Inferring Javascript types using Graph Neural Networks

Jessica V. Schrouff*
Prodo Tech Ltd
London, United Kingdom
jessica@prodo.ai

Kai Wohlfahrt
Prodo Tech Ltd
London, United Kingdom
kai@prodo.ai

Bruno Marnette
Prodo Tech Ltd
London, United Kingdom
bruno@prodo.ai

Liam Atkinson
Prodo Tech Ltd
London, United Kingdom
liam@prodo.ai

Abstract

The recent use of 'Big Code' with state-of-the-art deep learning methods offers promising avenues to ease program source code writing and correction. As a first step towards automatic code repair, we implemented a graph neural network model that predicts token types for Javascript programs. The predictions achieve an accuracy above 90%, which improves on previous similar work.

1 Introduction

Automatic bug detection or program correction are active fields of research, with recent developments at the intersection of programming languages and machine learning. These approaches have proven useful in the context of dynamic programming languages for which safeguards such as static types are lacking (Xu et al., 2016). For instance, in Python or Javascript, inferring the types of code tokens can be challenging and lead to undetected errors. Inferring token types and potentially notifying of type mismatches in a program would hence help preventing undesirable behaviour.

Recent advances in the field of *Big Code* have allowed to e.g. perform code or comment completion, detect code defects or automatically determine token properties, with encouraging success (Allamanis et al., 2018). Modelling large amounts of program code is hence a promising avenue to perform automatic source code fixing or provide recommendations during code writing.

In the present work, we use state-of-the-art modelling strategies of program source code to infer Javascript types. Our approach outperforms the current baseline in the field and can easily be deployed in e.g. an online editor or code review tool.

1.1 Related work

In (Allamanis et al., 2017), C# code is represented as a graph including syntactic and semantic information to predict variable names or misuse. While this work provides an interesting framework using graph neural networks and convincing results, the type of each token is used as a feature, which can only be obtained for statically typed languages. On the other hand, previous work on inferring types (e.g. (Raychev et al., 2015)) did not make use of novel deep learning methods and focused on specific subtasks (e.g. predicting types for function parameters only). While their results are promising, deep learning, and especially Graph Neural Network (GNN, (Scarselli et al., 2009)) models present the advantages of being typically high accuracy and agnostic models (i.e. the level of pre or post-processing is low).

*https://prodo.ai/

In this work, we represent each Javascript source code as a graph and use state-of-the-art graph neural networks to predict from eight Javascript types for each token.

2 Materials and methods

2.1 Data and code representation

Our dataset consists of Javascript source files gathered from Github, excluding files that significantly overlapped[1]. The dataset is then split once into train and test partitions according to repositories. The training set includes 10,268 files from 660 repositories and 578 organizations, while the testing set includes 3,539 files from 136 repositories and 127 organizations.

We represent each program source code as a graph including syntactic and semantic edges. More specifically, we identify the nodes of the graph[2] from the tokens in the abstract syntax tree (AST, min. nodes = 3, max. = 19,516), each node having a feature vector including:

- *ast-node-type*: derived from the program's grammar, e.g. 'DeclareClass'. We considered 144 possible AST node types (see Appendix A).
- *property*: some types have an associated property, e.g. nodes of type 'Assignement-Expression' can have the property 'operator:='. 107 properties were considered.
- *values*: each leaf node of the AST has either a name, value or pattern associated. These are stored as strings (capped at 16 characters) for each node.

We then construct different types of edges between the obtained nodes based on syntactic (e.g. 'child' relationships from the AST) and semantic (e.g. 'defined by') relationships between tokens. While our representation differentiates between 96 relationship types (Appendix A), this initial work groups node edges in two categories (AST and reference/traverse).

For each node covered by the test suite of a repository, a label is extracted during run time analysis. These labels correspond to the following JavaScript types: object, string, function, number, undefined, array, boolean and null. Nodes that cannot be associated to a type are assigned an 'unknown' type. These are masked out during training to avoid biasing our model towards predicting 'unknown'. In addition, nodes with an explicit type-label correspondence (e.g. 'DeclareFunction' type and 'function' label) are included during the training phase, but excluded from the evaluation phase for a more objective assessment of model performance.

2.2 Neural Network modelling

Node embeddings: The node vectors for each feature (i.e. type, property and string) are forwarded to specific embedding layers. For the node types, an embedding of size 144 to size *hidden-size* is used, as nodes can have multiple types. The node properties are encoded as a one-hot vector of size 107 then passed through a linear layer of size *hidden-size*. The strings (of fixed size 16, padded with white space if needed) are first embedded in a lookup table from size 104 symbols to *hidden-size* and then passed through a Gated Recurrent Unit (GRU) of size *hidden-size*, *hidden-size* being a hyper-parameter fixed across all model layers. The different embeddings are then concatenated, before being passed through a number of feed forward neural network blocks consisting of batch-norm normalization, linear layer with dropout and ReLU non-linearity.

Graph NNs: We consider two graph neural network models: the Graph Convolutional Network (GCN, (Kipf & Welling, 2016; van den Berg et al., 2017)) and the Gated Graph Neural Network (GGNN, (Li et al., 2015; Gilmer et al., 2017)).

[1] Based on a similarity analysis as performed using `https://www.harukizaemon.com/simian/`

[2] Please note that we exclude graphs with over 20,000 nodes (i.e. 4 files) for computational reasons.

In the GCN model, a message passing layer collects the messages from a node's neighbours for each edge category (separately), before passing them into a linear layer of size *hidden-size* with dropout. The messages from different edge categories are then summed and passed through a ReLU non-linearity. The number of message passing layers is treated as a hyper-parameter, with weights and dropout untied.

The GGNN model uses a GRU cell to update the node states in each message passing layer. Weights in the message passing phase are tied across layers and messages from different edge categories are passed through a specific linear layer then added together before GRU update, as in (Gilmer et al., 2017). Dropout is implemented in the message passing, in which random parts of the messages and of the node vector are dropped before GRU update. The dropout masks are identical across layers, following (Gal & Ghahramani, 2016). In addition, a 'master node' is considered for each graph, on which all nodes can write and from which all nodes can read. This master node allows long-distance information to be shared across nodes (Gilmer et al., 2017). To this setting, we add a dropout parameter on the nodes that can read or write from the master node, hence creating random 'skip connections' across the graph. This dropout value, the dimension of the master node and whether or not to include a master node are considered as hyper-parameters.

Decoding: The decoder, identical for all networks, consists of a log softmax layer preceded by a number of feed forward neural network blocks, as previously described.

2.2.1 Implementation

The data is split in mini-batches of 50 graphs or 20,000 nodes (whichever happens first during the random sampling) before entering the embedding, message passing and decoding steps. Accuracy is reported as the F1-score, micro-averaged over the batches in the validation set or over the test set (not batched). The ADAM optimizer (Kingma & Ba, 2014) updates the model parameters, with an initial learning rate as manually selected from section B.1. The models include a learning rate scheduler, decreasing the learning rate by a factor of 0.1 when encountering a plateau in validation accuracy. The best configuration for each model is selected based on a hyper-parameter search, as described in section B.2. All data representations and models were implemented in Python 3.6 using Pytorch 1.0.0.

3 Results

The final results are displayed in Table 1 in terms of F1-score and illustrated in Figure 1. The test set includes a total of $1,043,333$ tokens, among which $751,824$ tokens do not have a ground truth type and $64,631$ have explicit types, hence evaluation is performed on $226,878$ nodes.

Table 1: Model performance (in %).

	Training	Validation	Test
GCN	95.04	87.69	87.25
GGNN	98.01	90.52	90.79

Both models reach high performance on the considered test set, significantly higher than the 81 % reported in (Raychev et al., 2015). We can further observe that the models perform well on all types apart from 'Null' which is less represented in the learning set, and are able to identify missing types on trained node types reasonably well (visual inspection through our web app, demonstrated during the workshop).

4 Discussion and future work

This work provides an example of using *Big Code* to infer token types in Javascript source code. Interestingly, both models performed well on this dataset, with only marginal improvement

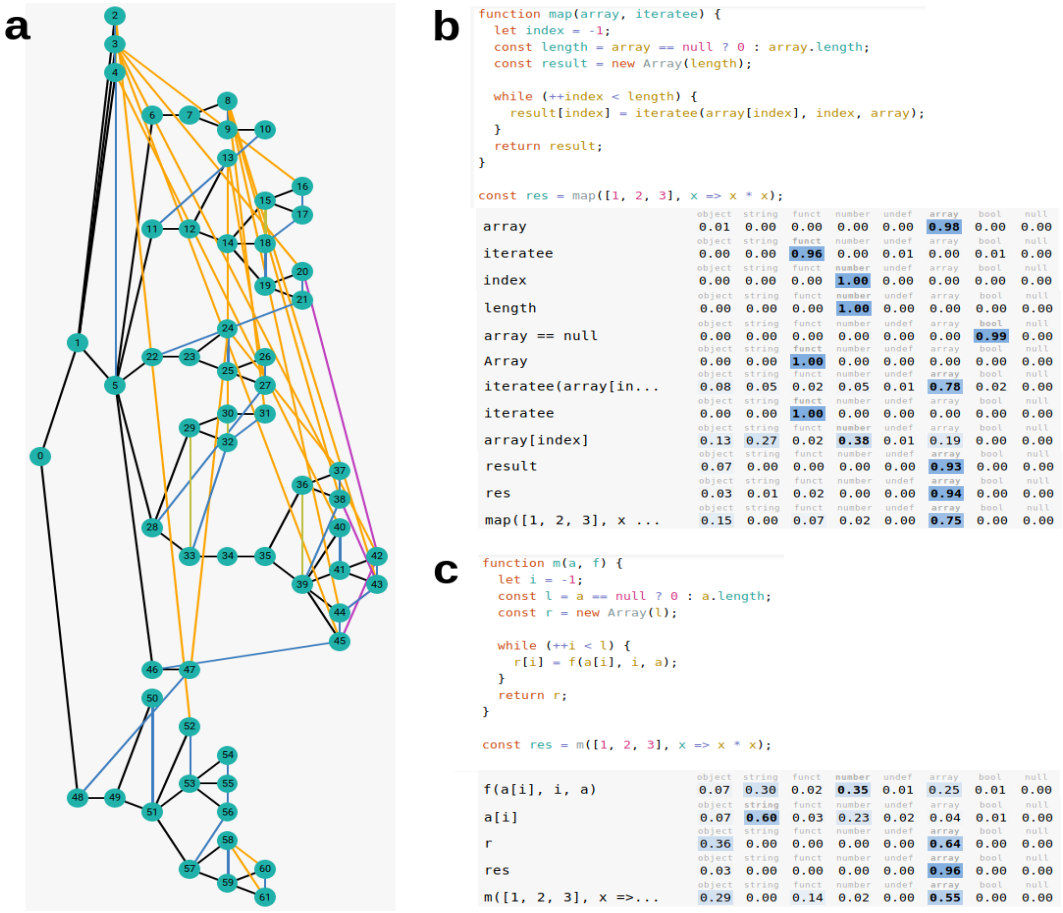


Figure 1: **a** Graph obtained from the *map* function displayed in **b**. Black lines represent AST child edges, blue represent 'next-node' edges, orange represent 'defined-by' edges, magenta represent 'next-use' edges and dark yellow represent 'next-sibling' edges. **b** Code with output type predictions (truncated). The model correctly identifies the variables 'array', 'iteratee' and 'result' and can differentiate 'Array' from 'array'. It however attributes the type 'array' to the return of the call to 'iteratee', which could be of any type. On the other hand, it is uncertain of its prediction for 'array[index]', which could also be of any type. **c** Same code but removing information from the variable names. In this case, most variables are still correctly identified but 'a[i]' is now predicted with some confidence as a string. The model is less confident about its predictions for 'r' and 'm([1,2,3, x=> x*x])'.

observed for GGNN. This high accuracy could be related to the simplicity of the task, i.e. predicting pre-defined and general types. This aspect limits the potential comparison with the work of Raychev et al. (2015), as they predicted more types, but for a limited number of tokens (i.e. function parameters only). We could extend our label set by adding more general types (e.g. 'RegExp' or 'Date') or by fine-tuning a model to take into account user specific types, potentially within a project or an organization.

Future work will investigate the effect of increasing the number of edge type categories on model performance, both in terms of accuracy and computational expenses. In addition, augmenting the semantic information might improve performance, for example 'return' relationships were shown to increase accuracy in (Allamanis et al., 2017).

Finally, the inferred types will be used to automatically detect *type mismatches*, e.g. notifying the programmer of potential issues in variable usage. In this context, obtaining uncertainty measures on the type predictions will be beneficial. This could be performed by modifying the dropout implementation (Gal & Ghahramani, 2015). Our next implementation will also

take semantic rule into accounts, a common strategy to improve performance and include domain knowledge (Raychev et al., 2015).

5 Data availability

The data set of this application will be released on our website `https://prodo.ai`, as soon as possible.

Acknowledgments

We thank all the team at Prodo Tech for their help on this project, especially Sergio Giro, Mani Sarkar and Jake Runzer.

References

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

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4):81, 2018.

Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. *arXiv:1506.02142 [cs, stat]*, June 2015.

Yarin Gal and Zoubin Ghahramani. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *Proceedings of the 30th Conference on Neural Information Processing Systems*, pp. 9, 2016.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. *arXiv:1704.01212 [cs]*, April 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the ICLR 2015*, December 2014.

Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat]*, September 2016.

Lisha Li, Kevin Jamieson, Afshin Rostamizadeh, Katya Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively Parallel Hyperparameter Tuning. February 2018.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. *arXiv:1511.05493 [cs, stat]*, November 2015.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv:1807.05118 [cs, stat]*, July 2018.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*, pp. 111–124, Mumbai, India, 2015. ACM Press. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009. ISSN 1045-9227. doi: 10.1109/TNN.2008.2005605.

Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. *arXiv:1803.09820 [cs, stat]*, March 2018.

Rianne van den Berg, Thomas N. Kipf, and Max Welling. Graph Convolutional Matrix Completion. *arXiv:1706.02263 [cs, stat]*, June 2017.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 607–618, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950343.

A Data representation


```
{
  "nodeTypes": [
    "AnyTypeAnnotation",
    "ArrayExpression",
    "ArrayPattern",
    "ArrayTypeAnnotation",
    "ArrowFunctionExpression",
    "AssignmentExpression",
    "AssignmentPattern",
    "AwaitExpression",
    "BinaryExpression",
    "BlockStatement",
    "BooleanLiteralTypeAnnotation",
    "BooleanTypeAnnotation",
    "BreakStatement",
    "CallExpression",
    "CatchClause",
    "ClassBody",
    "ClassDeclaration",
    "ClassExpression",
    "ClassImplements",
    "ClassProperty",
    "ConditionalExpression",
    "ContinueStatement",
    "DebuggerStatement",
    "DeclareClass",
    "DeclareExportDeclaration",
    "DeclareFunction",
    "DeclareInterface",
    "DeclareModule",
    "DeclareModuleExports",
    "DeclareOpaqueType",
    "DeclareTypeAlias",
    "DeclareVariable",
    "Decorator",
    "DoExpression",
    "DoWhileStatement",
    "EmptyStatement",
    "EmptyTypeAnnotation",
    "ExistentialTypeParam",
    "ExportAllDeclaration",
    "ExportDefaultDeclaration",
    "ExportDefaultSpecifier",
    "ExportNamedDeclaration",
    "ExportNamespaceSpecifier",
    "ExportSpecifier",
    "ExpressionStatement",
```

```
"ForAwaitStatement",
"ForInStatement",
"ForOfStatement",
"ForStatement",
"FunctionDeclaration",
"FunctionExpression",
"FunctionTypeAnnotation",
"FunctionTypeParam",
"GenericTypeAnnotation",
"Identifier",
"IfStatement",
"Import",
"ImportDeclaration",
"ImportDefaultSpecifier",
"ImportNamespaceSpecifier",
"ImportSpecifier",
"InterfaceDeclaration",
"InterfaceExtends",
"IntersectionTypeAnnotation",
"JSXAttribute",
"JSXClosingElement",
"JSXElement",
"JSXEmptyExpression",
"JSXExpressionContainer",
"JSXIdentifier",
"JSXMemberExpression",
"JSXNamespacedName",
"JSXOpeningElement",
"JSXSpreadAttribute",
"JSXSpreadChild",
"JSXText",
"LabeledStatement",
"Literal",
"LogicalExpression",
"MemberExpression",
"MetaProperty",
"MethodDefinition",
"MixedTypeAnnotation",
"NewExpression",
"NullableTypeAnnotation",
"NullLiteralTypeAnnotation",
"NumberTypeAnnotation",
"NumericLiteralTypeAnnotation",
"ObjectExpression",
"ObjectPattern",
"ObjectTypeAnnotation",
"ObjectTypeCallProperty",
"ObjectTypeIndexer",
"ObjectTypeProperty",
"ObjectTypeSpreadProperty",
"OpaqueType",
"Program",
"Property",
"QualifiedTypeIdentifier",
"RestElement",
"RestProperty",
"ReturnStatement",
"SequenceExpression",
"SpreadElement",
```

```
    "SpreadProperty",
    "StringLiteralTypeAnnotation",
    "StringTypeAnnotation",
    "Super",
    "SwitchCase",
    "SwitchStatement",
    "TaggedTemplateExpression",
    "TemplateElement",
    "TemplateLiteral",
    "ThisExpression",
    "ThisTypeAnnotation",
    "ThrowStatement",
    "TryStatement",
    "TupleTypeAnnotation",
    "TypeAlias",
    "TypeAnnotation",
    "TypeCastExpression",
    "TypeofTypeAnnotation",
    "TypeParameter",
    "TypeParameterDeclaration",
    "TypeParameterInstantiation",
    "UnaryExpression",
    "UnionTypeAnnotation",
    "UpdateExpression",
    "VariableDeclaration",
    "VariableDeclarator",
    "VoidTypeAnnotation",
    "WhileStatement",
    "YieldExpression",
    "BindExpression",
    "ObjectProperty",
    "StringLiteral",
    "NumericLiteral",
    "NullLiteral",
    "BooleanLiteral",
    "Directive",
    "DirectiveLiteral",
    "ObjectMethod",
    "RegExpLiteral",
    "ClassMethod"
  ],
    "propTypes": [
    "{async:true}",
    "{computed:true}",
    "{delegate:true}",
    "{exact:true}",
    "{exportKind:type}",
    "{exportKind:value}",
    "{expression:true}",
    "{generator:true}",
    "{importKind:type}",
    "{importKind:typeof}",
    "{importKind:value}",
    "{kind:const}",
    "{kind:constructor}",
    "{kind:get}",
    "{kind:init}",
    "{kind:let}",
    "{kind:method}",
```

```
"{kind:set}",
"{kind:var}",
"{method:true}",
"{operator:--}",
"{operator:-}",
"{operator:-=}",
"{operator:!}",
"{operator:!=}",
"{operator:!==}",
"{operator:*}",
"{operator:**}",
"{operator:**=}",
"{operator:*=}",
"{operator:/}",
"{operator:/=}",
"{operator:&}",
"{operator:&&}",
"{operator:&=}",
"{operator:%}",
"{operator:%*}",
"{operator:%*=}",
"{operator:%=}",
"{operator:^}",
"{operator:^=}",
"{operator:+}",
"{operator:++}",
"{operator:+=}",
"{operator:<}",
"{operator:<<}",
"{operator:<<=}",
"{operator:<=}",
"{operator:=}",
"{operator:==}",
"{operator:===}",
"{operator:>}",
"{operator:>=}",
"{operator:>>}",
"{operator:>>=}",
"{operator:>>>}",
"{operator:>>>=}",
"{operator:|}",
"{operator:|=}",
"{operator:||}",
"{operator:~}",
"{operator:in}",
"{operator:instanceof}",
"{operator:typeof}",
"{operator:void}",
"{optional:true}",
"{prefix:true}",
"{selfClosing:true}",
"{shorthand:true}",
"{sourceType:module}",
"{static:true}",
"{tail:true}",
"{value:true}",
"{variance:minus}",
"{variance:plus}",
"{operator:delete}",
```

```
    "{flags:}",
    "{flags:g}",
    "{flags:i}",
    "{flags:gi}",
    "{flags:gm}",
    "{flags:m}",
    "{flags:ig}",
    "{flags:mg}",
    "{flags:mi}",
    "{flags:im}",
    "{flags:u}",
    "{flags:y}",
    "{flags:mig}",
    "{flags:s}",
    "{flags:sm}",
    "{flags:iu}",
    "{flags:ug}",
    "{flags:yg}",
    "{flags:my}",
    "{flags:su}",
    "{flags:sum}",
    "{flags:gim}",
    "{flags:gmi}",
    "{flags:iy}",
    "{flags:iyg}",
    "{flags:um}",
    "{flags:iug}",
    "{flags:is}",
    "{flags:sg}",
    "{flags:sy}",
    "{flags:ms}"
  ],
  "edgeTypes": [
    "ast.child.alternate",
    "ast.child.argument",
    "ast.child.arguments",
    "ast.child.attributes",
    "ast.child.block",
    "ast.child.body",
    "ast.child.bound",
    "ast.child.callee",
    "ast.child.callProperties",
    "ast.child.cases",
    "ast.child.children",
    "ast.child.closingElement",
    "ast.child.consequent",
    "ast.child.declaration",
    "ast.child.declarations",
    "ast.child.decorators",
    "ast.child.discriminant",
    "ast.child.elements",
    "ast.child.elementType",
    "ast.child.exported",
    "ast.child.expression",
    "ast.child.expressions",
    "ast.child.extends",
    "ast.child.finalizer",
    "ast.child.handler",
    "ast.child.id",
```

```
"ast.child.implements",
"ast.child.impltype",
"ast.child.imported",
"ast.child.indexers",
"ast.child.init",
"ast.child.key",
"ast.child.label",
"ast.child.left",
"ast.child.local",
"ast.child.meta",
"ast.child.name",
"ast.child.namespace",
"ast.child.object",
"ast.child.openingElement",
"ast.child.param",
"ast.child.params",
"ast.child.properties",
"ast.child.property",
"ast.child.qualification",
"ast.child.quasi",
"ast.child.quasis",
"ast.child.rest",
"ast.child.returnType",
"ast.child.right",
"ast.child.source",
"ast.child.specifiers",
"ast.child.superClass",
"ast.child.supertype",
"ast.child.superTypeParameters",
"ast.child.tag",
"ast.child.test",
"ast.child.typeAnnotation",
"ast.child.typeParameters",
"ast.child.types",
"ast.child.update",
"ast.child.value",
"ast.child",
"ast.next-in-list.arguments",
"ast.next-in-list.attributes",
"ast.next-in-list.body",
"ast.next-in-list.callProperties",
"ast.next-in-list.cases",
"ast.next-in-list.children",
"ast.next-in-list.consequent",
"ast.next-in-list.declarations",
"ast.next-in-list.decorators",
"ast.next-in-list.elements",
"ast.next-in-list.expressions",
"ast.next-in-list.extends",
"ast.next-in-list.implements",
"ast.next-in-list.indexers",
"ast.next-in-list.params",
"ast.next-in-list.properties",
"ast.next-in-list.quasis",
"ast.next-in-list.specifiers",
"ast.next-in-list.types",
"ast.next-in-list",
"ast.position-from-last.*",
"ast.position-from-last.1",
```

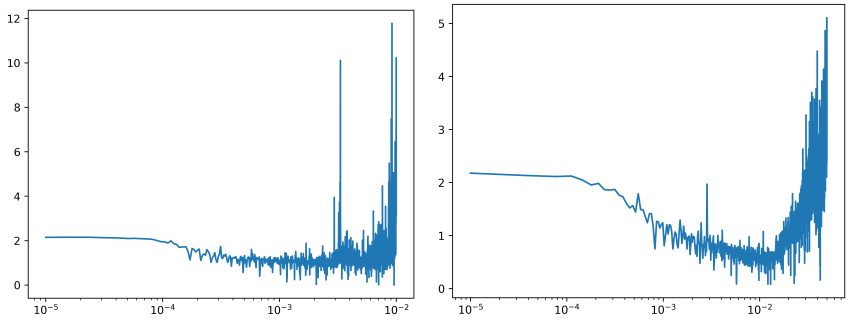
```
    "ast.position-from-last.2",
    "ast.position-from-last.3",
    "ast.position-from-last.4",
    "ast.position.*",
    "ast.position.0",
    "ast.position.1",
    "ast.position.2",
    "ast.position.3",
    "ast.position.4",
    "reference.defined-by",
    "reference.next-use",
    "traverse.next-node",
    "traverse.next-sibling",
    "ast.child.directives",
    "ast.next-in-list.directives"
  ],
}
```

B Hyper-parameter setting

We split the training set (based on repositories) into a single 80% - 20% partition (fixed random seed) to perform hyper-parameter search.

B.1 Learning rate

We first use the training set to identify a reasonable starting point for the learning rate of each model, as proposed in (Smith, 2018). The learning rate finders are presented in Figure 2. These plots display the loss on the training set while slowly increasing the learning rate after each forward pass on a minibatch, across a total of eight epochs. The initial learning rate value is selected at the middle of the 'acceptable' range, where the minimum is identified as the moment the model starts learning and the maximum as the moment the curve becomes too rough or training loss is at its minimum.



(a) GCN. (b) GGNN.

Figure 2: Learning rate finders for the considered models. The x-axis represents slowly increasing the learning rate after each batch while recording the training loss (y-axis).

The selected values are 0.0004 for GCN and 0.001 for GGNN.

B.2 Model hyper-parameters

This learning rate is then used to search over the space of hyper-parameters identified for each model. More specifically, we use Ray `https://ray.readthedocs.io/en/latest/index.html` and Tune (Liaw et al., 2018) to perform an Asychronous HyperBand search (Li et al., 2018) with a maximum of 50 epochs. 200 configurations are tested, uniformly sampled from:

- node representation size (*hidden-size*): 32, 64 or 128
- number of encoding feed forward blocks: 0 to 3
- dropout rate (constant across the model): 0, 0.1, 0.2, 0.3, 0.4, 0.5
- number of message passing steps: 1 to 10
- number of decoding feed forward blocks: 0 to 3

In addition, for GGNN we randomly include a master node of size 20 to 200 by steps of 20 with a dropout rate of 0 to 0.5 by steps of 0.1.

For each model, the validation accuracy for each configuration is extracted and plotted against each value of a hyper-parameter. The plot overlays a box plot and a scatter plot for more insight. It is expected that configurations leading to overall lower performance would be stopped earlier, leading to a decreased average for a specific value of the hyper-parameter. Please note that this plot does not investigate potential interactions between hyper-parameters. In addition, we manually explore the 10 configurations leading to highest performance.

These 10 configurations for the GCN model include no encoding layer, a node representation size of 64 and a low but preferably non-null dropout rate (0: 1, 0.1: 4, 0.2: 5). The number of message passings is preferred as high, including between 7 and 10 convolutions. The number of decoding layers is also favoured as high (3: 7, 2: 1, 1: 2). Those results are illustrated in Figure 3. Based on this figure and the 10 best configurations, a sensible configuration for the GCN model would hence be: no encoding layer, hidden size of 64, dropout of 0.1, 7 layers of message passing and 1 decoding layer.

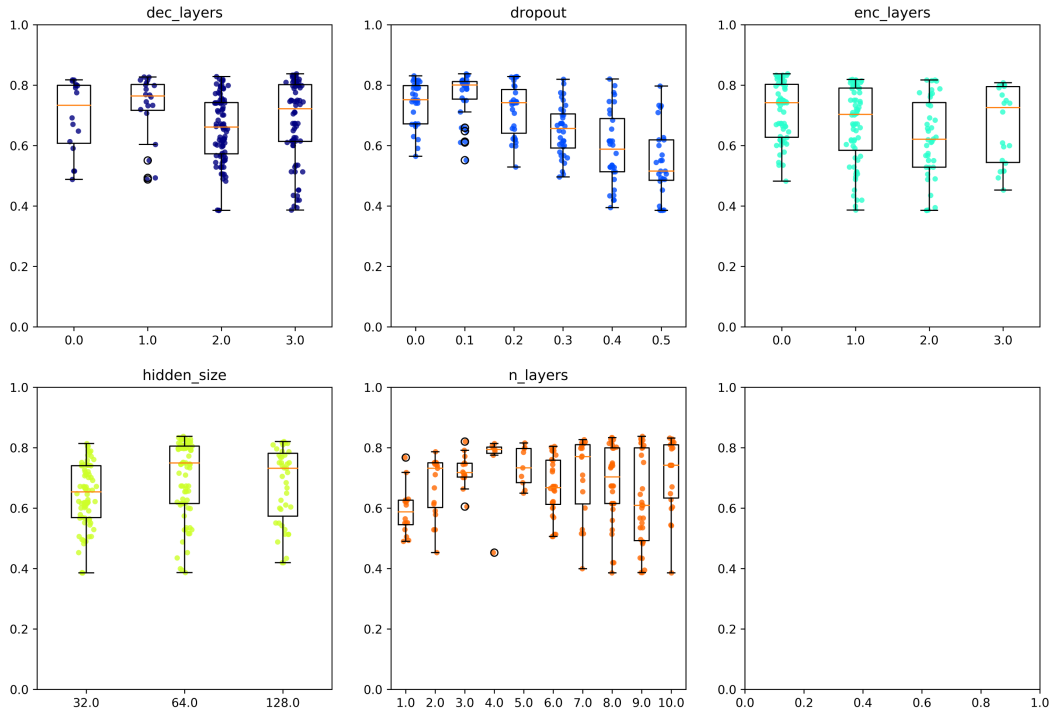


Figure 3: Hyper-parameter search for the GCN model. The space includes the number of encoding layers (enc_{layers}), the size of node representation ($hidden_{size}$), the dropout rate ($dropout$), the number of message passing layers (n_{layers}) and the number of decoding layers (dec_{layers}). This figure is based on 200 successful trials.

For the GGNN model, the 10 best configurations include a low number of encoding layers after the embedding (0 layers: 1, 1: 6, 2: 2, 3: 1) and prefer a higher node representation size (128: 9 out of 10, 64: 1) with low dropout (0: 7 out of 10, 0.1: 3). As displayed in

Figure 4, the number of message passing time steps is variable, with a reasonable choice being between 4 and 8 (8 out of 10). Surprisingly, the master node does not seem to improve classification accuracy (9 out of 10 configurations without master). This result suggests that adding long range connections does not improve on the graph representation. This might be due to the specific tree structure established from the AST that is relatively brittle (i.e. not all permutations can be performed on the graph and it does not make sense to connect all nodes). Alternatively, the model might become too large, overfit and needs more data. Finally, a low number of decoding layers is preferred (0: 4, 1: 4, 2: 2). The configuration chosen for the GGNN includes 1 encoding layer, a hidden size of 128, no dropout, 5 message passing layers without master node and 1 decoding layer.

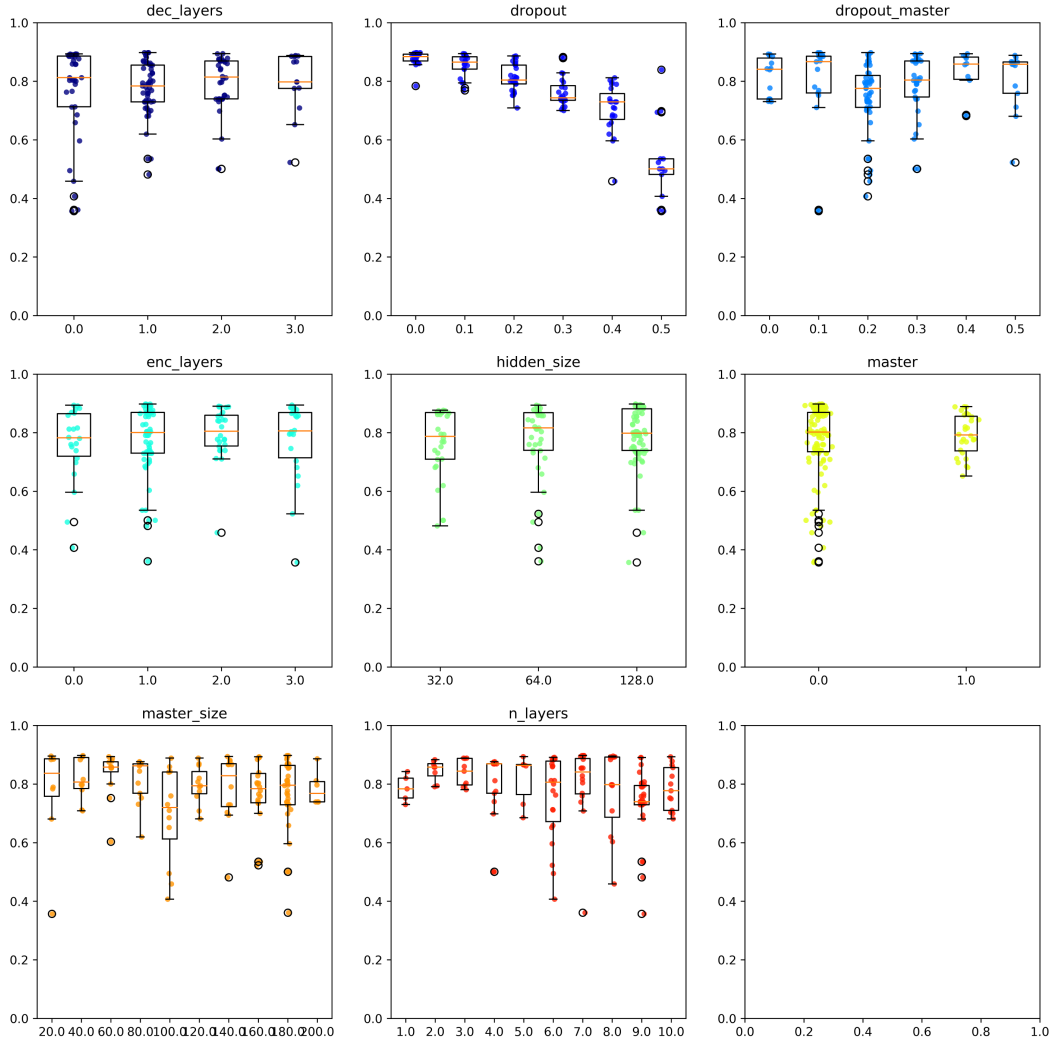


Figure 4: Hyper-parameter search for the GGNN model. The space includes the number of encoding layers (enc_{layers}), the size of node representation ($hidden_{size}$), the dropout rate ($dropout$), the number of message passing layers (n_{layers}), whether to include a master node ($master$, with $master_{size}$ and $dropout_{master}$) and the number of decoding layers (dec_{layers}). This figure is based on 138 successful trials (out of memory errors encountered).