COMPARATIVE CASE STUDY BETWEEN GATED GRAPH NEURAL NETWORKS VERSUS RELATIONAL GRAPH CONVOLUTIONAL NETWORKS FOR THE VARIABLE MISUSE TASK USING PYTHON AND DEEP LEARNING

By

Mukund Raghav Sharma

A Capstone Project Paper Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in

Data Science

University of Wisconsin – La Crosse

La Crosse, Wisconsin

May 2020

DEDICATION

To my grandparents and my favorite planet, Saturn.

ABSTRACT

GATED GRAPH NEURAL NETWORKS VS RELATIONAL GRAPH CONVOLUTIONAL NETWORKS FOR THE VARIABLE MISUSE TASK

MUKUND RAGHAV SHARMA

Identifying bugs in source code has been an extremely important part of software development since the inception of the industry. The majority of static analysis, the analysis of software without actually executing programs, is rule based without much involvement of deep learning until fairly recently. This paper engages in a comparative study of determining the more performant graph neural network model on the basis of test accuracy between Gated Graph Neural Network (GGNN) models and Relational Graph Convolutional Network (RGCN) models on the Variable Misuse Task, a prediction task involving choosing the correct variable based on all the variables of the same type in a particular scope.

The data is of source code from the files of 25 trending C# repositories that are converted into a modified Abstract Syntax Tree to represent a directed graph whose vertices that represent the tokens and relationships between the tokens are represented by edges. Each of these vertices are associated with one of the aforementioned type of networks for the training phase after a particular embedding is computed for each token.

The comparison to decide the more efficient model is based on the test accuracy of all the repositories, an esoteric repository and an extremely popular repository to cover the spectrum of different types of repositories. The results show that the RGCN based models outperformed the GGNN models for all cases, albeit, within < 5% range.

Keywords: Deep Learning, Graph Neural Networks, Tensorflow, Sequence Models, Convolutional Models, Learning from Code, Static Analysis.

TABLE OF CONTENTS

[TODO: FIX]

CHAPTER 1

INTRODUCTION

Project Background

Since the inception of the computing industry more than 70 years ago, the need for correct and efficient verification tools has been extremely desirous in the software engineering world. Static Analysis tools or tools that analyze the code without executing the program boost the productivity of the developer by highlighting some bugs at development time rather than during when the code is running in production.

Deep learning application in the field of learning from source code and static analysis is still in its nascent phase and a large part of the current work doesn't take advantage of the representational power of both the syntactic and semantic nature of the code. For example, shallow representations of source code is prevalent in recent research such a simple sequence of tokens such as by work done by Hindle et al. (2012) [1] or flat dependency networks of variables by Raychev et al. (2015) [2].

Recent work from Microsoft Research involves inculcating both the semantic and syntactic nature of source code by representing programs as directed graphs [3] and applying deep learning via sequential or convolutional neural networks to create graph neural network models. Using graph neural network models has proved to more easily solve state-of-the-art problems in the space of learning from code as the representational capacity of these models reduces the need for a large training set as well as meticulously encapsulates the semantic relationships between variables and types of the source code.

In this paper, I use the work by Microsoft Research in the field of Graph Neural Networks to conduct a comparative study between two different graph neural networks namely, Gated Graph Neural Networks (GGNN) and Relational Graph Convolutional Networks (RGCN), on the Variable Misuse Task to discern the better performing model on the basis of test accuracy. The Variable Misuse Task is a prediction based task on source code involving predicting the variable that most accuracy fits the current context from all the variables of the same type in a particular scope.

The Variable Misuse Task is a simple yet an extremely important task that has its uses in the world of static analysis. Microsoft Research's application of Graph Neural Networks on the Variable Misuse task has caught bugs that had been deployed in production for important repositories such as RavenDB and Rosyln [3]. Further application of Graph

Neural Networks on newer tasks show great promise and have already started changing the way code is tested and validated.

The input data used for this model is that of the top 24 trending C# repositories on Github. There are three main experiments conducted to discern the better performing model on the basis of test accuracy and these are training and testing on the source code of all the repositories, an esoteric repository and an extremely popular repository. The rationale here was to gain insight about how the GGNN and RGCN models would fare in different cases and which one of the two would prove to be the better predictor.

Objectives

The main objective of this project is to compute the test accuracies for the GGNN and RGCN models for three experiments namely, training and testing on data from 25 of the most popular C# repositories on Github, an esoteric repository and a popular repository, in an effort to discern the more performant model.

This process of computing the test accuracies involved enumerating through some minor, more granular, objectives and these are:

1. Obtaining, Cleaning and Understanding the Input Data.
2. Choosing, from the trending C# repositories, which one would characterize as the esoteric and popular one to conduct our experiments on.
3. Establishing a working pipeline through which the experiments will be conducted.
4. Interpreting results i.e. test accuracy of the experiments to land on a conclusion.

Rationale and Inspiration for undertaking the project

The constant strive to build better development tools that will improve the overall user experience is one I am starting to familiarize myself with more since I joined Microsoft's Developer Division as a software engineer where I work on the Performance and Reliability of Visual Studio. My role involves creating processes that improve the prevention of performance regressions by early detection before they reach the end user. My experience with performance engineering and testing on a large code base coupled with my strong inclination towards deep learning got me going down a path to choose a topic for my capstone that would fit the said intersection.

After discovering Microsoft Research's ground breaking work, I was convinced I needed to know the ins and outs of the details of how they improved state-of-the-art research models. And as a result to add more specificity to my topic, I decided to choose between

a sequence based model (GGNN) and one convolutional based model (RGCN) on a the Variable Misuse Task. Also, I was lucky enough to have all my questions answered within a couple of hours by the good people at Microsoft Research and this was another inspiration booster i.e. to work on ground breaking work with individuals who were extremely eager to help out in improving the understanding of their work.

The choice of repositories were amongst the top trending C# repositories that I have used professionally which made pursuing the data exploration all the more exciting as it provided me with a reason to dive deep into the code, as well. Conducting the comparison on repositories like Newtonsoft.Json was enticing as these are libraries I use on a daily basis and have used all my career being a .NET developer.

CHAPTER 2

LITERATURE REVIEW

Previous Work

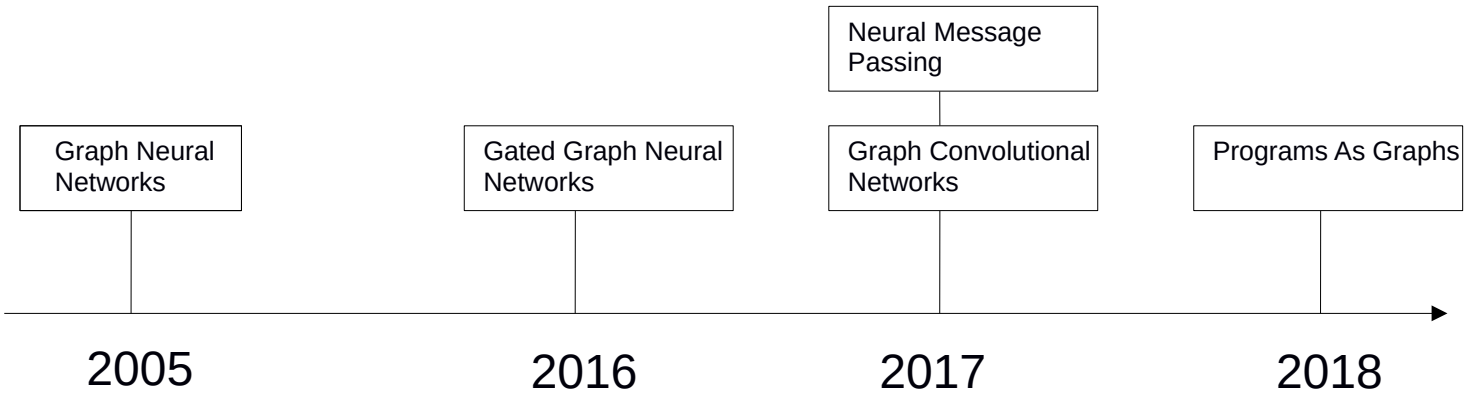


Fig. 1: Timeline of Graph Neural Networks Research

Researching through a lot of the related past work in the field of Graph Neural I was able to narrow down the timeline of key developments in this field details of which are as follows:

2005: Gori et al. proposed the notion of Graph Neural Networks that was the seminal research paper in this field that laid down the foundation of future work. Although, deep learning wasn't at the forefront of statistical research when this paper was written at it is

now, important fundamental concepts such as training of neural networks on graphs as well message passing emerged.

2016: Li et al. took the core concept of Graph based networks and applied sequence based concepts such as gated cells to neural networks to prevent loss of importance of information in large, extremely spread out sequences represented as graphs. The most important aspect of this model was the propagation model that took the messaging passing logic and applied recurrent gated cells at each vertex.

2017: Liao et al proposed idea of Neural Messaging Passing where information between the neighbors of vertices is shared and therefore, not only does each vertex in a graph, through training, have information about its initial state but also other vertices. I will be expounding on this a lot more in a later chapter as this concept has be explained thoroughly for full effect but to summarize, neural message passing allows graph neural network models to make use of their initial state but as well as the state of other vertices in an effort to increase overall representational capacity that requires fewer training data points.

2017: Kipf et al. introduced the concept of Graph Convolutional Networks that took plain vanilla graph neural networks and in state update training step used convolution of the state of the neighbors as the mechanism of message passing. The premise here is that the

Convolution operation that is typically used in the case of computer vision can be applied in the case of graph neural networks in the case of message passing where each vertex can gain further insight about it's position with respect to other vertices.

2018: Allamanis et al. applied concepts from previous research in the context of learning from programs following the simple yet powerful assertion that source code can be represented as graphs where the vertices represent the variables and it's associated information and the edges represent the relationship between the variables. Additionally, taking advantage of the semantic and syntactic nature of source code adds more detail to the input of the graphs that can result in quicker training. The majority of this report is built on this paper and its applications and further details will be elucidated upon in future chapters.

The aforementioned concepts at this point might seem difficult to grasp as they aren't fully elucidated on; I plan to cover the basics of all the concepts in one of the future chapters that'll simplify the subject matter. Needless to say, the extensive research on this topic of graph neural networks is truly inspiring and exciting as well as growing exponentially with developments in multiple dimensions.

CHAPTER 3

PROJECT OVERVIEW

General Approach / Layout

1. The basic steps undertaken to approach this project include:
2. Defining the Experience, Task and Performance Metric in the style of the formal machine learning once the problem statement was identified that we'll base the comparison on.
3. Acquisition and exploration of the input data source.
4. Deep diving into the source code associated with the Learning to Represent Graphs library to gain insight about how the code is executed.
5. Once the insight is gained to run the code, customize the code to accommodate for ease of use for my experiments by constructing a working pipeline that's based on ease of use.
6. Defining the 3 experiments and which data will be used for them.

7. Training models with the training and validation input data on GGNN and RGCN models for the experiments.
8. Training with different hyperparameters that can be used to improve the overall validation accuracy.
9. Using the weights from the training process to compute the test accuracy for RGCN and GGNNs.
10. Aggregate results to discern which model was the more performant one.
11. Conclude with possible next steps.

Now that the goals and outline are set, I want to shed light on what's actually happening under the hood during the training and evaluation of these models and define the task at hand with more detail.

CHAPTER 5

GRAPH NEURAL NETWORKS 101

Goals

In this section I plan to achieve the following goals:

1. Define and Describe the Variable Misuse Task in detail.
2. Explain how Graph Neural Networks work.
3. Deep Dive into Gated Graph Neural Networks.
4. Deep Dive into Relational Convolutional Neural Networks.

By the end of this chapter, the reader should have gained some knowledge about graph neural networks and their details with relation to the comparative study between GGNNs and RGCNs. As a note, I made it a point to not be too mathematically dense as this topic can be fairly involved. My secondary goal here is for everyone with even a novice understanding of Machine Learning to pick up on at least the basics involved.

The Variable Misuse Task

Properly defining and understand the task associated with our statistical learning algorithm is of paramount importance. The Variable Misuse Task, in a nutshell, is a task that requires the algorithm to predict the most fitting variable in a particular scope among all the variables that are of the same type. If there is a discrepancy between what the code highlights and our confidence level of the prediction from our algorithm, the intended action to flag that line of code as a potential bug. To get a better understanding of this task, two examples would be helpful:

Fig 2: Example 1 of Application of the VarMisuse Task in C#

```
 1 private readonly object _syncRoot = new object();
 2 private bool _isDisposed = false;
 3
 4 public override bool IsDisposed
 5 {
 6     get
 7     {
 8         lock ( #1 )
 9         {
10             return #2;
11         }
12     }
13 }
```

This snippet of code is locking on a variable to provide synchronized access and then returning another variable. As a part of this task, predicting which variable is most suited for #1 and #2. And if there is a discrepancy between our confidence levels from the model and what the code indicates, that there is a case of variable misusage. To be more specific, if the GGNN model was tested on this code that has been trained on other repositories, the following results are obtained:

For #1: **_syncRoot** as the correct variable with a confidence of 95% and **_isDisposed** as 5%.

For #2: **_isDisposed** as the correct variable with a confidence of 99% and **_syncRoot** as the correct variable as 1%.

and these results imply there was no variable misuse bug in this case.

As a counterexample where we do detect a discrepancy between what's predicted and the actual variable used in the code is:

Fig 3: Example 2 of Application of the VarMisuse Task in C#

```
1 int[] results = new int[] { 1, 2, 3, 4, 5 };
2 int MAGIC_NUMBER = 200;
3 for(int idx = 0; idx < results.Length; idx++ )
4 {
5     results[ MAGIC_NUMBER ] += 2;
6 }
```

This simple snippet of code essentially adds two to all the numbers in the array while looping through it. The fact that the code is trying to add 2 to the 200^{th} index of the results array wouldn't be highlighted as a compilation error and therefore, would require other less rule based methods to detect this at develop or compile time. The results from a well trained hypothetical model would illuminate the at the low confidence of the actual variable used in the source code and complain to the user that there could potentially be a bug with the code while suggesting the variable i.e. idx in this case, with the highest confidence.

I'd like to point out here that this task would be extremely difficult to achieve with high accuracy with a plain vanilla sequence model. The representative capacity of the graph neural network helps with the inference of the role and functions of the variables of the program and allows the efficient learning and retaining of pertinent features. Additionally, the variable misuse task can be used as a seminal example as it is a proxy for a considerable number of static analysis related tasks that can be similarly incorporated due to its similarity to code completion.

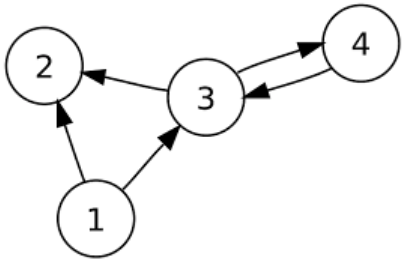
Graph Neural Networks

Now that the task at hand has been well defined, it is important to shift gears and talk about Graph Neural Network Models by first going over what a graph is and then how neural networks can be applied for prediction for any feasible task.

Graphs and Representation of Programs

A graph is defined by a set a nodes or vertices and a list of edges connecting the nodes. A simple example of a graph is as follows where 1,2,3 and 4 are the nodes and the arrows are the edges connecting these nodes.

Fig. 4: Graph Example



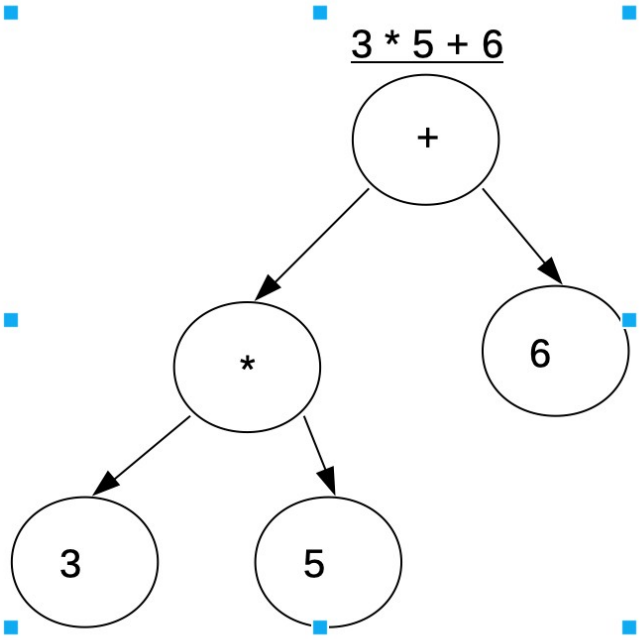
In the context of representation of programs as graphs, directed graphs, or graphs with edges that can be pointing from one node to another, are specifically considered contrasted with undirected graph where all edges are bi-directional. We further add more characteristics to the plain vanilla directed graph to accommodate the representational

capacity we want to achieve. For each node or vertex, we include associated features that will represent the state of the node associated with a token that could be a variable. Additionally, we introduce the concept of edge types that will come in handy later when we want to add different types of edges that represent disparate characteristics of the program such as a different edge type for representing if a variable is reading from another variable and a different one if the variable is writing to it.

Graph Construction

The construction of the graphs that represent programs is a modified version of the Abstract Syntax Tree represented by Rosyln, the .NET compiler framework. The Abstract Syntax Tree is a representation of the abstract syntactic structure of the code where the **syntax nodes** correspond to the programming language's grammar while the **syntax tokens** correspond to the leafs of the tree that represent the string from the source code.

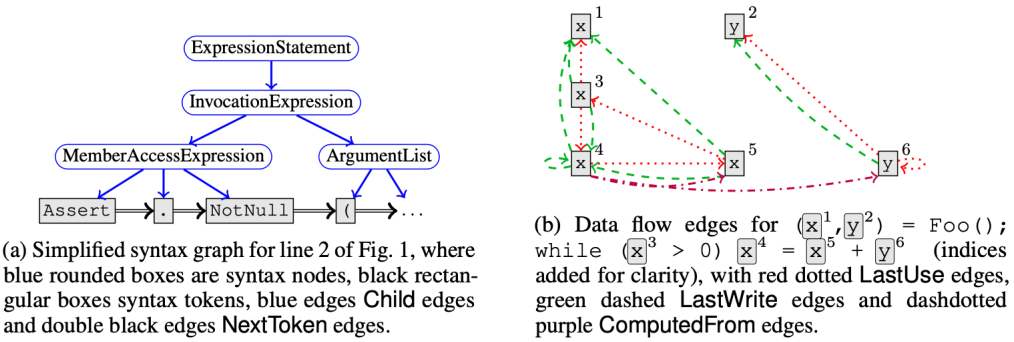
Fig 5: Simple Abstract Syntax Tree



The example highlights that for the line of code 3 * 5 + 6, we have the syntactic tokens i.e. 3,5 and 6 are leafs of the tree and * and + that are the corresponding arithmetical operators.

It is worth noting that the type of modified syntax trees we will be working with will be a lot more complex. As an example of how these graphs can be visualized, we borrow an example from Allamanis et al. (2018):

Fig 6: Modified Abstract Syntax Tree that Represent Programs



(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: Examples of graph edges used in program representation.

[TODO]: Fix the image

From the figure on the left, we observe the corresponding Program Graph for a line of code that asserts if a particular variable isn't null. The figure on the left highlights the modifications done to the edges among the vertices that are based on the type of relationship exemplified by the code. This modification step is what transforms the abstract syntax tree into a graph by adding different types of directed edges between components.

Node Feature Embeddings

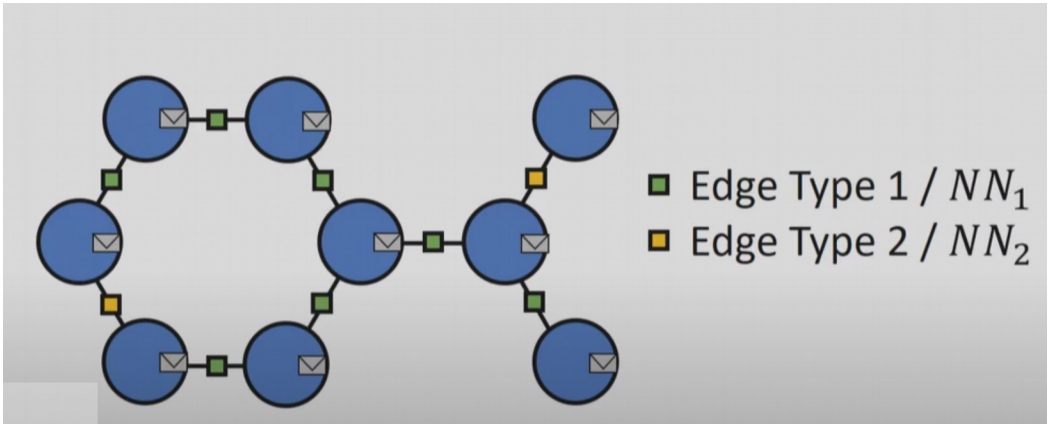
Each node feature is initialized by taking the initial embedding of the token that's obtained as a function of the embedding of the textual representation of the token and it's associated type. To be more specific, the embeddings of subtokens of the token are averaged and then the embedding of type of the node is concatenated with the averaged value and passed through a linear layer.

Training

The training is described as follows:

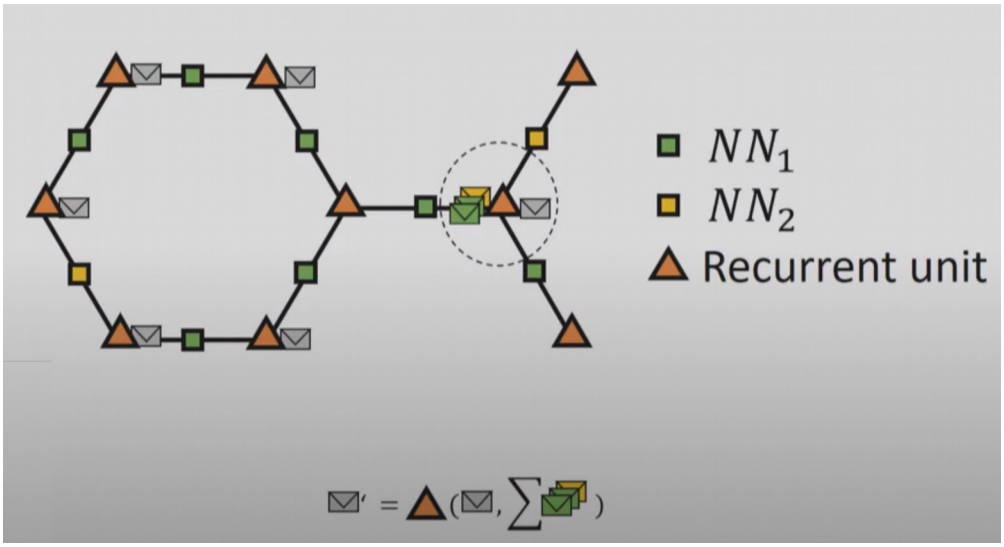
1. Each node is associated with a state i.e. its features that will be sent to each of its neighbors based on the type of edge. There is a unique neural network for each of the type of edges associated with a particular node that demarcates the learning behavior.

Fig 7: Figure Highlighting the Messages For Each Node



2. Once the data is sent from one node to its neighbors, it is aggregated using a summing function and is subsequently passed through an additional recurrent unit. This unit applies an additional mathematic function based on the previous state and the summed up messages from its neighbors and eventually other nodes. The intuition here is that the features of a node after each time step become are some mathematical function of its original state and states of other nodes of the graph

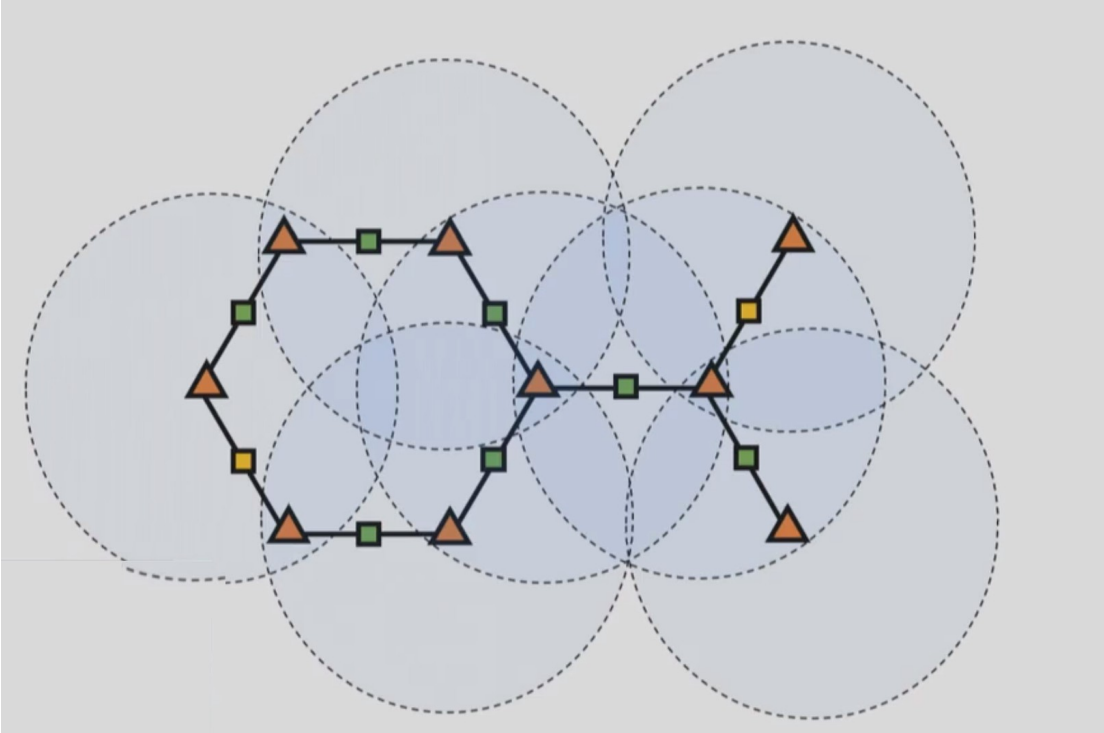
Fig 8: Message Passing And Aggregation by Nodes via a Recurrent Unit



3. The unrolling stage is when, after some number of time steps that can be a hyperparameter, the individual nodes are “aware” of not only the features of its

neighbors but also other nodes in further locations on the graph and these are indicated by the large dotted circles in the figure below. By the end of a certain number of time steps, all nodes should be acquainted with their own position as well as all other positions on the graph.

Fig 9: Unrolled Effect Of the Graph Where Nodes Become Positionally Aware



The objective function used for the training is that of a max-margin objective on the concatenation of the state of a slot variable i.e. the state of an empty spot we are trying to predict and the state of all possible same typed variables in the same scope. In other

words, using the state representations, we are trying to maximize the similarity of the state of the nodes representing a variable of the same type in the scope matches the ones that would be characteristic of the slot.

This trained graph neural network can also be fed to higher layers for more complicated representations for predictions based on more involved tasks. Specific to our comparative study, we'll be making use of the trained graph neural networks with different units and aggregation functions to discern the more performant model and therefore, the two sections build on the concepts.

The difference in the models is based on the mathematic functions involved in generating the next time step's state.

Gated Graph Neural Networks

The GGNNs take advantage of a Recurrent Cell function of the Gated Recurrent Unit as the mathematical function to compute the next state and is responsible for keeping important features as the distance between nodes increase in a large graph.

Fig 10: Formula for the GGNN State Update Per Node

$$h_v^{t+1} := Cell(h_v^t, \sum_\ell \sum_{(u,v)\in A_\ell} W_\ell * h_u^t)$$

To explain the formula, for node ‘v’, we compute the state by running the previous state of ‘v’ and the aggregated messages of all the neighbors of ‘v’ multiplied by the weights related to edge type ‘l’ through a gated recurrent unit. We apply this same formula to compute the state of all nodes to get our gated graph neural network based model to a trained state for inference.

Relational Graph Neural Networks

Rather than a gated recurrent unit, the relational graph neural network makes use of the convolution function using a non-linear function on the previous state of the neighbors, weights and problem specific normalization constant that serves as a hyperparameter.

Fig 11: Formula for the RGCN Node State Update Per Node

$$\mathrm{h}_v^{t+1} := \sigma(\sum_\ell \sum_{(u,v)\in A_\ell} 1/c_{v,\ell} * (W_\ell * h_u^t))$$

Like before, the next state of the node ‘v’ is computed by passing the normalized convolved aggregated states of all neighbors of ‘v’ through a non-linear unit such as a Rectified Linear Unit (ReLu) for all graph edge types ‘l’.

The intuition here is that accumulation of the transformed feature vectors of neighboring nodes through the normalized sum provide us with apt representations. Additionally, the Relational aspect of the RGCN is closely associated with the different edge types that have been covered before as per the research paper [TODO]

Now that we have expounded on the basics of the models we’ll be comparing and providing a bird eyes view of what’s happening under the hood, we proceed with getting into details about the comparative case study.

CHAPTER 6

PRELIMINARY STEPS IN CASE STUDY

Problem Definition

Defining the problem as meticulously as possible is an important step in Data Science and furthermore, even more important in the case of a comparison study to avoid ambiguity in the criteria of comparison.

Our problem statement is to demonstrate which model between the Gated Graph Neural Network and Relational Graph Convolutional Network based models has a higher predictive power on the basis of the test accuracy based on data from:

1. All 25 top trending C# Repositories
2. Esoteric Repository namely “CommonMark.NET”.
3. A Popular Repository namely “Dapper”.

The general goal is to identify if there is a better model out of the two i.e. which model has a higher representational capacity, one associated with a gated recurrent unit and one with a convolutional operation. Once the better model is identified, doing further research to apply the model to more sophisticated tasks by pushing the boundaries of the intersection of deep learning and static analysis.

The impetus behind choosing the GGNN and RGCN models was to prove a definitive hypothesis about the better model between two very disparate models from sequence

models and convolutional models respectively. In a sense, *caeteris parbirus*, what type of mathematical construct generalizes and predicts better.

Interview Notes

As a part of the interview process for this project, I was lucky enough to exchange emails with one of the authors of the paper my comparative study is based off of, Marc Brockschmidt who extremely graciously answered all my very specific questions about the paper and methodology. Additionally, I spoke to Anmol Joshi, a data scientist from C-1 at Boeing to get a more general perspective about deploying models in production. The main takeaways from the conversations were:

1. There is no silver bullet. The evaluation depends on the data.
2. Keep the code simple and highly decoupled from the data.
3. Treat the code that generates the initial representations of your data with as much importance as your modeling code.
4. The code that can be used for initial representation can be made decoupled from our modeling code if the pipeline is setup correctly.
5. Experiment as much as you can. You will never get the right answer right away as there is no right answer. It's only improvement of representation.

6. Keep researching and reading white papers to learn ideas from across other subfields.

These conversations helped me gain a good foundation as to how to go through with my experimentation. The insight I gained about the methodology was extremely helpful as well.

The choice of the experiments will be highlighted in the next chapter where the source data will be described.

CHAPTER 7

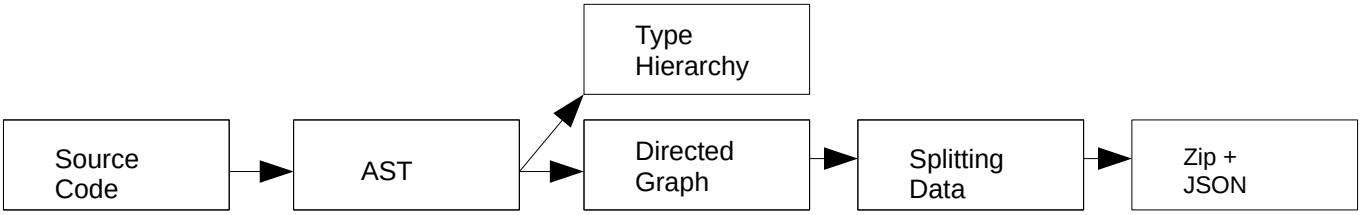
SOURCE DATA EXPLAINED

Introduction

The input data for this project is very different from the typical data frame based exploratory data analysis approach as it involves making use of the unstructured source code from repositories.

The absolute first stage of the data is that of the raw source code found in Github of the 24 top trending C# repositories. The source code from these top 24 trending C# repositories is then transformed into an Abstract Syntax Tree (AST) using the Rosyln, the .NET Compiler. Once the AST is obtained, the next step is to convert that it into a customized directed graph with the tokens representing the nodes and relationships between the variables represented by edges.

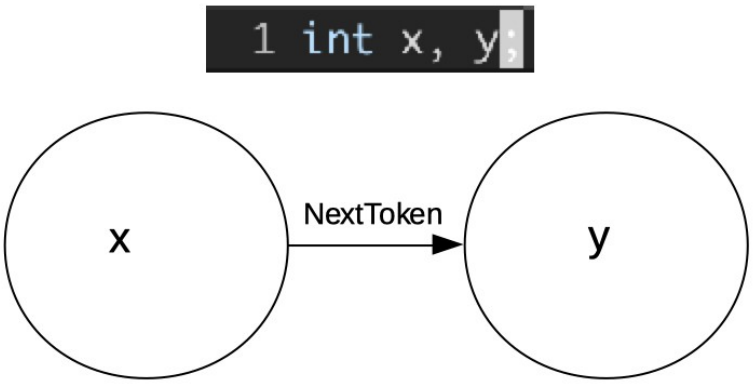
Fig 12: Data Transformation Pipeline



Examples of the different type of edges include:

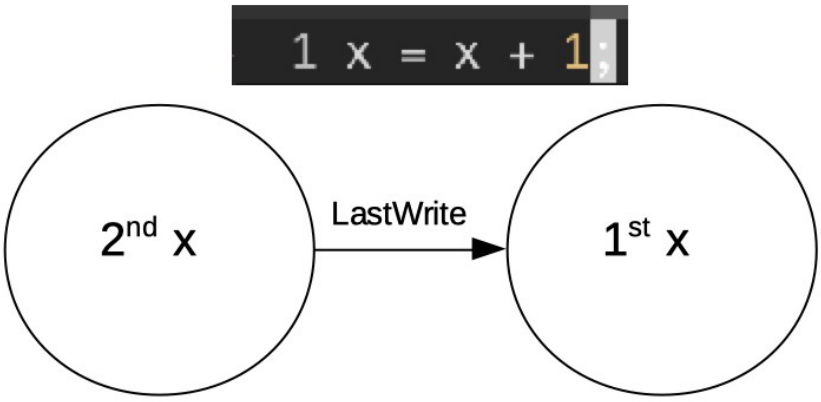
NextToken: The next token in the source code. In the example below, the x is pointing at the y as it is the next token in the source code.

Fig : Representing the Next Token Of the Directed Graph



LastWrite: Represents the relationship between variables where the directed edge points to the variable that was last written to by the node that was pointing to it. For example, in the figure below, the second ‘x’ would point to the first one as it is the last instance in the source code that has written to the variable.

Fig : Representing the LastWrite in the Source Code As a Graph



Once the graphs are successfully computed, they are converted into JSON and then zipped up using gzip. The code involved with acquiring the source data from the repositories till we get all the JSON files can be found here: https://github.com/microsoft/dpu-utils/tree/master/dotnet.

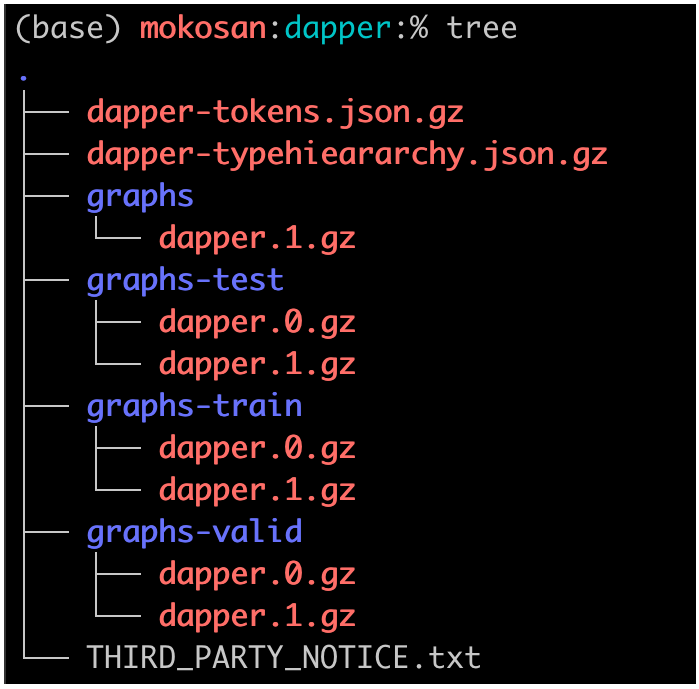
Thankfully, the transformed data for the top 25 trending C# repositories were easily available through Microsoft Research. The data obtained was a zip file consisting of 25 directories corresponding to each of the top trending repositories.

Fig : Listing of all the Directories From the Graph Data Set.

```
(base) mokosan:graph-dataset:% ls -l
total 104
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 akka.net
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 automapper
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 benchmarkdotnet
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 botbuilder
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 choco
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 commandline
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 commonmark
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 28 09:50 dapper
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 entityframework
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 humanizer
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 lean
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 nancy
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 newtonsoft
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 ninject
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 nlog
drwxr-xr-x 3 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 openlivewriter
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 opserver
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 orleans
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 polly
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 quartznet
-rw-r--r-- 1 mukundraghavsharma mukundraghavsharma 3871 Apr 13  2018 README.txt
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 restsharp
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 rxnet
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 scriptcs
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 signalr
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13  2018 wox
```

The structure of each of the directories is very similar where we have 2 zipped JSON files consisting of the token and type hierarchy information for the entire repository. Additionally, the data is split into testing, training and validation sets. The training and validation sets are used primarily for the training and hyperparameter validation and as suggested by the name, the testing files will be used to compute the test accuracy. This split is based on a percentage that's typically set to 60-20-20 for the training, validation and testing respectively.

Fig : Typical Directory Structure of Data From A Repository



Next, I'll be going over the contents and structure of the zipped JSON files above. The three types of files that can be found in the data folder are the tokens file, type hierarchy and the actual graph.

Token JSON

The tokens files consist of metadata associated with all the tokens as well as the data edges associated with the customized directed graph. This metadata along with the type hierarchy gives enough background information to the program to construct the graph in memory to conduct the training and testing. Examples of the type of properties are included in the token files include the types of the tokens, locations in terms of row and columns where the tokens were used, other variables of the same type in scope with a particular token and so on.

Type Hierarchy

The type hierarchy file consists of the type hierarchy associated with types i.e. the literal type used in the source code as well as the base classes, if any, that they were derived from. Additionally, an array of the type hierarchy where the indices correspond to the types used in the source code were added.

Structure of Typical Graph File

The graph file consists of the context directed graph that contains all the information needed for both the training and testing. The main part of the file consists of a

representation of the graph as a dictionary of the different edges, nodes and combinations of the cases where each variable we are trying to predict is replaced by a “slot” or empty spot for which we will maximizing the match on with all the other variables of the same type and in the same scope.

Studying these files in depth gave me a considerable insight as to how meticulously the input data is structured. Verbosity in the data seems to be the prevailing trend here where there is considerable amount of emphasis on is made on over specification of the details so that there is minimal transformation during training and testing.

I would like to note that it is a bit difficult to specify all the properties in each of these classes and for that reason I stuck with a basic description. Nonetheless, I have added typical samples to the repository that can be found in the assets/samples/ directory.

Other Data Sources

The other data source considered was that of the top trending 25 C# repositories on Github that I added to a CSV file called “RepositoryInfo” and was the only structured data in this project compared to the unstructured nature of the source code. The purpose of this data was to add form to and define different aspects of my experiments;

specifically answering questions such as what is an esoteric repository? Or what should be characterized as a popular repository? Acquiring this data involved getting the information from Github on the top 25 trending C# repository such as making use of the Github API.

The features considered in this case were the following to add form and definition to the three experiments I plan to consider:

1. **Repository**: The name of the repository considered.
2. **Link:** A link or URL of the repository on Github.
3. **Stars:** A proxy for the crowd-sourced popularity of the repository. Equivalent to the number of likes on Twitter or Facebook.
4. **Watchers:** The number of users who are actively getting updates about the repository such as issues posted by other users.
5. **Forks:** The number of copies of repositories made by other users for the purposes of independently developing the repository with their own changes.
6. **Commits:** The number of snapshots indicating some unit of work done by the contributors.
7. **Contributors:** The number of unique users who contributed to the repository since inception.

An example of a record from this data is the following:

Fig: Example of a Record of the RepositoryInfo

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
akka.net	https://github.com/akkadotnet/akka.net	3.5k	304	891	5547	197

CHAPTER 8

METHODOLOGY AND RESULTS

Introduction

Now that we have gone over the fundamentals of Graph Neural Networks, highlighted the preliminary case study and described the source data, the next step would be to elucidate on the methodology leading up to the obtaining the results. This involves a number of steps are these are:

1. Experimental Design
2. Details Related to the Pipeline
3. Conducting the Experiments
4. Results
5. Troubleshooting

Experimental Design

Designing the experiments to properly fit the problem at hand is of paramount importance. I decided to conduct three experiments for this comparison between GGNN and RGCNs that are designed to answer the following three questions:

1. Which type of model provides a higher test accuracy across all the top trending C# repositories?
2. Which type of model provides a higher test accuracy for an esoteric repository?
3. Which type of model provides a higher test accuracy for a popular repository?

The rationale for choosing the concomitant experiments relating to the aforementioned questions was to examine the results across the entire spectrum of the type of repositories. Not all the repositories garner the same number of contributors or the number of stars, a crowd-sourced metric that's a proxy of the popularity of the repository. on Github and I wanted my comparative experiments to revolve around other dimensions of generality apart from looking at results across all repositories.

To be as specific as possible, the following three experiments conducted were in an effort to obtain the test accuracy for both the GGNN and RGCN for:

1. All repositories
2. An Esoteric Repository
3. A Popular Repository

The 25 top trending C# repositories considered were the following:

Fig : All Repositories Considered

RepositoryInfo

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
akka.net	https://github.com/akkadotnet/akka.net	3.5k	304	891	5547	197
automapper	https://github.com/AutoMapper/AutoMapper	7.1k	403	1500	3512	137
benchmarkdotnet	https://github.com/dotnet/BenchmarkDotNet	5.2k	261	554	1980	129
botbuilder	https://github.com/microsoft/botbuilder-dotnet	0.496k	96	294	9010	124
choco	https://github.com/chocolatey/choco	6.4k	237	642	2965	76
commandline	https://github.com/commandlineparser/commandline	2k	64	261	1809	73
commonmark	https://github.com/Knagis/CommonMark.NET	0.929k	66	137	392	11
dapper	https://github.com/StackExchange/Dapper	12.1k	1000	3100	1387	151
entityframework	https://github.com/dotnet/ef6	1.1k	160	474	2921	79
humanizer	https://github.com/Humanizr/Humanizer	4.6k	213	672	1904	167
lean	https://github.com/QuantConnect/Lean	3.3k	347	1600	9887	111
nancy	https://github.com/NancyFx/Nancy	7.1k	460	1500	5473	263
newtonsoft	https://github.com/JamesNK/Newtonsoft.Json	7.7k	527	2600	1821	95
ninject	https://github.com/ninject/Ninject	2.3k	172	526	851	27
nlog	https://github.com/NLog/NLog	4.4k	288	1100	5735	152
openlivewriter	https://github.com/OpenLiveWriter/OpenLiveWriter	2.3k	208	484	696	46
opserver	https://github.com/opserver/Opserver	3.9k	331	785	998	55
orleans	https://github.com/dotnet/orleans	6.4k	550	1500	4824	210
polly	https://github.com/App-vNext/Polly	7.3k	374	743	961	54
quartznet	https://github.com/quartznet/quartznet	3.8k	326	1200	1295	65
restsharp	https://github.com/restsharp/RestSharp	7.1k	459	2000	1743	199
rxnet	https://github.com/dotnet/reactive	4.2k	295	548	3316	59
scriptcs	https://github.com/scriptcs/scriptcs	2.2k	152	375	1961	61
signalr	https://github.com/SignalR/SignalR	8k	712	2200	4684	81

Using the data from the “RepositoryInfo” table, I define two terms that were mentioned multiple times in our experiments:

Esoteric Repository

An esoteric repository is defined as one with a relatively low number of stars and a low number of contributors whose purpose is extremely specific, however, the interest from the general C# populous is low. The choice of esoteric repository for the training process is that of CommonMark.NET, a repository that converts Markdown to HTML. The source code for this library is extremely elegantly written and the usage is simple and easy to follow.

Quantitatively, CommonMark.NET had < 1,000 stars, 11 contributors and 66 watchers. These numbers are significantly lower than the other repositories considered and therefore, considering this repository as our ideal candidate of an esoteric repository seemed apt.

Fig: CommonMark.NET Details

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
commonmark	https://github.com/Knagis/CommonMark.NET	0.929k	66	137	392	11

Qualitatively, based on the nature of the code of the repository i.e. a .NET port of an extremely popular library, CommonMark, indicates that this was probably someone's weekend project or needed for another project that gained some traction over time. Therefore, CommonMark.NET seems like the ideal candidate from even a qualitative perspective to be characterized as an esoteric repository.

Popular Repository

A popular repository is one that has a lot of stars, contributors and users watching for updates. The choice of popular repository is that of Dapper, a simple object mapper for .NET that could be used to abstract out the layer between the database and plain old class objects (POCO).

Quantitatively, Dapper has around 12.1 thousand stars, 1000 watchers, 151 contributors and 3100 forks indicating it is a well maintained and liked repository with a lot of interest in further development as it is used by many .NET developers in production.

Fig: Dapper Details

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
dapper	https://github.com/StackExchange/Dapper	12.1k	1000	3100	1387	151

From a qualitative perspective, since an uncontested amount of code involves accessing a database and retrieving objects, Dapper seems like the perfect candidate for a popular repository. Additionally, the main repository has branched out into many other packages under the same umbrella.

Now that the experiments and definitions associated with the experiments are well defined, the next step to detail how the code was run on the data.

Details Related to the Pipeline

This section involves the following in an effort to get the reader quickly up to speed with the abstractions and the pipeline to conduct experiments to generate the test accuracies:

1. Describing the Source Code To Run The Experiments
2. Preparation of the Data
3. Training
4. Testing

Describing the Source Code To Run The Experiments

The code used to run the experiments can be found here: https://github.com/mokosan/tf-gnn-samples. The link is to my fork of the original "tf-gnn-samples" repository or the Tensorflow based implementation of the GNN based sample library by Microsoft Research can be found here: https://github.com/microsoft/tf-gnn-samples. My fork contains additional modifications to the code to fit the pipeline I wanted to run as well as additional assets supplementing this paper.

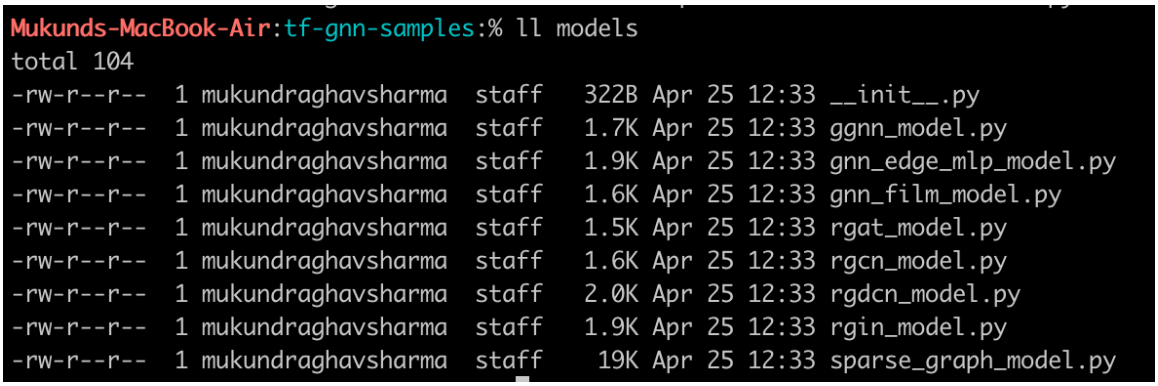
The original source code that my implementation is based off of is written in Python 3.x using Tensorflow 1. There is some discussion that indicates that the code is getting converted into Tensorflow 2 that, in my opinion, is considerably more user friendly than Tensorflow 1.

The major abstractions include the following:

Models:

The models encapsulate different types of graph based neural networks and are essentially representations of the directed graphs containing the edges and nodes with the graph neural networks.

Fig: Models



Graph Neural Network Types:

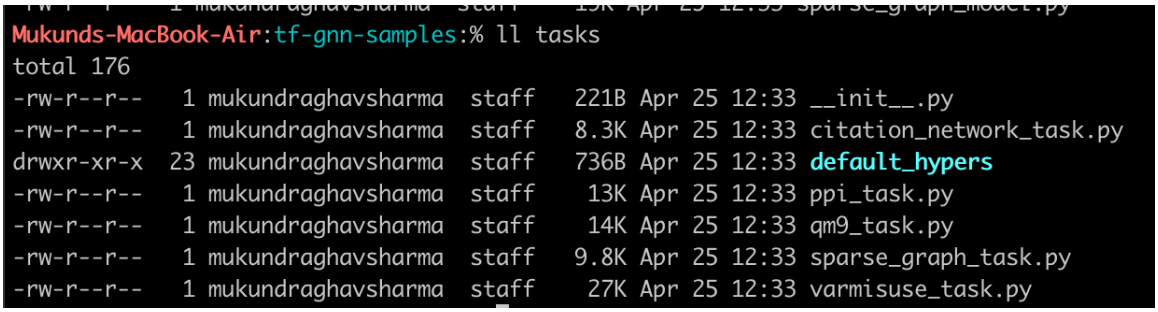
The code in sparse_graph_model.py provides a general entry point to make use of other models based on the inputs from the user. The code paths made use of from these abstractions are rgcn_model.py and gnn_model.py.

Tasks:

As mentioned before, this repository contains more than just the task we are working with i.e. the Variable Misuse Task. This directory contains the other tasks along with dictionaries containing the default hyperparameters associated with the tasks and the respective models.

I'll touch on this briefly but I didn't find much benefit in tweaking the default hyperparameters based on my experimentation as the results were very similar with negligible overall improvement. The hyperparameters were stored as a JSON file and then fed into the code as a dictionary in typical python fashion. The specific files that I made use of from the tasks directory were varmisuse_task.py and the hyperparameters were VarMisuse_RGCN.json and VarMisuse_GGNN.json.

Fig: Tasks And Hyperparameters

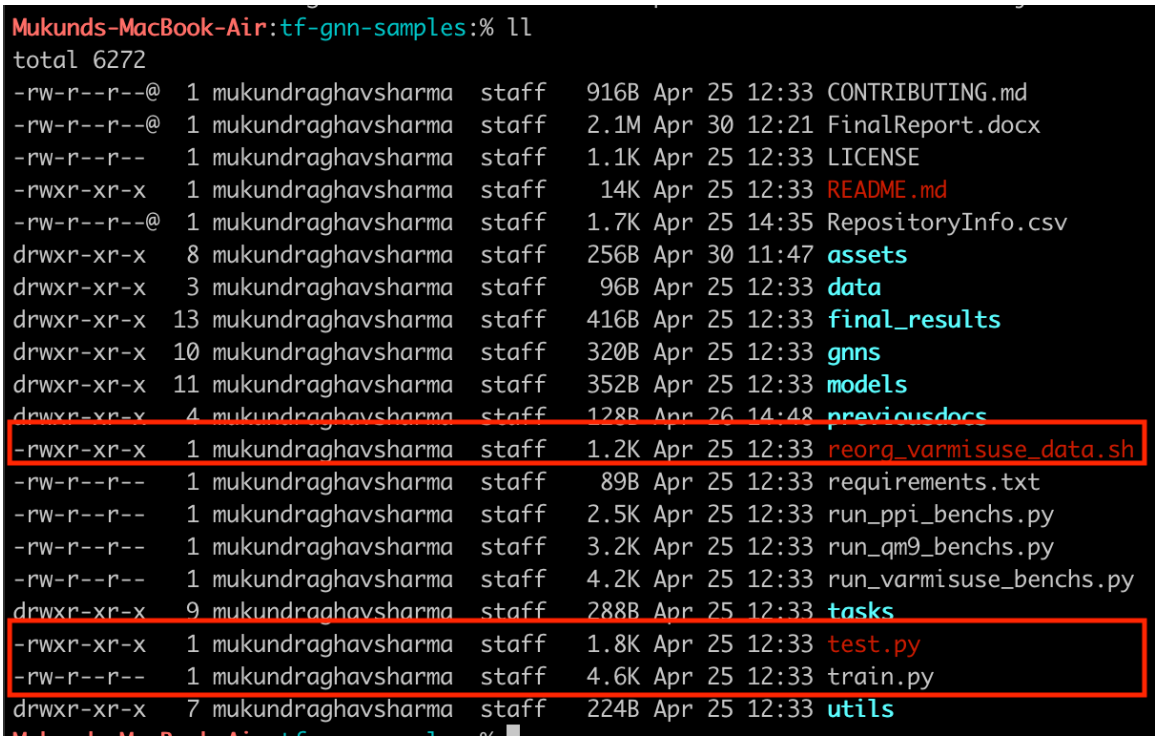


Running Scripts:

The three main scripts used to run the code were the reorg_varmisuse_data.sh script that I handcrafted to tailor specifically to the VarMisuse Task with some adjustments that were needed to run. This script basically converts the raw JSONized directed program graphs into digestible chunks that can be pulled into memory to create the models.

The two other scripts that run as command line programs are train.py and test.py. The source code from train.py takes the outputted data from reorg_varmiuse_data.sh and trains the models. Additionally, optimization is done on the basis of the validation data covered in the Source Data section. The weights after each epoch of training are serialized and saved as pickle files. These pickle files form the basis of which the testing is commenced by the test.py file that first loads the weights and runs a round of inference in the form of prediction to compute the test accuracy.

Fig : Scripts used Demarcated in Red



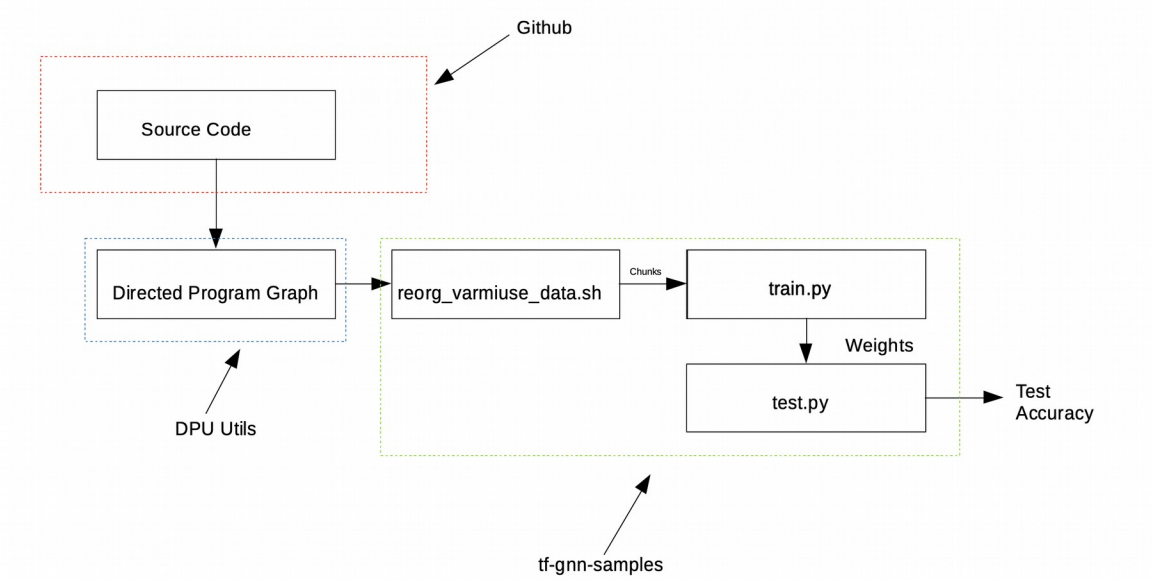
Misc Utilities:

Miscellaneous utility files include all the auxiliary functions used throughout the repository from code that is responsible for bringing the data into memory and constructing the representation of the directed program graph to dictionary look up functions used throughout the repository that map the name of the model type to its functional equivalent.

In general, I'd like to conclude by highlighting that the source code is incredibly cleanly written with emphasis on generality of the type of task, model used and the graph neural network considered to bolster separation of concerns. I also appreciated the documentation associated with all the graph neural network types that supplemented my theoretical understanding by easily being able to reason about the mathematical formulae. One other aspect of the source code that I considered to be done incredibly well done was the clearly separated utilities that are used across a lot of the files. Marc Brockshmidt, the contributor I reached out to was also incredibly welcoming in others contributing, which made this exploration experience all the more worthwhile.

Pipeline Mockup

The entire pipeline



Conducting The Experiments

This section involves adding specificity to the abstractions described in the previous section to be applied to the experiments that were conducted.

TODO: Pipeline Mock up specific to the experiment

Results

Add Table

Explain Results

CHAPTER 9

CONCLUSION

REFERENCES

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The graph neural network model. IEEE Transactions on Neural Networks 20(1):61–80.

[1] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In International Conference on Software Engineering (ICSE), 2012.

[2] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In Conference on Computer and Communications Security (CCS), 2016.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. arXiv:cs.LG/1711.00740