most often, variable misuse leads to compilation errors; sometimes it leads to runtime errors; and sometimes there are no errors at all, but the program acts in an unintended way. Allamanis et al. train a Graph Neural Network to detect locations in the code, where an incorrect variable might have been used.

To do this, they first parse the piece of code they want to examine to obtain its abstract syntax tree (AST), which is a tree representation of source code with each node corresponding to a programming language construct (such as a while loop or an identifier). They then modify the AST by connecting various nodes with edges according to a set of predefined rules. For instance, for every variable, they introduce edges to connect together all the references to that variable. Figure 2.1.1, which is reproduced from their paper, shows how Allamanis et al. convert an AST to a program graph.



(a) Simplified syntax graph for line 2 of Fig. 1, where blue rounded boxes are syntax nodes, black rectangular boxes syntax tokens, blue edges Child edges and double black edges NextToken edges.



(b) Data flow edges for (x¹, y²) = Foo(); while (x³ > 0) x⁴ = x⁵ + y⁶ (indices added for clarity), with red dotted LastUse edges, green dashed LastWrite edges and dashdotted purple ComputedFrom edges.

Figure 2.1.1. "Examples of graph edges used in program representation." This figure along with its title is reproduced from the paper by Allamanis et al. (2017). Note in particular the LastUse edges with which all the references to the same variable are connected together.

Having constructed a program graph in this way, Allamanis et al. specify a single location in the code that they want to check for variable misuse. That location corresponds to some node in the program graph, and Allamanis et al. remove any edges between this and other nodes that depend on the particular variable referenced at that location. In this manner, information about the variable used in the code at the given location is completely erased from the program

graph. Next, Allamanis et al. run a GGNN to obtain an embedding for the node in question. That embedding encodes the variable usage context.

Finally, Allamanis et al. compile a list of n possible variable identifiers that could have been inserted at the given location (which includes the variable identifier appearing in the code). For each candidate variable, they then update the graph as if that variable had actually been referenced and run the network to obtain a new embedding for that location.

In the end, Allamanis et al. compute $n + 1$ embeddings: a context embedding and n variable-specific embeddings. The variable that the network selects as "correct" is the one that corresponds to an embedding most similar[1] to the context embedding. If an already trained network cannot accurately predict the variable name for some location, it might be the case that the programmer made a mistake and typed an incorrect variable name. With their networks, Allamanis et al. have been able to locate several bugs in widely used software libraries.

Allamanis et al. write that they train the whole system end-to-end with a max-margin objective. It is unclear from the paper what kind of max-margin objective they use, but I have opted for a modified version of the triplet loss described in the next section.

2.2 Adapting GNNs for the import prediction task

The motivation behind using GNNs for the VarMisuse task rests on the assumption that the most logical variable choice must correspond to the smallest change in the structure of the program graph. Graph structure serves as input to GNNs and, therefore, is encoded in the representations GNNs create, meaning that this type of neural network is perfectly suited for the task. The same logic can be applied to import prediction and repository graphs: the addition of a "plausible" import statement to a file should lead to smaller changes in the graph structure and the corresponding embedding than the addition of a completely random import statement. My implementation of GNNs is, therefore, quite similar to that of Allamanis et al.

[1] Similarity is computed by a single-layer neural network trained jointly with the GGNN

Given a repository and a set of potential import statements to be added to a certain file in that repository, I first run the network on the repository graph and record the contextual embedding of the file in question. For every potential import, I then update the graph by adding the corresponding edge to it and run the network on this new graph recording the new embedding for the file under consideration. The import option selected as "correct" corresponds to the embedding most similar to the original one.

During the training stage, I randomly select a file for which the model will make a prediction and a particular import statement in that file which the model will learn to predict. I then remove from the repository graph the edge encoding that import statement and perform the steps described in the previous paragraph.

I train the network with a slightly modified version of the triplet loss, which makes the network learn that embeddings of similar objects must also be similar. In its canonical form, triplet loss takes three embeddings as input. Two embeddings, referred to as the anchor (A) and the positive (P), encode objects that are elements of the same class. The third embedding, known as the negative (N), encodes an object from a different class. Triplet loss minimizes the squared Euclidean distance between the anchor and the positive and maximizes the squared distance between the anchor and the negative. More formally, triplet loss is defined as:

$$L(A, P, N) = max(|A - P|^2 - |N - P|^2 + \alpha, 0)$$

For the purposes of the input prediction task, the anchor is the contextual embedding computed when the target import edge is removed from the graph. The positive is the embedding computed when the graph is left unmodified. There can be arbitrary many negatives, one for each potential import that the network must recognize as illogical. I compute the triplet loss for every one of these and make the network backpropagate the mean. Following Allamanis et al., I use a linear layer to compute the distance between a pair of embeddings. Since this linear layer is trained jointly with the network, I formulate my version of the triplet loss in terms of distances rather than embeddings themselves. The loss function used can, therefore, be described by the following equation:

$$L(D_{a,p}, \{D_{a,n} | n \in N\}) = \frac{\sum_{n \in N} max(D_{a,p} - D_{a,n} + \alpha, 0)}{|N|}$$

Here a is the anchor, p is the positive, N is the set of negatives, $D_{x,y}$ is the distance between x and y, and α is a hyper-parameter, which I set to 0.5 as is commonly done.

2.3 PyTorch vs. Tensorflow

The code for both the original paper on Gated Graph Neural Networks (Li et al., 2017) and the paper on the VarMisuse task (Allamanis, Brockschmidt, et al., 2017) makes use of the TensorFlow library. Nevertheless, I decided to use PyTorch instead of TensorFlow because I have more experience with the former and because PyTorch allows for more convenient debugging, which makes it easy to track a program's execution and to verify that the program behaves as intended.

As the basis for my implementation, I selected one of several GitHub repositories implementing Gated Graph Neural Networks' functionality.[2]. Modifications I made to the network architecture pertain to the step where node embeddings are converted to the output; the code computing node embeddings themselves is left unchanged. I have, however, verified that the implementation is correct by comparing the code to the equations I refer to in the next section and making sure that the network can learn to predict simple things such as node degree or distance between nodes.

2.4 Computing Node Representations

This section outlines the algorithm by which Gated Graph Neural Networks compute node embeddings. I begin with a high-level overview of the process and turn to specific equations at the end.

Before a graph can be passed to a Gated Graph Neural Network, each node in it must be annotated with a feature vector.[3] The network updates each vector a fixed number of times

[2] github.com/chingyaoc/ggnn.pytorch

[3] I describe the node annotation process in Sections 4.4, 4.5, and 4.6.

with a Gated Recurrent Unit (GRU) and the resulting embeddings are then used to compute the output as described in Sections 2.1 and 2.2.

A Gated Recurrent Unit is a type of Recurrent Neural Network with two kinds of input. The output produced by a GRU at timestep i is referred to as the "hidden state" and is passed to the GRU at timestep $i + 1$. Along with the hidden state from the previous iteration, GRU also takes as input a second vector obtained from an external source. In the context of the import prediction task, the hidden state is the node embedding computed at a previous iteration and the second input vector contains the messages passed along the edges from the neighbouring nodes. These messages are collected by passing representations of the neighbouring nodes through a linear layer and then summing the results.

Edges in the graph can represent different kinds of relationships between nodes, and separate linear layers are learned for each type of edges. For each type, a GGNN automatically constructs a reverse edge type such that if there is an edge from v to u of type e, there will always be an edge from u to v of the "reverse e" type. For the purposes of the import prediction task, there are three non-reverse edge types. First, there are edges that represent import statements. Edges of the second type link together files defined within the same package. Finally, there are edges that mark extension or implementation of one public class or interface by another.

Table 2.4.1 contains the equations which summarize the process of updating an embedding of a single node in the graph. The equations are reproduced from the original paper about GGNNs (Li et al., 2017). D stands for the dimensionality of a node embedding and $\mathcal{V}$ is the set of nodes in the graph. Vector $\mathbf{h}_v^i \in \mathbb{R}^D$ is the embedding of node $v \in \mathcal{V}$ at timestep i. The original node annotation constructed prior to running the neural network is denoted as $\mathbf{x}_v$. Li et al. allow the original node annotation to be of smaller size than the resulting word embedding (Equation 1), but for the purposes of this project, $\mathbf{h}_v^{(1)} = \mathbf{x}_v$.

Equations 3 to 6 form the canonical definition of a GRU. $\mathbf{W}$, $\mathbf{U}$, $\mathbf{W}^z$, $\mathbf{U}^z$, $\mathbf{W}^r$, and $\mathbf{U}^r$ are learnable parameters, $\mathbf{z}_v^t$ is the update gate vector, and $\mathbf{r}_v^t$ is the reset gate vector. The update vector regulates how much the embedding is changed at each timestep: if $\mathbf{z}_v^t = \mathbf{0}$, then

$$\mathbf{h}_v^{(1)} = \left[\mathbf{x}_v^T, \mathbf{0}\right]^T \quad (1)$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^T \left[\mathbf{h}_1^{(t-1)^T} ... \mathbf{h}_{|\mathcal{V}|}^{(t-1)^T}\right]^T + b \quad (2)$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \quad (3)$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \quad (4)$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \quad (5)$$

$$\mathbf{h}_v^{(t)} = \left(1 - \mathbf{z}_v^t\right) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}} \quad (6)$$

Table 2.4.1. Equations describing an update of the embedding of a single node in the graph. These equations can be found in the original paper on Gated Graph Neural Networks (Li et al., 2017).

$\mathbf{h}_v^{(t)} = \mathbf{h}_v^{(t-1)}$ and the embedding stays the same. The reset gate vector determines whether the changes made to the embedding should depend on the embedding itself: if $\mathbf{r}_v^t = \mathbf{0}$, the changes only depend on the new information. Figure 2.4.1, which can be found on Wikipedia (Gated recurrent unit, 2020), visualizes the process described in Equations 3 to 6.

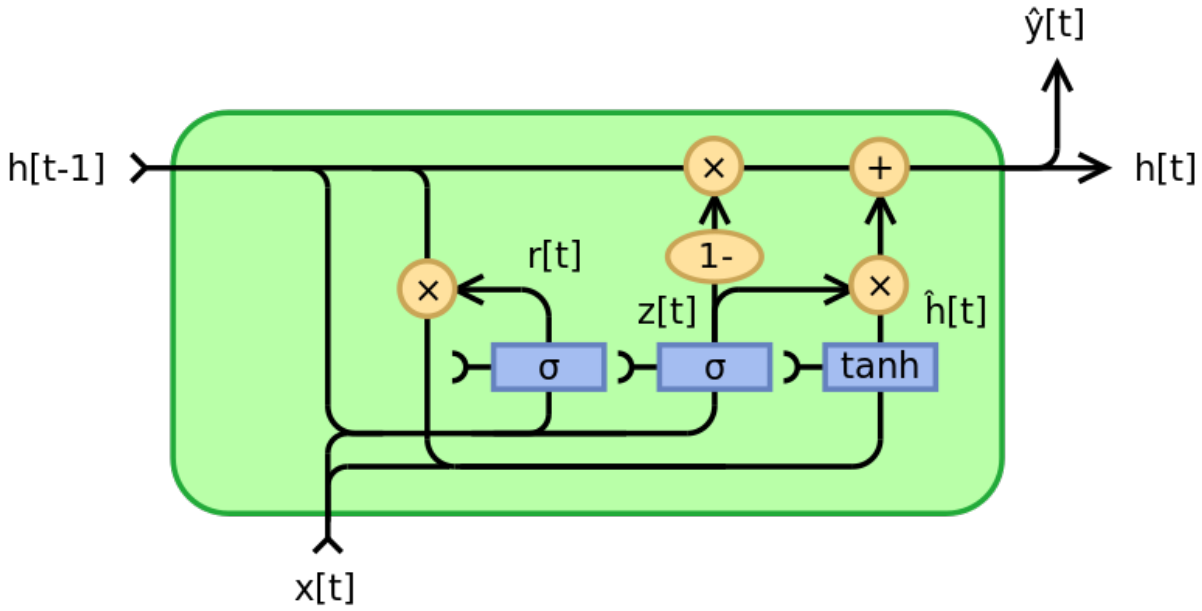


Figure 2.4.1. Visual representation of a GRU, which is formally defined by Equations 3 to 6. Note that the output $\widetilde{y^t}$ is irrelevant in the Graph Neural Network context. The figure can be found on Wikipedia (Gated recurrent unit, 2020).

The matrix $\mathbf{A} \in \mathbb{R}^{D|\mathcal{V}| \times 2D|\mathcal{V}|}$ in Equation 2 is two-dimensional only because this makes the equation shorter and easier to read. In practice, implementations of GGNNs, including the one used in this project, split the data stored in $\mathbf{A}$ between two matrices: an adjacency matrix and an edge type matrix. For each edge type and a pair of nodes, the adjacency matrix encodes

whether there is an edge of the given type between the given nodes. The edge type matrix contains parameters for linear layers corresponding to different edge types. It might be easier to think of $\mathbf{A}$, which combines all of this information together, as a resized version of the five-dimensional matrix $\widetilde{\mathbf{A}} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|\times 2\times D\times D}$. The entry $\widetilde{\mathbf{A}}_{v,u,0}$ is a zero matrix if and only if there are no edges from v to u in the graph. Similarly, the entry $\widetilde{\mathbf{A}}_{v,u,1}$ is a zero matrix if and only if there are no edges from u to v in the graph. If there is an edge of type e from v to u, then the entry $\widetilde{\mathbf{A}}_{v,u,0}$ will contain the parameters of the linear layer corresponding to type e, and the entry $\widetilde{\mathbf{A}}_{u,v,1}$ will contain the parameters of the linear layer corresponding to the reverse e type. Note that this formulation of graph neural networks allows there to be only one pair of edges between any two nodes. Figure 2.4.2 reproduced from the original GGNN paper along with its caption might help to visualize the structure of matrix $\mathbf{A}$.

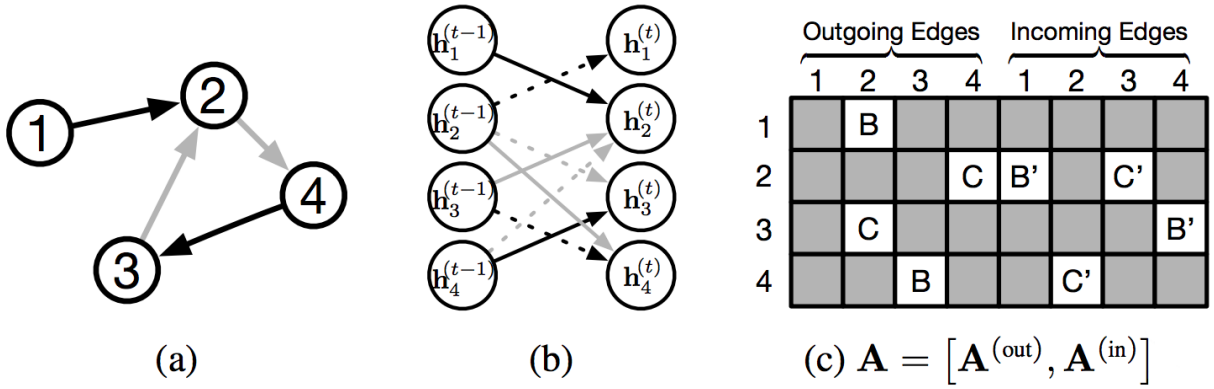


Figure 2.4.2. (a) Example graph. Color denotes edge types. (b) Unrolled one timestep. (c) Parameter tying and sparsity in recurrent matrix. Letters denote edge types with B' corresponding to the reverse edge of type B. B and B' denote distinct parameters.

In Equation 2, $\mathbf{A}_{v:} \in \mathbb{R}^{|\mathcal{V}|D\times 2D}$ stands for what could also be denoted as $\widetilde{\mathbf{A}}_v$. Consequently, $\mathbf{a}_v^{(t)} \in \mathbb{R}^{2D}$ contains the output of the linear layers from edges in both directions, which is the information based on which the GRU updates the node embeddings.

As I hope it is clear from this chapter, data preparation becomes a complicated process when GGNNs are involved. One has to define different types of edges, convert the data to a graph format, annotate every node in every graph with a feature vector and perform many other pre-processing steps before training and evaluating the networks themselves. The next chapter is

devoted to the way the data for this project was collected, and the following chapter deals, among other things, with the node annotation process.

3
Data Collection and Preliminary Analysis

In order to successfully train a neural network to predict imports, one has to compile a large enough dataset of Java projects. In recent years, mining GitHub has become the go-to method for collecting code-related data (Lacomis et al., 2019; Vasilescu et al., 2017; Allamanis and Sutton, 2013; Allamanis, Brockschmidt, et al., 2017). As of August 2019, GitHub houses an impressive 100 million repositories, many of which are publicly available, making the website an invaluable source of information. Google's BigQuery provides an efficient way to examine GitHub in its current state, and GH Archive records all the user activity since 2011. Even so, it can be difficult to select the repositories relevant to a particular research task. Moreover, much of the GitHub code is obsolete, cannot be compiled or is otherwise deficient, and there is little agreement over how to filter out the repositories that are likely to contain faulty code.

3.1 Selecting the Repositories

I examine the information available through BigQuery and GH Archive and use the following criteria to select the repositories for my dataset:

1. **If a repository is a fork, it is not included it the dataset.** Forks are a major source of duplicated code on GitHub, which is why it is best to avoid them. I identify forks by

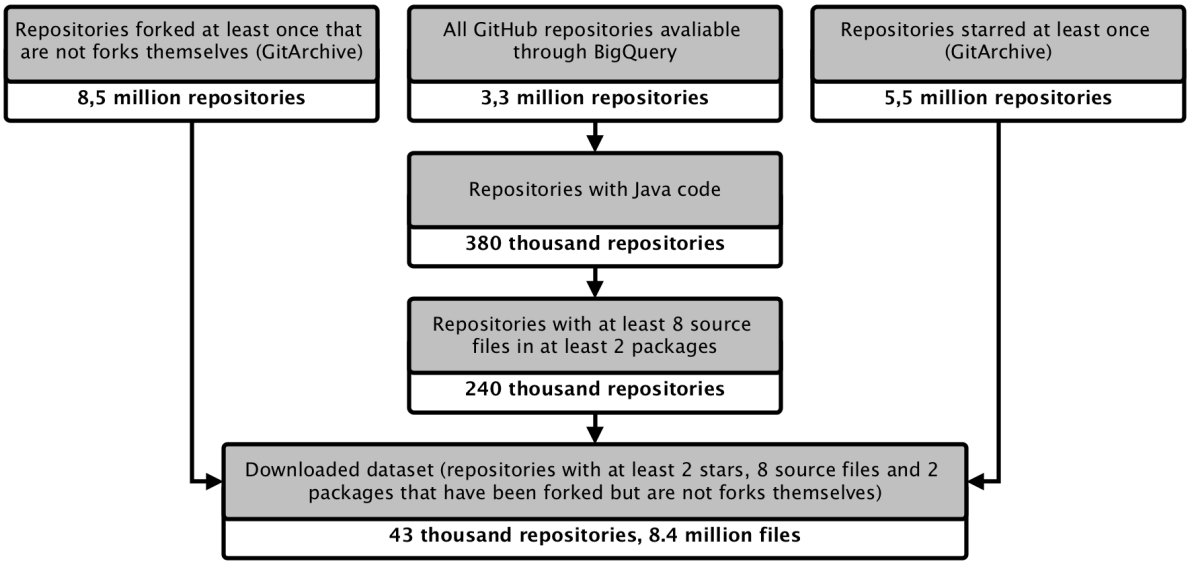


Figure 3.1.1. The pipeline for selecting repositories for the dataset. Each arrow represents a process of filtering out repositories that for one reason or another should not be included. The final 43 repositories are the intersection of the 8,5 million forked repositories, 5,5 million starred repositories, and 240 thousand repositories relevant to the import prediction task.

scanning GH Archive, which records a ForkEvent, whenever someone forks a repository. It is worth mentioning that GH Archive does not record events that occurred prior to 2011. This, coupled with the fact that the syntax of ForkEvents has changed multiple times throughout GitHub's history, means that some forks might be inadvertently included. Moreover, GitHub users often publish modified versions of cloned repositories without explicitly marking them as forks. For these reasons, I return to the problem of filtering out duplicates at a later stage (Section 3.3).

2. **For a repository to be included in the dataset it has to be starred at least twice and forked at least once.** These requirements are aimed to ensure that selected repositories are of sufficient quality. It is, of course, impossible to design a perfect criterion for filtering out faulty code, just as it is impossible to define what faulty code is; however, the two criteria I use have been previously employed by Theeten et al. (2019) and Allamanis and Sutton (2013), respectively.

3. **Each repository in the dataset must contain at least 8 Java source files in at least 2 packages.** Since there is no need to explicitly import classes from within the same package in Java, each repository in the dataset must have at least two packages. I also require that each repository has at least 8 source files to filter out potentially incomplete projects. Establishing such a threshold makes sense for another reason as well: for projects with 8 files or less, import prediction could potentially be guided by simple heuristics. I chose the value of 8 because the number of possible different undirected graphs of n nodes (OEIS, n.d.) becomes larger than the number of GitHub repositories with n Java source files at n=8 (12346 vs. 7995).

The resulting data amounts to 43 thousand repositories, 8.4 million files and 52 GB of code. Figure 3.1.1 illustrates the process of selecting this set of repositories relevant to the import prediction task.

3.2 Parsing

After downloading these 8.4 million files, I parse them with JavaParser. For each file, the parser extracts the name of the corresponding package and all of the import statements. Additionally, for each public non-static class or interface (which can only be one per file or compilation unit), the parser extracts the names of all extended or implemented classes or interfaces. This is the information later used to construct repository graphs.

JavaParser fails to parse a small portion of these files. Manual inspection of several dozens of such files revealed syntactical errors in every one of them. For this reason, I consider repositories containing at least one non-parsable file to be faulty and remove them from the dataset. There are also some files whose names do not match the names of the public non-static classes or interfaces defined in them. Such files would never compile, which is why any repository containing such files is removed from the dataset.

3.3 Filtering

After parsing the code, I proceed to remove duplicate repositories. Duplicates can be identified by packages: packages are supposed to be unique, so if two repositories share a package, one of them must be a duplicate. Given a pair of such repositories, one must decide which one to keep and which one to remove. Lacking a better criterion, I keep the repository that has been forked most and remove the one that has been forked least. If both repositories have been forked the same number of times, one repository is removed at random. I also remove repositories that have classes in unnamed packages or have several identically-named classes located in the same package.

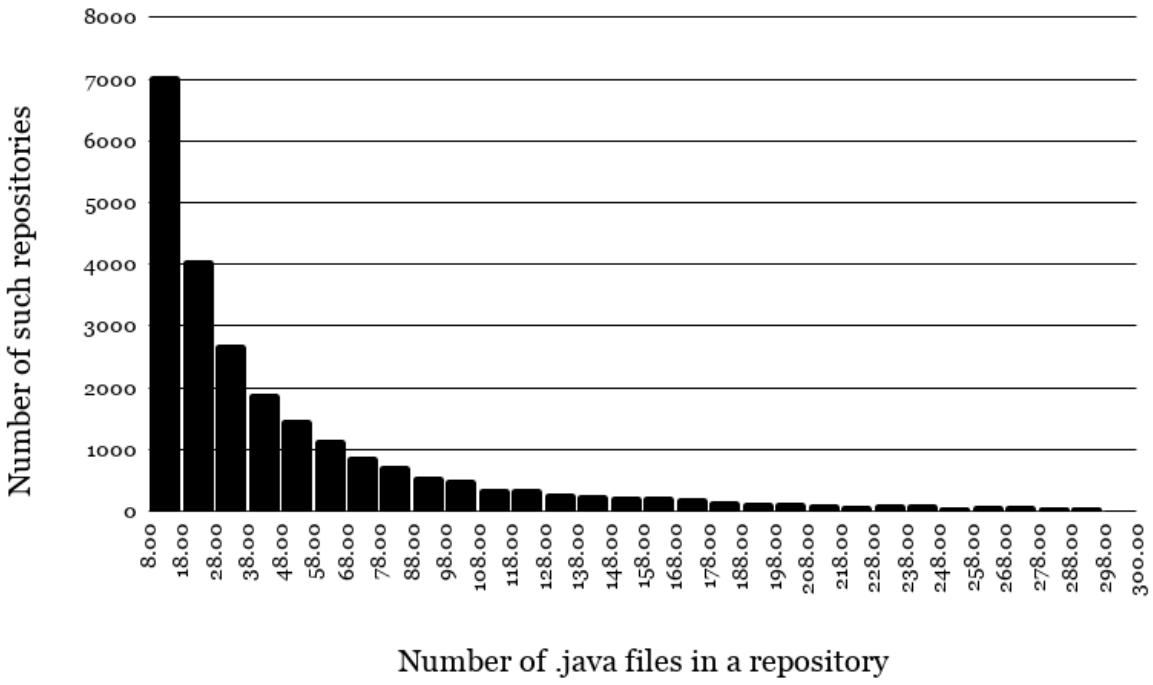


Figure 3.3.1. Distribution of repository sizes in the final dataset. The bucket size is 10.

This leaves a dataset of approximately 25 thousand repositories. Of these, 95 percent have 300 or fewer source files. Several repositories in the remaining five percent are very large. In particular, 3 repositories have over ten thousand files each. Running graph neural networks on such repositories is problematic because standard GNN implementations represent graphs with

adjacency matrices. Hence, to run a GNN on a graph with thousands of nodes one would have to support sparse matrix computation. There exists a TensorFlow implementation of GGNNs that supports sparse matrices, but for several reasons - one being that this implementation is inherently slower than others - I do not employ it. Instead, I remove from the dataset any repository that has more than 300 files. This results in the final dataset of 24,269 repositories.

Figure 3.3.1 shows the distribution of repositories in the dataset by the number of source files in them. One can immediately see that most repositories are small: the median is 33 files per repository. This raises the question of whether a network trained on such a dataset would perform well on large repositories, when there are many of different import candidates to choose from. Perhaps surprisingly, both the network and the baselines performs better in such scenarios, as is discussed in Section 5.2.

4
Features and Baselines

To prove that Gated Graph Neural Networks are efficient at predicting future imports I compare their performance to that of several widely-used classifiers. Conceptually, all these classifiers make predictions in the same way that GNNs do. Given a repository R, a file f to which a new import statement must be added and a set C of classes that are candidates for import, the classifier evaluates each possible import on its own with a classifier-specific function g. The correct candidate is then computed as $argmax_{c \in C} g(R, f, c)$. The performance of a classifier, therefore, depends entirely on the way the classifier-specific function g is defined.

The two types of machine learning methods employed as baselines are Random Forests (RFs) and simple Feed-Forward Neural Networks (FFNNs). Feed Forward Neural Networks are a good baseline because they are what Gated Graph Neural Networks default to when there are no edges in the graph.[1] Were GGNNs to perform no better than FFNNs, it would mean that representing repositories with graphs is a useless enterprise. Random Forests are good at making predictions based on sets of unrelated features, such as some of the features described below, which is why I pick them as a second baseline. Both methods take a single feature vector normalized as a z-score as input. Below I described the features of which this vector is composed.

[1] More precisely, in the absence of other edges in the graph, GGNNs make predictions based on two embeddings: that of the import candidate and that of the file for which the new import is to be predicted. A GGNN would make such a prediction with a GRU, but whatever function a GRU can express, an FFNN should be able to express as well. Given that

4.1 Import frequency

In the absence of other information future import prediction can be based on prior likelihood: one selects a class that has been imported most frequently in the past. On the repository graph, such class would correspond to the node with the highest indegree.

4.2 Shortest Path

Another feature based on the repository graph is the minimum distance between the file f and a given candidate c. The assumption is that nodes that are close to each other on the graph correspond to compilation units that share some functionality or are created to solve some common problem. The closer two classes are on the graph, therefore, the more likely it is that there is an import statement linking the two together.

4.3 Edit Distance

A programmer will often use similar names for classes with similar functionality. Sometimes, similarity between two class-names betrays their import relationship: it is not at all surprising that the `AudioSystem` class in `javax.sound.sampled` package imports the `AudioFileReader` class from a different package. One of the most popular measures of string similarity is edit distance, and I include the normalized edit distance in the list of features based on which the classifiers make their predictions.

4.4 Filename Embeddings

While edit distance can be useful for detecting pairs of filenames that share words or prefixes, it does not reflect the semantic similarity between filenames. A way to capture the semantic similarity between a pair of filenames is to construct embeddings for both strings. The similarity

an FFNN would have fewer parameters than a GRU, it should perform better in such a scenario, which makes it a good baseline classifier.

can then be defined as Euclidean distance between the embeddings: the smaller the distance, the more semantically similar the strings are.

To construct a filename embedding, I first split a filename into individual words based on camelCase (e.g. `HashMap` is split into `Hash` and `Map`). I then look up an embedding for each of these words in the GloVe dataset (Pennington et al., 2014)[2] and take the unweighted average to be the resulting filename embedding. This is similar to how Koc et al. (2019) construct embeddings for program dependency graph nodes and how Allamanis et al. (2017) construct embeddings for variable identifiers, except that the latter use a linear layer instead of the unweighted average to compute the final embedding. Whether using a linear layer is a more adequate approach can only be determined experimentally.

Their usefulness in obtaining a semantic similarity feature is only one reason why filename embeddings are relevant to this project. The other reason is that each node passed to a GGNN must be annotated with a vector, as explained in Chapter 2. It is quite common to annotate nodes with vectors ultimately derived from pre-trained word embeddings. Among others, Allamanis et al. (2017) and Koc et al. (2019) take this approach. I follow them and annotate nodes in my network with filename embeddings, albeit with one difference which I discuss in the next section.

4.5 External Import Embeddings

The goal of this project is to predict local imports, that is imports of files defined in a repository to other files within that same repository. For each class in the repository, there is a corresponding node in the repository graph that serves as input to graph neural networks. However, classes defined in external libraries and edges representing external imports do not appear in this graph: putting them there would dramatically increase the number of nodes and slow down the network's performance. Moreover, it is not always possible to obtain the source code for an arbitrary external library and one cannot always determine the relationship between files within

[2]While there are many similar datasets available on the web, GloVe is easy to download and is maintained by a group of well-known NLP researchers at Stanford.

it. In other words, the geometry of repository graphs as I define them does not encode external imports.

Intuitively, however, external imports must facilitate future import prediction and it is important that the classifiers have access to information about external imports in one form or another. For this reason, I compute what I call external import embeddings, which one can compare to learn whether two files make similar external imports. There are several ways such embeddings could be constructed, as discussed in Chapter 6. At present. however, I use the simplest method possible. For each external import in a file, a class-name embedding is computed using the approach used for filenames (Section 4.4). These class-name embeddings are then averaged to get a single external import embedding for the file in question.

For each file in a repository, therefore, there are two embeddings: a filename embedding and an external import embedding. The node annotations passed to GGNNs are concatenations of these two embeddings. If embeddings have the dimensionality of 10, for instance, the node annotations have dimensionality of 20. Similarly, these two embeddings are concatenated into a single vector and passed to Feed-Forward Neural Networks. Unlike FFNNs and GGNNs, Random Forests cannot compare embeddings, which is why I compute the Euclidean distance between embeddings and pass that distance as a feature to the Random Forest baseline.

4.6 Reducing Embedding Dimensionality

This gives rise to the final question addressed in this chapter: what dimensionality is optimal for node annotations? Previous studies show that the dimensionality of word embeddings used in NLP is usually much higher than that needed for node annotations in a GGNN. All publicly distributed GloVe embeddings, for example, have at least 25 dimensions. By contrast, Koc et al. (2019) report that they found embedding dimensionality of 8 to be optimal for their problem.

Unlike Koc et al. I do not construct word embeddings myself, because using a well-known dataset trained on natural text seems preferable to training an entirely new set of embeddings of unknown quality. However, I do conduct a series of experiments to determine the optimal

embedding size for the import prediction task. To reduce the dimensionality of filename- and external import embeddings I use simple autoencoders, Feed-Forward Neural Networks with a single hidden layer trained to predict their own input. The loss function typically used for this task is Mean Squared Error Loss. The size of the hidden layer is smaller than the size of the input layer and so the output of the former can be used as an embedding of its own.

By reducing embedding dimensionality one unavoidably loses some information preserved in the original vector: Figure 4.6.1 shows how MSE Loss increases with decreasing hidden layer size. Nonetheless, large embeddings are not necessarily better: the larger the embeddings are, the more free parameters a graph neural network has. An excessive number of free parameters can prevent the network from converging and decreases the training speed.

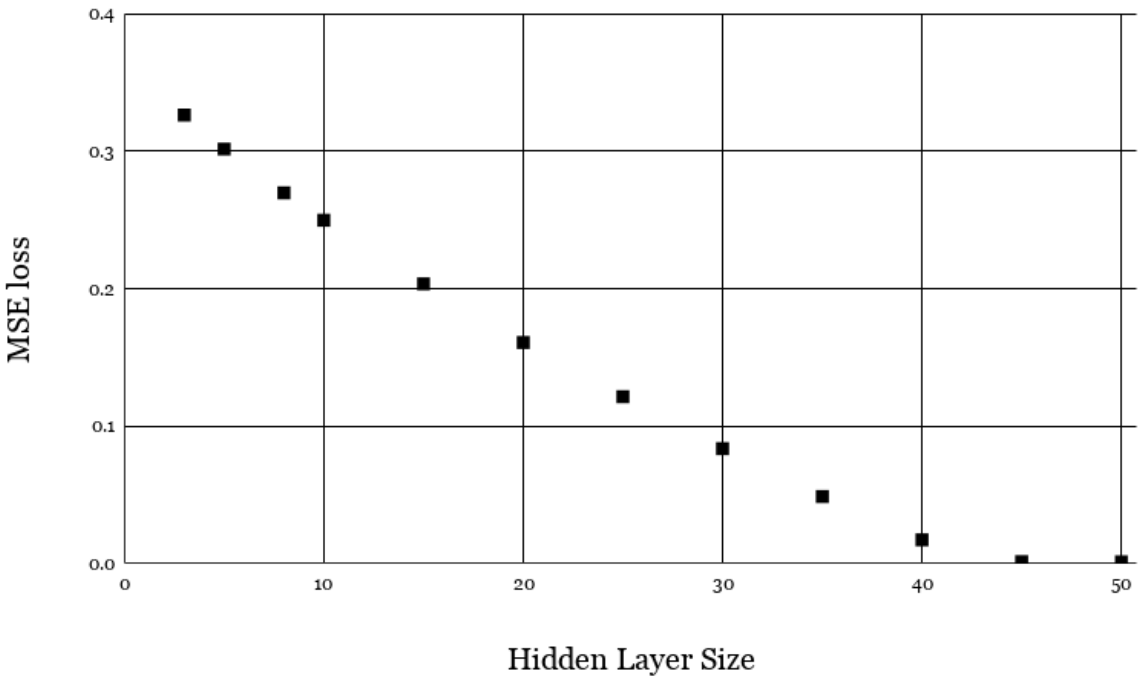


Figure 4.6.1. The influence of the hidden layer size (horizontal axis) on the MSE Loss of embedding autoencoder (vertical axis) trained on the Glove dataset (400K words, each encoded with an embedding of size 50). Because the mean square error appears to be increasing linearly with decreasing hidden layer size, the mean absolute error must increase ever more slowly as hidden layer size is reduced.

I experimented with different embedding sizes and found embedding dimensionality of 8 to be optimal for the import prediction task.[3] Figure 4.6.2 shows how network accuracy on the binary import prediction task increases as embedding size decreases until the accuracy reaches the peak at an embedding size of 8.

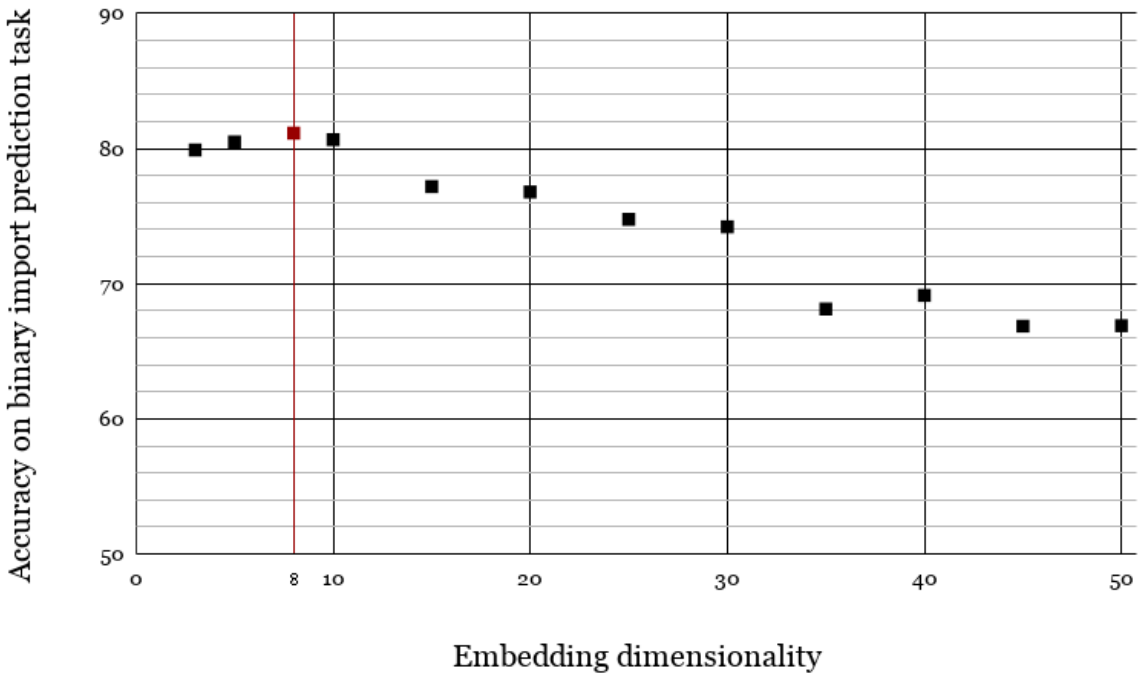


Figure 4.6.2. Correlation between GNN accuracy on the binary prediction task (in percentages, vertical axis) and embedding size in use (horizontal axis). Embedding size is the only hyper-parameter changed; all other hyper-parameters are fixed. Note that the hyper-parameters used to create this graph were slightly sub-optimal and so it is possible to achieve a slightly better performance in all cases.

[3]Note that embedding dimensionality of 8 translates to node annotation dimensionality of 16, as described in Section 4.5.

5
Results

In this chapter, I compare the results that the baselines and the GGNNs achieve on the import prediction task. I also discuss the effect that the values of several hyper-parameters have on GGNNs' performance and analyze the usefulness of different features defined in Chapter 4. The results presented here shed light on some interesting aspects of the import prediction problem.

5.1 Best Hyper-Parameters

To ensure optimal performance from the GGNNs, I perform greedy hyper-parameter search to determine the learning rate, embedding size, and the number of node embedding updates that lead to the best performance. I start by determining the optimal base learning rate because this is the parameter to which neural networks are most sensitive. Note that the networks are trained with Adam optimizer, which adjusts the learning rate for each individual parameter. These individual learning rates are, however, expressed in terms of the base learning rate which must be specified manually. As can be seen from Figure 5.1.1, the accuracy of the network on the binary import prediction task increases with a decreasing base learning rate until it reaches a plateau at a learning rate of approximately 10^{-3}.

Next, I determine the optimal embedding size. As discussed in Chapter 4 and illustrated by Figure 4.6.2, the network performs best with embeddings of size 8.

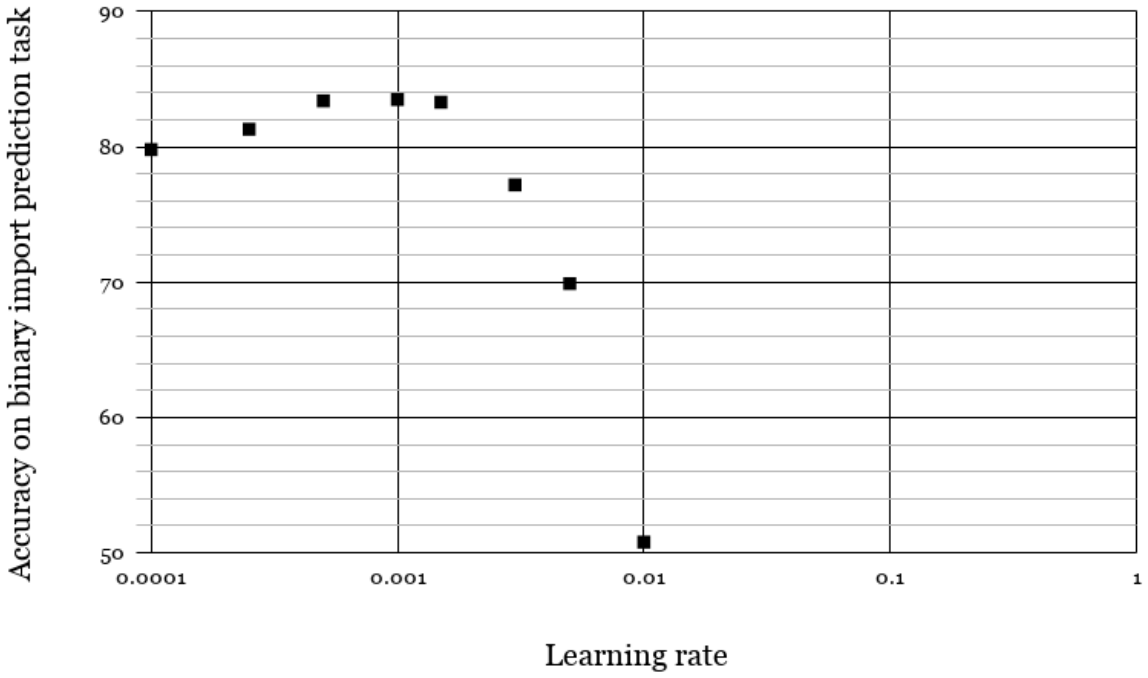


Figure 5.1.1. Correlation between GGNN accuracy on the binary prediction task (in percentages, vertical axis) and the base learning rate used (horizontal axis, log scale). All other hyper-parameters are fixed.

Finally, I identify the number of node embedding updates (see Section 2.4) that correlates with the highest GGNN accuracy. Allamanis et al. (2017) note that the performance of their networks increases with the number of updates but that after a certain point, the increase becomes negligible. In my case, as is evident from Figure 5.1.2, the performance stops improving after just three updates. One update is associated with passing messages from the neighbouring nodes to the target node as described in Section 2.4. Therefore, when the number of updates is equal to three, the prediction that a GGNN makes for a certain node on the graph depends on embeddings of all the nodes that are within the distance of at most three edges from the target node. It is often the case that any node on a repository graph is within the distance of three edges from any other node, in which case an additional update would not introduce new information to the system.

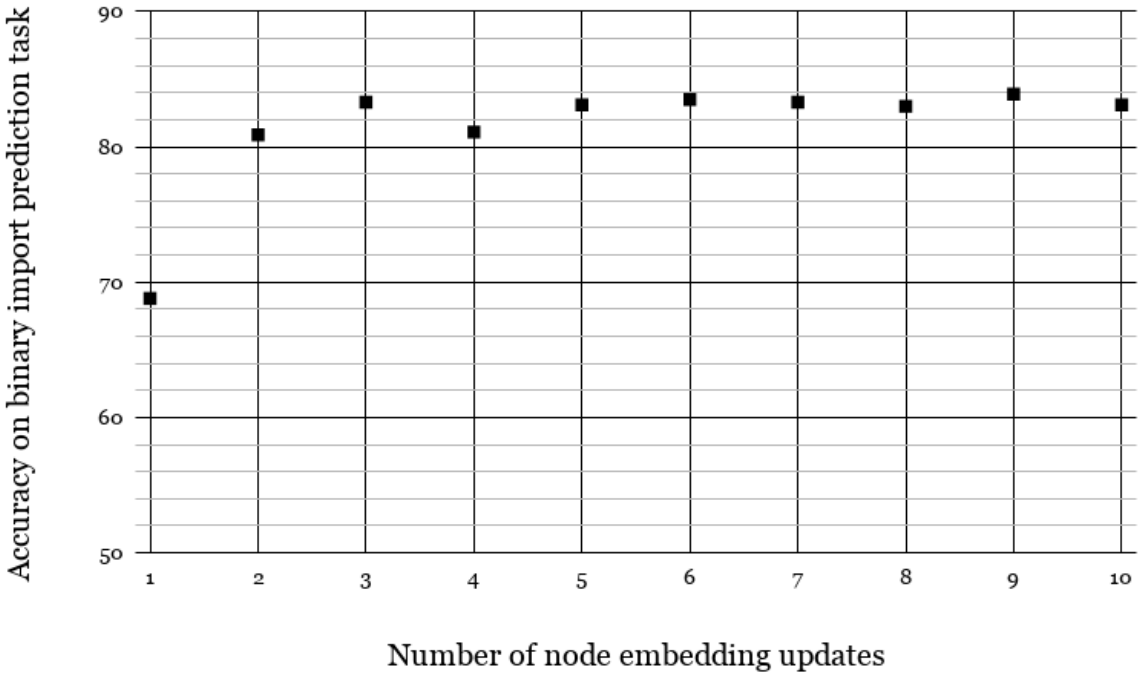


Figure 5.1.2. Correlation between GGNN accuracy on the binary prediction task (in percentages, vertical axis) and the number of times node embeddings are updates (horizontal axis). All other hyper-parameters are fixed.

5.2 Predictive Power of Individual Features

Before comparing the results that the baselines and the GGNNs achieve on the import prediction task, I discuss the predictive power of individual features defined in Chapter 4. Table 5.2.1 summarizes the accuracy that one can achieve on the import prediction task by selecting an import candidate that maximizes or minimizes some particular feature. The accuracy is given for four scenarios which differ in the number of import candidates (classes) the classifier is presented with.

While the accuracy decreases with the number of classes - as it should - it decreases more slowly than one might expect it to. The classifier based on the edit distance feature, for instance, achieves the accuracy of .619 on the binary classification task. One would, therefore, expect it to have the accuracy of $.619^4 \approx .147$ when presented with five candidates to choose from. Nonetheless, the figure is significantly higher (.314), which means that relationships between

N of classes	Rand.	Edit dist.	Import freq.	Shortest path	Filename embeds. distance	Import embeds. distance	FFNN (embeds. only)
2	.5	.619	.742	.623	.508	.502	.605
5	.2	.314	.518	.312	.251	.244	.283
25	.04	.129	.253	.091	.093	.067	.063
125	.008	.065	.098	.025	.041	.017	.017

Table 5.2.1. Comparison of the predictive power of individual features on the import prediction task. The value recorded in each cell is the accuracy that a particular baseline (column) achieves given a certain number of import candidates to choose from (row). The "Rand." column records the performance of a hypothetical random classifier. The following five columns correspond to baselines that select an import candidate by minimizing or maximizing a particular feature. The final column records the accuracy that Feed-Forward Neural Networks achieve when given concatenated import- and filename embeddings of the target file and hypothetical import candidate as input.

import statements are not independent. Perhaps the difficulty of predicting imports has more to do with the particular import one is learning to predict than with the fake candidate imports one must learn to ignore.

Consider the example drawn in Section 4.3 to illustrate the importance of the edit distance feature. Suppose that a classifier is given a set of import statements only one of which really appears in the `AudioSystem` class and that the classifier has to identify the correct import. If that correct import happens to be the `AudioFileReader` class, the classifier would have no problem identifying it based on the edit distance between the two class-names. In this case, the deciding factor would be that "`AudioSystem`" is similar to "`AudioFileReader`"; the number of fake import statements the classifier is presented with would not play much of a role.

While edit distance is useful, it is not the most reliable of features. For any number of classes, the highest accuracy is achieved by selecting the import candidate that has been imported most frequently in the past (see Section 4.1) . This reflects the fact that certain classes and interfaces are designed to be imported more frequently than others.

The features that seem to be least important are those based on the Euclidean distance between pairs of embeddings (see Sections 4.4 and 4.5). This underscores the need for having a separate linear layer to measure embedding similarity, a layer trained jointly with the GNN

N of classes	Import freq.	RF	FFNN	GNN
2	.742	.775	.778	.835
5	.518	.553	.54	.614
25	.253	.281	.279	.368
125	.098	.108	.083	.133

Table 5.3.1. Summary of the accuracy achieved by different classifiers on the import prediction task. The value recorded in each cell is the accuracy that a particular classifier (column) achieves given a certain number of import candidates to choose from (row). The "Import freq." column records the performance of a classifier that selects a candidate that has been imported most in the part. The next three columns summarize the performance of baselines that take all features into account. The last column records the accuracy achieved by the GNNs. "RF" stands for "Random Forest", "FFNN" - for "Feed-Forward Neural Network".

as described in Section 2.2. The failure of Euclidean distance as a feature does not mean that embeddings themselves do not encode relevant information: a Feed-Forward Neural Network trained on pairs of embeddings achieves results that are noticeably better than random (see Table 5.2.1). Rather, the relationship between embeddings relevant to import prediction and learned by Feed-Forward Neural Networks must have very little to do with Euclidean distance.

5.3 Baselines vs. GNNs

Table 5.3.1 shows the accuracy that a Random Forest Classifier, a Feed-Forward Neural Network, and a Graph Neural Network achieve on the import prediction task. The GNN outperforms both baselines in all scenarios. Because it takes a relatively long time to train a GNN, I was not able to run the network with optimal parameters enough times to prove that the difference is statistically significant. Nevertheless, one might expect statistical significance because GNNs outperform the baselines on the binary prediction task even when hyper-parameters are sub-optimal: see, for instance, the results recorded in Figure 5.1.2 for different number of embedding updates.

The difference between the baselines and the GNN is most pronounced on 25- and 125-class classification tasks. So, for instance, the accuracy achieved by the GNN on the 25- classification task is $\frac{.368}{.281} \approx 1.3$ times the accuracy of the best baseline (RF), whereas for the binary classification task, the GNN is just 1.07 times better than FFNN. This is probably because 25- and

125-class classification is done on relatively large repository graphs which allows the GNN to take the most out of the graph geometry. Large repositories can also have more local imports per file: a follow-up experiment could be conducted to determine whether higher mean node indegree is associated with better GNN performance.

6
Future Work and Conclusion

This project consists of three major components: graph neural network implementation, data pre-processing, and comparison between GNNs and various baselines. Numerous changes could have been made to the way each of these sub-problems is addressed in the code and this text but for each sub-problem, there is one particular avenue that would be most important to explore going forward. In this chapter, I discuss what the three potential problems with the import prediction project are and offer tentative solutions to each of them.

6.1 GNN Deployment: the Issues of Speed and Memory

Speed is paramount to any code completion system because code completion has to be done in real-time as the user types new code. For this reason, even though Graph Neural Networks outperform other classifiers by a large margin, they are unlikely to be deployed by IDEs any time soon, unless the computation process can be accelerated.

The bottleneck of the current implementation both in terms of speed and in terms of memory is associated with the need to store information about edges in a fixed-size adjacency matrix. In the current implementation, all the matrices are padded to be of the same size allowing for up to 300 nodes per graph, which was originally done to facilitate batch computation. However, given that the mean repository size in the dataset is 33 source files (3.3.1), this means that half

of the time, $1 - \frac{33^2}{300^2} \approx 99\%$ of the entries in a matrix are zeros, and most of the computations are done on zero vectors. This is not to mention that even without padding, adjacency matrices tend to be quite sparse: a single file would rarely import all the other files from all the other packages in a repository. Finally, because the matrix size grows quadratically with the number of files in a repository, the computation process becomes intractable for repositories that have more than 300 files.

For these reasons, as mentioned in Section 3.1, supporting sparse matrix computation would significantly reduce the amount of processing time and memory needed to make a prediction.

6.2 Node Annotations: Additional Features would Help

When a Graph Neural Network computes an embedding for a particular node, it takes into account the embeddings of all the neighbouring nodes. Sometimes, however, there might only be a few neighbouring nodes or even none at all, in which case the prediction will mostly be based on the node annotation constructed before the network was run. This means that node annotations must be designed to encapsulate as much information as possible.

Currently, node annotations are constructed by concatenating filename- and import embeddings into one vector. Import embeddings are in turn constructed by taking the weighted average of embeddings of all the words found in external import statements (Section 4.5). This approach is easy to understand and replicate, but it has one potential drawback: when there are many external imports in a file, computing import embeddings involves averaging over a large set of words. This is problematic because by averaging over a large set of vectors, one might lose information. In fact, the average of all known word embeddings is commonly used as an embedding for out-of-vocabulary words because, being like any other embedding in the dataset, such a vector has little meaning.

An alternative way of constructing import embeddings would be to first annotate each file with a vector that for each potential external import would indicate whether this import is present in the file or not. One could then use an autoencoder to reduce the dimensionality of

these new vectors to the desired size. Files with common import statements would then have similar embeddings.

One might also try to encode information about identifier names or commonly used syntactic structures - it is best to include as much of the relevant information about a file as possible when constructing node annotations.

6.3 New Baseline: Item to Item Collaborative Filtering

While Graph Neural Networks do perform significantly better than the baseline classifiers described in Chapter 4, there is one prediction method to which the GNNs have not been compared yet. This method is referred to as item-to-item collaborative filtering and has been invented by Amazon.com to make personalized recommendations by showing a customer the products similar to those that the customer had bought in the past. In the context of the import prediction task, products correspond to import statements and customers - to Java source files. The idea is to compute similarities between different import statements and to then rank potential imports based on their similarity to the imports already present in the file. Without delving into too much detail, item-to-item collaboration filtering should work significantly faster than a Graph Neural Network. However, this method would arguably under-perform whenever there are not enough import statements already present in the file because existing import statements are the sole source of information on which the predictions can be based. The same is only partially true for GNNs, which can make predictions based on filename embedding similarity or based on the import statements in other files in the same package. In any case, this project would definitely benefit from a direct comparison between the two methods.

6.4 Conclusion

This chapter shows that there is still a long way to go before Graph Neural Networks could be deployed by IDEs to predict future import statements. Nonetheless, it is not inconceivable that this might happen in the future because, as I hope I have demonstrated in this project,

GNNs can successfully reason about Java programs. Their application should not be restricted to Java or the import prediction task only; one can, for example, imagine a GNN being used as an encoder in the pipeline generating natural language descriptions of repositories.

References

Allamanis, M., & Sutton, C. (2013). Mining source code repositories at massive scale using language modeling, In *Proceedings of the 10th working conference on mining software repositories*, San Francisco, CA, USA, IEEE Press.

Mikolov, T., Yih, W.-t., & Zweig, G. (2013). Linguistic regularities in continuous space word representations, In *Proceedings of the 2013 conference of the north American chapter of the association for computational linguistics: Human language technologies*.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation, In *Empirical methods in natural language processing (emnlp)*.

Allamanis, M., Barr, E. T., Devanbu, P. T., & Sutton, C. A. (2017). A survey of machine learning for big code and naturalness. *CoRR*.

Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to represent programs with graphs. *CoRR*.

Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2017). Gated graph sequence neural networks. *ICLR 2016*.

Vasilescu, B., Casalnuovo, C., & Devanbu, P. (2017). Recovering clear, natural identifiers from obfuscated js names, In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, New York, NY, USA, Association for Computing Machinery.

Bavishi, R., Pradel, M., & Sen, K. (2018). Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *CoRR*.

Koc, U., Wei, S., Foster, J. S., Carpuat, M., & Porter, A. A. (2019). An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool, In *2019 12th ieee conference on software testing, validation and verification (icst)*.

Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., & Vasilescu, B. (2019). Dire: A neural approach to decompiled identifier naming. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Theeten, B., Vandeputte, F., & Van Cutsem, T. (2019). Import2vec: Learning embeddings for software libraries. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*.

Gated recurrent unit. (2020). *Gated recurrent unit — Wikipedia, the free encyclopedia*. Retrieved April 20, 2020, from https://en.wikipedia.org/wiki/Gated_recurrent_unit

OEIS. (n.d.). *A000088. number of graphs on n unlabeled nodes*. Retrieved February 10, 2020, from http://oeis.org/A000088