# A Convolutional Neural Attention Approach to the Identifier Naming Problem in Program Code

Wout Elsinghorst

Master Thesis Computing Science
(ICA-3344819)

Supervisors:

dr. J. Hage dr. A. J. Feelders

Utrecht University
Department of Information and Computing Science

**Abstract**

Conditional random fields were previously used by Raychev, Vechev, and Krause, 2015 in a solution to the *VarNaming* problem (Allamanis, Barr, Bird, & Sutton, 2014), which is defined as the problem of assigning good, mutually consistent names to the various locally defined *variable* identifiers within a piece of source code. The conditional random field model assigns probabilities to each possible assignment of names to the various variable identifiers and then tries to find the assignment of maximum probability. By design, the implementation of the conditional random field model of Raychev et al., 2015 makes some locality assumptions during inference which might preclude it from obtaining a more global picture of the optimal names that should be assigned to each identifier.

In recent years, convolutional neural networks have become invaluable tools in the area of computer vision and image processing. Convolutional neural networks allow for state of the art prediction performance in the task of image classification by aggregating local information into a hierarchy of increasingly higher level features that eventually inform the final prediction. In this thesis, we propose (and experimentally verify) an application of convolutional neural network architectures to the domain of inferring good identifier names by interpreting a particular graph based representation of the source code as some sort of *generalized image*. From this perspective, source code identifiers are to be seen as pixels belonging to some higher dimensional picture, with the normal adjacency relations between pixels being replaced by various syntactic and semantic relations that are extracted from the source code. Interpreted this way, standard convolutional architectures become applicable to the source code domain, and the goal is to design a convolutional architecture that can complement the conditional random field model by extracting a set of distinctive -higher order- features that can be used to improve the initial predictions made by the conditional random field model.

Although the idea is seemingly straightforward, a concrete implementation of this idea raises many technical and conceptual questions. Topologically, these generalized images are a lot more complicated than their ordinary two dimensional counterparts, with the generalized variant possibly being as complicated as any general multigraph. The adjacency relation between the pixels is very sparse, by which we mean that almost all 'pixels' lie on the border, that, even in the conventional setting, makes them more difficult to handle. Typical concepts like $K$-by-$K$ receptive fields, stride, pooling, etc. need to be re-evaluated in this new environment. Another serious difference is the fact that in our generalized setting, pixel values are now categorical instead of numerical. At first glance this prevents us from applying any numerical kernel operations, like averaging over the pixel values in a neighborhood, but by adequately embedding the categorical labels into a finite dimensional vector space, the use of numerical kernels can be recovered.

This thesis is written to elaborate and verify the ideas sketched above. We define the problem and survey the most important previous works in the area. We then rigorously introduce the tools and concepts needed to understand the proposed framework. Alongside these specifics, a general introduction to the field of machine learning is also given, so that our work might also be understood by the uninitiated. We assume no previous machine learning experience. Once the stage is set, we will give a high level overview of the proposed architecture, elaborating on the individual components and on their various interactions. As a companion to the theoretical description of the proposed architecture, we also provide a technical description of a concrete framework that implements these ideas. Finally, the utility of this framework is demonstrated by a series of carefully crafted experiments.

# Contents

# Chapter 1

# Introduction

Writing code is easy, but reading code is hard. Understanding unfamiliar code is aided by recognizing common idioms and trusting identifier names in explaining intended meaning. Studies (Liblit, Begel, & Sweetser, 2006; Lawrie, Morrell, Feild, & Binkley, 2006; Binkley et al., 2013) have shown that good identifier naming is crucial here: the name of an identifier denotes its intended purpose, and if the names of identifiers match the structure of the surrounding code, then an experienced programmer will immediately recognize the function of the fragment without actively having to think about it, thus reducing mental strain on programmers unfamiliar with the codebase.

```
function findNode(tree, targetKey) {
  var currentNode = tree;

  while (currentNode != null) {
    if (targetKey == currentNode.key) {
       break;
    } else
    if (targetKey < currentNode.key) {
       currentNode = currentNode.leftChild
    } else
    if (targetKey > currentNode.key) {
       currentNode = currentNode.rightChild
    }
  }

  return currentNode;
}
```

Figure 1.1: Clear, beautiful code.

Let us take a look at the code snipped shown in figure 1.1. Even programmers unfamiliar with the JavaScript language will most likely infer the correct meaning of this code snippet due to the recognizable code structure combined with the informative names attached to the identifiers. More importantly, experienced programmers will probably do so almost instantly. Having seen the common pattern before, they subconsciously verify the correctness of the piece by recursively performing a top-down deconstruction of the fragment into its individual components while simultaneously verifying at each level that the individual components indeed combine as was advertised by their encompassing description. The snippet above can be contrasted to the variant shown in figure 1.2, which is semantically and operationally the same, but with undescriptive names being assigned to the various identifiers. To understand the meaning of this version, the programmer has to mentally execute the fragment by systematically analyzing the meaning of each atomic component and by then bottom-up reconstructing the meaning for the whole by combining the meaning found for the parts. For larger fragments this process quickly becomes infeasible for most programmers, resulting in them being unable to obtain a clear mental picture of

```
function a(b, c) {
  var d = b;

  while (d != null) {
    if (c == d.e) {
       break;
    } else
    if (c < d.e) {
       d = d.x
    } else
    if (c > d.e) {
       d = d.y
    }
  }

  return d;
}
```

Figure 1.2: Unreadable, unmaintainable code

the overall operation. It is clear that to obtain understandable and maintainable code, some thought must be given to the names assigned to the various identifiers.

Unfortunately giving good names to identifiers is not an easy task. Different languages have different naming conventions, complex data types have complex purposes and thus a wide variety of names will usually seem to qualify. There is no real objective way to select the best of these possible names, and picking the right name is more of an art than a science. Nonetheless, some names are picked more often than others, and choosing a name which matches other programmer's expectations as best as possible is important as they can then more easily match the code with other similar pieces of code which they might have encountered earlier, thus allowing them to leverage previous experience in the current setting.

It now becomes clear that having good tooling which helps the programmer make these naming decisions is paramount. With this thesis we explore the design of a *deep convolutional neural architecture* which helps the programmer with assigning good, mutually consistent names to the identifiers within a piece of code. The model will be informed by the names assigned to similar identifiers which were found in similar pieces of code, all obtained from a large corpus of pre-existing code bases. This allows the model to make suggestions which are in line with the names previously given by other programmers to identifiers which occurred in similar contexts.

## 1.1 Previous Work

We are certainly not the first researchers to tackle the problem of identifier name prediction. Previously, Allamanis, Barr, Bird, and Sutton, 2015, Raychev et al., 2015 and later Alon, Zilberstein, Levy, and Yahav, 2018 and Alon, Zilberstein, Levy, and Yahav, 2019 have pursued this problem in one form or another. For our work here we will take the precise problem attacked in (Raychev et al., 2015) as our problem definition, which is the problem of assigning good names to any *locally declared* variables in the program (the `VarNaming` problem, see section 1.3). In their paper, Raychev et al., 2015 show how conditional random fields can be used to obtain state-of-the-art results in this specific problem domain.

For our work, we started with using this conditional random field model to explore how much room we have for improvement. When using their model, configured with the optimal hyperparameters that were recorded in their paper, we achieved a `TOP-1` validation accuracy of around 69 percent on our dataset. When looking at their `TOP-2` and `TOP-3` validation accuracies, however, we found that their model attains accuracies of around respectively 73 and 74 percent. It seems that in around 5 percent of the cases, the conditional random field makes a wrong prediction while the correct answer was actually within reach, with the correct answer belonging to the three candidates which the model deemed the most likely to be correct. We've written down these findings in table 1.1. Our main goal for this thesis is to construct an auxiliary model which can improve the ranking of these `TOP-3` (or actually `TOP-K` for any choice of $K \in \mathbb{N}$) candidate predictions by utilizing a (convolutional)

neural network architecture that is able to use both local and global source code-derived information to make improvements to the already given conditional random field predictions. We will note here that the existing conditional random field model is explicitly *not* a neural network model; we hope that the utilization of a neural network will offer a complementary view on the situation that allows for overall better informed predictions.

| TOP-1 | TOP-2 | TOP-3 |
|---|---|---|
| 69.00 | 72.78 | 73.87 |

Table 1.1: Validation accuracies for the baseline model.

## 1.2 Convolutional Neural Networks

The main idea behind our line of research is to interpret a particular graph-based representation of the source code as a *generalized image*, where the nodes in the (multi)graph are to be interpreted as pixels in a higher-dimensional image and where the edges between the nodes correspond to generalized adjacency relations between these pixels. The set of values from which the pixels now take their values is no longer a set of ordered pixel intensities (e.g. red-green-blue values in either the range 0-255 or in the 0.0-1.0 range) but a discrete, unordered set of all allowed identifier names.

Ordinary image-based convolution is based on using local averaging operators (so called *kernels*) which calculate for each pixel position some quantity which represents how strongly some particular feature associated to the respective kernel is present at that specific position. A kernel takes as input a neighborhood of pixels centered at a specific location (typically a 3-by-3 or a 5-by-5 grid) and then calculates a weighted sum of the values of the pixels in that neighborhood. The exact combination of weights given to the relative positions around the center point determines how strongly the kernel responds (i.e. the magnitude of the resulting sum) to a particular configuration of pixels, and thus how strongly the kernel thinks the corresponding feature is present. Each kernel is evaluated at each pixel position, and by combining a broad ensemble of kernels we obtain varied characterisations of the contents of the neighborhoods around each location. Individual kernels can be designed to respond to the presence of sharp edges, lines, color combinations, etc., and by feeding the output of a layer of kernels as input to a subsequent layer of (different) kernels, we can detect the presence of features of increasingly higher complexity. If the first layer detects the presence of particularly oriented line segments, the second layer might be able to detect whether the right combination of line segments is present to constitute the presence a single finger, and the kernels in the third layer might try to respond strongly to the presence of precisely five fingers to detect a complete hand. The purpose of deep convolutional networks in the context of image processing is to detect a hierarchy of features, of increasingly higher level, which ultimately serve as good indicators to make a confident final prediction.

In the case of image processing, the prediction made can be the assignment of a label to the picture which classifies its contents (e.g. 'dog', 'cat' or 'human'), it can be a numerical value (e.g. the number of people in the picture), or the prediction can itself even be an image (e.g. the prediction of the face of a celebrity superimposed on a picture of yourself making funny facial expressions, with the imposed celebrity having matching facial expressions). With the perspective of interpreting a graph-based representation of a piece of source code as a generalized image, we would like to implement a convolutional architecture which takes as input a graph-represented version of the source code (i.e. the picture), stripped of a subset of the identifier *names* (but with all other names and relations intact), and which outputs a reconstructed version of the graph where all missing identifier names have been given names which make sense given the context in which they were used. The idea here is that the stacked convolutional layers will be able to obtain a higher level, global, overview of the program, which, together with the local first-order information from the direct neighbors, will be sufficient to make accurate predictions.

## 1.3 Problem Statement

In the previous sections we've argued that predicting good identifier names is very important. As we mentioned before, for our purposes we will limit ourselves to tackle the more restricted `VarNaming` problem (Allamanis et al., 2014; Raychev et al., 2015) for whole programs, where the objective is to assign good, semantically rich

names to precisely the identifiers identifying *locally declared variables*. Moreover, we will limit ourselves to the local (i.e. non-global) variables. The names of globally defined variables, object properties, contants, etc. will not be predicted, but will serve as extra information for the model to base its predictions on.

The limitation to (local) variable identifiers is mainly one of focus: in principle we could allow the prediction of method names, class names, etc. to occur jointly with the variable names, but this would probably entail extracting additional features specific to these situations, which is not something which we want to concern ourselves with here. Also, holding these non-variable identifiers static while predicting names for the variable identifiers provides context to the predictions; knowing the *named* methods which are invoked on some reference to an object greatly informs the right names which can be assigned to the reference itself. Jointly predicting both method and variable names is thus a much harder task, giving less information to the prediction engine and having a much larger combination of names to optimize for. Nonetheless, we expect most techniques described here to also be applicable to tasks where an extended variety of identifiers is predicted, given that the size of the training set is sufficiently extended to compensate for the increase in problem complexity.

The limitation to only focus on identifiers which were declared in local scope is mostly just a technical one: these identifiers can be freely renamed without having to take non-local parts of the program into consideration. The extent to which these variables can influence the program is strictly localized, assuring that the renamings will never change program semantics. Globally declared variables on the other hand can be shared between modules, forcing the renaming engine to take all concerning modules into account simultaneously to obtain a consistent renaming which preserves the original program semantics. Our framework (and those described in previous works) will be designed to work on a per-source file basis, thus requiring it to be able to treat each file independently. Of course, given some efforts, most real life projects can easily (and mostly mechanically) be brought into some self-contained local scope, for example by concatenation of the individual source files and lifting everything to occur inside some encapsulating container function.

From the previous it follows that the various identifiers within a program can be separated into two distinct categories: those identifiers which we wish to predict (i.e. local variable names), and those identifiers which we use as context to base the predictions on (global variable names, class names, method names, etc). To make the roles of these identifiers more explicit, we will now introduce some notational conventions which will be respected throughout this document. All variables introduced here are to be taken with respect to a specific program $P$ which should be clear from the context. The category of **unknown identifiers** in $P$ will be enumerated $Y_1$, $Y_2$, …, $Y_u$. These variables are to be predicted and won't have any meaningful name assigned to them at the start of the prediction task. The category of **known identifiers**, which consists of all identifiers which were not in the previous category, will be enumerated $X_1$, $X_2$, …, $X_v$. These identifiers are always static and only serve as extra information to make more informed predictions. Finally, we need access to the combined category of **all identifiers**, which will be enumerated $Z_1$, $Z_2$, …, $Z_{(u+v)}$. Individual identifiers will normally be named by a single letter corresponding to the most specific category it belongs to, e.g. $Y$ will indicate some identifier whose name we wish to predict, $X$ will indicate some contextual identifier, etc.

Using the notation just introduced, we can now state our goal as follows: we wish to obtain a method of finding labels $y_1$, $y_2$, …, $y_u$ such that label $y_i$ is a good name for the unknown identifier $Y_i$ in the program $P$, given the context of the labels $x_1$, $x_2$, …, $x_v$ already assigned to the known identifiers $X_1$, $X_2$, …, $X_v$.

## 1.4 Abstract Representation

Following Raychev et al., 2015; Alon et al., 2018, we represent our programs in the form of *undirected* knowledge (multi-)graphs. Figure 1.3 shows a possible representation constructed for the program shown in figure 1.1. This graph is an abstracted form of the original program which only contains information relevant to our problem of predicting names for the identifiers occurring within the program. Here 'relevant' is subjective and determined by the model designer; see the next paragraph for a discussion on obtaining the exact feature set. White nodes indicate the identifiers to which we want to assign good names (i.e. the *unknown identifiers*), while blue nodes indicate static context which is used to inform the predictions made for the white nodes (i.e. the *known identifiers*). The labels on the edges between the different nodes indicate how the respective nodes are related to each other within the program. For example, the *hasProperty* edge between the nodes named *currentNode* and *leftChild* indicates that the *leftChild* property was syntactically accessed on the variable currently identified with the name *currentNode*.

In general, the relations between entities can be both syntactic and non-syntactic. The relations in the graph shown here are all syntactic, being derived directly from the abstract syntax tree without requiring

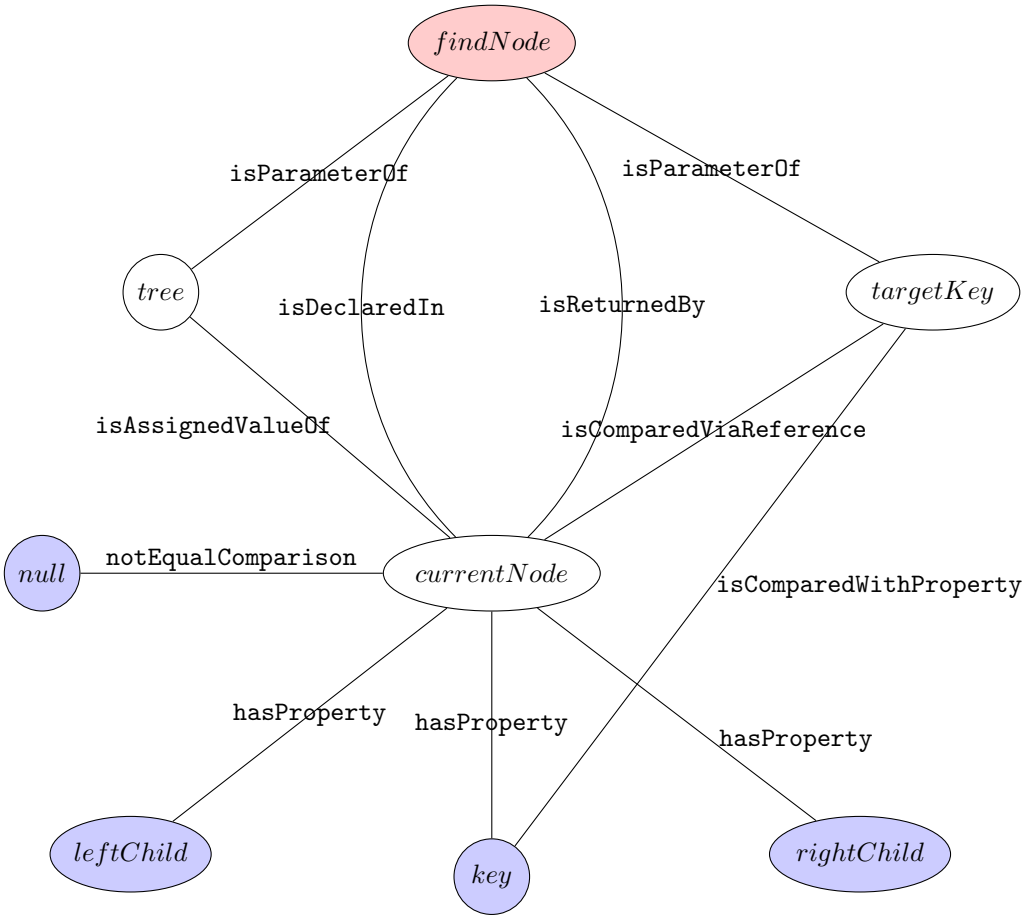


Figure 1.3: Abstract representation of the `findNode` program. Some edges have been removed or simplified for presentation purposes.

any additional analysis. Non-syntactic relations (e.g. pointer aliasing) can also be derived, for example by performing static analysis prior to the construction of the knowledge graph. Non-syntactic relations usually convey information about the actual *meaning* of the program, which can make them quite useful in determining good names for the involved identifiers.

Knowledge graphs like the one shown here can be constructed for programs written in any language, but for our work we restrict ourselves to programs written in JavaScript. The exact information available within the constructed knowledge graphs obviously depends on choices made during the construction of the graph. In principle, these knowledge graphs can be made mostly language agnostic when using only syntactic features which are common to most programming languages. In theory this allows prediction models to be used across language boundaries, but in practice it pays off to use language specific features for better information representation. We refer to section 4.1 for more information on knowledge graphs and about how to obtain them.

## 1.5 Conditional Random Field Foundations

The original inspiration for this thesis is the conditional random field model described by Raychev et al., 2015. This model is based on using the information available in the undirected knowledge graphs described in the previous section to derive a probabilistic model. Here the nodes in the graph are interpreted as (probabilistic) random variables, and the graph structure is used to make explicit the dependencies (and more importantly the *independencies*) between the individual variables. Each random variable takes on values from the set of labels which are allowed for that particular node (i.e. the set of acceptable variable names for some variable), and the goal of the model is to find an assignment of names $y_i$ to the nodes $Y_i$ such that the **joint conditional probability**

$$\Pr\left(Y_1 = y_1, Y_2 = y_2, \ldots, Y_u = y_u \mid G^P, X_1 = x_1, X_2 = x_2, \ldots, X_v = x_v\right) \tag{1.1}$$

of that assignment (also conditioned on the graph structure) according to the model is maximum among all possible label assignments. The graph structure is used to simplify the model, by requiring the probabilities of each of the possible labels for a single random variable $Y$ to be fixed as soon as the labels for the direct neighbors are known. The combination of a set of random variables $Z_i$ together with a graph structure $G$ on these variables that respects the (in)dependencies between the variables is called a **Markov random field**. A **conditional random field** (Lafferty, McCallum, & Pereira, 2001) is a generalization of this which allows the graph structure itself to be dependent on the data, which is in our case the program $P$. We plan on using the predictions made by this model as a baseline which is used by our model to bootstrap its own predictions. To be able to better explain the motivations behind our own model, we will now also briefly describe how the conditional random field model by Raychev et al., 2015 is obtained from the training data and how it can subsequently be used to make predictions for unseen data. We refer to section 2.5.3 for a more theoretical perspective on conditional random fields.

### 1.5.1 Construction

The construction of the model starts by constructing a knowledge graph $G^P$ for each of the programs $P$ in the training set (see the previous section). The labeled nodes in a knowledge graph $G^P$ are connected to each other via (possibly a multitude of) labeled edges. The nodes in the training set each already have a single known-to-be-correct reference label assigned to them. Each connection between two nodes $Z_i$ and $Z_j$ via an edge labeled with the label $r$ induces a triple of information $(z_i, r, z_j)$ for the concrete assignments of the label $z_i$ to node $Z_i$ and the label $z_j$ to node $Z_j$. For each program $P$, we obtain a bag (multiset) of these triples, denoted $\texttt{triples}(P)$, that were induced by the edges in the graph $G^P$. When we collect all triples induced by the programs $P$ in the training set $\mathcal{D}$ together and throw away their bag structure, we obtain a flat set of triples, denoted $\texttt{features}(\mathcal{D})$, that will serve as the base features of the model. Each triple $f \in \texttt{features}(\mathcal{D})$ is assigned a score $\omega_f \in \mathbb{R}_+$ that somehow measures the importance of the occurrence of this triple in some program $P$. The bag $\texttt{triples}(P)$ contains precisely the triples that were found in $G^P$. By summing over the scores $\omega_f$ assigned to the features $f \in \texttt{triples}(P)$ found in $G^P$ (counting with multiplicity), we obtain a total score for the current label assignment to the nodes of $G^P$. Of course, each concrete assignment of labels to the *unknown* nodes $Y$ of $G^P$ leads to a different bag of triples, which in turn leads to a different calculated score. In the full conditional random field model, the total score assigned to the bag of triples $\texttt{triples}(P)$ for a

particular assignment of labels to the nodes of $G^P$ directly translates to the probability given to this assignment by the model; the higher the score, the higher the calculated probability. In the end, the goal of their use of the model is to find an assignment of labels to the *unknown* nodes of maximum probability (equivalently: score), while leaving the *known* nodes in their original state. Because higher scores directly lead to higher probabilities, it suffices to find the label assignment which maximizes the total score; it is never necessary to calculate the actual probability while searching for the assignment that maximizes this probability.

### 1.5.2 Training

Of course, the scores $\omega_f$ assigned to the individual triples $f \in \texttt{features}(\mathcal{D})$ still need to be determined. The goal of the training phase is to assign these scores in such a way that the actual labelings of programs $P \in \mathcal{D}$ induce triple bags $\texttt{triples}(P)$ which are scored higher according to the model than the triple bags induced by any of the other alternative, *false*, labelings (false with respect to the truth found in the training set). Of course, adjusting the scores of some triples such that the score of one program is improved might result in the scores of many other programs to become worse. From the probabilistic perspective, a principled approach would be to assign the scores in such a way that the likelihood of the data is maximized. Unfortunately, calculating the likelihood, let alone maximizing it, is not actually feasible. The implementation given by Raychev et al., 2015 finds a good global assignment for these scores (but which probably won't maximize the likelihood) by performing a global optimization step through the use of **Structured Support Vector Machines**, which globally minimizes the *structured hinge-loss* (see section 2.7.4) of the predictions made by the model with respect to the actual assignments found in the training set. The structured hinge-loss is an upper bound on the 0-1 classification loss, and thus minimizing the structured hinge loss on the training set also guarantees a bounded prediction loss on the training set, which will hopefully translate to a good overall prediction accuracy, even on unseen data.

### 1.5.3 Inference

The calculated probability $\Pr(Y_1 = y_1, Y_2 = y_2, \ldots, Y_u = y_u \mid G^P, X_1 = x_1, X_2 = x_2, \ldots, X_v = x_v)$ by the model for a concrete assignment of labels $Y_i = y_i$ to program points $Y_i$ in an *unseen* program $P$ is then directly related to the sum of the scores assigned to each of the triples $(z_i, r, z_j)$ extracted form $G^P$. Changing the assigned label of a single node $Y$ from $y^1$ to a different label $y^2$ thus replaces all attached triples $(y^1, r, z_j)$ with different triples $(y^2, r, z_j)$, for each of the nodes $Z_j$ that are directly connected to $Y$. Thus, the changing of a single label at a particular node $Y$ replaces all triples attached to this node. These new triples have a new set of scores assigned to them, and the total sum of these scores might be lower or higher than the sum of the triple scores corresponding to the previous assigned label. The goal of inference now is to find the assignment of labels $y$ to the unknown nodes $Y$ of $G^P$ that *jointly* maximizes the score obtained from the induced bag of triples.

In their work, Raychev et al., 2015 use the Markov structure for a local *greedy* maximization algorithm, which starts with a random label assignment to the unknown nodes $Y$ and then iteratively tries to improve each the label assignment to a single node *while keeping the direct neighbors constant*, until no more improvements are possible. With the neighbors held constant, each particular label assignment $y$ to $Y$ results in a concrete scored bag of triples. Finding the best label for $Y$ under this constant neighbors assumption can be done very quickly, and each improvement made to $Y$ directly results in an overall increase in the score calculated for the whole program $P$. Thus, randomly making improvements to random nodes in the graph results in a monotonically increasing score for the overall label assignment. This random improvement of nodes can be continued until either no more improvements can be found or until a limit on the number of steps is reached.

Unfortunately, although is procedure can find a good assignment relatively quickly, this assignment is found in a *greedy* way and as such it is by no means guaranteed to be optimal, with the optimization procedure running the risk of getting stuck in local maxima which are not globally optimal. We will illustrate this situation by means of a simple example. Figure 1.2 shows a simple scoring table for the label assignment to two nodes $Y_1$ and $Y_2$. Obviously, without knowing anything about any other nodes that may or may not be connected to either of these nodes, the optimal assignment in this situation would be $Y_1 = c$ and $Y_2 = c$ with given score 97. However, starting from the assignments $Y_1 = a$ and $Y_2 = a$, this solution will never be found. This initial assignment is given the score 13. When we start by fixing $Y_2$ and looking for the best assignment to $Y_1$, we immediately arrive at the assignment $Y_1 = b$, which would result in a score of 19 for the new situation. When we

now hold $Y_1$ constant and vary $Y_2$, we now find that the assignment $Y_2 = b$ is optimal with a score of 23. We've now reached an equilibrium. When we again swap the roles of $Y_1$ and $Y_2$, the best assignment for $Y_1$ is still its current assignment of $Y_1 = b$. The jump to the global optimum will not be made, because setting $Y_1 = c$ will first significantly lower the score of this assignment to 11. Thus, the inference algorithm is stuck at a solution which is locally optimal but not globally so. Of course, in this simple situation without any other relevant nodes, the optimal solution could be read directly from the table, but when multiple nodes are connected to the same node, it's much harder to see the overall effect of a simple change.

| label $Y_1$ | relation | label $Y_2$ | score |
|---|---|---|---|
| a | r | a | 13 |
| a | r | b | 19 |
| a | r | c | 17 |
| b | r | a | 19 |
| b | r | b | 23 |
| b | r | c | 11 |
| c | r | a | 17 |
| c | r | b | 11 |
| c | r | c | 97 |

Table 1.2: Scoring table for two nodes $Y_1$ and $Y_2$.

## 1.6 Research Questions

The discussion in the previous sections lead us to the following three concrete research questions that we would like to answer with this thesis.

**Can we (significantly) improve upon the conditional random field model by developing a complementary convolutional neural network for increased information utilization?**

In the previous section we gave a short introduction to the conditional random fields used by Raychev et al., 2015 in their solution to the `VarNaming` problem. Our goal in this thesis is to develop an auxiliary model to this baseline conditional random field that uses a convolutional architecture to refine and improve upon the predictions made by the initial model. In the first section we saw that there is quite a large gap of around 4 percent between the `TOP-1` and the `TOP-2` and `TOP-3` accuracies of the baseline model. This gap means that in a about 4 percent of the cases, the conditional random field model makes an incorrect prediction while it already had the actual correct answer within eye sight. We hope that our complementary convolutional neural network can offer a different perspective on the situation that can help move some of these 4 percent wrongly predicted labels to the `TOP-1` position.

**Can we get our model to synergize with the original model, i.e. can the combination of two models overcome each other's weaknesses for a better overall result?**

In the previous section we also explained the inference algorithm used by the conditional random field model. This inference algorithm is *greedy* and as such it is known to get stuck in local optima which are not globally optimal. Our hypothesis is that at least some of these wrongly predicted labels are caused by the inference algorithm getting stuck in a local optimum that precluded it from making the jump the correct label, and as such it would be interesting to see how the conditional random field values (in calculated score) the new label assignment predicted by our own convolutional model. When the score assigned to our refined prediction by the conditional random field is higher than the score assigned to its own initial prediction, we know we've found a new situation that is better than the previous local optimum. When we allow the conditional random field to continue its inference algorithm starting from this new situation, it will arrive at a new, better local optimum (that is probably still not globally optimal). When iterate this, we might get a synergistic effect were each model can help the other break through its current local optimum.

**Can we do all this while maintaining the real-time inference property of the original conditional random field framework?**

For some practical purposes (e.g. name inference in an Integrated Development Environment) it is very important that the predictions made by the fully trained (combination of) our models on unseen data can be done approximately in real time, i.e. the final labeling of all identifiers within a given piece of code is done within a few seconds at most.

## 1.7 Document Overview

The rest of this document is structured as follows. We will start the next chapter ('Preliminaries') by first giving some general background information on the various general techniques, methodologies and terminologies which we will be using during the rest of the document. In chapter 3 ('Previous Work') we will review the papers which most directly influenced our current research direction. Chapter 4 ('Approach') then gives a detailed exposition of the various ideas and technologies that together constitute our architecture. The subsequent chapter, chapter 5 ('Architecture') formalizes our proposed architecture in terms of an abstract implementation that fixes the general flow of information within the model. We continue in chapter 6 ('Experimental Verification') with the main course: we discuss the execution and interpretation of a number of experiments that we created with our own concrete, fully fleshed out Python/Tensorflow implementation of this previously still abstract architecture. Chapter 8 ('Conclusion') finally returns to the research questions that were posed in the previous section. Here we give a short summary of the things that we've achieved in this thesis, and we also discus some possibilities for future research. We end this document with a short discussion on some related work in chapter 7 ('Related Work').

# Chapter 2

# Preliminaries

This chapter provides some context to some of the principles behind machine learning and general experimental (computer) science.

## 2.1 What We Are Learning

### 2.1.1 Supervised vs. Unsupervised Learning

In supervised learning we work with labeled data. The goal of the learning process is to create a model from a pre-labeled training set that can be used to accurately predict the labels for data where the correct labels are not yet known. For unsupervised learning we don't have any information on what is right or wrong, but we are interested in organizing and modeling our data, such that we can understand it better or such that it is more easily digestible by machinery further down the pipeline. Some examples of unsupervised learning are data clustering where we need to learn how to group data into clusters for which the points within each cluster share some characteristics, or learning a probabilistic model over the data from which we might sample new data. For our project we'll be performing supervised learning, where the task is to learn a model that can predict labels (identifier names) for program points from a dataset where the expected output is already supplied.

### 2.1.2 Discriminative vs. Generative Models

Similar to the distinction between supervised and unsupervised learning, with generative modeling we create a model for the complete data set, both the dependent and the independent variables. With discriminative learning we only create a model for dependent variables, which is conditioned and can depend on the independent variables. When we are only interested in making predictions given some observations, it's usually wasteful to also model the direct observations, so discriminative learning is usually preferred in this case.

For us the learning task is discriminative: both the existing conditional random field model as well as our own model gives us a probability distribution of the labels *conditioned* on the structure of the surrounding program context. We explicitly do not model and are not interested in the distribution of this contextual information; we are not interested in generating programs, only in generating names for specific locations within otherwise static programs.

### 2.1.3 Overfitting vs. Underfitting

We want to make good predictions for unseen data. When overfitting, we have mostly memorized the training set and are only able to make good predictions for data which was already seen while training. When underfitting, we haven't found the true patterns underlying the data, thus making poor predictions. Finding the right balance in model complexity so that the resulting model both generalizes to unseen data and also doesn't do so oversimplistically is crucial.

### 2.1.4 Parameteric vs. non-Parametric Models

In the specific sense, **parametric models** are families of probability distributions indexed by some finite number of real parameters. The model consists of the entire family of distributions, and for each parameter configuration we obtain a specific distribution within the model. For example, for a fixed dimensional $n$, the family of normal distributions on $\mathbb{R}^n$ is a parametric model that is indexed by a vector $\vec{\mu}$ of $n$ means and a covariance matrix $\Sigma$ with $\frac{1}{2} \cdot n(n+1)$ (co-)variances. For each value of $n$ there exists a different model of normal distributions on $\mathbb{R}^n$, and each specific configuration of $\vec{\mu}$ and $\Sigma$ corresponds to precisely one distribution. More generally, parametric models are **hypothesis classes** that are parameterized by a finite number of real parameters. Each hypothesis within such a class is a concrete relation between the relevant dependent and the independent variables. For example, the set of polynomials of a given degree $k$ over $\mathbb{R}$ in $n$ variables forms a model in this broader sense, with each polynomial (hypothesis) within this model being parameterized by $\binom{k+n}{k}$ coefficients. Each concrete choice of coefficients results in a specific hypothesis on the relation between the independent variables $\vec{x} \in \mathbb{R}^n$ the dependent variable $y \in \mathbb{R}$. We will usually refer to the precise hypothesis obtained by optimizing the parameters within a specific model as the **fitted model**.

The goal of parametric learning is both selecting the right model family out of the various possible sensible families and of finding the specific set of parameters that results in a concrete hypothesis that best reflects the actual data. Having a fixed number of parameters ensures that the hypotheses within a specific model cannot become arbitrarily well adjusted to the training set. For *non-parametric models*, there is no such limitation on the nature of the parameterization of the model; models can become arbitrarily complex, and without adequate regularization they are very prone to overfit. Examples of non-parameteric models are *decision trees*, *random forests* and the $k$-nearest-neighbors classifiers. In principle, these models can just memorize the complete training set, allowing their decision procedures to be as complex as the data on which is was trained.

All models considered by us will be parametric. The construction of a conditional random field extracts a finite set of features from the training set, and these features together with their assigned scalar weights fully determine the model, which thus has a finite, *fixed* set of parameters. Similarly, each graph structure underlying a neural network (up to choice of transfer functions, etc.) fully determines a model that is completely parameterized by the neural network weights, each of which are in direct correspondence with the neurons in the network.

### 2.1.5 Parameters vs. Hyperparameters

Hyperparameters determine the actual model (for example the *degree* $k$ of a family of polynomials, or the dimension $n$ of the normal distribution) while parameters determine the specific configuration of the model (for example the precise coefficients of some polynomial of fixed degree $k$, or the means and covariances for the normal distribution on $\mathbb{R}^n$). More generally, hyperparameters can also be parameters of the fitting process, which indirectly, through the precise execution of the fitting algorithm, determine the final output hypothesis. For each configuration of hyperparameters we usually fit the corresponding model to the data as best as we can. Then we test the performance of the various models corresponding to the various hyperparameters on some fresh set of data which was not used for fitting, and then select the hyperparameter combination whose resulting fitted model performs best on this fresh set.

### 2.1.6 Distributions vs. Classifiers

The prediction results of the models discussed in this paper (both conditional random fields and our neural network) all come in the form of probability distributions on the total set of identifier names that was found in the training set. For each identifier name prediction, the generated probability distribution indicates for each of the possible labels the confidence that the model has in that particular label being the correct one. When we use this probability distribution to order the candidate labels with respect to the confidences given to them by the model, we obtain a ranking of labels, with the top ranked label seen to be the most probable choice by the model. When we return this top candidate as the final prediction result by throwing away the other candidates, we obtain a *classifier* which just assigns a label class to each point of prediction without remembering anything on how it got there. In general, we will not only be interested in this top candidate, but also in the candidates that were ranked on the second and third positions.

## 2.2 Performance Metrics

During the training phase we adjust the parameters of our models as to make predictions that match our expectations as well as possible. Our training set contains lots of source code data together with the expected labels (names) for the various *unknown* identifiers, and our goal is to find and tune a model that can accurately predict those labels given only the raw source code where the labels are still missing. We use the training set to find and tune the model, but the goal is to perform well on new, unseen, *fresh* data. That is, we want to *generalize*. There are various ways in which we can directly and objectively measure how our model performs. We compare the identifier name predictions made by our model to the reference names assigned to the same identifiers on the *gold standard* data set obtained from real world source code (see section 6.5). How useful these *objective* measurements are, is however, a *subjective* matter; different requirements on the quality of the predictions leads to different ways of assessing performance. Whether a measured difference in any of the following performance measures between two models is actually significant, is also something which needs to be verified separately; see section 2.4.

### 2.2.1 `TOP-K` Accuracy

The `TOP-K` prediction accuracy of a classifier is defined as the proportion of cases where the actual true label that was found on the gold standard is among the $K$ highest ranked candidates by the classifier. For each of these accuracy classes, either the actual truth label is within the `TOP-K`, or it is not. In the first case, the prediction is scored 1 point, and in the other case it is scored 0 points. The total sum of these scores is taken over all data points in the dataset, and is subsequently divided by the total number of data points to obtain the final `TOP-K` accuracy.

### 2.2.2 Precision, Recall and $F_1$

Precision measures the proportion of true positives in the set of all positively matched results. Recall measures the proportion of true positives in the set of all actual positives. The $F_1$ score is the harmonic mean of precision and recall, which assures that there is a good balance between precision and recall when both are valued equally.

With our prediction problem we are dealing with complicated predictions (descriptive variable names) which are themselves composed of smaller subpredictions (subtokens). For each unknown identifier $Y$ within some labeled program $P$ within the training set, we have a set $Y_{expected}$ of subtokens that occur in the given gold standard label. We also have a set $Y_{predicted}$ of subtokens of the final output predicted by the model. We can measure precision and recall with respect to these two sets: precision is calculated as the proportion of subtokens predicted that also occur in the expected set, while recall is the proportion of expected subtokens that were actually predicted.

For example, when the expected name is *getEmployee* and the predicted name is *getCustomerName*, we obtain the sets $Y_{expected} = \{get, employee\}$ and $Y_{predicted} = \{get, customer, name\}$. From the predicted set, only the token *get* is correctly predicted, which is a third of the whole set, so the precision is 0.33. From the expected set, we've predicted the token *get* correctly, which is half of that set, so our recall is 0.50. Calculating the harmonic mean of these two gives us a $F_1$ score of 0.41. It would seem here that the $F_1$ measure gives a better picture of this situation than the plain accuracy score discussed above, which would have rated this prediction with a score of 0.

Each program point contributes values for these measures that lie between 1 (completely correct) and 0 (at least partially incorrect), which are then averaged over all prediction points. The average precision, recall and $F_1$ scores are thus all upper bounds on the overall accuracy, coinciding only in the case of a perfect prediction of the whole test set.

## 2.3 Model Validation

Usually the model optimalization process depends on some hyperparameters, with each hyperparameter resulting in a different model. From all possible parameters of the model corresponding to a single hyperparameter configuration, we select the optimal configuration (see section 2.6) for that model. Each set of hyperparameters will result in a different tuned model, and we somehow need to evaluate the fitted models and see how they compare.

### 2.3.1 Evaluation

We reserve part of our data set for a final evaluation of our model. The reserved part is explicitly *not* used in any way while fitting the models or while determining the optimal values for their respective hyperparameters. The final score reported for our work will be based on the performance of our model on this **evaluation set**.

### 2.3.2 Training

The rest of the data (i.e. the data outside the evaluation set) is used to fit and compare various models corresponding to the different hyperparameters. From these hyperparameters, we would like to select the hyperparameters that result in the best fitted model. Because models have the tendency to overfit (which means they are too much adjusted to the training set while performing poorly on unseen data), we cannot, in good conscience, evaluate a fitted model (i.e. the quality of the hyperparameters) on the same data we used to train the model; we need some additional unseen data. We further divide the non-evaluation data into a **training set** and a **validation set** on which the models corresponding to particular hyperparameter configurations are respectively fitted and compared on performance. Reserving a part of the training set for validation is wasteful because the information in this set cannot be used to inform the model. Normally, when working with datasets that are not too unwieldy, we would use **cross validation** to estimate the optimal hyperparameters in a way that would still allow us to use the whole training set for model fitting. Unfortunately, this technique requires too many extra auxiliary models to be fitted to be feasible with our available computing capabilities.

## 2.4 Significance Testing

By interpreting our problem as a binomial classification problem where the predictions for each program point can be either correct or incorrect, we can compare our model $M$ to a different model $N$ by considering the set of all program points in our evaluation set (taken over all source files) where either $M$ or $N$, **but not both**, make a correct prediction. This is the set of data points where the predictions made by both classifiers differ in correctness, and our goal here is to find out whether this set is *significantly* skewed in favor or against our own model $M$. Under the (false) assumption that all predictions were made independently (we actually perform *structured prediction*, so the predictions done within a certain source file are most certainly not independent), we can model this question of significance by a sequence of Bernoulli trials, with the 'success' event meaning our model $M$ was correct and the other model $N$ was incorrect (and with the 'failure' event being the opposite). Under the null hypothesis that both events are equally likely (i.e. both models perform equally well), we can calculate the probability of having obtained our Bernoulli sequence, or any more extreme sequence, using a binomial distribution, and if this probability is sufficiently small ($< 0.005$) we deem our results to be significant.

## 2.5 Parametric Models

A parametric model was previously defined as a family of hypotheses indexed by some fixed dimensional vector of parameters. This section describes a few concrete models within this class that are relevant for our architecture.

### 2.5.1 Logistic Regression

The logistic model is a discriminative probabilistic model used to relate a binary dependent variable $Y \in \{y_1, y_2\}$ to a vector of independent continuous variables $\vec{X} \in \mathbb{R}^n$. The model starts from the assumption that the space of measurements $\mathbb{R}^m$ where the independent variables take their values can be linearly separated into two disjoint parts by a hyperplane. The two parts are respectively labeled $y_1$ and $y_2$, and measurements $\vec{X}$ are assigned labels by the model corresponding the side of the hyperplane where they are located (measurements that lie on the hyperplane itself are assigned the label $y_2$ by convention). The hyperplane itself is determined by a vector $\vec{\omega} = (\omega_1, \omega_2, \ldots, \omega_n)$ that is normal to the plane. When the normal vector is normalized, $||\vec{\omega}||_2 = 1$, the signed distance from the plane to the location of the measurement can be calculated by taking the dot product between the normal vector $\vec{\omega}$ and the vector of measurements $\vec{X}$:

$$H = \left\langle \vec{X}, \vec{\omega} \right\rangle \tag{2.1}$$

When this value is positive the measurement is location on one side of the hyperplane, when it is negative it is located on the other side, and when this value is 0 the measurement actually lies on the hyperplane. We obtain the following decision procedure to decide which label to assign to $\vec{X}$:

$$Y = \begin{cases} y_1 & \text{if } H > 0 \\ y_2 & \text{otherwise} \end{cases} \tag{2.2}$$

Of course, in practice, we can never completely measure all factors that determine the right label to assign to a situation; there are always small influences that add some unpredictability to the final label assignment. In logistic regression, this unpredictability is modeled by adding an *unobserved* error term $\epsilon$ that randomly changes the distance from the measurement $\vec{X}$ to the hyperplane:

$$H = \left\langle \vec{X}, \vec{\omega} \right\rangle + \epsilon \tag{2.3}$$

The decision procedure from equation 2.2 is still used in the background to decide which label to assign to the measurements $\vec{X}$, but because the error term $\epsilon$ is not directly observed, we can now no longer directly observe $H$ itself, and thus we cannot directly obtain the sign of $H$ to determine the right label. The distance $H$ to the hyperplane has become a random variable, whose probability distribution directly depends on the probability distribution of $\epsilon$. If we could somehow estimate this probability distribution, then we could use this distribution to decide on the result label by just choosing the label that has the highest probability:

$$Y = \begin{cases} y_1 & \text{if } \Pr(Y = y_1 \mid \vec{X} = \vec{x}) > \Pr(Y = y_2 \mid \vec{X} = \vec{x}) \\ y_2 & \text{otherwise} \end{cases} \tag{2.4}$$

Of course, this distribution would still need to be estimated, which is not feasible without making any additional assumptions on the random error term $\epsilon$. The assumption made by logistic regression on this error term is that it is distributed according to the logistic distribution (hence its name) with scale parameter $s = 1$ with center $\mu = b$: $\epsilon \sim$ `Logistic(b, 1)`. Under this assumption, we can now directly calculate the probability of assigning label $Y = y_1$ using the logistic function:

$$\begin{aligned}
\Pr(Y = y_1 \mid \vec{X} = \vec{x}, \mu = b, s = 1) &= \Pr(H + \epsilon > 0 \mid \vec{X}, \mu = b, s = 1) && \text{(definition)} && (2.5)\\
&= \Pr(\epsilon > -H \mid \vec{X}, \mu = b, s = 1) && \text{(moving term to the right)} && (2.6)\\
&= \Pr(\epsilon' > -H - b \mid \vec{X}, \mu = 0, s = 1) && \text{(shifting distribution to zero)} && (2.7)\\
&= \Pr(\epsilon' < H + b \mid \vec{X}, \mu = 0, s = 1) && \text{(symmetric around zero)} && (2.8)\\
&= \int_{-\infty}^{H+b} f(t)dt && \left(\begin{smallmatrix}\text{cumulative}\\ \text{distribution function}\end{smallmatrix}\right) && (2.9)\\
&= \frac{1}{1 + e^{-(H+b)}} && \text{(logistic function)} && (2.10)\\
&= \frac{1}{1 + \exp\left(-(\langle \vec{\omega}, \vec{x} \rangle + b)\right)} && \text{(definition)} && (2.11)
\end{aligned}$$

Thus, by making the assumption that the error term $\epsilon$ is logistically distributed, we can directly calculate the probability of the *unobserved* signed distance $H$ being positive or negative, and with these two probabilities we

can use the decision procedure of equation 2.4 to directly decide the label:

$$Y = \begin{cases} y_1 & \text{if } \Pr(Y = y_1 \mid \vec{X} = \vec{x}) > \Pr(Y = y_2 \mid \vec{X} = \vec{x}) \\ y_2 & \text{otherwise} \end{cases} \tag{2.12}$$

$$= \begin{cases} y_1 & \text{if } \dfrac{\Pr(Y = y_1 \mid \vec{X} = \vec{x})}{\Pr(Y = y_2 \mid \vec{X} = \vec{x})} > 1 \\ y_2 & \text{otherwise} \end{cases} \tag{2.13}$$

$$= \begin{cases} y_1 & \text{if } \exp\left(\langle \vec{\omega}, \vec{x} \rangle + b\right) > 1 \\ y_2 & \text{otherwise} \end{cases} \tag{2.14}$$

$$= \begin{cases} y_1 & \text{if } \langle \vec{\omega}, \vec{x} \rangle + b > 0 \\ y_2 & \text{otherwise} \end{cases} \tag{2.15}$$

$$\tag{2.16}$$

We again end up with a linear decision boundary, which is a direct result of assuming the error term $\epsilon$ to be logistically distributed. When we make other assumptions on the distribution of $\epsilon$, the cumulative distribution function of equation 2.9 changes accordingly, but then we're not performing logistic regression anymore, and the decision boundaries will accordingly not necessarily be linear anymore. For example, when we assume a normally distributed error term, we obtain *probit* regression.

Figure 2.1 graphically depicts how the measurements $X_i$ are combined with the coordinates $\omega_i$ of the normal vector $\vec{\omega}$ to first obtain the signed distance $H$ from $\vec{X}$ to the hyperplane, and then to obtain the probability that the sign of this distance is positive when the random error term $\epsilon$ is added. The graph depicted here is on of the the simplest examples of a *neural network*. In a general neural network, the function $\sigma(\cdot)$ is not restricted to be a cumulative distribution function, but it is allowed to be any transformation from $\mathbb{R}$ to $\mathbb{R}$. Of course the probabilistic interpretation we gave in this section will be lost then. In section 2.5.4 we will discuss how the simple type of neural network shown here can be used as a basic building block to construct a much more general class of prediction devices.

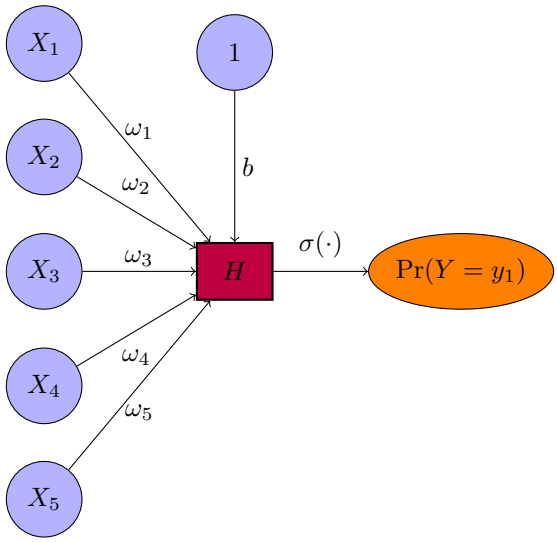


Figure 2.1: Graph-based representation of a general hyperplane + error term classifier. The function $\sigma(\cdot)$ depicts the cumulative distribution function that corresponds to the distributional assumptions on the error term.

Thus far we've discussed the assumptions made by logistic regression about the existence of a separating hyperplane in the space of measurements and on the random error term that gives the decision procedure a non-deterministic dependency on this hyperplane. Of course, the positioning of this hyperplane is still not specified. In practice, *maximum likelihood estimation* or *maximum a posterori estimation* are usually performed to estimate the coordinates of the vector $\vec{\omega}$ that determines the model. We will discuss these estimation methods respectively in sections 2.6.1 and 2.6.2.

### 2.5.2 Multinomial Logistic Regression

In the previous subsection we discussed the binary classification model called logistic regression that relates a binary dependent variable $Y \in \{y_1, y_2\}$ to a vector of continuous measurements $\vec{X} \in \mathbb{R}^n$. Multinomial logistic regression is a generalization of this binary classifier to the case of categorical dependent variables with more than two categories. The independent variables $X_i$ are still assumed to be continuous, but the dependent variable $Y$ is now allowed to take values from any finite discrete set of $K$ values: $Y \in \{y_1, y_2, \ldots, y_K\}$.

The main assumption of the binary logistic regression model was the logistic distribution of the error term $\epsilon$, which was equivalent to the logarithmic odds, $\log\left(\frac{\Pr(Y=y_1|\vec{X}=\vec{x})}{\Pr(Y=y_2|\vec{X}=\vec{x})}\right)$, being a linear combination of the continuous measurements $X_i$. This last property generalizes easily to the categorical setting, were we require the logarithmic odds $\log\left(\frac{\Pr(Y=y_i|\vec{X}=\vec{x})}{\Pr(Y=y_j|\vec{X}=\vec{x})}\right)$ between any two labels $y_i$ and $y_j$ to be linear combinations of the measurements $X_i$. Under this assumption, there exist parameter vectors $\vec{\omega}_j$ for each label $y_j$ such that the probability calculated by the multinomial model for label $y_j$ can be obtained as:

$$\begin{aligned}\Pr(Y = y_j \mid \vec{X} = \vec{x}) &= \frac{\exp\left(\langle\vec{\omega}_j, \vec{x}\rangle\right)}{\sum_k \exp\left(\langle\vec{\omega}_k, \vec{x}\rangle\right)} \\ &= \frac{1}{Z(\vec{x})} \cdot \exp\left(\langle\vec{\omega}_j, \vec{x}\rangle\right)\end{aligned} \tag{2.17}$$

Here the normalizing sum $\sum_k \exp(\langle\vec{\omega}_k, \vec{x}\rangle)$ in the denominator of the first equation is factored out as $Z(x)$ to make the resemblance between multinomial regression and conditional random fields (discussed below) more explicit. In the binomial case of two labels where $K = 2$, setting $\vec{\omega}_1 = \vec{0}$ recovers ordinary logistic regression (compare with equation 2.11). In general, we always have that one of the $K$ parameter vectors is superfluous; subtracting any vector $\phi$ (for example $\phi = \omega_1$) from all parameter vectors $\omega_j$ does not change the calculated probabilities; the model is not *identifiable*.

For our own model, we would like to predict a name for each unknown identifier $Y$ from the information that we obtained from the neighborhood surrounding $Y$. Initially this information will be discrete, but after we've *embedded the neighborhood* (see section 4.5), we're left with a number of continuous measurements $X_i$ from the neighborhood that we'd like to use to predict the correct name for $Y$. For our model we'll restrict ourselves to the task of generating identifier names from a pre-defined finite vocabulary of identifier names that was constructed at training time from the identifiers that were found in the training set. Thus, we would like to relate the categorical dependent variable $Y \in \{\texttt{identifier names that were found in the training set}\}$ to the independently obtained neighborhood measurements $X_i$, and this is precisely what multinomial regresssion allows us to do.

Just like with logistic regression, parameter estimation is usually done via maximum likelihood estimation or via maximum a posteriori estimation. For our own model, however, we will incorporate the multinomial classifier as the final layer to our (deep) neural network, and here a combined parameter estimation of all network parameters by maximum likelihood or maximum a posteriori estimation is generally infeasible. In this setting, we jointly optimize the multinomial prediction layer with the rest of the parameters of the neural network using stochastic gradient descent (see section 2.6.3).

### 2.5.3 Conditional Random Fields

Conditional random fields are discriminative probabilistic models that can be used for *structured classification*. With multinomial logistic regression, the set of target labels $\{y_1, y_2, \ldots, y_K\}$ is discrete, and the labels $y_j$ themselves have *no internal structure* that relates any two of them together. Two distinct labels $y_i$ and $y_j$ are only related to each other through their respective parameter vectors $\omega_i \in \mathbb{R}^n$ and $\omega_j \in \mathbb{R}^n$ that were found during the model fitting phase. The algebraic relations between these parameter vectors reflect the semantic relations that were found between these labels in the training set; when the labels $y_i$ and $y_j$ are usually used in similar contexts, the coordinates of these two labels will also tend to be similar, with the vectors pointing in similar directions in $\mathbb{R}^n$. However, these are *the only relations* between the labels $y_i$ and $y_j$. Any prior information between two distinct labels $y_i$ and $y_j$ is discarded by the multinomial model. Conditional random fields allow us to solve this problem by allowing parameters to be shared between the different labels.

For our purposes, conditional random fields are probabilistic models whose probability distributions are

given by the following formula:

$$\begin{aligned}\Pr(Y = y_j \mid X = x) &= \frac{\exp\left(\langle \vec{\omega}, \vec{f_j}(x)\rangle\right)}{\sum_k \exp\left(\langle \vec{\omega}, \vec{f_k}(x)\rangle\right)} \\ &= \frac{1}{Z(x)} \cdot \exp\left(\langle \vec{\omega}, \vec{f_j}(x)\rangle\right)\end{aligned} \tag{2.18}$$

The functional form of this formula is very similar to that of multinomial logistic regression. Just like with multinomial regression, the model assigns probabilities to a categorical variable with $K$ categories: $Y \in \{y_1, y_2, \ldots, y_K\}$. Both models invoke a linear response from the labels $y_j$ and the measurement $x$ with respect to the parameters of the model, which is subsequently exponentiated and normalized to yield a probability distribution. The only difference between the two models is how the parameters, the labels $y_j$ and the measurements $x$ are precisely aligned. In multinomial regression, each label $y_j$ has its *dedicated* parameter vector $\vec{\omega}_j \in \mathbb{R}^n$ which is compared to a measurement vector $\vec{x} \in \mathbb{R}^n$ that is shared between the labels. Conditional random fields, on the other hand, have a single parameter vector $\vec{\omega} \in \mathbb{R}^n$ that is *shared* between the labels $y_j$, but each label $y_j$ has a dedicated measurement vector $\vec{f_j}(x) \in \mathbb{R}^n$ that is a label-specific refinement of the shared conditional $x$. The functions $\vec{f_j}$ that implement the label-specific measurements allow the model designer to incorporate prior knowledge of relations between different labels by making explicit the way parameters are shared between them. Shared parameters correspond to shared structure, and the more parameters two labels have in common the more they are structurally the same. The multinomial model can be recovered from the conditional random field model by declaring there to be zero structural overlap between the labels. Zero structural overlap corresponds to the parameters that are used when calculating the probabilities to be orthogonal between the labels $y_j$, and this is achieved by requiring the measurement vectors $\vec{f_j}(\vec{x})$ to be mutually orthogonal. Then the individual parameters vectors $\vec{\omega}_j$ of the multinomial model combine into a single parameter vector $\vec{\omega} := \vec{\omega}_1 \oplus \vec{\omega}_2 \oplus \ldots \oplus \vec{\omega}_K \in (\mathbb{R}^n)^K$ when the corresponding measurement vectors $\vec{f_j}(\vec{x})$ are defined as orthogonal embeddings of $\vec{x}$ into the $j$-th orthogonal subspace of $(\mathbb{R}^n)^K$: $\vec{f_j}(\vec{x}) := (\vec{0}, \vec{0}, \ldots, \vec{x}, \ldots \vec{0}, \vec{0})$.

Conditional random fields were originally introduced in (Lafferty et al., 2001), where they are defined as consisting of a collection of categorical random variables $\mathcal{Y}$ together with a graph structure $G = (V, E)$ on these variables such that the nodes $a \in V$ index the random variables, $V_a \in \mathcal{Y}$, and such that the random variables obey the Markov properties with respect to the graph $G$. The Markov properties dictate that the probability distribution of a single random variable $V_a \in \mathcal{Y}$ is completely determined when conditioned on the random variables $V_b$ that are direct neighbors of $V_a$ in the graph $G$. Under some mild assumptions on the random variables, this approach can be seen to be equivalent to the approach we gave in this section.

The conditional random field model that we use as a baseline for our own experiments is a slightly generalized version of the conditional random fields described here. In the version we use, the restriction that the set of target labels should be finite is lifted. Instead of specifying the measurement functions $\vec{f_j}(x)$ corresponding to the labels $y_j$ upfront, we now have a single measurement function $\vec{f}(Y, x)$ that is allowed to directly inspect any concrete label $y$:

$$\Pr(Y = y \mid X = x) = \frac{1}{Z(x)} \cdot \exp\left(\langle \vec{\omega}, \vec{f}(y, x)\rangle\right) \tag{2.19}$$

For our baseline experiment, the label set is now roughly defined as the set of all labeled directed knowledge graphs, and the measurement function $\vec{f}(\tilde{G}, P)$ measures for each labeling $\tilde{G}$ of the knowledge graph $G = (V^P, E^P)$ that was extracted from $P$ how many times each unique *triple* (see introduction section 1.5) occurs in $\tilde{G}$.

### 2.5.4 Feed Forward Neural Networks

Feed-forward neural networks employ a hierarchical structure of basic building blocks similar to the logistic regression prediction layer to make complex predictions. In section 2.5.1 we discussed how logistic regression separates the space of measurements $\vec{x} \in \mathbb{R}^n$ into two labeled halves by a hyperplane, and how it then makes assumptions on the random error term to calculate the probabilities that a concrete measurement $\vec{x}$ should actually lie in either half space. Of course, the assumption that the measurement space consists of only two parts that precisely determine the label is very simplistic.

The first step to allow for labeling more realistic scenarios is by allowing a number of $k$ independent hyperplanes that each separate the total measurement space into their own respective pair of half spaces. Each hyperplane measures its own binary property, and by taking the intersection of half spaces we obtain a conjunction of binary properties. The individual probabilities multiply to obtain the probability of the conjunction. The main idea behind neural networks now is that the calculated probabilities for each of these $k$ binary properties together also constitute a vector of measurements $M(\vec{x}) \in \mathbb{R}^k$. Again, we can divide this space into two labeled half spaces by a hyperplane, and again we can make some assumptions on the random error term that makes the actual label assigned to the measurement $M(\vec{x})$ be different to the label on the half space where $M(\vec{x})$ is actually located. The main power of neural networks now comes from the fact that the *linear* decision boundary in this latter feature space $\mathbb{R}^k$ corresponds to a *non-linear* decision boundary in the original feature space $\mathbb{R}^n$, due to the fact that the calculated probabilities calculated form the original measurements all depend non-linearly on $\vec{x}$.

Thus, adding an array of $k$ logistic regression predictors that each take as input some measurement $\vec{x} \in \mathbb{R}^n$, we obtain a vector of derived probability measures $M(\vec{x}) \in \mathbb{R}^k$, and linear decisions on the derived quantity $M(\vec{x})$ translate to non-linear decisions on the original measurements $\vec{x}$. Of course, the last logistic regression layer also outputs the probability of its decision being correct, and this again can be seen as some derived measurement. By combining the first array with an additional array of $t$ logistic regression predictors that each take the derived measurement $M(\vec{x}) \in \mathbb{R}^k$, we obtain another vector of measurements $N(M(\vec{x})) \in \mathbb{R}^t$ that are twice-derived from the original input vector $\vec{x}$. The linear decisions made in this space now correspond to non-linear decisions in the previous space $\mathbb{R}^k$, which then correspond to an even more sophisticated decision boundary in the original space $\mathbb{R}^n$.

In general, this pattern can be repeated over any number of predictor arrays, with each array containing any number of predictors. The general setup is shown in figure 2.2. Due to the increased complexity of the network, some pieces of information that were explicitly depicted in the graphical depiction of the logistic regression neural network in figure 2.1 are left out to avoid cluttering the picture.

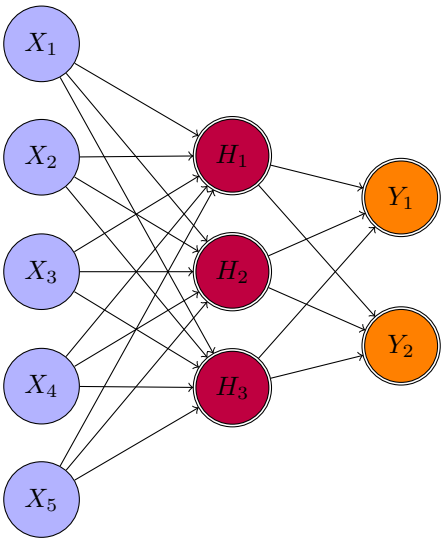


Figure 2.2: Graph-based representation of a small neural network.

### Activation Functions

In the previous we've explained neural networks in terms of a layered stack of simple logistic networks in which each layer delivers a number of probability measurements which are used as inputs to the subsequent layer. Like we discussed previously in section 2.5.1, the final probability calculated by the logistic regression network by an application of the **activation function** $\sigma(\cdot) =$ `< logistic function >` to the linear response $H$ was based on the assumption of the presence of a logistically distributed error term $\epsilon$. For general neural networks, the output of the network need not be a probability, and even when it is, the distribution of the error term need not be assumed to be logistic. When a different assumption is made on the distribution of $\epsilon$, the current activation

function gets replaced by the cumulative distribution function corresponding to these new assumptions. In practice, when neural networks are configured to output probabilities, these probabilities will almost always be given by setting $\sigma(\cdot)$ to the logistic function, which is due to the convenient algebraic properties of this function in combination with a lack of evidence that any other distribution should be assumed.

**Hyperbolic Tangent** A frequently occurring alternative to the logistic activation function that does not arise from any probabilistic assumptions is the hyperbolic tangent activation function `tanh`. This function maps the real line to the open interval $(-1, 1)$, instead of to the unit interval $(0, 1)$. The activation of the linear response $H$ can be interpreted as measuring the extent to which some input is correlated $(+1)$, uncorrelated $(0)$ or anti-correlated (-1) with some feature. The hyperbolic tangent function preserves the vector space negation operation, $\tanh(-t) = -\tanh(t)$, which causes it to treat the positive and negative directions on equal footing when the activated responses are used in a multiplicative way further down the pipeline: the result of multiplying something with $\tanh(-t)$ has the same norm as when performing the multiplication with $\tanh(t)$, while the logistic function exchanges preservation and absorption properties of the norm. Figure 2.3 shows a graph of the hyperbolic tangent function.

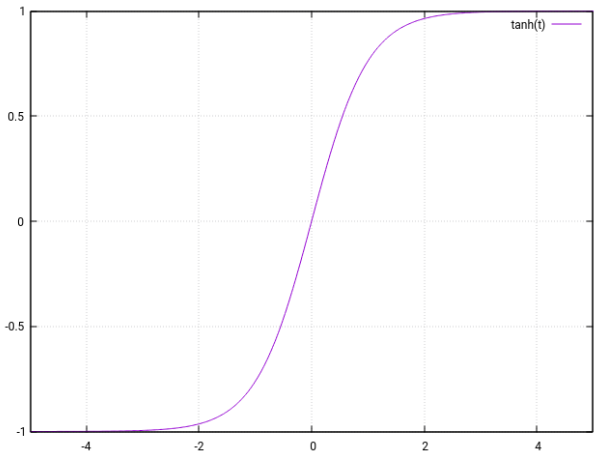


Figure 2.3: Graph showing the hyperbolic tangent function.

**Rectified Linear Unit** Another commonly used activation function is the *rectified linear unit*, `relu`, which is defined as:

$$\texttt{relu}(t) := \max(0, t) \tag{2.20}$$

The function is linear on the positive real line and identically zero on the negative real line. Even though this activation function is very simple, by the universal approximation theorem for neural networks (Sonoda & Murata, 2015), it is still sufficiently strong to allow neural networks to approximate any function (meeting some mild conditions) to any desired accuracy by constructing sufficiently wide neural networks using only `relu` activations. Compared to the previously discussed sigmoidal activation functions, the `relu` function has the desirable property that its derivative is almost everywhere constant, which solves some of the problems that sigmoidally activated neural networks sometimes suffer from. This, together with its implementation simplicity, makes the `relu` activation function a popular choice of activation function in neural networks.

### Soft-Max Layer

The soft-max layer is a special layer used in neural networks to normalize the outputs of a previous layer to be positive (even when the inputs were not) and have a total mass of 1. These two properties together allow the outputs of the soft-max layer to be interpreted as a probability distribution on the individual output positions. The main property of soft-max normalization is that the mass of the resulting probability distribution is mostly located at the single position that had the largest input value, while all smaller inputs values will

have exponentially less mass after normalization. As a result, and as its name suggests, soft-max can be used as a smooth substitute for the usual max-function by calculating a weighted average of the input values using their respective calculated weights. The final result will then lie most closely to the input which had the highest value.

The outputs of the soft-max layer for a number of inputs $L_k$ are calculated as follows. For each input $L_j$, the corresponding soft-max weight is calculated as:

$$p_j := \frac{\exp(L_j)}{\sum_k \exp(L_k)} \tag{2.21}$$

The soft-max layer is used in multinomial logistic regression to calculate the label probability distribution from the dot products between the measurement vector $\vec{x} \in \mathbb{R}^n$ and the parameter vectors $\vec{\omega}_j \in \mathbb{R}^n$ for each label $y_j$.

## 2.6 Parameter Estimation/Optimization

The previous section gave an overview of a number of (probabilistic) parametric models that are parameterized by finite dimensional parameter vector. The goal of the training phase is to instantiate these parameters in such a way that the fitted model becomes good at making predictions. There are various methods and principles available to guide this search for optimal parameters, and some of them will be discussed in this section.

### 2.6.1 Maximum Likelihood Estimation

When fitting parametric probabilistic models on a given set of training data $\mathcal{D}$, the *principle of maximum likelihood* states that we should instantiate the parameters in such a way that the probability of obtaining the actual training data, according to the parameterized model, is maximized over all possible parameter instantiations. Let us assume the data points in $(y, x) \in \mathcal{D}$ are all obtained independently. The probability of obtaining the training set $\mathcal{D}$ under this assumption then factorizes over the probabilities of the individual data points. The goal of parameter estimation via likelihood maximization is then to estimate the optimal parameters $\hat{\vec{\omega}}$ as:

$$\begin{aligned}
\hat{\vec{\omega}} &= \arg\max_{\vec{\omega}} \Pr(\mathcal{D} \mid \vec{\omega}) \\
&= \arg\max_{\vec{\omega}} \prod_{(y,x)\in\mathcal{D}} \Pr(Y = y \mid X = x, \vec{\omega}) \\
&= \arg\min_{\vec{\omega}} \sum_{(y,x)\in\mathcal{D}} -\log \Pr(Y = y \mid X = x, \vec{\omega})
\end{aligned} \tag{2.22}$$

Unfortunately, a closed-form solution that minimizes this quantity is usually not available. Numerical optimization procedures are then employed to try to minimize this objective, but unfortunately this minimization objective is usually not convex and thus a globally optimal solution will usually be out of reach.

### 2.6.2 Maximum A Posteriori Estimation

A different estimation principle called *maximum a posteriori estimation* states that we should directly try to calculate the probability of the parameter vector $\vec{\omega}$ when conditioned on the training data $\mathcal{D}$. We again assume that all data points in $(y, x) \in \mathcal{D}$ are obtained independently. By using Bayes' theorem, we can write this objective as:

$$\begin{aligned}
\hat{\vec{\omega}} &= \arg\max_{\vec{\omega}} \Pr(\vec{\omega} \mid \mathcal{D}) \\
&= \arg\max_{\vec{\omega}} \Pr(\mathcal{D} \mid \vec{\omega}) \cdot \Pr(\vec{\omega}) \\
&= \arg\max_{\vec{\omega}} \Big( \prod_{(y,x)\in\mathcal{D}} \Pr(Y = y \mid X = x, \vec{\omega}) \Big) \cdot \Pr(\vec{\omega}) \\
&= \arg\min_{\vec{\omega}} \Big( \sum_{(y,x)\in\mathcal{D}} -\log \Pr(Y = y \mid X = x, \vec{\omega}) \Big) - \log \Pr(\vec{\omega})
\end{aligned} \tag{2.23}$$

The prior probability of the data $\Pr(\mathcal{D})$ in Bayes' theorem is independent of the parameters and thus a constant that can be removed from optimization. The distribution on the parameter space $\Pr(\vec{\omega})$ is called the *prior distribution*. The choice of prior distribution on the parameter space is something that needs to be specified explicitly in this method, and each choice corresponds to a specific regularization method. Choosing the coordinates of the parameter vector to be either normally distributed or Laplace distributed results in the term $-\log \Pr(\vec{\omega})$ being proportional to respectively the $L^1$-norm or the square of the $L^2$-norm of the parameter vector $\vec{\omega}$. In the case of linear regression, calculating this maximum a posteriori estimate using these two priors would be called respective lasso regression and ridge regression.

### 2.6.3 (Stochastic) Gradient Descent

As we mentioned earlier, finding closed form solutions that maximizes the likelihood of the model parameters with respect to the training data is not feasible in general. The alternative is to employ numerical optimization algorithms that explore the parameter landscape until a solution of a sufficiently high quality is found. The optimization algorithm that we employ for our model is a variant of *stochastic gradient descent*, which itself is an instance of the general family of gradient descent optimization algorithms.

Gradient descent algorithms are used in supervised learning, where there is some *loss function* $L$ that quantifies the differences between the expected answer and the solution that was currently returned by the model. We can write $L_x^y(\vec{\omega})$ that signifies the dependency of this scalar loss on the model parameters with respect to a single data point $(y, x) \in \mathcal{D}$ that is held constant. If the dependency of the loss on the model parameters $\vec{\omega}$ is continuous and differentiable, we can use the *derivative* of this function to find the direction in the parameter space in which the function is decreasing the strongest. The (total) derivative of a function $L : \mathbb{R}^n \to \mathbb{R}$ at a point $p \in \mathbb{R}^n$ is given by the *gradient vector* whose components are the partial derivatives of the function $L$ with respect to each of the coordinate dimensions. The gradient vector $\nabla_p L$ of $L$ at the point $p$ is known to point in the direction in which $L$ is increasing the fastest. If the function $L$ is well-behaved, then adjusting the parameters $p$ by a small step in the direction $\nabla_p L$ would have the strongest effect on the value of $L(p)$. Of course our goal is to *minimize* the loss function $L_x^y$, so taking the negative of the gradient $\nabla_{\vec{\omega}} L_x^y$ would indicate the direction in which $L_x^y$ is currently the quickest decreasing at its current parameter location $\vec{\omega}$. If $L_x^y$ is well-behaved, then nudging $\vec{\omega}$ slightly in the direction $\nabla_{\vec{\omega}} L_x^y$ would usually decrease the loss of the data point $(y, x)$. If we then re-evaluate the gradient at the nudged location, we again see in which direction it is decreasing the strongest, and again we can nudge this vector a bit in the direction of this gradient to (hopefully) lower it even further. Using the gradient of the loss function to nudge the parameters in a direction which decreases the loss calculated by the loss function is the main idea behind the family of gradient descent algorithm, where we *descent* the parameter space on a path that has increasingly smaller calculated loss. We can continue the descent until we've reached a local optimum where the gradient vanishes, or we can stop if following the gradient has no more positive (or even negative) effect.

Stochastic gradient descent is a variant of this basic gradient descent algorithm where the gradients are taken with respect to a small, randomly sampled batch of $N$ data points at the same time. The negative gradient vectors for each of the $N$ data points are averaged to obtain a single vector that represents the direction which would overall be the most beneficial for the batch. The parameter update step can be written as:

$$\vec{\omega}_{k+1} = \vec{\omega}_k - \frac{\alpha}{N} \cdot \sum_i^N \nabla_{\vec{\omega}_k} L_{x_i}^{y_i} \tag{2.24}$$

Here the *hyperparameter* $\alpha$ is the learning rate which controls the size of the jumps we make. When $\alpha$ is too large, we jump over the landscape too quickly and we never reach a valley. When $\alpha$ is set too low, we don't learn anything.

The actual optimization algorithm we will be using for our experiments is a variant of stochastic gradient descent called *Adam*. The precise details of Adam are beyond the scope of this document, but the main idea is that it keeps track of individual learning rates for each parameter. Section 2.7 below discusses a few common ways to quantify loss. Some of these are differentiable and thus allow gradient descent to be used to optimize them.

## 2.7 Loss Functions

When doing supervised learning, there are various ways in which the predictions made by the model can be compared to the reference values in the training set. The goal of supervised learning is to use the known-to-be-correct values in the training set to inform the construction of a model that can make predictions for new, unseen data as well as possible. Loss functions are a way to quantify the difference between the correct value in the training set and the value predicted by the model. Various such functions exist, each one suited for a specific domain and each having different operational characteristics.

In the following, let $M_{\vec{\omega}}$ be a model parameterized by a parameter vector $\vec{\omega}$ and let $M_{\vec{\omega}}(x)$ be the label predicted by $M_{\vec{\omega}}$ for the data point $x$.

### 2.7.1 0-1 Loss

$$\mathcal{L}_{\vec{\omega}}(y \mid x) := \begin{cases} 0 & \text{when } M_{\vec{\omega}}(x) \text{ is equal to } y \\ 1 & \text{when } M_{\vec{\omega}}(x) \text{ is not equal to } y \end{cases} \tag{2.25}$$

Discrete. Counts one error for each specimen in the training set that is predicted incorrectly. Usually this is what we actually want to minimize (i.e. minimize the total number of errors made), but unfortunately this isn't actually feasible in practice. The number of prediction errors made is not a continuous function of the model parameters; changing the parameters slightly can totally and unpredictably change the predictions made over the complete training set. There is no way to systematically search the parameter landscape to find a parameter configuration which is at least somewhat good, let alone to find the global optimum. In practice most applications use more tractable loss functions, which either upper bound the 0-1 loss (which thus makes it good objective functions to minimize instead), or which are based on some other principle that make it likely that the 0-1 loss function will also be somewhat minimized with it (e.g. negative log likelihood minimization).

### 2.7.2 Hinge Loss

$$\mathcal{L}_{\vec{\omega}}(y \mid x) := \max\big(0, 1 - y * f_{\vec{\omega}}(x)\big) \quad y \in \{+1, -1\} \tag{2.26}$$

Calculates a direct upper bound to the 0-1 loss. Used in Support Vector Machines where the goal is to find a hyperplane (which is determined by $f_{\vec{\omega}}(x)$) in some $d$ dimensional space that separates the positive samples from the negative samples as well as possible. There is a margin on both sides of the hyperplane, and for each labeled point the signed distance between the point and the margin of the side where it should be located is calculated, with the distance being positive when the point is located on the wrong side. The hinge loss is defined as this distance when the distance is positive and zero otherwise. This way, points that are on the correct side of the hyperplane and that are separated from it by at least the margin will contribute zero loss, while points which are on the correct side but within the margin will contribute a loss between zero and one. Points which are on the wrong side of the margin will contribute a loss of more than one. It is easy to see that this loss is equal to the 0-1 loss when the hyperplane perfectly classifies all data points with no points within the margin, and it is strictly greater otherwise. Minimizing this loss thus bounds the 0-1 loss, and thus this loss function can serve as a good replacement loss function to minimize when minimizing the actual 0-1 loss is unfeasible.

### 2.7.3 Hamming Loss

$$\mathcal{L}_{\vec{\omega}}(y \mid x) := d_{Hamming}\big(y, M_{\vec{\omega}}(x)\big) \tag{2.27}$$

Structured version of 0-1 loss. When predicting structured output, count the number of subcomponents which were predicted wrongly. Equals the Hamming distance when the output can be interpreted as a sequence of symbols. Just like the 0-1 loss, minimizing the hamming loss directly usually isn't feasible because the predictions don't continuously depend on the parameters.

### 2.7.4 Structured Hinge Loss

$$\mathcal{L}_{\vec{\omega}}(y \mid x) := \max_{y' \in \Omega_x} \Delta(y, y') + [\text{score}_{\vec{\omega}}(y', x) - \text{score}_{\vec{\omega}}(y, x)] \tag{2.28}$$

Calculates a direct upper bound to the Hamming loss. Used in Structured Support Vector Machines where the goal is to find the parameters to a (linear) score function $\text{score}_{\vec{\omega}}(y, x))$, such that the score $\text{score}_{\vec{\omega}}(y, x)$ assigned

to the actual labeling $y$ of a data point $x$ is higher than the score $\text{score}_{\vec{\omega}}(y', x)$ assigned to an alternative $y'$ by a margin of at least $\Delta(y, y')$, for as many data points as possible. This forces the score function to *convincingly* assign the highest score to the correct labeling, with a data point only contributing zero loss when all alternative labelings have scores that are at least as much worse as the number of errors they make compared to the actual labeling. The loss functions measures how much a single data point violates this margin, and the whole training process minimizes the average violation over all data points. The total loss function is globally convex and thus has a unique minimum that can easily be found by stochastic sub-gradient descent. This loss function is used for fitting conditional random fields in (Raychev et al., 2015).

### 2.7.5 Negative Log Likelihood Loss

$$\mathcal{L}_{\vec{\omega}}(y \mid x) := -\log \Pr_{\vec{\omega}}(Y = y \mid X = x) \tag{2.29}$$

This is the loss function that corresponds to maximum likelihood maximization that was discussed in section 2.6.1.

# Chapter 3

# Previous Work

The research described in this document is based primarily on the contributions made by papers listed in the sections below. We outline the relevant parts of the individual papers and we describe to what extent we adapted their machinery to suit our own needs.

## 3.1 Predicting Program Properties from "Big Code"

Raychev et al., 2015 used an undirected knowledge graph to represent the program. In this representation, the vertices of the graph correspond to the program points of interest and the edges between the vertices correspond to various relations between the program points that were extracted from the program. The main idea of their approach is that the predictions made for the various program points should be made in a mutually consistent way. The relations were chosen in such a way that the presence or absence of a relation between two program points would (hopefully) give a good indication of the compatibility of various possible assignment of names to those variables.

For example, in the expression x = (a + b)/2, a consistent naming for the variables x, a and b would be something like midPoint, beginPoint and endPoint while an inconsistent naming would be iter, speed and isActivated: averaging a scalar and a boolean and calling the result iter usually doesn't make sense (even though JavaScript would probably allow it). In this example the variables a and b are related by the averaging relation $r = [\_ = (? + ?)/2]$, which indicates that there is some other program point (denoted by the wildcard underscore) such that the average of the two endpoints (denoted by the two question marks) is assigned to the other program point. This relation is purely *syntactic*, but in general non-syntactic relations are also possible (e.g. if by static analysis it can be determined that two pointers p and q always point to the same object, we might add an aliasing relation between their nodes in the graph).

For each program in our training set we can construct such a **knowledge graph**, and when a specific relation (edge) with a specific assignment of labels (variable names) to the connecting program points occurs a lot in the training set then this triple $(z_i, r, z_j)$ of a relation $r$ and two labels $z_i$ and $z_j$ will get assigned a high score. The goal of the framework is to first determine good scores for all triples $(z_i, r, z_j)$ that occur in the training set (this is the training phase) and then to find an assignment of labels to all the vertices in the knowledge graph constructed for an unseen program such that the total sum of scores for the induced triples $(z_i, r, z_j)$ for that assignment is greater than or equal to the induced total score of all other possible assignments (the inference phase).

The authors formulate this problem in the language of **conditional random fields**, which are probabilistic models akin to **Markov random fields** where the vertices of a graph correspond to random variables and the edges correspond to dependencies between these random variables. More precisely, the random variables obey the local (and also the global) Markov property with respect to the graph: when conditioned on its direct neighbors in the graph, a random variable $Y$ becomes independent of all other variables in the graph. Conditional random fields differ from Markov random fields in that the probability distributions on the random variables can depend (be conditioned) on contextual information which is specific to the program under consideration. In our situation this means that for each program $P$ we get a distinct set of random variables $Y_1$, $Y_2$, .., $Y_n$ corresponding to the various program points within our program, each of them taking values in the space of possible variable names, $Y_i \in \mathit{VarNames}$ (seen in the training set), where the distribution of each of these variables is dependent on

the specifics of the program structure. The knowledge graph constructed for $P$ specifies the (in)dependencies between the variables.

Given the distribution for each of these variables, we can ask which combined assignment of labels to each of these variables is most likely, i.e. which assignment $(y_1, y_2, .., y_n)$ of labels to the program points (labeled 1 through $n$) maximizes the probability $P(Y_1 = y_1 \wedge Y_2 = y_2 \wedge .. \wedge Y_n = y_n \mid P)$. This directly corresponds to the maximization of the total score over all triples $(y_i, R, y_j)$ in the previous paragraph. Fitting the distributions $P(Y_2 \wedge Y_2 \wedge .. \wedge Y_n \mid P)$ to best match the distribution found in the training set corresponds to finding optimal scores for each of the possible triples $(y_i, r, y_j)$ that occur in the training set.

Trying all possible combinations of assignments $y_i$ of variable names to each individual program point $Y_i$ is combinatorically infeasable. Using the Markov property of the conditional random field, Raychev et al., 2015 have implemented a greedy inference algorithm which does local optimizations which iteratively finds the optimal value for one single program point assuming all neighbors already have their optimal values set. Because of the local Markov property, the score contributed by a specific assignment to a program point $Y_i$ is only dependent on the values assigned to its direct (radius-1) neighbors $Y_j \in N(Y_i)$ in the knowledge graph, so only scores corresponding to the triples $(y_i, R, y_j)$ directly attached to $Y_i$ have to be taken into consideration. The whole algorithm then consists of first randomly assigning labels to all program points and then iteratively visiting all nodes $Y_i$ (possibly in random order), updating $Y_i$ to the label $y_i$ which attains the maximum score with the direct neighbors held constant. This process can go on until either an iteration limit has been reached or when convergence has occurred.

The framework developed by Raychev et al., 2015 consists of a language-agnostic backend called *Nice2Predict* which consumes a training set of labeled undirected knowledge graphs and outputs a trained conditional random field model which can be used to make identifier name predictions for partially unlabeled knowledge graphs. The knowledge graphs are constructed by a language-specific frontend, which in their JavaScript demonstration implementation is called *UnuglifyJS*.

## 3.2 A General Path-based Representation for Predicting Program Properties

Alon et al., 2018 demonstrated that syntactic paths between identifiers in the program's abstract syntax tree can usefully serve as features in predicting names for identifiers. They showed this by adapting the conditional random field framework that was described in the previous section with relational edges derived purely from syntactic paths between identifiers, and their results show that these path-based features substantially increase the prediction accuracy.

The adapted *UnuglifyJS* JavaScript frontend to *Nice2Predict* which outputs these path-based features is dubbed *PidgeonJS*. Originally we had hoped to use this improved feature set in our own experiments instead of the original feature set extracted by `UnuglifyJS`, but unfortunately this resulted in a data set which was too large for us to efficiently experiment on.

## 3.3 Code2Vec: Learning Distributed Representations of Code

Alon et al., 2019 subsequently showed how the path representations that were described in the previous section can be used to continuously embed discrete code fragments into a finite dimensional real vector space using a novel attention mechanism which allows these fragments to serve as input features for a deep neural network. Their embedding technique is heavily inspired the `word2vec` (Mikolov, Chen, Corrado, & Dean, 2013) embedding mechanism was was used previously to embed discrete `textual` fragments into some real vector space. Their main addition to the existing `word2vec` method is the addition of an *attention mechanism* which allows the model to learn which parts of the context are important and which are not.

The authors showed that their code embeddings are good at preserving semantic relationships between related code fragments and that the obtained features allow accurate predictions of method names given the embedding of the method body. Our thesis starts by adapting their code fragment embeddings technique to instead embed the *local graph neighborhoods* surrounding the various identifiers in a knowledge graph, with the goal of using these embedded neighborhoods as features to base our own predictions on.

## 3.4 Efficient Estimation of Word Representations in Vector Space

The embedding technique used in the `code2vec` paper was itself heavily inspired by the Continuous-Bag-of-Words (CBOW) design that was first introduced in (Mikolov, Chen, et al., 2013; Mikolov, Sutskever, Chen, Corrado, & Dean, 2013). The CBOW model (dubbed `word2vec` in this document) obtains vector representations for natural language words by training a shallow neural network to predict center words from the vector representations of their respective context words. These so-called center words are obtained from large training corpus of semantically rich text (for example Wikipedia). The individual words in the training text are extracted together with a context of words that surround them in the text, and this derived dataset is used by the neural network to optimize the coordinates of the vector representations of the individual words in the dictionary such that they become good predictors for the center word. This optimization forces the context words and the center words to occupy similar locations in the embedding vector space, thus obtaining a *distributed representation* where the meaning of a single word is represented as the superpositioning of some basic concepts. Since its introduction, `word2vec` and related techniques have revolutionized the field of natural language processing.

# Chapter 4

# Approach

As mentioned in the introduction, our main idea is to generalize the convolutional architectures used in image processing to operate on general graphs. We abstractly represent our source code as multigraphs where the nodes represent the source code identifiers and the labeled edges represent various syntactic and semantic relations between them. Some of the nodes (i.e. identifiers) are already named while other nodes need to have have their names predicted for them. Nodes in the first category will be refered to as the **known** nodes, while nodes in the latter category will be referred to as the **unknown** nodes. The overall task of our convolutional neural architecture is to accumulate enough local and non-local information at each of the graph nodes such that the names for the *unknown* nodes can be accurately predicted from the available information.

The purpose of this chapter is to give a detailed overview of the various ideas and technologies that together constitute our proposed architecture. A subsequent chapter will subsequently verify the viability of this architecture via a number of carefully designed experiments.

Throughout this chapter we assume a basic familiarity with ordinary feed-forward neural networks. We refer the reader to section 2.5.4 for a gentle introduction to this subject.

## 4.1 Knowledge Representation

Knowledge graphs are a general way to represent information between various entities. A knowledge graph can either be directed or undirected and consists of a regular (un)directed multi-graph structure $G = (V, E)$ together with two mappings $\mathit{nodeName} : V \to \mathit{String}$ and $\mathit{edgeName} : E \to \mathit{String}$ that each assign a unique name to respectively the nodes and the edges of the graph. Multiple edges between a single pair of nodes are allowed.

The entities are the various objects of interest and the edges between them record the various ways they are known to be related. For our purposes we will model the relevant information known for a program $P$ by the construction of a knowledge graph $G^P = (V^P, E^P)$ where the nodes $Z \in V^P$ correspond to various identifiers within the program $P$ and the labeled edges $(Z_i, Z_j)^{label_k} \in E$ correspond to the various ways the identifiers $Z_i$ and $Z_j$ are related to each other. Figure 4.1 shows the sample program from the introduction, side-by-side with its corresponding knowledge graph representation. We see here that the nodes are labeled with the various identifier names that were found in the program, and that the named (multi)edges record the syntactic relations between objects attached to these identifiers. The knowledge graph constructed for a specific program is by design an *abstraction* of the original program; it only contains that part of the program that we deem relevant to the task at hand. We specifically do not include any of the program's concrete syntax, and in fact, we also don't fully necessarily include all information that was in the original abstract syntax tree.

The exact information contained in a knowledge graph is something that needs to be explicitly specified. The knowledge graphs constructed in (Raychev et al., 2015) used a set of manually engineered relations, some of which were explicitly identified within the abstract syntax tree, and others were found using static analysis. In (Alon et al., 2018), the authors replaced the explicitly derived syntactic features by a general set of **path features** found by considering general paths within the abstract syntax tree connecting various identifiers. No manual considerations were given to these paths, only that they should connect two identifiers and be limited to be of some pre-specified length. Whether the graphs used are directed or undirected depends on the application. The graphs shown thus far naturally seem to be of the directed kind. In (Raychev et al., 2015), however, the

```
function findNode(tree, targetKey) {
  var currentNode = tree;

  while (currentNode != null) {
    if (targetKey == currentNode.key) {
       break;
    } else
    if (targetKey < currentNode.key) {
       currentNode = currentNode.leftChild
    } else
    if (targetKey > currentNode.key) {
       currentNode = currentNode.rightChild
    }
  }

  return currentNode;
}
```

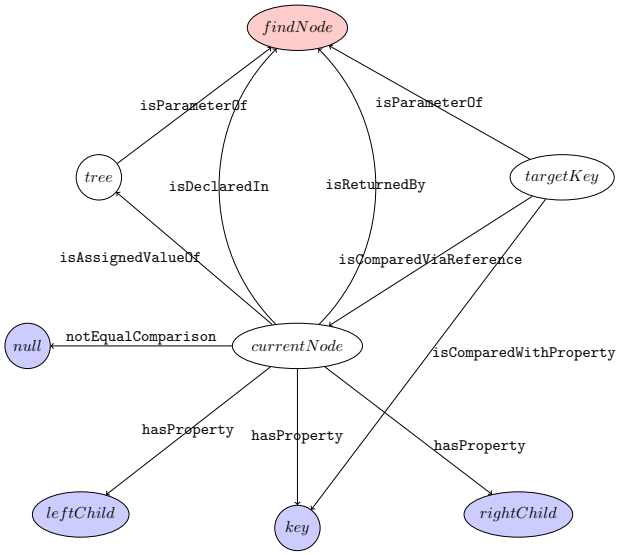


Figure 4.1: The *findNode* program together with its knowledge graph representation.

*conditional random field* framework uses the knowledge graph structure to model the *independencies* between random variables, and in this setting it is more natural to consider knowledge graphs of the undirected kind. For our architecture we adopt their choice of using directed knowledge graphs. The concrete knowledge graphs we plan to use for our experiments will be extracted from the source code by the program `UnuglifyJS` (Raychev et al., 2015) that was also used to test the conditional random field model of the same authors.

```
function findNode(b, c) {
  var d = b;

  while (d != null) {
    if (c == d.key) {
       break;
    } else
    if (c < d.key) {
       d = d.leftChild
    } else
    if (c > d.key) {
       d = d.rightChild
    }
  }

  return d;
}
```

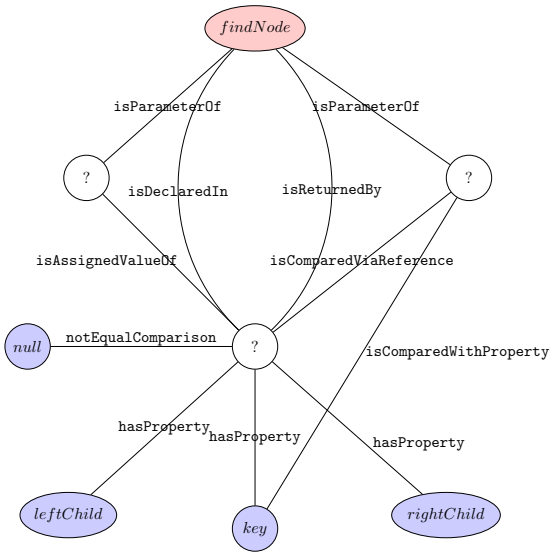


Figure 4.2: The *findNode* program and knowledge graph with local variable names erased.

In figure 4.2 we've shown a version of the *findNode* program where only the local variable identifiers have had their names removed. When programs are presented in this form, the prediction task is easy to state: fill in the question marks in such a way that the various relations between the instantiated nodes make the most sense.

## 4.2 Neighborhood Information Aggregation

In computer vision, convolutional neural architectures are used to summarize neighborhood information for various locations on a two dimensional grid by applying a simple single-layer neural network (the **kernel**) to the information contained in small neighborhoods surrounding the location where the neural network is currently being evaluated. The input grid has fixed finite dimensions $w$ and $h$, and each grid location $(x, y)$ contains some finite amount of information represented by a number of real numbers. In computer vision applications, the input grid typically represents an image, and the information stored at each grid location $(x, y)$ correspond to RGB color intensities.

When used in one of its simplest forms, the neural network is evaluated for each grid location $(x, y)$ and takes (for example) the information from a 3-by-3 block of grid positions centered at $(x, y)$ as input and then outputs a number of activated responses. The individual responses each detect the presence or absence of some features in the input neighborhood, and by evaluating this network for all grid position we obtain information on where precisely these features are most strongly present and where they are mostly absent. We can store the output of the neural network evaluated at the various positions in a fresh grid of the same dimensions to obtain an abstracted version of the original input where the information present at each position is now limited to the information from a neighborhood around the same position in the original grid that was deemed relevant by the neural network. Figure 4.3 shows how the linear (unactivated) response for such 3-by-3 kernel calculated from the neighborhood surrounding some location, and how the result is subsequently stored in a fresh grid.

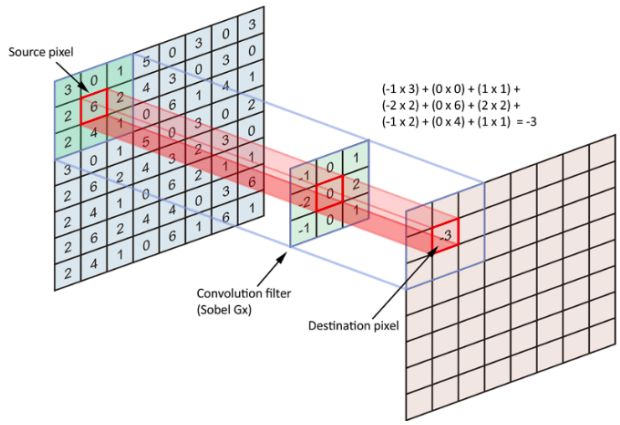


Figure 4.3: Single 3x3 feature map evaluated at a specific position. Source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

The information in the transformed grid is of a higher level than the information in the original grid, with the resulting features corresponding to concepts that are expressed in terms of multiple more basic concepts in small neighborhood in the original grid. When we apply this process to images where the information in the input grid corresponds to RGB color intensities, the output features on the various grid positions are no longer plain pixel intensities, but measurements of concepts expressed in terms of pixel intensities. Some of these higher features might be interpreted as measuring the presence or absence of some specifically oriented line segments in the area surrounding the location, whereas others might specialize in detecting the homogeneity of the image at that location.

The application of such a single small neural network to the each of (the neighborhoods of) the various positions on the grid is called convolution, and the gadget that transforms a grid of measured features into a grid of higher level features is called a **convolutional layer**. By iterating the application of these convolutional layers (with each layer described by a *different* kernel), we obtain an hierarchy of higher level concepts for each grid position that in the end together hopefully describe the original input in a way such that it can easily be reasoned about, for example for the purpose of classifying the input as containing either a cat or a dog in the case of RGB images. Each of these layers produces features at each position that are expressed in terms of a neighborhood of features from the previous layer, and these are themselves obtained from their respective neighborhoods in their own previous layer. When we apply a 3-by-3 kernel to the positions in the first layer, and when we then also apply (a different) 3-by-3 kernel in the second layer, the output features of this second layer at each position contain information that was spread out over a 5-by5 grid in the original input. Adding a

third 3-by-3 layer would correspond to obtaining features that are derived from information originally located in a 7-by-7 grid around the initial source locations. Adding more convolution layers thus increases the *receptive field* of the network, which means that the information that becomes available for each grid location at each layer is of an increasingly more global nature.

In practice, the original resolution of the input grid is usually lowered substantially in between the layers (for example by *pooling* techniques that combine multiple source locations into a single target location). This is done to reduce the total number of parameters within the combined network, to create an information bottleneck that forces the network to distill only the relevant information, or just so that the final number of output features becomes more manageable.

For our own architecture, we want to instantiate this basic idea of iterated local neighborhood information aggregation to the source code domain. As we mentioned in the previous section, we've adopted a graph based representation for our programs. The main idea here that an image is just very specific type of graph, and by expressing the previously described grid-based convolutional architecture in terms of general (multi-)graphs we might obtain an architecture that can obtain a global view of the whole (graph-representated) program by iterating a number of convolutional layers that each individually just summarize the ways the information at each node is related to the information available at each of the directly connected nodes. At the final layer, we hope that enough local and non-local information has accumulated at the unknown nodes such that the labels (i.e. identifier names) for these nodes can be accurately predicted.

In the original grid-based convolutional architecture, we can interpret an information grid as a (directed) graph, where the nodes in the graph are obtained from the grid positions and the edges between the nodes are obtained from the various adjacency relations between the nodes. Each grid location $(x, y)$ induces a node $Z_{xy}$, and two nodes are connected by a labeled directed edge precisely when they are directly adjacent in the grid in one of the 8 cardinal and intercardinal directions. The labels on the edges record these relative orientations, so that the applied kernels can still differentiate between the various spatial configurations. A 3-by-3 kernel can now easily be lifted to apply to these grid-induced graphs. The kernel applied to the neighborhood around some node $Z_{xy}$ would then use the information on the connecting edges to consistently align its input from the adjacent nodes so that it always uses the same input neurons for digesting neighbors of the same adjacency type.

## 4.3 The Problems of Generalization

The graphs we use to represent our programs are a bit less structured than those induced by the rectangular grids of the original setting. Instead of having precisely 8 different possible orientations, we now have as many 'orientations' as the number of possible relations between identifiers in the original source code. Although we have a great number of *possible* relations (a subsequent chapter will show that the training set contains around 90 thousand of them), we still have that most nodes are actually connected to only a few others. Thus the adjacency relation is very *sparse*, and this means we can't just allocate input neurons for each possible adjacency type like we did in the original grid-based architecture, because then almost all kernel input positions would not receive any input for most neighborhoods. Another difference of course is that we are now working with multi-graphs, so neighboring nodes can actually be adjacent in multiple distinct ways.

Although these differences in graph topology are certainly non-trivial, there is a more immediate problem that prevents us from directly applying the convolutional setup to our graph-represented source code. In the original grid-based setting, the available information is always represented by a number of real numbers at each grid position. When we're working with input that consists of RGB images, each grid location contains real-valued pixel intensities for the red, green and blue color channels. When applying a convolutional layer to this input, the output of the layer is an array of real numbers, with each real number representing the degree in which some specific measured feature is present at that location. The convolutional $D$-by-$D$ kernels discussed in the previous paragraph are implemented via a small neural network that initially just takes a (signed) weighted average of the measured features in a neighborhood. Because both the input RGB intensities and the output features are all represented in the same way by an array of real numbers, there is never an impedance mismatch between the representation of the available information at a certain layer and the representation of information that is expected by a subsequent consuming layer. Also, the averaging of various real number features in a neighborhood is meaningful precisely because each real number represents the degree in which some measured feature is present or absent. Measuring a rescaling of these measured features by some factor $\lambda > 1$ has the interpretation that the presence of this feature has increased, while a rescaling by a factor $\lambda < 1$ has the

interpretation the the presence of this feature has decreased. The weighted averaging of a number of individual measurements can be interpreted as measuring to what degree a specific combination of these factors co-occur within the original input.

For our source code-derived knowledge graphs, however, none of the available information is directly represented by an array of real numbers. Each node is labeled by an identifier name, which to the neural network is nothing more than a structure-less string of characters. Of course, we could encode each character by some real number, and then we would indeed obtain an array of real numbers for each character string, but this encoding would be totally arbitrary, and the resulting 'features' would most definitely not have the interpretation of being measurements of the degree of presence of some (relevant) property. Rescaling the presence of the character `d` has no semantic meaning that the network can use to adjust its prediction. The similarities between the spellings of the words `danger` and `ranger` are (as far as we can tell) purely accidental, and we would not want our network to see the latter word as a more intense (on the first position) version of the former word; the arbitrary encoding of the labels introduces superfluous patterns in the resulting feature encodings that have no physical meaning.

A more meaningful way to feed the identifier names to the neural network would be to encode each unique label as an array of real numbers using a *one-hot encoding* mechanism that assigns a feature vector to each label without introducing any arbitrary structure or patterns in the process. The one-hot encoding reserves a dedicated coordinate axis for each unique label. Each discrete label is then represented by a coordinate vector that is 1 on its own allotted axis while it is 0 on every other coordinate axis. This encoding makes no further assumptions on any possible relations between the different labels, and as such it is completely safe; rescaling the encoding of a specific label by some factor only signals that this precise label is now more or less present, without it making any unwarranted claims on the presence of any of the other labels.

Unfortunately, this encoding requires each label to live in a vector space of total dimensionality that is equal to the total amount of labels that are taken under consideration, which will be about 70 thousand for our training set. With this representation, each node within a neighborhood would contribute around 70 thousand features, all of them being zero except for one. Assigning a dedicated coordinate axis to each label seems a bit wasteful. What we actually want is an encoding of our labels as vectors in some relatively low dimensional vector space in a way such that each coordinate offers a maximal amount of information on the semantics of the label while the overall encoding induces a minimal amount of additional structure between the labels that was not already there to begin with. Of course, using significantly fewer dimensions than there are labels in the vocabulary forces most labels to share some of their representation (which is a good thing when these labels are actually semantically related to each other), but we want the labels to be void of any mutual relations that are induced by our own design choices as much as possible. The only relations that should exist between the various labels are the semantic relationships obtained directly from the data. In the next section we will build towards a method that lets us use our training set to dictate the most efficient label encoding under the constraint that the vector space that houses these labels has a fixed dimension $d$.

## 4.4 Embeddings of Discrete Data

The encoding of a set of discrete labels as vectors that live in a fixed finite dimensional real vector space is called an **embedding**. As we mentioned in the previous section, there are many such encodings possible, but not all of them are equally desirable. One has to be careful to not let the chosen encoding induce relationships between the embedded labels that are purely artifacts of this specific encoding, because otherwise the prediction architecture has to go through hoops to workaround these non-existing connections. For our architecture we've chosen to adapt the `word2vec` (Mikolov, Chen, et al., 2013) embedding mechanism that has recently become very popular in the area of natural language processing. This mechanism was recently also used in the `code2vec` framework (Alon et al., 2019) for predicting method names in source code from syntactic constructs extracted from the abstract syntax trees of their respective method bodies. In this section we will first introduce the main `word2vec` embedding mechanism. Our own adaptation of this mechanism re-uses some key contributions of the `code2vec` paper that will be described in the next section.

As we mentioned, the goal of the general `word2vec` mechanism is to obtain a high quality encoding of discrete labels as vectors in some finite dimensional vector space. In the original paper (Mikolov, Chen, et al., 2013), `word2vec` was used to obtain embeddings of dictionary words of some natural language, with the goal of the embedded words being used as features in text related prediction tasks. The main idea behind obtaining the encoding is that the meaning of words is mostly related to how they are typically used in combination with

other words. When removing a single word from a non-trivial sentence, humans can usually reconstruct the missing word with pretty high accuracy. Figure 4.4 gives a small illustration of this phenomenon. Apparently the meaning of the missing words is somehow shared with the meaning of the words surrounding it, and this would make it not seem unreasonable that the encoding of the missing word shares some representation with the encodings of its surrounding words. Going back to the idea of letting the data dictate the encoding, the idea implemented by `word2vec` is to obtain a *maximally shared representation* between a word and all of its context words *over all words* in the whole dataset. When the representations (i.e. real vectors) of the context words are known, the best word to place at the center position is the word with the representation closest to the overall representation of the context. Of course, we still need to specify how we quantify the degree in which representations are shared, but once this is done, we can simply formulate the embedding problem as finding the assignment of coordinates to each word such that the average representation sharing between each word and its context words is maximized when considering all words in the dataset. Neither the measure that decides the agreement between representations, nor the choice of the number of context words to use, nor the size of the embedding vector space are uniquely determined, but overall, the idea of letting the data determine the (necessary) overlap between word representations, by forcing similar words (according to the contexts in which they are used) to have similar representations, seems much more principled than any manual encoding one could come up with.

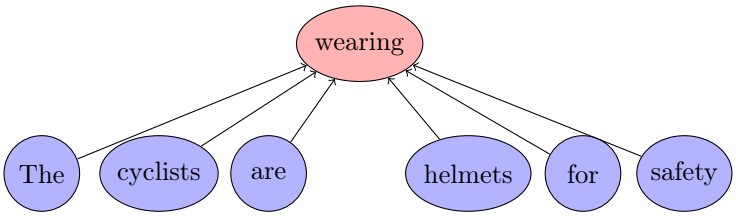


Figure 4.4: The word 'wearing' is semantically related to the words before and after it. A neural network is trained to optimize the coordinates of the words by making it use the coordinates of the words in the context to predict the word at the center. By using a large dataset of semantically rich sentences, the coordinates are organized such that related words are positioned close together.

The way `word2vec` actually quantifies how much representation is shared between (the encodings of) the context and the center word is via an application of a joint soft-max operation to the dot product between the averaged context encoding and the encodings of each of the words in the dictionary. This results in a probability distribution over the dictionary, and the probabilities assigned to the words are to be interpreted as measuring the degree in which the representations of each of these words overlap with the surrounding context. The goal is then to optimize the coordinates assigned by this dictionary to each word in such a way that the probabilities calculated in this way for the actual center words are generally as large as possible. `Word2vec` employs a neural network in an attempt to perform this optimization. The network is configured to minimize the negative log-likelihood of the calculated probability distributions with respect to the center labels in the dataset.

## 4.5 Graph Neighborhood Embeddings

In the previous section we've discussed a way of encoding discrete atomic words as real vectors such that the algebraic relations between the vectors correspond as closely as possible to the semantic relations between the usages of the words in the dataset. Initially, the words were mere *symbols* that did not have any internal structure, and as such there was no way any of the words could be mutually compared. Either two words are exactly the same or they are completely different. By training a neural network to assign vectors of real numbers to these symbols such that vectors assigned to contextual words become good predictors for their respective center words, we obtained a continuous representation for these words that actually has meaningful internal structure. Perhaps more importantly, the structure induced on these representations was controlled to be precisely the internal structure needed for the contextual words to be good predictors for their centers on the dataset, without implicitly imposing any additional assumptions on the representation except for a fixed context size and the very general hypothesis that angle measurements are all that is needed to differentiate between various degrees of dissimilarity between the labels.

For our architecture, we'd like to adapt this basic idea to the knowledge graph setting. In this setting, we're now dealing with labeled nodes that are surrounded by a context of directly connected labeled nodes. The connected nodes are connected via an edge that is itself also labeled, and our goal now is to find an encoding of the edge and node labels as real vectors in such a way that the averaged encodings of the labels on the connected edges and nodes become good predictors for the node label at their center.

In the previous section, the context of a center word $Z$ was obtained by looking at the $R$ words directly to the left of the center word and by looking at the $R$ words directly to the right of the center word. For example, in figure 4.4, $R$ was chosen to be 3, and the three words `the cyclists are` together with the three words `helmets for safety`, together seem to make `wearing` likely to be sandwiched inbetween. In our graph setting, we can let the various connected edges take over the roles of left and right. Instead of knowing to look at a static total of two sides, we now have as many 'sides' as there are edges connected to a center node $Z$. We can even generalize further and let the number of outgoing non-cyclic paths of length $R$ determine the number of sides to $Z$, where each such path of length $R$ contains precisely $R$ labeled edges and $R$ labeled nodes, in addition to the center node $Z$. When we remove the start node $Z$ from these paths, we obtain strict chunks of context related to $Z$. In this rest of this document, these chunks of context will be referred to as the **context rays** *of radius-$R$ originating at $Z$*. Figure 4.5 shows a fragment containing a center node together with three radius-2 rays. Each ray consists of $R = 2$ blocks at distances $r = 1$ and $r = 2$ from the center node $Z$. Each block contains precisely one edge and one node. The block containing the $X$ node and edge is shared between two rays.

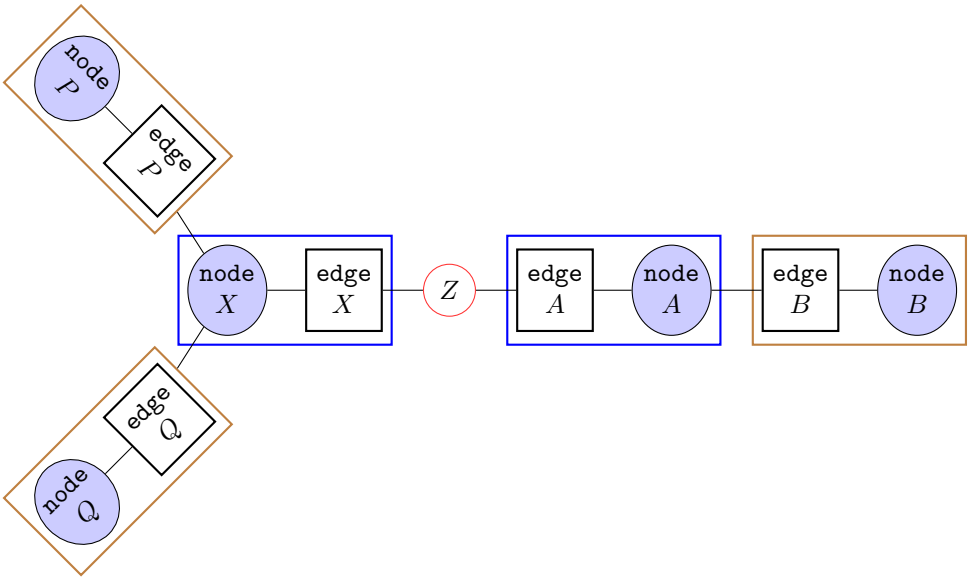


Figure 4.5: Center node $Z$ shown with three connected radius-2 rays. The left blue block is shared between two distinct rays.

The context rays originating at $Z$, together with their center node $Z$ will constitute the neighborhoods surrounding $Z$, and these neighborhoods will be the graph equivalents of the 3-by-3 and 5-by-5 grids that were used as neighborhoods around a center position in the original grid setting.

In principle, we can now directly apply the `word2vec` setup to this setting. The only differences we need to accomodate for is that we now have a variable number of context rays that we need to average over, and that we have two different kinds of 'words' instead of just one, namely the edge labels (source code features) and node labels (identifier names). Because we have these two semantic categories, we will keep track of two disjoint dictionaries; the encodings obtained for either category will live in different vector spaces that need not share their dimensionality. Figure 4.6 illustrates the prediction problem using a center-context combination obtained from the example used in the introduction chapter. Here the context size $R$ was chosen to be 1 (we only look at the directly connected edge/node pair) and the number of context rays used for this particular center node is 6.

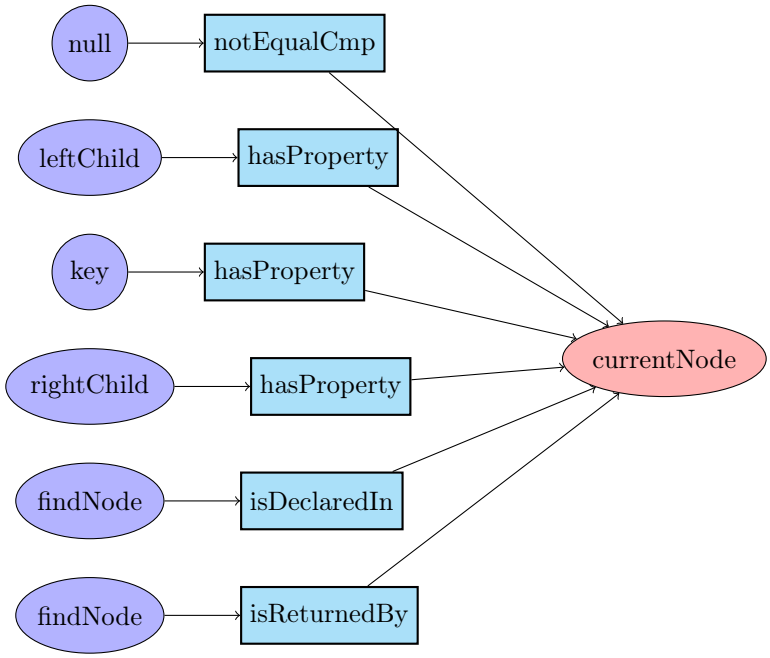


Figure 4.6: The label in the single node in the right column is predicted using the labels on the nodes and edges in its direct neighborhood which are shown in respectively the left and middle columns. Again the neural network is forced to assign good coordinates to these labels such that the coordinates of the labels in the context become good predictors for the coordinates of label in the center.

## 4.6 Averaging Neighborhoods with Attention

The original `word2vec` implementation used a uniform averaging of the various context pieces. Given that each word always has precisely two sides, we always have a total of $2 \cdot R$ different embedded labels that constitute a context (in this chapter we will keep on ignoring the details of dealing with contexts that have fewer than $R$ words available to them). When we average the embedded context labels, we just add together their respective embeddings and divide the total by $2 \cdot R$. This is a fair normalization that makes sure that each of the $2 \cdot R$ labels makes an equal contribution to the overall context. Unfortunately, as the total number of labels $R$ used at each size increases, the actual difference a single piece makes becomes negligible in the grand sum. In our application, the number of context rays is at least as large as the number of edges connected to the respective center node. Because the node degree is not bounded, the total number of labels from the context rays averaged over is also not bounded. So even if we keep our number of context rays $R$ relatively small, the varying and unbounded node degrees in the graph might still cause an overwhelming amount of information to enter the pool. When there is a lot going on in a particularly large neighborhood, all these different kinds of information might result in a very uninformative average.

In their `code2vec` (Alon et al., 2019) implementation, the authors adopted `word2vec` for predicting class method names from syntactic elements obtained from the abstract syntax tree of the method body. Just like in our application, these syntactic elements are all treated symbolically (i.e. they have no internal structure), and their use of `word2vec` allows them to construct vector representations for these context elements such that the algebraic relations between these vectors represent the semantics of these elements as they are seen in the source code. Similar to our situation, the number of context elements extracted from their method bodies is not bounded, and so they also have the problem that the naive uniform averaging of the individual context elements might destroy valuable information.

In their paper, Alon et al., 2019 instead employ an **attention** mechanism to calculate the usefulness of each of the context elements to the final prediction. The attention mechanism calculates for each context element a weight that determines the importance of that particular element of context. The calculated weights are all positive and their total sum is normalized to be 1. Instead of taking a uniform average over the embedded context elements, the calculated weights are now used to take a convex combination of the individual embedded

elements, with each element only contributing to the final result in accordance to the weight that was calculated for it. The weights themselves are calculated via a so-called **attention vector** $\vec{\alpha}$ that lives in the same vector space as each of the embedded context elements $\vec{c}_i$. The weight calculated for a context element $\vec{c}_i$ is directly related to the angle the vector $\vec{c}_i$ makes with the reference attention vector $\vec{\alpha}$. The coordinates of the attention vector are learned parameters of the model, and the optimization mechanism is encouraged to jointly organize the embeddings of the context elements and the attention vector in such a way that the differences between the calculated angles of the individual context elements with the attention vector is indicative of the relative importance of those elements. The attention mechanism allows the neural network to learn which of the context elements make important contributions to the final prediction and which context elements are just noise that should be canceled out. Thus, even though the number of context elements in a neighborhood might be arbitrarily large, the attention mechanism can learn to fade all unimportant, irrelevant or contradictory elements such that only the remaining relevant elements contribute significantly to the end result.

## 4.7 Graph Convolutions

In the previous sections we saw how a neural network can be used to encode discrete labels as vectors in some real vector space by forcing it to efficiently organize the coordinates in such a way that the coordinates of the context labels become good predictors for the coordinates of the center label. Once we've obtained these embeddings, our goal is to employ small neural network kernel that uses the (now continuously represented) information in the neighborhoods to extract a number of quality features that efficiently summarize what is going on in these neighborhoods. The neural network is evaluated at, for example, the radius-1 neighborhoods surrounding each graph node $Z$, and the features generated by this network when evaluated around $Z$ are subsequenly stored in $Z$ itself. Thus originally each node $Z$ only contains the (embeddings of) the labels that were assigned to it, but after the first convolutional layer it will also contain the features generated by the first layer from the information in the the neighborhood surrounding $Z$. If we would add an additional layer on top of this first layer, this second layer would be able to generate features for a node $Z$ using a neighborhood of nodes that now each have both the original label embeddings as well as the generated features from the previous layer available to them. Thus, indirectly, the features obtained from this last layer would contain information about the radius-2 neighborhoods of the original input graph. We see here again that the *receptive field* of the network increases by adding additional layers. Stacking $L$ radius-1 layers results in nodes containing information from radius-$L$ neighborhoods. The goal of this architecture of course is to accumulate enough information at each of the *unknown* nodes $Y$ such that this information useful in predicting the right identifier name to assign to the identifier represented by the node $Y$.

## 4.8 Training Objective

Of course, the parameters of the layered architecture described in the previous section still need to be tuned so that the features generated by the evaluations of the neural network kernels at the various neighborhoods at the various layers eventually lead to quality features that serve as good predictors for the labels of the unknown nodes $Y$ that we are actually interested in. We've already described how we obtain the coordinates for the label embeddings by training a shallow neural network to optimize the dictionaries such that the coordinates of the labels in the contexts are good features for predicting the labels in the respective center positions.

Ideally, all layers in the proposed architecture would be trained at the same time, such that the impact of changing a parameter in the lower layers is directly measurable in the output of the higher layers. This is done in the original grid-based setting, where the direct connections between the different layers allows the gradient of the final loss function to flow back all the way into the lower layers. Here, the stochastic gradient descent optimization algorithm can tune all layers at the same time, which allows it to make tweaks to the outputs of the lower layers if it can measure a beneficial response at the higher layers. Unfortunately, the irregular graph structure of our data prevents us from optimizing all layers at the same time. In contrast to the grid-based setting, we don't have a constant number of nodes in our input graphs, and each of these nodes also doesn't have a (non-trivial) bound on the number of adjacent nodes that it is connected to. Thus, the evaluation of the kernel neural network on a neighborhood requires a variable number of inputs, and each of these inputs themselves might also be generated by the application of a kernel neural network in the layers before. The total number of evaluations of the individual kernels is thus totally dynamic, and unrolling the layers such that

the outputs and inputs are all aligned in a single network would result in an enormous neural network whose topology would need to be re-instantiated for each concrete program.

So unfortunately, training all layers at once is not an option. For our architecture we've decided on a layer-by-layer approach where we mimic the way our semantic embedding layer was trained. Looking back at this first layer, we already see that this layer delivers a number of features in the form of the attention-averaged context rays from which it is trained to accurately predict the center label. These features, together with the embeddings of the context ray labels from which they originate, are already optimized to efficiently represent the neighborhood information that is relevant for predicting the label. For the higher layers, we repeat this training objective. The neural network kernel used at each layer is trained to use (a configurable subset of) the features generated by the lower layers to generate features that are useful in predicting the center labels of the neighborhoods on which the kernel is evaluated. The kernel itself is now (at the higher layers) not restricted to be a shallow single-layer neural network, but it can be as deep as is required to adequately refine the features gathered from the lower layers in the neighborhood to make the final prediction. Thus the lower layers are evaluated layer-by-layer until all features obtained from previous are available for all nodes $Z$, and then the kernel of the current layer is trained to predict the center label for each node $Z$ using the selected subset of features from these previous layers. The kernel is itself a fully connected feed forward neural network with possibly many layers, and we use the responses it generated at its final layer, just before making its center label prediction, as the designated output features of this layer.

In the end, all our convolutional layers are individually trained as classifiers that assign labels to the various nodes in the input graphs using information that was generated by any previous layers. We can compare the prediction accuracies between the internal layers to find the optimal balance between the extra time needed to train and evaluate each of the layers and the increase in prediction accuracy that (hopefully) comes with it.

# Chapter 5

# Architecture

In the previous chapter we've explained the various ideas that constitute the foundations of our architecture. In this chapter we will flesh out these ideas into a fully fledged architecture. The architecture presented here consists of a number of interconnected components and of the various sorts of data that flows through them. In the first section ('Discrete Data Types'), we will give a typed specification of the initial neighborhood data that we need to extract from the knowledge graphs to serve as input to the convolutional kernel. This discrete data is pulled through the initial embedding layer to be transformed into a continuous representation that will be used throughout the rest of the kernel. The second section ('Continuous Data Types') will describe the continuously represented variants of these discrete data types that are obtained from this embedding layer. We follow in the next section ('Implementation') with giving a typed abstract implementation of our framework that describes the complete data flow starting at the initial discrete knowledge graphs and which ends at the final prediction layer. The last section of this chapter ('Configuration') finally discusses the gaps that still need to be filled to turn this abstract implementation into something that something concrete.

Our prediction framework consists of a number of convolutional layers. Each convolutional layer uses a single kernel that is evaluated on the neighborhoods surrounding the graph nodes $Z$. Each kernel is a neural network that itself also consists of a number of internal layers. To prevent confusion between the *convolutional* layers we've used thus far in this thesis, and the internal layers used inside the kernel of a specific layer, we now introduce the concept of **stage** to refer to layers in the sense of layers internal to a kernel, while we keep using the word **layer** to refer to convolutional layers. Each convolutional layer uses a specific kernel that is evaluated on neighborhoods, and each kernel is built out of a number of stages.

## 5.1 Discrete Data Types

We begin with defining the two label sets used to supply our graphs with meaning. The first set we'll define is the set of node labels that contains all unique identifier names that were found in the training set, together with two special labels `UNKNOWN` and `MISSING` that are used to respectively indicate out-of-vocabulary labels and placeholders for nodes that don't currently have a label assigned:

$$\texttt{NodeLabel} := \{\texttt{identifier names found in the training set}\} \cup \{\texttt{MISSING}, \texttt{UNKNOWN}\} \tag{5.1}$$

The final prediction made by the neural network kernel for a single neighborhood is a probability distribution over this label set that indicates for each label in the node label set how confident the network is in that label being the correct label. The current design of our neural kernel only attempts to predict labels from this set, and it can only take input labels from this set, so our model is very much dependent on the diversity of the labels that were found in the training set.

Similar to the node label set, we also have a specific set containing all labels that were assigned to the edges in the training set. Again, our model can only deal with edge labels that are in this set; any labels that are found on knowledge graphs that are not in this set are replaced by the `UNKNOWN` placeholder. Edge labels correspond to various syntactic and semantic relations that were found in the JavaScript source code by the knowledge graph feature extraction program `UnuglifyJS`. The set of edge labels is defined as follows:

$$\texttt{EdgeLabel} := \{\texttt{edge features found in the training set}\} \cup \{\texttt{MISSING}, \texttt{UNKNOWN}\} \tag{5.2}$$

The actual architecture will be given in terms of a number of high level functions that together express the relationships between the original neighborhood input and final predicted output. The type of data consumed by each layer or stage will differ from layer to layer and from stage to stage, depending on the exact configuration of the framework. We've chosen to adopt an informal but mostly precise type system to make this dependency of the concrete types of the various components and pieces of data on the user configuration explicit.

We now begin by defining the type of information that is available at the center position when evaluating the kernel of layer $l$ on a neighborhood centered around a node $Z$ at stage $s$:

$$\texttt{CenterFeatures}(l, s) := \texttt{NodeLabel}^{K(l,s,0)} \times \texttt{CenterPool}(l, s) \tag{5.3}$$

Each stage $s$ in a layer $l$ can choose to add a user-definable number of $K(l, s, r)$ different candidate labels to its information pool at the nodes at distance $r \geq 0$ from the center:

$$\begin{aligned} &K(l, s, r) :: \mathbb{N} \\ &K(l, s, r) = \texttt{< user-defined number of candidates >} \end{aligned} \tag{5.4}$$

The value for $r = 0$ corresponds to the center node itself. For distances $r > 0$ from the center, we don't support a separate configuration for the number of candidates, so we can only configure for $s = 0$ here; we currently don't support a separately configured candidate configuration for the situation that both $r > 0$ and $s > 0$. The candidates are picked from the `TOP-K` predictions of either the baseline model or any of the preceding layers (this also configurable per layer). The $\texttt{CenterPool}(l, s)$ type contains a user-configurable selection of feature maps generated by any of the previous layers or stages:

$$\texttt{CenterPool}(l, s) := \texttt{< user-defined information pool >} \tag{5.5}$$

This type can be configured to house any combination of feature maps obtained from any of the preceding stages, either in the current layers or in any of the previous layers. By default, each stage after the initial stage is configured to only source from the feature maps obtained from the directly preceding stage, which results in a simple linear neural network structure for the kernel, but in principle much more complex information flows are possible. Usually the feature maps generated by the final stage of a particular layer are used as extra features in the first stage of the following layer, but need not be the case in general.

In section 4.5 we informally introduced the notion of radius-$R$ context rays. In figure 4.5 we saw that these rays could be visualized as a chain of $R$ blocks, with each block containing precisely one node and one edge. Our kernels will eventually use the attention mechanism to average over (the embeddings of) the information in these rays, but before we come to this we first have to make explicit the exact information contained on them. Each layer $l$ can be individually configured precisely as to what information is read from the neighborhood at various distances from the center node. We first start by defining the raw context type that contains all context rays that are available for a particular node $Z$:

$$\texttt{ContextFeatures}(l, Z) := \{c_i \in \texttt{Ray}(l) \mid c_i \texttt{ originates from } Z\} \tag{5.6}$$

The type $\texttt{Ray}(l)$ contains rays that are directly extracted from the knowledge graphs, before any embedding has taken place. Rays are just the Cartesian product of the type of the individual blocks from which the rays are built:

$$\texttt{Ray}(l) := \prod_{1 \leq r \leq R(l)} \texttt{Block}(l, r) \tag{5.7}$$

The product is taken over all blocks that lie at distances $r \in \{1, 2, \ldots, R(l)\}$ from a center node. The radius $R(l)$ used for layer $l$ is a layer specific configuration option:

$$\begin{aligned} &R(l) :: \mathbb{N} \\ &R(l) = \texttt{< user-defined context radius >} \end{aligned} \tag{5.8}$$

The blocks $\texttt{Block}(l, r)$ at each distance $r$ from the center node can all be individually configured, and their type thus depends on both the layer $l$ and the distance $r$ from the block to the center. This block type is defined as:

$$\texttt{Block}(l, r) := \texttt{NodeLabel}^{K(l,0,r)} \times \texttt{EdgeLabel} \times \texttt{ContextPool}(l, r) \tag{5.9}$$

Each block always contains precisely one edge label, one node label, and possibly any combination of feature maps that were delivered by any of the previous layers. This last part is user-configurable, and the exact type $\texttt{ContextPool}(l, r)$ depends on precisely what information the user wishes to make available to the kernel for layer $l$ at the node located at distance $r$ from a center node:

$$\texttt{ContextPool}(l, r) := \texttt{< user-defined information pool >} \tag{5.10}$$

Usually, each context node will be configured to use the final stage feature maps generated from only the layer directly preceding it, but some of our experiments also make available the feature maps from all preceding layers.

We've now defined the types $\texttt{CenterFeatures}(l, s)$ and $\texttt{ContextFeatures}(l, Z)$ that together describe all information that can be available in a neighborhood surrounding a center node $Z$. Together they combine into a type that specifies precisely what input a kernel is expected to consume when evaluated around $Z$:

$$\texttt{NeighborhoodFeatures}(l, s, Z) := \begin{cases} \texttt{CenterFeatures}(l, s) \times \texttt{ContextFeatures}(l, Z) & \texttt{if context is used at s} \\ \texttt{CenterFeatures}(l, s) & \texttt{otherwise} \end{cases} \tag{5.11}$$

Not every stage will want to make use of the information available in the context, and the branching shown here makes explicit this decision for a particular stage. As we can see, this type still depends on the specific center node $Z$ under consideration. Each kernel expects a (possibly padded) fixed amount of input data, and the variable number of rays attached to each center node poses a problem. Also, most of the data described by these types is still discrete, with the neighborhood data consisting mostly of node and edge labels. In the next section we will describe how the discrete values of these types can be 'lifted' to their embedded equivalents, and here we will also get rid of the dependency of the neighborhood features type on the center node $Z$ by forcing an upper bound on the number of context rays that participate in a single evaluation of the kernel.

## 5.2 Continuous Data Types

The previous section described the neighborhood data available to the neural network kernel when evaluated at a neighborhood around a specific center node $Z$. These data types are (at least partially) discrete and the values that live inside them are not directly amenable to be consumed by the kernel. In this section we will describe corresponding fully continuous data types that will house the embedded vector representations of there respective discrete originals. The continuous variants or their respective discrete counterparts are denoted by a superscript $\bigstar$. Usually the continuous variant of a discrete datatype is obtained by a recursive application of the $(\cdot)^\bigstar$ operator on the individual component types.

We first define the type in which the node labels are embedded:

$$\texttt{NodeLabel}^\bigstar := \mathbb{R}^p \tag{5.12}$$

This is just the ordinary $p$-dimensional real vector space, where $p$ is user-configurable and is constant over all layers:

$$\begin{aligned} &p :: \mathbb{N} \\ &p = \texttt{< user-defined node vector space embedding dimension >} \end{aligned} \tag{5.13}$$

We have a similar type for edge labels:

$$\texttt{EdgeLabel}^\bigstar := \mathbb{R}^q \tag{5.14}$$

Again, the vector space dimension $q$ is a global configuration option:

$$\begin{aligned} &q :: \mathbb{N} \\ &q = \texttt{< user-defined edge vector space embedding dimension >} \end{aligned} \tag{5.15}$$

The type of embedded center features are obtained by the embedding of the individual discrete candidates:

$$\texttt{CenterFeatures}^\bigstar(l, s) := \left(\texttt{NodeLabel}^\bigstar\right)^{K(l,s,0)} \times \texttt{CenterPool}(l, s) \tag{5.16}$$

The user configurable $\texttt{CenterPool}(l, s)$ type was already continuous and as such it is left untouched from the original in the previous section.

For the type of context features we break the pattern of simply applying the $(\cdot)^\bigstar$ operator to the individual components. The final representation of the context is obtained from attention-averaging the embeddings of the individual context rays. The continuous vector space where this averaging happens and where the final result lives is defined as follows:

$$\texttt{ContextFeatures}^\bigstar(l, s) := \mathbb{R}^{d(l,s)} \tag{5.17}$$

As opposed to the original, discrete, type of context features, this continuous type has lost its dependence on the center node $Z$. The dimension of this vector space again is user-configurable:

$$\begin{aligned} &d(l, s) :: \mathbb{N} \\ &d(l, s) = \texttt{< user-defined dimenson of vector space that embeds the context >} \end{aligned} \tag{5.18}$$

The attention-averaging is performed separately for each stage $s$ that wishes to source from the context, and each individual stage can specify how many features it wants to extract from the context presented at layer $l$. This is done through an intermediate stage-specific neural layer that summarized the features on a single ray into a $d(l, s)$-dimensional vector.

The final continuous pool of information available at stage $s$ when evaluating the kernel belonging to layer $l$ on the neighborhood surround a node $Z$ is now typed as follows:

$$\texttt{NeighborhoodFeatures}^\bigstar(l, s) := \texttt{CenterFeatures}^\bigstar(l, s) \oplus \texttt{ContextFeatures}^\bigstar(l, s) \tag{5.19}$$

Again, notice that this type has no dependence on the actual center node $Z$.

What remains now is to describe the types of the individual embedded context rays before they are combined in the context summary type $\texttt{ContextFeatures}^\bigstar(l, s)$. Rays are embedding by embedding their individual blocks:

$$\texttt{Ray}^\bigstar(l) := \prod_{1 \leq r \leq R(l)} \texttt{Block}^\bigstar(l, r) \tag{5.20}$$

Each block is itself embedded by embedding the corresponding edge and node labels:

$$\texttt{Block}^\bigstar(l, r) := \texttt{NodeLabel}^{\bigstar K(l,0,r)} \times \texttt{EdgeLabel}^\bigstar \times \texttt{ContextPool}(l, r) \tag{5.21}$$

The type $\texttt{ContextPool}(l, r)$ corresponding to the user-configurable extra information is already continuous, and as such it remains the same as in the previous section.

## 5.3 Implementation

### 5.3.1 Assumptions

In the previous sections we've described the various kinds of data our implementation as to deal with. In this section we will use these types to give a high level pseudo-implementation of our architecture. We start by assuming the existence of functions that can be used retrieve the original discrete labels from the various nodes in the current neighborhood. The $k$-th candidate label available at the current node $Z$ is retrievable via the `candidate_node_label` function:

$$\texttt{candidate\_node\_label}(l, s, Z, k) :: \texttt{NodeLabel} \tag{5.22}$$

When the $k$-th candidate is unavailable the `MISSING` stand-in label is returned. We assume all context rays in a neighborhood are indexed by an index $i \in \mathbb{N}$. Similarly, the $k$-th candidate labels on context positions at distance $r$ from the center node $Z$ on ray $i$ are retrieved by the `context_node_label` function:

$$\texttt{context\_node\_label}(l, Z, i, r, k) :: \texttt{NodeLabel} \tag{5.23}$$

The `context_edge_label` function returns the edge label of ray $i$ at distance $r$ from $Z$:

$$\texttt{context\_edge\_label}(l, Z, i, r) :: \texttt{EdgeLabel} \tag{5.24}$$

We also assume the existence of functions that retrieve the user-configured selection of input features that were generated by any of the previous layers or stages. The features available at the center position of the context surrounding a node $Z$ (i.e. on $Z$ itself) are retrieved by the `previous_center_features` function that is specific to configuration specifed for the current layer-stage combination:

$$\texttt{previous\_center\_features}(l, s, Z) :: \texttt{CenterPool}(l, s) \tag{5.25}$$

The features on the context positions at distance $r$ from the center node $Z$ on the ray $i$ are retrieved by the `previous_context_features` function:

$$\texttt{previous\_context\_features}(l, Z, i, r) :: \texttt{ContextPool}(l, r) \tag{5.26}$$

Finally, we assume efficient mappings between the discrete and continuous label types that are compatible with the coordinate optimizations performed by the neural network. In our actual implementation, these are primitives supplied by the Tensorflow framework. We first assume a dictionary mapping between the discrete node labels and their continuous encodings:

$$\texttt{node\_dict} :: \texttt{NodeLabel} \rightsquigarrow \texttt{NodeLabel}^\bigstar \tag{5.27}$$

Edge labels have their own dictionary:

$$\texttt{edge\_dict} :: \texttt{EdgeLabel} \rightsquigarrow \texttt{EdgeLabel}^\bigstar \tag{5.28}$$

Finally we need a way to lookup labels in their respective dictionaries:

$$\texttt{dictionary\_lookup} :: \forall \ell \,.\, \big(\ell \rightsquigarrow \ell^\bigstar\big) \to \big(\ell \to \ell^\bigstar\big) \tag{5.29}$$

### 5.3.2 Plumbing

In the previous subsection we've listed all the primitives needed to describe our architecture. We now continue with the actual implementation. First, the embedded $k$-th candidate label for node $Z$ is retrieved via a simple dictionary lookup of the label:

$$\begin{aligned}&\texttt{candidate\_node\_label}^\bigstar(l, s, Z, k) :: \texttt{NodeLabel}^\bigstar \\ &\texttt{candidate\_node\_label}^\bigstar(l, s, Z, k) = \texttt{dictionary\_lookup}\big(\texttt{node\_dict},\ \texttt{candidate\_node\_label}(l, s, Z, k)\big)\end{aligned} \tag{5.30}$$

The $k$-th node candidate labels on ray $i$ at distance $r$ from $Z$ are retrieved similarly via a dictionary lookup:

$$\begin{aligned}&\texttt{context\_node\_label}^\bigstar(l, Z, i, r, k) :: \texttt{NodeLabel}^\bigstar \\ &\texttt{context\_node\_label}^\bigstar(l, Z, i, r, k) = \texttt{dictionary\_lookup}\big(\texttt{node\_dict},\ \texttt{context\_node\_label}(l, Z, i, r, k)\big)\end{aligned} \tag{5.31}$$

The final type of dictionary lookup is performed to retrieve the similarly located edge labels:

$$\begin{aligned}&\texttt{context\_edge\_label}^\bigstar(l, Z, i, r) :: \texttt{NodeLabel}^\bigstar \\ &\texttt{context\_edge\_label}^\bigstar(l, Z, i, r) = \texttt{dictionary\_lookup}\big(\texttt{edge\_dict},\ \texttt{context\_edge\_label}(l, Z, i, r)\big)\end{aligned} \tag{5.32}$$

We can now directly obtain our final embedded center features by embedding the desired number of candidate labels:

$$\begin{aligned}&\texttt{center\_features}^\bigstar(l, s, Z) :: \texttt{CenterFeatures}^\bigstar(l, s) \\ &\texttt{center\_features}^\bigstar(l, s, Z) = \Big(\bigoplus_{1 \le k \le K(l,s,0)} \texttt{candidate\_node\_label}^\bigstar(l, s, Z, k)\Big) \\ &\qquad\qquad \oplus\, \texttt{previous\_center\_features}(l, s, Z)\end{aligned} \tag{5.33}$$

Here the '$\oplus$' symbol indicates the direct sum of the vectors on either side, i.e. the concatenation of the coordinates. Context embeddings are obtained via the attention-averaging of their embedded rays, and the embedding of a single ray is realized via the embedding of the individual blocks that constitute the ray:

$$\begin{aligned}
&\texttt{embedded_block}(l, Z, i, r) :: \texttt{Block}^{\bigstar}(l, r) \\
&\texttt{embedded_block}(l, Z, i, r) = \Big( \bigoplus_{1 \leq k \leq K(l,0,r)} \texttt{context_node_label}^{\bigstar}(l, Z, i, r, k) \Big) \\
&\qquad\qquad \oplus \texttt{context_edge_label}^{\bigstar}(l, Z, i, r) \\
&\qquad\qquad \oplus \texttt{previous_context_features}(l, Z, i, r)
\end{aligned} \tag{5.34}$$

The embedded ray is now just the concatenation of the blocks:

$$\begin{aligned}
&\texttt{embedded_ray}(l, Z, i) :: \texttt{Ray}^{\bigstar}(R(l)) \\
&\texttt{embedded_ray}(l, Z, i) = \bigoplus_{1 \leq r \leq R(l)} \texttt{embedded_block}(l, Z, i, r)
\end{aligned} \tag{5.35}$$

The original context rays contained labels originating from two distinct semantic categories, namely the category of edge labels and the category of node labels; their respective embedding were realized in two disjoint vector spaces that did not have any direct relations. To obtain a single representation of the context rays that combines these two different kinds of information, we now employ a single transformation that combines the information from the different spaces into a single vector space by the application of a single hidden layer:

$$\begin{aligned}
&\texttt{summarized_ray}(l, Z, i) :: \texttt{ContextFeatures}\bigstar(l, s) \\
&\texttt{summarized_ray}(l, Z, i) = \texttt{tanh}\Big( \big( \texttt{summary_layer}\big(l, \texttt{embedded_ray}(l, Z, i)\big) \big) \Big)
\end{aligned} \tag{5.36}$$

Besides combining the edge and node label categories, this transformation also allows the information at differently distanced blocks to be incorporated differently in the final representation. The hidden layer is `tanh`-activated, and the linear part is implemented via a matrix that transforms from the space where the ray embeddings live to the space where the final embedded context will live:

$$\begin{aligned}
&\texttt{summary_layer}\big(l\big) :: \texttt{Ray}^{\bigstar}(l) \to \texttt{ContextFeatures}\bigstar(l, s) \\
&\texttt{summary_layer}\big(l\big) = (w_l^{\dagger})_{ij}
\end{aligned} \tag{5.37}$$

The parameters of this matrix are found as part of the neural network training process. Each stage that decides to use features from the context gets its own dedicated `summary_layer` matrix. The final context embedding is now obtained via a weighted sum over the individual context rays:

$$\begin{aligned}
&\texttt{context_features}^{\bigstar}(l, s, Z) :: \texttt{ContextFeatures}^{\bigstar}(l, s) \\
&\texttt{context_features}^{\bigstar}(l, s, Z) = \sum_i \lambda(l, s, Z, i) \cdot \texttt{summarized_ray}(l, Z, i)
\end{aligned} \tag{5.38}$$

The weights used in this summation are obtained by comparing the summarized rays with the reference attention vector via a dot product. The calculated dot products are jointly soft-maxed to obtain weights for each of the context rays such that the rays that are the best aligned with the attention vector get the highest weights:

$$\begin{aligned}
&\lambda(l, s, Z, i) :: \mathbb{R} \\
&\lambda(l, s, Z, i) = \frac{\exp\Big(\big\langle \texttt{summarized_ray}(l, Z, i),\ \texttt{attention_vector} \big\rangle\Big)}{\sum_j \exp\Big(\big\langle \texttt{summarized_ray}(l, Z, j),\ \texttt{attention_vector} \big\rangle\Big)}
\end{aligned} \tag{5.39}$$

The attention vector is itself an array of learned parameters. Again, each stage that decides to use features from the context gets its own dedicated attention vector:

$$\texttt{attention\_vector} :: \texttt{ContextFeatures}\bigstar(l, s) \tag{5.40}$$

Now, the total feature vector available to the kernel of layer $l$ at stage $s$ for a given node $Z$ is given by combining the center and context feature vectors of the respective layers at the respective stages:

$$\begin{aligned}&\texttt{source\_features}(l, s, Z) :: \texttt{NeighborhoodFeatures}^{\bigstar}(l, s)\\&\texttt{source\_features}(l, s, Z) = \texttt{center\_features}^{\bigstar}(l, s, Z) \oplus \texttt{context\_features}^{\bigstar}(l, s, Z)\end{aligned} \tag{5.41}$$

The output feature vector of the application of this same kernel at stage $s$ is obtained by pulling these `source_features` through a $\sigma_l^s$-activated hidden layer:

$$\begin{aligned}&\texttt{result\_features}(l, s, Z) :: \texttt{NodeFeatures}(l, s)\\&\texttt{result\_features}(l, s, Z) = \sigma_l^s\Big(\texttt{linear\_layer}\big(l, s, \texttt{source\_features}(l, s, Z)\big)\Big)\end{aligned} \tag{5.42}$$

The activation function $\sigma_l^s$ used here is in principle configurable per layer per stage, but for our experiments we will always set $\sigma_l^s = \texttt{tanh}$. The linear part of this last hidden layer is again given by a matrix that differs from layer to layer and from stage to stage:

$$\begin{aligned}&\texttt{linear\_layer}(l, s) :: \texttt{NeighborhoodFeatures}(l, s) \to \texttt{NodeFeatures}(l, s)\\&\texttt{linear\_layer}(l, s) = (w_l^s)_{ij}\end{aligned} \tag{5.43}$$

The output of a specific layer-stage combination is contained in a dedicated vector space:

$$\texttt{NodeFeatures}(l, s) := \mathbb{R}^{f(l,s)} \tag{5.44}$$

The dimension of this vector space can again be configured on a per-stage basis:

$$\begin{aligned}&f(l, s) :: \mathbb{N}\\&f(l, s) = \texttt{< user-defined dimenson of vector space that houses the feature maps >}\end{aligned} \tag{5.45}$$

Finally, we obtain our predictions for the node $Z$ at layer $l$ by using the features obtained from the final stage of $l$ to calculate a probability distribution over the node label space:

$$\begin{aligned}&\texttt{predictions}(l, Z) :: \texttt{NodeLabel} \rightsquigarrow [0, 1]\\&\texttt{predictions}(l, Z) = \texttt{predict\_probabilities}\big(l, \texttt{result\_features}(l, \texttt{last\_stage}(l), Z)\big)\end{aligned} \tag{5.46}$$

The final `predict_probabilities` function is implemented by an additional linear layer from $\texttt{NodeFeatures}\big(l, \texttt{last\_stage}(l)\big)$ into the space $\texttt{NodeLabel}^{\bigstar}$ of embedded node labels, followed by a joint soft-max over the dot products between the predicted vector $\psi \in \texttt{NodeLabel}^{\bigstar}$ and the embeddings $\tilde{c}_t \in \texttt{NodeLabel}^{\bigstar}$ of each the node labels $c_t \in \texttt{NodeLabel}$:

$$\begin{aligned}&\texttt{predict\_probabilities}(l) :: \texttt{NodeFeatures}\big(l, \texttt{last\_stage}(l)\big) \to \big(\texttt{NodeLabel} \rightsquigarrow [0, 1]\big)\\&\texttt{predict\_probabilities}(l) = \lambda\, \psi\, .\, \left(c_t \mapsto \frac{\exp\Big(\big\langle \tilde{c}_t,\, \texttt{prediction\_layer}(l, \psi)\big\rangle\Big)}{\sum_j \exp\Big(\big\langle \tilde{c}_j,\, \texttt{prediction\_layer}(l, \psi)\big\rangle\Big)}\right)\end{aligned} \tag{5.47}$$

The linear layer is as usual implemented via a simple matrix whose parameters need to be learned:

$$\begin{aligned}&\texttt{prediction\_layer}(l) :: \texttt{NodeFeatures}\big(l, \texttt{last\_stage}(l)\big) \to \texttt{NodeLabel}^{\bigstar}\\&\texttt{prediction\_layer}(l) = (w_l^{\odot})_{ij}\end{aligned} \tag{5.48}$$

## 5.4 Configuration

Each stage $s$ of the kernel corresponding to layer $l$ delivers a number of features for the various nodes in the input graph that are captured by the type $\texttt{NodeFeatures}(l, s)$. Any subsequent stage $s' > s$ at some layer $l' \geq l$ for some context position $r' \geq 0$ can make use of these features by an adequate instantiation of its configuration types $\texttt{CenterPool}(l', s')$ and $\texttt{ContextPool}(l', r')$. Besides these choices in information availability, there are a number of other hyperparameters that need to be determined before we can actually use this framework. Assuming a complete implementation of the primitives that were left open in the previous section, we are left with the freely choosable hyperparameters shown in table 5.1. The next chapter will discuss the experiments we performed to verify the usability of our architecture. Each of these experiments will have been given a specific concrete instantiation for these hyperparameters.

| hyper parameter | meaning |
|---|---|
| $p :: \mathbb{N}$ | node embedding dimension |
| $q :: \mathbb{N}$ | edge embedding dimension |
| $R(l) :: \mathbb{N}$ | context rays size |
| $K(l, s, r) :: \mathbb{N}$ | number of candidates used |
| $d(l, s) :: \mathbb{N}$ | context embedding dimension |
| $f(l, s) :: \mathbb{N}$ | number of feature maps |
| $\sigma_l^s :: \mathbb{R} \to \mathbb{R}$ | activation function |
| $\texttt{CenterPool}(l, s) :: \texttt{Type}$ | configuration of center source features |
| $\texttt{ContextPool}(l, r) :: \texttt{Type}$ | configuration of context source features |

Table 5.1: Hyperparameters for the convolutional prediction architecture.

# Chapter 6

# Experimental Verification

In the previous chapter we've given a detailed description of a general architecture that can be used to perform attention-based graph convolution experiments on general knowledge graphs. To verify the utility of this architecture, we've created a general framework which can be used to construct a variety of experiments to test the architecture by simple acts of configuration. Using this framework, we subsequently designed and executed a number of experiments that try to probe the strengths and weaknesses of the architecture when it is used for the purpose of identifier name inference. These experiments fall into two distinct categories. The main research question of this thesis asks whether we can improve upon the existing conditional random field model by the use of a convolutional neural network architecture, and the first category consists of experiments that take the predictions made by the conditional random field as a given and then try to refine these predictions by using the convolutional architecture to get an improved sense of which of the original conditional random field predictions should actually be the top prediction. These experiments are discussed in section 6.9 ('Extension Experiments'). The second category consists of experiments that are designed to find out how our architecture performs when used as a self-sufficient model that doesn't have any candidate suggestions delivered by a pre-existing model. These experiments are discussed in section 6.10 ('Standalone Experiments').

Before we can discuss these experiments in full detail, however, we will first need to discuss the capabilities and configuration options that are available when designing experiments using this framework. This is done in section 6.3 ('Design Space'), which lists and motivates the available settings and which also gives default values for these settings when they are the same for all performed experiments.

## 6.1 Implementation

The framework implemented by us to verify the utility of our architecture has been made available online to aid future research in this area. Our framework is based on the Tensorflow-based implementation of `code2vec` (Alon et al., 2019) that was generously provided by the authors.

Besides the construction of our own framework, we've also made some small enhancements to two existing tools, `Nice2Predict` and `UnuglifyJS`, that are used by our framework. These enhancements were only made to streamline the integration of the respective tools in our own specific framework and as such they don't change the existing tools in any non-trivial way. Any possible future reimplementation of this architecture will probably want to revisit these adaptations, and as such we would recommend this implementation to abandon our changes and to directly checkout the original source code. Links to both our enhancement versions and the original versions of these programs are given in table 6.1.

| software | ours | original |
|---|---|---|
| conditional attention field | https://bitbucket.org/wlelsing/conditionalneurology | https://github.com/tech-srl/code2vec |
| adapted `Nice2Predict` | https://bitbucket.org/wlelsing/nice2predict | https://github.com/eth-sri/Nice2Predict |
| adapted `UnuglifyJS` | https://bitbucket.org/wlelsing/unuglifyjs | https://github.com/eth-sri/UnuglifyJS |

Table 6.1: Framework implementation.

## 6.2 Baseline Experiments

Before we can start with our own experiments, we first need to properly setup the baseline experiment to which all of our results can be compared. Besides providing these baseline accuracy results, the baseline model also serves to provide candidate predictions which will bootstrap our own model. Before we explain our candidate generation procedure, we will now first explain the configuration settings that are used by all our versions of the conditional random field classifier.

### 6.2.1 Configuration

The baseline conditional random field model ('`Nice2Predict`'[1]) was optimized on our training set using the default hyperparameter settings of the main training program (`n2p/training/train_json`). The most important hyperparameters are described in (Raychev et al., 2015), and we'll summarize their description here. The results in their paper were also obtained by using these precise settings. The rest of the available (hyper)parameters are unfortunately too technical to discuss here. We refer the reader to the linked source code for more information.

The main training procedure of the conditional random field model by Raychev et al., 2015 is based on *projected stochastic gradient descent*. This algorithm is in broad strokes exactly the same as the stochastic gradient descent algorithm used when training neural networks (see section 2.6.3). Just as is the case with neural networks, the gradient of some error function with respect to the model parameters and a minibatch of data points is calculated, and the (negative) direction of this gradient is used to slightly adjust the model parameters in the direction which would lower the calculated loss of this model with respect to these data points and this error function. The error function used by Raychev et al., 2015 in their the conditional random field model is the *Structured Hinge Loss* (see section 2.7.4)[2], and just as with any gradient descent algorithm there is a learning rate parameter $\alpha$ that controls how much of the gradient is actually used in the parameter adjustment.

The authors use the following learning rate regime on which they obtained their best results. The learning rate $\alpha$ is initially set to 0.1, and is halved at the end of each pass in which there was no observed improvement in accuracy on the training set. The training of the model is ended when either the learning rate drops below 0.0001 or when the limit of 24 epochs is reached.

The training of the conditional random field model also employs a regularization mechanism in an attempt to prevent the model from becoming overly adjusted to the peculiarities of the training set. They employ a regularization hyperparameter $\lambda$ that individually bounds each weight $w$ to lie in the interval $[0, \frac{1}{\lambda}]$. The larger $\lambda$ is set, the lower the allowed values for the parameters $w$ and the less power the model has to focus on any particular feature. They found the optimal value of $\lambda$ for their name inference experiments to be 2.0 in their paper, and we will stick to this value for our experiments.

The last hyperparameter we discuss here is the number of threads used by the model during training. The model is designed to compute the gradients over a number of data points in parallel, and then to combine the results by averaging these gradients at the end of each pass. Adding more threads increases the number of data points that can be handled in parallel, but it also prevents the training contributions of these data points to influence each other. Thus the final result of the training phase is influenced by the number of threads used, so the number of threads becomes an additional hyperparameter. For our experiments we were limited to a 16 core system which made us decide to allocate 16 threads to this model, but the experiments by Raychev et al., 2015 actually performed with 32 allocated threads. It is unclear to us to what extent this influences the final model accuracies. Our choice of hyperparameters for the conditional random field model have been recorded in table 6.2.

### 6.2.2 Candidate Extraction

The main purpose of our framework is to improve upon the existing candidate suggestions provided by an existing baseline classifier. We've chosen to implement this idea by allowing the neighborhood consuming convolutional kernels to have access to (the embeddings of) the full `TOP-K` (for some choice of $K$) candidates, in addition to information obtained from the averaged context rays in the neighborhood and any feature maps generated at the lower layers (if any). We preprocess our datasets by using an adequately trained baseline

[1] https://github.com/eth-sri/Nice2Predict/tree/9086fb6f8b318bd3ab0b7df434547da85165d475

[2] This is different from the negative log likelihood loss usually used to fit probabilistic models.

| hyper parameter | value |
|---|---|
| training method | ssvm |
| start learning rate | 0.1 |
| stop learning rate | 0.0001 |
| regularization const | 2.0 |
| cpu_count | 16 |
| epoch limit | 24 |

Table 6.2: Conditional Random Field hyperparameters.

model (see the subsection below) to generate candidates for each point of prediction. Unfortunately, whilst the conditional random field implementation of Raychev et al., 2015 keeps an internal list of scored candidates for each prediction point, the external interfaces only support the retrieval of the *combined label assignment* that maximizes the *joint probability* of the combination of all predictions over a *complete program*. This means that we only get a single (best) candidate per prediction point, instead of the complete list of (internally available) candidates for that point.

To obtain the actual internal candidate lists, we've extended the main loop of the batch evaluation program (`n2p/training/eval`) to store the internal candidate list of each program as soon as it is done with inference. The score obtained for a single candidate label $c$ at a specific unknown node $Y$ is equal to the maximum possible score the radius-1 neighborhood centered at $Y$ can contribute to the overall program score when all of the direct neighbors $Y'$ of $Y$ get assigned labels which are maximized in mutual consistency with respect *only* to the assignment of $c$ to $Y$ (by ignoring any other possible neighbors $Y''$ of the $Y'$). Thus each label candidate $c$ for a node $Y$ gets assigned a score that indicates its maximum potential contribution to the overall score when all direct neighbors of $Y$ are perfectly aligned with this assignment of $c$ to $Y$. The label assignments obtained from choosing the `TOP-1` candidate from each candidate list are thus all *locally* optimal, but they still might not be *globally* optimal with respect to the probability given to this joint assignment by the full conditional random field model. To obtain our final candidate lists, we find the globally optimal candidate (as given by the external interface discussed in the previous paragraph) in each of the candidate lists ranked on local optimality, and we then move all these to the `TOP-1` position. At this point we can make no further improvements to the rankings of all remaining (locally ranked) candidates; this is something we leave for our own convolutional neural model to figure out. A performance comparison between the locally and the globally optimized baseline classifiers is given in table 6.3. The definitions of the performance measures displayed in this table are discussed in section 2.2 of the preliminaries chapter.

| layer | epochs | precision | recall | $F_1$ | TOP-1 | TOP-2 | TOP-3 |
|---|---|---|---|---|---|---|---|
| `baseline-local` | 22 | 58.67 | 58.60 | 58.58 | 57.81 | 60.76 | 62.08 |
| `baseline-global` | 22 | 68.89 | 68.82 | 68.82 | 68.38 | 71.91 | 72.98 |

Table 6.3: Accuracy results for the Baseline Classifiers on the evaluation set (around 2 million predictions total).

### 6.2.3 Obtaining Representative Candidates

To obtain a representative list of candidates for our model to train on, we'd like to use the baseline model to generate these candidates for the *unknown* nodes in the training set. Unfortunately, we can't just use the baseline model trained on the training set to generate candidates for the training set, because the trained model has the tendency to somewhat memorize the correct answers on the set it was trained on, which causes it to perform much better on this set than it would on unseen data. Teaching our own *neural* model to refine candidates predicted in this way would be problematic because the skewed distribution of candidates could encourage the model to make unwarranted shortcuts, like for example placing too much trust in the first ranked candidate. Ideally, we'd have a second training set on which we can train a separate model that we can use to generate candidates for the actual training set, which would then have the right distribution, but unfortunately we don't have the data available to do this.

To still be able to obtain candidates with approximately the right distribution, we first note that we observed during our initial experiments that most of the accuracy of the conditional random field model can be obtained

from a relatively small amount of data. Training a model on a dataset of 5000 programs already gives us a classifier which attains prediction accuracy of around 55%, while training the model using the same settings on the full training set of 100.000 programs only increases this accuracy to around 70%. This observation led us to conclude that using half of the training dataset might actually result in a model that can attain a generalization accuracy close to the generalization accuracy of the model trained on the full dataset, but which isn't over-adapted to the data in the other half of the training set. We've divided the training set into two folds, $A$ and $B$, that each contain complementary halves of the training set. On each of the folds $A$ and $B$ we train the conditional random field model using the hyperparameters mentioned in the previous subsection, and then we use this model to generate candidates for the complementary fold. We've recorded our validation results on these folds in table 6.4.

| training set | validation set | validation accuracy |
|---|---|---|
| fold $A$ | fold $B$ | 70.32% |
| | fold $A$ | 84.68% |
| fold $B$ | fold $A$ | 70.31% |
| | fold $B$ | 84.11% |
| fold $A \cup$ fold $B$ | fold $A \cup$ fold $B$ | 84.39% |
| | actual validation set | 68.38% |

Table 6.4: Validation accuracies on parts of the training set with models trained on a complementary set.

We can see that, indeed, the validation accuracies of the models trained on fold $A$ (respectively $B$) and evaluated on the complementary fold $B$ (respectively $A$) results in candidates which are distributed much closer to the candidates generated by the full model on the validation set. We use the union of the candidate-augmented training folds $A$ and $B$ as our complete training set, whose overall candidate distribution is the average of the candidate distributions over the individual folds. The candidates on the validation and evaluation sets are obtained from the full baseline model trained on the complete training set.

## 6.3 Design Space

In this section we discuss the different pieces of configuration that are needed to fully describe a single experiment. Many of the configuration options mentioned here are shared between all of our experiments, and for these options the shared settings are recorded directly after the respective option is introduced.

### 6.3.1 Embedding Vector Space

The first part of our framework tries to assign good continuous coordinates to each unique discrete label so that these coordinates can adequately represent the label further down the pipeline. To do this, the framework needs to know how many coordinates to assign to each label, i.e. it needs to know the dimensions of the real vector spaces in which the labels will live. In the chosen knowledge graph representation, there are in principle two different kinds of labels which convey semantic information: the labels on the nodes, that correspond to assignments of names to the identifier represented by the nodes, and the labels on the edges, that correspond to specific relations between the nodes which were extracted from the original source code by the feature extraction program. These two types of labels have no direct connections and will normally live in different vector spaces, whose dimensions can be chosen independently. These dimensions are hyperparameters for our framework.

Besides these two options, there is a third type of vector space which can be used. Node labels (i.e. identifier names) serve two distinct purposes: they are both used as *features* found in a neighborhood that serve as input to the neural network, and as *target labels* which need to be correctly predicted by the neural network. In (Alon et al., 2019), the source and target labels live in two different semantic categories, the former being assigned to locally declared variables and the latter being the names of class methods. Here the authors create two separate embeddings for each node label: one embedding for when a label is used in the *input position* and one embedding for the case when the label is used in the *target position*. This distinction adds another possible hyperparameter which can be chosen freely. For our framework, however, we've chosen to use a shared embedding between usages of labels in the input and target positions. A shared representation reduces the number of parameters

of the final model, thus reducing training time and chance of overfitting. Semantically this is justified because for us the input and target labels do indeed belong to the same category.

All our experiments use the same choice of vector space dimensions. We've listed these choices in table 6.5.

| hyperparameter | value |
|---|---|
| edge dimension | 150 |
| node dimension | 150 |
| target dimension | `shared` |

Table 6.5: Embedding Dimension hyperparameters used in all experiments.

These dimensions are a bit higher than those used in `code2vec`, where both edge labels and node input labels were given vector spaces of dimension 128 to live in. On the other hand, node labels used in the target position were given $3 \cdot 128$ dimensions in the `code2vec` paper, which is much higher than our uniform choice of 150 for both the input and the target roles. Our reasons for choosing 150 dimensions are mostly accidental. Later offline experiments have not shown any noticeable difference in final validation accuracy, so we stuck to our initial choices.

### 6.3.2 Context Span

Each convolutional layer extracts for each node $Z$ in the knowledge graph a number of features from the neighborhood surrounding $Z$ via an attention-based averaging mechanism over the radius-$R$ rays originating at $Z$. The rays originating at $Z$ of radius $R \in \mathbb{Z}$ ($R \geq 1$) are precisely the non-cyclic paths in the graph from $Z$ to some other node $T$ containing precisely $R$ edges, with $Z$ itself removed from the path to obtain the ray. In other words, they are sequences of $R$ consecutive `edge-node` pairs, with each edge within a pair connecting the paired node with the node from the previous pair in the sequence. The edge in the first pair connects the pair to $Z$ itself. Figure 6.1 shows a node $Z$ with two connected radius-1 rays. The blue blocks constitute a single `edge-node` pair. Figure 6.2 shows a node with three connected radius-2 rays. Note the overlap between the two left rays originating at $Z$.

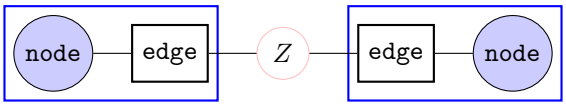


Figure 6.1: Center node $Z$ shown with two connected radius-1 rays.

For our experiments we need to decide on the values for $R$ to use at each layer. To obtain quality embeddings, it is necessary for the neural network to have as much contextual information available as possible, as the neural network tries to jointly optimize the embeddings of the context rays to become good predictors for the center label, and the more information it has, the better it should be able to decide which pieces of information are related and which are not. On the other hand, too much available information can also be problematic, as it might pollute the information pool and increase processing time.

In the previous subsection we've decided to set the embedding dimensions of both the nodes and the edges to 150. Concatenating the embedded node and edge labels in a radius-$R$ ray gives us that each ray is represented by $(150 + 150) \cdot R$ features when each ray contains just the information available in the original knowledge graph. This means that, in this case, each radius-2 ray supplies $d = 600$ reals of information while radius-1 rays each supply $d = 300$ reals of information. However, at higher layers we have more information available to us than just the information initially obtained from the knowledge graph: each ray can also contain any of the feature maps obtained from the previous layers for each node on the ray. To keep the amount of information available in check, we've chosen to only use radius-2 neighborhoods during the initial embedding phase, and use radius-1 neighborhoods in all subsequent layers. This choice keeps the amount of information supplied by each ray always between 450 and 750 reals for all of our experiments.

(Our use of) TensorFlow also requires us to choose up front the exact number of context rays $K$ which we take into consideration. When the number of rays $k$ in a neighborhood is larger than $K$, the excessive $(k - K)$ rays are thrown away and can't be used. On the other hand, when the number of rays $k$ in a neighborhood is smaller than $K$, the difference $(K - k)$ of missing rays need to be padded by dummy placeholder labels.

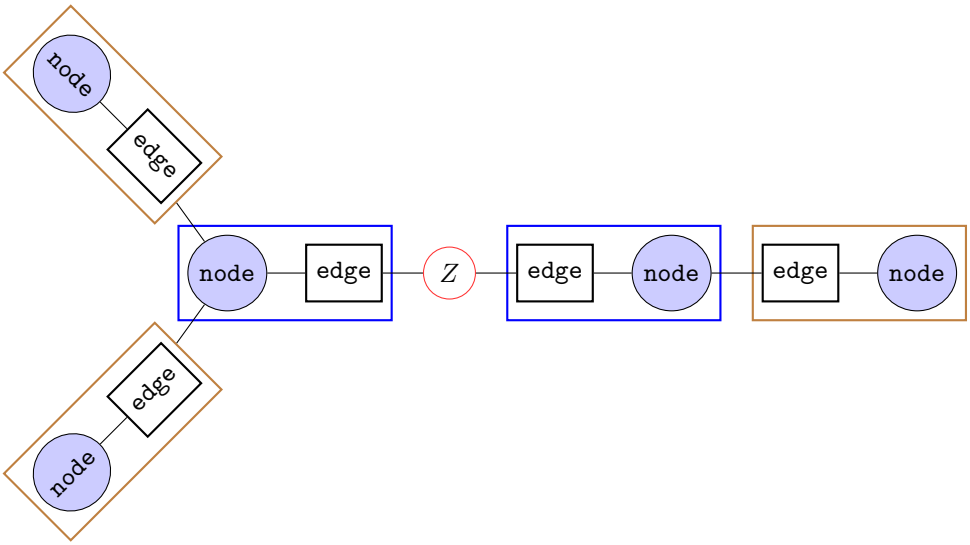


Figure 6.2: Center node $Z$ shown with three connected radius-2 rays. The left blue block is shared between two distinct rays.

Eventually these dummy labels are masked out to ensure that they do not add any unwanted information to the context, but for most of the pipeline they are still present which means that the total number of rays processed per context is always precisely $K$. Thus, choosing $K$ too small makes us lose valuable information while choosing $K$ too large makes us perform unnecessary calculations. A quick calculation over the training set has shown us that the average number of originating radius-1 rays is around 13 rays per node[3], while the average number of originating radius-2 rays is around 150 rays per node[4]. Of course, we use the attention mechanism to take a weighted average over all embedded rays in a neighborhood, so in the end the whole neighborhood is represented by a single vector of $d$ real numbers, independently of how many rays were extracted from the neighborhood.

The previous observations led us to choose a bound of $K = 50$ on the number of radius-1 rays and a bound of $K = 200$ on the number of radius-2 rays. With this choice, we have more than enough room allocated for all radius-$R$ rays in a neighborhood to be used in the average case, and there is also quite some room to spare to fully accommodate neighborhoods which are a bit larger. The choices made in this discussion are summarized in 6.6.

| layer | radius | ray count $K$ |
|---|---|---|
| 0 | 2 | 200 |
| $\geq 1$ | 1 | 50 |

Table 6.6: Neighborhood sizes used in all experiments.

Ideally we would also have performed some experiments with radius-1 neighborhoods during the initial layer-0 embedding, but unfortunately, we didn't have time any time for this.

### 6.3.3 Activation Functions

Activation functions are put in between the different layers in the neural network to allow the layers to learn non-linear effects. Without any non-linear activation functions, each neural network would effectively collapse into a single linear layer, because in that case the connections between the layers can be seen as direct matrix compositions, and the composition of two matrices can itself also be written as a matrix by simple linear algebra. Thus, to be able to learn interesting, non-linear patterns, we need to choose activation functions (see section 2.5.4) for each of our layers.

[3]When ignoring rare nodes of extremely large degrees which disproportionately skew this number.
[4]Again ignoring nodes with extremely large degrees.

There are many choices of activation functions available to us, but for all our experiments we've chosen to activate our layers with the hyperbolic tangent function, `tanh` (see section 2.5.4). The main benefit of using the hyperbolic tangent function for activation is its intuitive interpretation of the activated linear response as measuring whether some feature is somewhere between fully correlated (+1), absent (0) or fully anti-correlated (-1) with the input data. `ReLU` activations (see section 2.5.4) are known to be much faster to evaluate than the (transcedental) hyperbolic tangent functions, but in our initial (offline) experiments (on CPUs, as opposed to GPUs), there didn't seem to be much of a difference in overall running time between the two activation functions; it would seem that the bottleneck in training lies somewhere else. Hyperbolic tangent functions were also the type of activation functions used for the hidden layer in `code2vec` (Alon et al., 2019). Due to the elegant interpretation, the negligible performance downsides, and the consistency with the `code2vec` paper, we've chosen to use the hyperbolic tangent function as activation function in all hidden layers in all of our experiments.

| hyper parameter | value |
|---|---|
| activation function | `tanh` |

Table 6.7: Activation function hyperparameter used for all hidden layers in all of our experiments.

### 6.3.4 Regularization

Dropout randomly disables some selection of responses in a particular layer (with the selection being different for each batch), which forces the network to create redundant features and which prevents it from being too reliant on any single feature. For all of our experiments we've dropped 25 percent of the nodes in the first hidden layer directly following the context embeddings. All subsequent hidden layers have had 20 percent of their nodes dropped. The main reason to go for a 25 percent drop rate in the first hidden layer was due to the existing embedding framework described in (Alon et al., 2019) also setting dropout to this same value. In our initial experiments, none of the other layers besides the first one had any dropout applied. The reason we later went for 20 percent dropout in these layers (as opposed to also setting it to 25 percent) was mainly due to it seeming to be a safer option. Unfortunately we were unable to measure any noticeable differences in validation accuracy between the situations where dropout was respectively disabled or enabled in the non-initial layers. The exact effects of dropout on our final validation accuracy are still unknown.

Batch normalization separates the learning of the magnitudes and variances of the linear response in each layer from the learning of the relative effects of each of the responses. For each batch normalized layer, all outputs of the layer are renormalized on a per batch basis to have means of zero and variances of one over the whole batch. An intermediate layer is subsequently added to re-scale and re-center these normalized outputs (using learned center and scale variables $\vec{\beta}$ and $\vec{\gamma}$) so that any desired response might still be learned by the layer. Batch normalization is mainly used to improve the flow of the gradient (which improves learning), but it also has a small regularization effect due to randomness arising from normalizing on a per-batch basis. When the same data point is seen again in a different epoch, the actual *absolute* responses within the network will be different because the responses are normalized with respect to the corresponding responses of the rest of the data in the batch, and when the contents of the batches is random (which for us it usually is), the normalizations will also be random. This randomness over the responses of data which is seen multiple times over multiple epochs prevents the network from becoming too reliant on the exact values of the responses, and it encourages it to focus on just the relative differences between the responses.

| hyper parameter | value |
|---|---|
| dropout rate embedding hidden layer | 0.25 |
| dropout rate other hidden layers | 0.20 |
| batch normalization | Yes |

Table 6.8: Regularization hyperparameters used for all hidden layers in all of our experiments.

### 6.3.5 Weight Initialization

#### Hidden Layers

The initial choice of network parameters is very important for the efficiency of training and for the final network performance. When the initial weights are too large we suffer from the *exploding gradient* problem, while if the initial weights are too small the network might exhibit *vanishing gradients*. Given our choice of activation functions to always being the (positive-negative symmetric) hyperbolic tangent function, a natural choice for us is to initialize all weights in `tanh`-activated layers by the Xavier (Glorot) initialization procedure that samples the weights from the `Norm`(0, 1) distribution but rescales the sampled weights in accordance to the number of inputs and outputs to the layer such that the output of the layer initially is expected to have approximately the same variance as the input. Empirically, this seems to work very well (see (Glorot & Bengio, 2010) for further details).

#### Embedding Vocabulary

Dictionary weights (i.e. the coordinates of the embedded labels) are initialized by sampling from the uniform distribution on the interval $[\frac{-3.0}{d}, \frac{3.0}{d}]$, where $d \in \mathbb{N}$ is the dimension of the embedding vector space. This is the initialization procedure used by `code2vec`, which we maintained in our adaptation of their embedding strategy. Unfortunately we've not been able to find a convincing theoretical motivation in the literature for this exact choice.

| parameter | initial values |
|---|---|
| hidden layer weights $W_{ij}$ | Xavier initialization procedure |

Table 6.9: Hidden layer initial parameter initialization.

#### Batch Normalization

The batch normalization layers adjust the linear responses of each layer to have a mean value of 0 and a variance of 1 over each training mini batch. To maintain the expressive power of the network, each batch normalized (linear) layer is directly connected to a simple layer that performs a pointwise affine transformation on each of the individual normalized outputs by re-scaling the $i$th component of the linear response $\vec{r}$ by a learned scale factor $\gamma_i$ and subsequently re-centering the output by adding a translation of $\beta_i$. Setting $\beta_i$ and $\gamma_i$ to respectively the mean and the standard deviations of the output $r_i$ over the mini batch recovers their original non-normalized responses over said batch. The parameter vectors $\vec{\beta}$ and $\vec{\gamma}$ are learned during training and as such they need to be properly initialized. For all our experiments we initially set to the values shown in table 6.10.

| parameters | initial values |
|---|---|
| batch normalization $\vec{\beta}$ | $\vec{0}$ |
| batch normalization $\vec{\gamma}$ | $\vec{1}$ |

Table 6.10: Batch Normalization initial parameter initialization.

#### Layer Bias

None of the layers in any of our experiments have an explicit bias term added to their linear responses, although all hidden layers except for the initial embedding layer have batch normalization enabled, which implicitly adds a bias term to those layers in the form of the $\vec{\beta}$ parameter vector.

| hyper parameter | value |
|---|---|
| explicit bias term | No |

Table 6.11: Bias hyperparameter.

## 6.4 Model Fitting

Given the features extracted from our training set, a specific combination of hyperparameters fixes a parametric family of hypotheses (called the model) that is parameterized by some parameter vector $\vec{\omega}$. In our case the hyperparameters determine, among others, the embedding vector spaces, the choice of labels in the edge and node feature sets, and the precise topology of the neural network. With all these variables fixed, we obtain a model that is parameterized by the total of all the weights used at each of the internal layers in the neural network and by the coordinates assigned to each of the labels in their embedding vector spaces. All these parameters together are then jointly referred to by the vector $\vec{\omega}$. The goal of the training procedure is then to find an estimation $\hat{\vec{\omega}}$ of $\vec{\omega}$ such that the model instantiated with $\hat{\vec{\omega}}$ has the best prediction performance on unseen data as possible.

To find this parameter estimation $\hat{\vec{\omega}}$, we employ the Adam (Kingma & Ba, 2014) optimization procedure to systematically explore the parameter landscape. Adam is a variant of stochastic gradient descent that uses the gradient (derivative) of the loss function (see section 2.7) with respect to a small batch of data to see in which direction (in the total parameter space) the current value of $\vec{\omega}$ needs to be moved to reduce the loss of the model with respect to this mini batch. The parameters are slightly nudged for each mini batch in the training set, for a number of iterations over the training set, and the hope is that all these slight adjustments move $\vec{\omega}$ to a new position that also reduces the loss (and thus prediction error) for unseen data. Adam differs from normal stochastic gradient descent in that it maintains per-weight learning rates instead of using a single learning rate for all weights at the same time. Adam employs a number of hyperparameters that determine its behavior while optimizing. Our choice for these values are shown in table 6.12. These values are the same as the defaults given by TensorFlow. We refer to (Kingma & Ba, 2014) and the TensorFlow manual for a more detailed discussion on the meaning of these hyperparameters.

| hyper parameter | value |
|---|---|
| step size $\alpha$ | 0.001 |
| first moment decay rate $\beta_1$ | 0.9 |
| second moment decay rate $\beta_2$ | 0.999 |
| safety constant $\epsilon$ | $1.0 \cdot 10^{-8}$ |

Table 6.12: Adam hyperparameters.

## 6.5 Dataset Preparation

For our experiments we've chosen to use a dataset of 150 thousand JavaScript programs which was first used in (Raychev, Bielik, Vechev, & Krause, 2016) on learning how to generate programs from noisy datasets. This dataset consists of precisely 150 thousand JavaScript source files which were systematically ripped from online code repositories like GitHub and Bitbucket. The authors of the cited paper took care of sanitizing the dataset by eliminating code overlap (possibly resulting from source files that are shared between multiple repositories) and by making sure that there are no pre-obfuscated/minified files present. We should note here that our chosen dataset is different from the dataset originally used by the authors of the `jsNice` paper. In fact, our dataset is only half as big as the dataset used by Raychev et al., 2015 in their original experiments. When we repeated their experiments on our own dataset, our final (baseline) model evaluation accuracy was around 5% higher than the evaluation accuracy reported in their paper. It would seem that there is some quality difference between the two datasets that results in a discrepancy between the respective measurements, but we don't think this difference will matter much in the assessment of the final quality of our own models. In the end we are mostly interested in improving the baseline performance by refining its `TOP-K` candidates, and as we've shown in the introduction (section 1.1, table 1.1), there is still plenty of opportunity to improve upon the existing results by rearranging within the `TOP-3` alone.

### 6.5.1 Knowledge Graph Extraction

We use the JavaScript knowledge graph extraction program `UnuglifyJS`[5] to transform the raw JavaScript source files $P$ into their respective knowledge graph representations $G(P)$ that contain precisely the information we care about for our predictions. The nodes in the graph $G(P)$ correspond directly to the various identifiers in $P$, and the labels attached to them are the names given to the corresponding identifiers within the original source code. Some of these labeled nodes are static and serve as information to inform the predictions while others correspond to the locally declared identifiers whose names we'd like to predict. The labeled edges connecting the nodes represent certain syntactic and semantic relations derived in the original source code. The exact nature of the information represented by the labeled edges is irrelevant for our current experiments; the only thing we care about here is that they were carefully engineered by the authors of (Raychev et al., 2015) and that they contain enough information for the conditional random field to make useful predictions.

We invoked the main extraction program to generate the graph for a single program using the following commandline:

```
> node bin/unuglifyjs --extract_features --skip_minified $PATH_TO_PROGRAM
```

This outputs a JSON representation of the generated graph to the standard output that we subsequently capture and store. The output generated by `UnuglifyJS` is adapted to `Nice2Predict` and can thus directly be used to train or evaluate the conditional random field model. For our own framework we use a different graph representation for which it is easier to generate neighborhoods. Besides the labeled edges between nodes, the original graph structure also contains inequality annotations that indicate whether two identifiers were declared in the same scope, which means that they should not be given equal names if the goal is to preserve the original program semantics. Unfortunately, for our model we currently make no efforts of preventing these name clashes, and as such these constraints are not preserved by our graph conversion. The conditional random field model does indeed use them, which means that in contrast to our models, their model can be safely used for example for automatic variable renaming purposes. Their model *does not* use these inequality constraints for determining the initial label scores, so it seemed safe for us to also ignore them as potential features.

### 6.5.2 Splitting for Training, Validation and Evaluation

To obtain an accurate estimation of the performance of both our model and of the existing baseline model, we've divided the dataset into three parts that are respectively used for training, validation and evaluation (see section 2.3). The *training set* is used for tuning the parameters of the model obtained from each particular hyperparameter configuration while the *validation set* is used to mutually compare the performances of the models obtained from each of these hyperparameter configurations. The best performing configuration for this set is chosen as our final model. This selection procedure results in a model that is optimally tuned for the validation set, having been chosen precisely because of its high performance on this dataset. Unfortunately, it may be the case that the hyperparameters of our optimal model are too well adjusted to the particularities of the validation set; to obtain a fair estimation of this model's performance on general *unseen* data, we make a final accuracy measurement on the *evaluation set*.

To obtain these respective datasets, we've initially allocated two-thirds of the dataset for training purposes. The remaining third part is divided into two parts by a proportion of $5 : 7$ which are then assigned to become respectively the validation and the evaluation sets. The evaluation set was chosen to be slightly bigger due to its greater importance, but in hindsight this choice seems not to matter that much. The dataset was pre-randomized by Raychev et al., 2016 and the two-third part that was selected to become the training set was also already allocated for this purpose by the same authors. Unfortunately, validation on the full validation set was too expensive to probe the model performance at certain intervals while training. The validation accuracy graphs shown below were obtained from periodic measurements at four evenly spread points per epoch on a smaller validation set that only contains the initial 5000 *usable* programs of the full validation set.

Unfortunately, not all programs in the dataset were usable for our experiments. The feature extraction program (`UnuglifyJS`) that constructs knowledge graphs from the raw source files was not able to generate non-trivial knowledge graphs from all source files in the dataset. For the training set (100.000 files total) we've recorded some observations concerning this failure to generate usable graphs. In most cases the reason was that the source files were just too simple to have any features extracted from them (18748 cases). In other cases the

[5] https://github.com/eth-sri/UnuglifyJS/tree/0b657a3ecff0d3ce24fe10b91d7d7152c5ac80ac

extraction program was unable to parse the source files (3674 cases) or the extraction program took too long (224 cases, timeout set at 10 seconds). The remaining failures (4912 cases total) were due to unknown reasons. In total, out of the 100.000 source files originally assigned to the training set, around one fourth (27558 cases) were not usable for training purposes. Table 6.13 summarizes the discussion concerning these three datasets.

| dataset | source file split | usable files | unknown identifiers |
|---|---|---|---|
| training | 100000 | 72442 | 6896091 |
| validation | 20834 | 15005 | 1367930 |
| evaluation | 29166 | 21244 | 1906061 |

Table 6.13: Dataset characteristics.

### 6.5.3 Pruning Rare Labels

Patterns concerning labels which occur in very low frequencies in the training set (e.g. only in a single source file, or only a few times over all source files) might not be likely to be learned correctly, given that there is not much information to generalize over. Trusting that usages of these labels within the training set are representative for their usages in general data might be dangerous if too much weight is given to these patterns. We've opted to prune any labels which are mentioned in fewer than 10 source files within the training set (around 100.000 files total). Unfortunately this number is quite arbitrary, and, in hindsight, the pruning itself might be unwarranted. It is not clear to us at this moment whether this pruning was beneficial or harmful to our experimental results. We made this choice in the earliest phases of our experimentation and we are now stuck with it; we don't have any data on how many unique labels we've thrown away because of this choice. All pruned labels within the training, validation or evaluation sets are identified and replaced by the generic `UNKNOWN` label placeholder, indicating that the label was originally something not found in our vocabulary. Conflating all these pruned labels into a single `UNKNOWN` label removes all information from the original label and replaces it with the information that something *rare* used to be there.

Another way of reducing the number of labels used by the model is to just set a hard limit $F$ on the number of (edge or node) labels used, by only keeping the $F$ most frequently occuring labels in either of the node and the edge label sets. Labels that don't belong to the $F$ most frequently occurring labels will then be replaced by the generic `UNKNOWN` label (again both in the training set as well as in the validation and evaluation sets). We've chosen not to limit the number of labels in this way, because we had no prior information on deciding how to set the thresholds. Our choices for these pruning hyperparameters have been recorded in table 6.14. The number of remaining labels of each type after pruning is shown in table 6.15.

| hyperparameter | value |
|---|---|
| pruning threshold | 10 |
| node feature limit | none |
| edge feature limit | none |

Table 6.14: Pruning hyperparameters.

| unique labels | count |
|---|---|
| nodes | 72654 |
| edges | 89253 |

Table 6.15: Number of unique labels remaining of each type after pruning.

### 6.5.4 Indexing Labels

Neither the baseline model nor our own model directly inspects the contents of any (edge or node) label. Labels are treated as atomic entities which only support the operation of being mutually compared for equality. All predictions are basically made by advanced bookkeeping procedures which in the end only need to check the

presence or absence of each label in a specific context, which can all be expressed in terms of basic equality queries. Because the labels themselves are never inspected, there is no need to keep their exact contents available to the framework; we only need to store enough information for the equality check to be able to be performed. Assigning to each uniquely occurring label in the training set a unique integer index keeps precisely enough information for this equality comparison to be performed while greatly reducing the amount of space needed to store each label.

Our label assignment is implemented via the construction of a histogram which counts the number of occurences of each label in the training set. The sets of node and edge labels are treated separately, creating two separate histograms for respectively the node and the edge labels. The calculated histograms are then used to assign an increasing sequence of consecutive integers to each label of the respective type that was found in the training set, giving the most frequently occurring labels the lowest indices[6].

### 6.5.5 Reserved Indices

We've reserved two special indices in this encoding for special situations. As mentioned previously, we use the `UNKNOWN` label as a stand-in for any label which was not in the initial training vocabulary. Our indexed label representation uses the index 0 to represent such out-of-vocabulary labels. In addition to the special `UNKNOWN` label, we've also reserved a special index for the situation when the name of an identifier is used while there is no such name available. This situation occurs for example when a node whose name still needs to be predicted is used as a feature in the context of another node when no candidates are available (in the zeroth layer for example), or when a radius-1 ray cannot be extended to a radius-2 ray, even though a radius-2 ray is requested. In this case, instead of throwing away such a partial radius-2 ray, the second edge and the second node of the ray are padded by a special label indicating that the second half of the ray does not contain any actual information. The use of this special label for padding locations with missing information allows the neural network to use these partial contexts without any further special treatment while still explicitly recording that some part of the context is missing. This can be compared with filling such positions with the generic `UNKNOWN` label, which would instead indicate that the position initially actually contained something, but which was too rare to preserve any further. Our indexed label representation uses the index 1 to represent this special `MISSING` label. The exact choice of representing index for these special labels facilitates efficient pruning of empty neighborhoods (see section 6.5.7), but otherwise it is immaterial to our use of labels within the framework. Our choice of label represention for these special labels is shown in table 6.16.

| label | index |
|---|---|
| `UNKNOWN` | 0 |
| `MISSING` | 1 |

Table 6.16: Indices assigned to special labels.

### 6.5.6 Neighborhood Extraction

In section 6.3.2 we saw that we need to limit the number of context rays we extract from a neighborhood to some predefined number $K$. Neighborhoods having more than $K$ rays will have to be reduced to have at most $K$ rays before they can be processed by the neural network, and this means a choice will have to be made on which of the $k > K$ rays will actually be representing the neighborhood. For our framework we've chosen to use a randomized *depth-first* approach for making this selection in which each edge list is first randomly shuffled before exploration is continued. The algorithm we use can be described by the snippets shown in figures 6.3 and 6.4 that respectively show the pseudo-code for generating radius-1 and radius-2 neighborhoods.

For training we use both the unknown and the static nodes to serve as neighborhood centers, while for validation and evaluation purposes we only test the accuracy of the framework in predicting labels for the unknown nodes. The rays are extracted on-the-fly when the neighborhood data is needed; when the same data point is encountered for a second time in a different epoch, the generation algorithm is run again and a (slightly different) neighborhood will be generated.

[6]The associated indices thus also encode a frequency partial order, which for example might be used as a tie-breaker when two labels are otherwise indistinguishable.

```
# input:  node Z, context size K
# output: iterator yielding precisely K context rays of the form (edge_1, node_1)

count = 0
E = edges(Z)
for connections (e, P) in random_shuffle(E):
  yield (e, P)
  count += 1; if count == K: return;

while count < K:
  yield (MISSING, MISSING)
  count += 1
```

Figure 6.3: Pseudo-code for extracting $K$ radius-1 neighborhoods for a center node $Z$.

```
# input:  node Z, context size K
# output: iterator yielding precisely K context rays of the form (edge_1, node_1, edge_2, node_2)

count = 0
E = edges(Z)
for connections (e, P) in random_shuffle(E):
  F = edges(P)

  if length(F) == 0:
    yield (e, P, MISSING, MISSING)
    count += 1; if count == K: return;
    continue

  for connections (f, Q) in random_shuffle(F) s.t. (Q != Y):
    yield (e, P, f, Q)
    count += 1; if count == K: return;

while count < K:
  yield (MISSING, MISSING, MISSING, MISSING)
  count += 1
```

Figure 6.4: Pseudo-code for extracting $K$ radius-2 neighborhoods for a center node $Z$.

In the code shown in figures 6.3 and 6.3, the *references* to the nodes $P$ and $Q$ which lie on a returned ray are returned, which allows code that consumes these extracted neighborhoods to extract precisely the information from the nodes that it requires. Our framework keeps track of the data generated for each node at each layer, and via some bookkeeping all generated data from any of the previous layers can in principle be accessed from these returned references to the returned nodes. This includes the `TOP-K` predictions made at the end of each layer as well as the neural network activations from any of the intermediate neural network layers (these will be called *stages* later on in this text, see section 6.6).

### 6.5.7 Neighborhood Pruning

Neighborhoods that either have zero context rays or neighborhoods where all context rays contain only `UNKNOWN` or `MISSING` labels contain no inherent predictive information that can be used to predict the center label from its context at the initial layer. In this case there is no actual pattern to relate the center label to its context, and to prevent the network from adapting to non-patterns, the empty neighborhood should be pruned from the training set. At higher layers, more information might become available at some of the nodes in these (initially) empty neighborhoods, so that an actual pattern might still be discovered, but due to an implementation deficiency these empty neighborhoods will still be pruned during training.

On the other hand, empty neighborhoods are always preserved during the validation and evaluation phases as to allow the framework to make at least a random guess in these situations. At the zeroth layer these predictions will be totally uninformed, but at higher layers they might still be pretty accurate when there is enough accumulated information available from the previous layers.

## 6.6 Layer Connectivity

In the previous sections we've given an overview of a number of generally applicable configuration options and design decisions that are shared between all our experiments. The remaining configuration options are all concerned with describing the precise topologies of the neural networks that are used within our overarching prediction architecture, and of the data flow between them. In this section we will explain two general configuration table formats for these remaining configuration options, so that each of our actual experiments can be described almost completely by simply filling out these tables. An illustration of the main configuration format that globally governs the inputs and outputs of the prediction layers is given by table 6.17. Table 6.18 contains an illustration of the context configuration format that precisely specifies the nature of the information extracted from the contexts and the dimensionalities in which it is made available to the rest of the pipeline.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| B | - | n/a | n/a | n/a | n/a | **150** | n/a | n/a |
| 0 | 0 | - | - | - | 300 | n/a | 300 | `source` |
| | 1 | - | 300 | 300 | - | **150** | **150** | `classifier` |
| 1 | 0 | 150 (`source`)$_1$ ⋯ 150 (`source`)$_K$ | 150 | 600 | 270 | n/a | 425 | |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | - |
| | $S(1)-2$ | - | 400 | 400 | 230 | n/a | 300 | |
| | $S(1)-1$ | - | 300 | 300 | - | **150** | **150** | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| L | 0 | 150 (`source`)$_1$ ⋯ 150 (`source`)$_K$ | 150 | 600 | 250 | n/a | 500 | |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | - |
| | $S(L)-2$ | - | 475 | 475 | 210 | n/a | 350 | |
| | $S(L)-1$ | - | 350 | 350 | - | **150** | **150** | |

Table 6.17: Configuration example for layer connectivity.

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 $(\texttt{source})_1$ | - | 150 | 150 $(\texttt{source})_1$ | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 1 | 0 | 50 | 150 | 150 $(\texttt{source})_1$ | 150 | - | - | - | 450 | 270 |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | $S(1)-2$ | 50 | 150 | 150 $(\texttt{source})_1$ | 150 | - | - | - | 450 | 230 |
| | $S(1)-1$ | 0 | - | - | - | - | - | - | - | - |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| L | 0 | 50 | 150 | 150 $(\texttt{source})_1$ | 150 | - | - | - | 450 | 250 |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | $S(L)-2$ | 50 | 150 | 150 $(\texttt{source})_1$ | 150 | - | - | - | 450 | 210 |
| | $S(L)-1$ | 0 | - | - | - | - | - | - | - | - |

Table 6.18: Configuration example for context information specification.

### 6.6.1 Layers and Stages

For the description of our architecture we will now introduce a few concepts which deviate a bit from the nomenclature normally used to describe neural networks. Our architecture consists of a layered stack of prediction engines, each of which takes a graph representation of the program under prediction as input, possibly augmented with some extra features and candidate predictions for any of the nodes, and which then outputs for each node within this graph a number of scored predictions, together with any of the intermediately generated features that were used to come to this prediction. Layers within this stack are connected to each other by feeding the information generated by one layer as input to the following layer. The first layer of this stack can optionally be chosen to be the conditional random field model, but any of the higher layers will always be of the convolutional kind that were introduced in this thesis. Each of these convolutional layers internally invokes a neural network for each of the nodes $Z$ within its input graph. The neural network consumes the available information within the neighborhood surrounding $Z$, and uses this information to generate the scored list of predictions for the label that should be assigned to $Z$. The neural network internally of course also consists of a number of (fully connected) layers. Thus we now have two different notions of *layer*: 1) a single input-out engine in a layered stack of prediction engines and 2) a single neural network layer consisting of a number of neurons that measure the presence or absence of features in the output of the previous layer. To be able to easily differentiate between these two notions of *layer*, we now define a **stage** to be a layer in the latter (neural) sense, while we reserve the word **layer** to refer to the one in the former (stack) sense. These conventions will hold until the end of this document.

Thus, our architecture consists of a number of *layers* that each augment their input graphs with additional candidate predictions and features for each graph node $Z$. These candidate predictions and features for $Z$ are generated by evaluating a neural network on a neighborhood surrounding $Z$, and this neural network itself is composed of a number *stages* that generate higher level features from the outputs of the layers below. Table 6.17 shows the general form of a table describing the layer and stage configurations for a particular experiment. The experiment shown there consists of a total of $L$ layers, and each layer itself utilizes a neural network consisting of $S(L)$ stages. The first two columns together index a specific layer-stage combination, and the remaining columns specify the precise origins of the inputs and outputs for the neural network when evaluated on a neighborhood surrounding a particular point of prediction $Z$. The exact numbers used here are shown only for demonstration purposes; for our actual experiments they will be different between the experiments.

### 6.6.2 Input Features

Let $Z$ be the node under consideration when making a prediction using the internal neural network. There are two distinct categories of information that can be used to inform this prediction: the category of information

currently available at the node $Z$, and the category of information available in the attention-averaged radius-$R$ context rays originating from $Z$. Each category supplies a number of real numbers that directly serve as input to the neural network, and the number of real numbers supplied by each of these categories is shown respectively in the column **total center input** and in the column **summarized context input**. These two categories are explained in more detail in the subsections below.

### 6.6.3 Center Input

The category of information shown in the *total center input* columns can be divided further into two subcategories: the subcategory of information obtained from the concatenation of the embeddings of the `TOP-K` candidate labels from one of the previous layers (when available) and the subcategory of information obtained from the feature maps generated by any of the previous stages in any of the previous layers. The configuration of the first of these subcategories is shown in column **center candidate labels** while the second subcategory is specified in the column **center features**. Each stage can be independently configured whether and how it sources from the candidate labels obtained from some previous layer. When a stage uses candidate labels from a previous layer, the enumeration of the candidate ranks is shown in the column *center candidate labels* for that particular stage. Each candidate rank specifies the number of reals contributed by the embedding of the label at that rank, together with a single character indicating the layer that originally generated these candidates (shown between brackets). Table 6.19 shows the mapping between the layers and these characters.

Our prediction problem consists of filling in the missing information on labeled graphs, were some nodes already have labels assigned to them and other nodes need to have their labels assigned to them by our framework. The nodes that already have labels assigned to them are referred to as the **known** nodes while the other nodes whose labels we need to predict are referred to as the **unknown** nodes. When the source of a label at a specific rank is `M`, we don't have any candidate labels available for that rank. When the label of the node is *known* (i.e. it doesn't need to be predicted) and when the requested rank is 1, we return the static label of that node as the `TOP-1` candidate. For all other ranks $K > 1$ and for the any of the ranks for nodes that are *not* statically known, we always return the `MISSING` label (see section 6.5.5). When the source of the label is either `B` or a layer $l \in \mathbb{N}$, we have a number of candidates for the *unknown* nodes that need to be predicted, while again we have just a single static label for the nodes that are *known*. In the case of a *known* node, we again return its static label for a `TOP-1` request while we return `MISSING` at the higher ranks for these nodes. For the *unknown* nodes, we return the candidate of the requested rank when a candidate of this rank is available at that position, or otherwise we also return `MISSING` (the baseline model does not always return the same number of candidates; sometimes there might be fewer candidates available for the node than are needed for the requested rank).

All labels are embedded in an 150 dimensional vector space (see section 6.3.1), so this will be the number of reals contributed by each label for all our experiments.

| character | description |
|---|---|
| `M` | `missing`: position only contains statically known labels; no actual candidates |
| `B` | `baseline`: position contains candidate labels generated by the baseline model |
| $l \in \mathbb{N}$ | `layer`: position contains the candidate labels generated by layer $l$ |

Table 6.19: Shortened indicators for the possible sources of candidate labels.

The subcategory of *center features* specifies the origin of the features used by the current stage that were generated by some previous stage. In principle each stage can be configured to source from any combination of previous stages in any of the current or previous layers, but for our experiments we've restricted the stages to only source from the stage listed directly before it in the table. We've used a color coding to highlight the correspondence between the features used at the center and the output stage that previously generated them.

### 6.6.4 Context Input

The *summarized context input* column of table 6.17 specifies the number of real numbers obtained by averaging over the information in the radius-$R$ context rays originating at the center of prediction. Each stage can independently specify whether or not it sources from this context, and if so, in how many real numbers it wishes to summarize the information contained therein. Each stage utilizes its own attention vector to determine the

weights given to each ray, so each stage can decide for itself on precisely what kind of information it focuses when summarizing the neighborhood.

The attention mechanism averages over the information contained in the radius-$R$ rays originating at the center location, and each ray is itself composed of $R$ edges and $R$ nodes. The edges are static and remain the same over all layers and stages, but the nodes acquire additional features for each additional stage added to the pipeline. Always utilizing the total amount of information available on the nodes on the ray quickly becomes unwieldy at the higher layers, so we need to be able to specify the precise information that we're interested in averaging over. Table 6.18 shows the general form of a table describing the information chosen to be extracted from the nodes and edges on a single radius-1 or radius-2 ray.

The first two columns index a specific layer-stage combination and the third column specifies how many rays are extracted from the neighborhood (see section 6.5.6). The remaining columns now specify for each of the possible edge or node positions on a radius-1 (or radius-2, for the zeroth layer) ray how many reals are used and where they come from. The **edge** $r$ **labels** columns ($r \leq R$) specify the number of reals contributed by the edge position at distance $r$ from the center. All edges contain precisely one label that is always embedded in a 150 dimensional vector space, so these positions always contribute precisely 150 reals when present on the ray. The **node** $r$ **labels** columns ($r \leq R$) specify the candidates used at the node lying at distance $r$ from the center. In principle this could be any number of $K$ candidates, but for our experiments we've limited these candidates to only the candidate at the `TOP-1` position. Again we use a single character (see table 6.19) between brackets to indicate the layer where these candidates originated. The **node** $r$ **features** column ($r \leq R$) specifies which of the feature maps generated by any of the previous layers and stages are used for the node lying at distance $r$ from the center. Due to the context rays being dynamically generated from the results of the previous layer, the *node r features* are limited to be taken from stages in layers previous to the current layer. Again we use color codes to highlight the correspondence between the choice of node features made here and the output of the stages that previously generated them. In this particular example the stage only sources from a single stage in a previous layer, but in general there may be more than a single source.

The number of features resulting from the concatenation of all these edge and node features on a single ray is shown in the column **raw context features**. These features are transformed and summarized via a single (`tanh`-activated) hidden layer to a vector of condensed features living in a vector space of dimensionality shown in the column **summarized context input**. Finally, the averaging attention mechanism is used to obtain a weighted average over all of these summarized context rays, which results in a single vector, again of this same dimensionality, that represents a summarized version of the information in the complete neighborhood. This is the vector that is used as input the stages shown in table 6.17.

### 6.6.5 Output Features

Now consider table 6.17. A single stage is itself (by our definition) a single hidden layer within the neural network that makes predictions for a specific node $Z$ by using the neighborhood surrounding $Z$. The stage concatenates the features obtained from the *total center input* with the features obtained from the *summarized context input* that were described in the previous subsections, and then it outputs a number of features obtained by performing a single linear (or affine) transformation of the concatenated input followed by a point-wise application of the non-linear activation function `tanh`. These activations are to be interpreted as higher order features that can be used by any subsequent layer to make high level predictions. The number of features (i.e. activated responses) delivered by a particular stage are shown in column **output features**. Here we again use color coding to indicate the relationships between the features generated by a particular stage and the features consumed at a different stage.

### 6.6.6 Training Objective

All experiments we perform in this chapter are configured to output precisely 150 features per node at the last stage of each layer. These features are considered to be the main deliverable of the layer and they are usually consumed as input by some successor layer.

To be able to train the layers so that the generated features are actually useful, we decided to optimize the neural network for creating features that are directly useful at predicting the center label. The last 150 `tanh`-activated responses that constitute the deliverable of the final layer are transformed via a single additional linear transformation to the same vector space that embeds the node labels, and it is here that the output of

the layer is compared with the labels in the embedding dictionary. The precise comparison is made via the measurement of the dot product between the output vector predicted by this neural network and each of the embedded labels from the node label dictionary. The calculated dot products are subsequently jointly pulled through a soft-max layer (see section 2.5.4) that results in a probability distribution over these labels that assigns higher probabilities to labels that have greater dot products (i.e. smaller angles or greater magnitudes) with the predicted vector. When predicting, we order our predictions by the probabilities thus obtained, and when training we try to optimize the parameters of the network so that the negative log likelihood of these parameters is minimized with respect to the training data (see section 2.7.5)

### 6.6.7 Inheriting Parameters

Each of the various layers in a single experiment are themselves autonomous classifiers that take a (feature and candidate augmented) graph as input and then give a list of scored predictions as output. Even though the various classifiers (layers) within a single experiment all make independent predictions, it is wasteful to train them completely independently. For our experiments, we always start the training of a higher layer $l > 0$ with the respective embedding vocabularies initialized to the same values as they had at the end of training the last layer $(l - 1)$. Thus each higher layer in any experiment always starts with label embeddings that are at least as good at those in the previous layer. Besides inheriting the parameters for the label embeddings, some layers also (partially) inherit the weights used in the stages from the corresponding stages in the previous layer, or sometimes even from the corresponding stages in the same layer in a previous experiment. A note of this fact will have been made in the column **inherited layer parameters** in the respective configuration table when this is the case for some specific layer. The main idea here is that we can inherit parameters whenever there is some overlap between the domains of the matrices that implement the linear transformation part of the hidden layer by selecting the corresponding rectangular blocks of submatrices that correspond to the linear transformation restricted to this intersecting of domains. The precise choices made for what to do when the dimensions mismatch (i.e. the choice of projection or injection into a smaller or respectively larger space) are not very interesting. We refer the reader to the actual implementation (see section 6.1) for more details.

## 6.7 Combining Classifiers

Sometimes different classifiers have different perspectives on what the correct answers should be for a certain prediction. When most classifiers agree on the `TOP-1` label, one can return this label as the final answer with high confidence. When many of the classifiers each propose their own, different, `TOP-1` label, a further inspection of the confidences of each of these disagreeing classifiers might offer some insight, and some clever strategy might still allow a quality label to be returned with high confidence. Combining the predictions of a number of classifiers might result in a super classifier that performs better than each of the individual classifiers that were used to construct it. The layers in our framework are all individually trained as classifiers, and as such they each have their own view on the desired labeling of a program. Below we discuss two mechanisms that we used to combine our own layer classifiers with the baseline classifier.

### 6.7.1 Averaging of Classifiers (Soft Voting)

Each of the layers used in our experiment assigns a probability distribution to the set of all possible node labels for each unknown graph node $Y$. The collection of node labels is ranked according to their probabilities, and the ordered list of the $K$ most likely labels is returned as the final prediction for the node $Y$ for that layer. By a point-wise averaging of the probability distributions obtained from each layer, we obtain a new probability distribution where most of the mass is located on labels where most original layers already had most of their respective mass located, while labels that were only thought to be very probable by a few layers now take a relatively small chunk of the total mass. We can now rank our labels according to this averaged distribution, and the *soft-voted* ranking thus obtained has the good property that it preserves the pair-wise rankings of any two labels when all other layers already agreed on this.

Not all layers are equally accurate in their predictions, and to give more power to the more accurate classifiers, we calculate a *weighted average* over the distributions, with the weights for each probability distribution being normalized with respect to *validation* accuracies of the respective classifiers that generated them. The scores returned by the baseline model are not normalized and can't directly be interpreted as probabilities, but we can

forcefully normalize them via a soft-max operation. Since the scores returned by the baseline model are always positive, we could also have normalized these scores by dividing each of them by their total mass, but this would have resulted in a distribution that has its mass more spread out compared to the distributions generated by our own layers, given that our own layers also perform a soft-max on their respective final scores.

### 6.7.2 Heuristic Combination with Baseline

Besides the soft-voting combination of classifiers, we also tried a different strategy where we compare the confidences of our own classifiers with those of the baseline classifier to decide how we can slightly nudge the baseline candidate predictions into a hopefully better ranking. For this combination we employ two heuristics. For our first heuristic, we'd like to de-prioritize baseline candidates that are seen to be unimportant to our own model. To do this, we employ some proximity constant $p$, and when the baseline classifier has a candidate prediction $c_k$ at rank $k$ but our own classifier has the same prediction at some rank $d$ that is not within proximity, $d > p + k$, we move this candidate to the end of the baseline candidate list. We start with the higher ranked candidates when we iterate the candidate list so that we preserve the relative rankings of all candidates that are pushed to the back. The optimal value for this proximity parameter on the validation set using our best classifier was found to be $p = 3$ or $p = 4$, depending on whether you value `TOP-1` or `TOP-2` accuracy more. For all our experiments we went with $p = 3$.

For the second heuristic, we reorder any pair of candidates within the `TOP-3` where the baseline does not think very strongly about this particular relative ranking, but where our own classifier is strongly opinionated that the ranking should actually be reversed. The first of these two criteria was checked by measuring the scores assigned by the baseline model to the two labels: when these scores lie within distance $\delta$ from each other, we think that their relative rankings are somewhat accidental, and so we would be willing to swap them around whenever our own model thinks they actually should be reversed. The second of the two criteria is checked by checking the scores assigned to the two labels by our own model: whenever our model assigns a higher score to the lower baseline-ranked label than to the other label, and this higher score is at least $\epsilon$ higher, we'd think our own model feels very strongly about the incorrectness of the current ranking. When both criteria are met, we make the swap.

The values for the thresholds $\delta$ and $\epsilon$ are hyperparameters for this combination classifier, and thus their optimal values need to be determined from the validation set. To obtain initial estimates, we measured the mean scores of the respective classifiers for our best performing classifier over the cases in the validation set where this swap would actually have improved the situation. The **validation means** column in table 6.20 contains the results of these measurements. Unfortunately we did not have enough time for a thorough investigation on the optimal values for the corresponding hyperparameters. The same table also shows our final choice for these hyperparameters that we used for all our experiments, and these were obtained mainly by a conservative instantiation of about three hyperparameter combinations around these means.

| hyperparameter | validation means (stds) | value |
|---|---:|---:|
| proximity | - | 3 |
| closeness $\delta$ | 0.0243 (0.0431) | 0.0150 |
| separation $\epsilon$ | 0.1534 (0.0760) | 0.1640 |

Table 6.20: Hyperparameters measurements for the heuristic combination classifier.

## 6.8 Evaluation / Performance

The various layers of our framework are *trained* as classifiers to predict the node labels for *all* knowledge graph nodes, i.e. we use the respective surrounding contexts to predict the labels for both the statically known nodes, and the unknown nodes whose labels actually need to be determined. As far as the framework is concerned, there is no actual difference between the two types of nodes, besides the possible presence of a candidate list at nodes of the unknown variety. The predictions made for the statically known nodes are not actually interesting to us, but training the network to be able to make these predictions forces it to also learn the semantic relations between these nodes and their respective neighborhoods. Also, the feature maps generated to make these predictions serve as neighborhood summaries, and these are as valuable features for the higher layers. So even

when validating or evaluating, we allow the framework to make predictions and generate feature maps for all nodes within the graph, irrespective of whether the node is known or unknown. However, when measuring the validation or evaluation performances of the framework, we don't care about mispredicted statically known nodes. In this case we only care about the performance of the framework on the *unknown* nodes, and so all our performance measures are *restricted to the class of unknown nodes.*

For each of the classifiers used in our experiments we measure both `TOP-1`, `TOP-2` and `TOP-3` accuracies and precision, recall and $F_1$ scores over the evaluation set. We define the `TOP-K` accuracy for a classifier as the fraction of cases where the correct answer was within the $K$ highest valued predictions according to the classifier. The precision, recall and $F_1$ scores are initially calculated on a per-prediction basis, and the summary scores calculated for each experiment are obtained by averaging over the respective per-prediction calculated scores. For a single point of prediction, we divide both the predicted and the expected label in a number of lower-case subtokens, obtained from splitting the respective labels first by underscores and then by camel case. From these two subtoken lists, the precision is calculated as the fraction of subtokens predicted that were also expected to be present, while the recall is calculated as the fraction of subtokens from the expected label were actually predicted. The point-wise $F_1$ score is then calculated as the harmonic mean of these point-wise precision and recall scores. The averaged $F_1$ score is an upper bound on the `TOP-1` accuracy, but it is more nuanced in that it can also give points to partially correct answers. We refer to section 2.2 for more information on these calculated performance measurements. In table 6.3 of section 6.2 we already showed the performance measurements for the locally and globally optimized baseline classifiers. The results obtained from the globally optimized baseline classifier are repeated in all main performance tables for all performed experiments to serve as reference values to contrast our own results against. The showed performance metrics for this baseline classifier will be exactly the same over all experiments due to the fact that these baseline predictions were generated for the evaluation set just once as a pre-processing step.

Each of the convolutional layers within an experiment is trained as a classifier, and as such, each layer will have a different opinion on what the correct labeling should be. For our experiments we've compared the mutual performances of each pair of classifiers within a single experiment. When two classifiers have similar accuracies, it is interesting to find out how these accuracies relate to each other. If both classifiers tend to make exactly the same mistakes, then both classifiers are mostly indistinguishable, and keeping both of them around is now redundant because each of them just repeats what the other had already said. Things become more interesting when two classifiers with similar accuracy scores disagree on a large portion of the dataset, because then the fact that they disagree on this large part can be used as additional information to further refine the predictions. Each classifier might be seen as an expert on the portion of the dataset where it performs better, and by identifying each classifier's area of expertise, a combination classifier might be obtained that delegates each prediction to the classifier that is most skilled at answering the specific question (see section 6.7).

To gain some insight in the agreements and disagreements between the various classifiers, we've compiled a number of tables (shown in appendix chapter A) for each experiment that show for each pair of classifiers how many times one classifier in the pair dominated the prediction made by the other classifier in the same pair. When both classifiers are performing equally well, we would expect these domination events to be well balanced, so that in about half of the cases one classifier would dominate while in the other half the other classifier would dominate.

Under the null hypothesis that these two events indeed each occur with probability $p = 0.5$(and the assumption that all predictions are made independently) we've also calculated the probability of having obtained this set, or any more extreme set, of disagreements between the two classifiers, by performing a simple binomial test. This probability is the p-value value for this pair of classifiers, and we interpret the difference in accuracy between them as statistically significant whenever this probability would indicate that obtaining these results was very rare (p-value $< 0.005$).

Each of the main experiments performed in the sections below will have the pair-wise performances of each of the used classifiers compared on `TOP-1`, `TOP-2` and `TOP-3` accuracies. The pair-wise performances for these three performance categories will be shown in three corresponding tables. The domination events shown in these tables are defined as follows. Classifier $A$ dominates classifier $B$ for some prediction in the `TOP-K` category whenever the actual known-to-be-correct label is found within the $K$ highest valued answers returned by $A$, and the rank given to this label by $A$ is strictly better than the rank given to the same label by $B$. It is important to note here that under this definition, the correct label can still lie within the `TOP-K` for $B$ for this event to take place, it just needs to be worse than the rank of this label according to $A$. Thus the difference between the `TOP-K` accuracies of two classifiers (as measured for the main performance tables) does not need to be the same

as the difference in the number of domination events of the two classifiers; the pair-wise TOP-K performance differences and the main TOP-K performance differences do not record the same information.

## 6.9 Extension Experiments

The main research question that we'd like to answer with this thesis is whether we can construct a convolutional neural architecture that can use and improve upon the predictions made by the conditional random field model. In this section we describe three different experiments that were designed to test to what extent our proposed architecture is capable of this improvement.

The first experiment ('Initial Semantic Embedding Classifier') trains a single layer convolutional neural network with the intent of obtaining good label embeddings by making the network optimize the label coordinates so that the attention-averaged radius-2 context rays containing these embedded labels become good predictors for the labels at the respective center points. The label predictions made by this single layer are not good enough to be useful on their own, but the feature maps used in making these predictions provide for excellent first-order summaries of their respective surrounding neighborhoods. The other two experiments in this section build upon this first layer by utilizing the generated feature maps as additional neighborhood information. They also re-use the label embeddings generated by the first layer to initialize their own label embeddings.

The second experiment ('Augmented Classifier') extends the single prediction layer obtained from the first experiment by a number of additional layers, where each additional layer also has access to some of the feature maps generated by the previous layer to better inform its own final predictions. The feature maps extracted from nodes in context positions themselves contain information that is a summarized version of the information in the contexts centered around these respective context nodes, so the higher layers in this experiment aggregate information that originated far beyond the first order neighborhoods where they explicitly read their information.

The first two experiments only use the baseline candidates to fill in *context* positions when making predictions for any particular node, without taking into consideration the predictions made by the baseline model for the center node itself. The third experiment ('Refinement Classifier') is designed to see what happens when we also allow the neural network to peek at the TOP-3 candidate predictions made by the baseline model when making a prediction. The idea here is that the network learns which of the three candidates is most consistent with the available context and that this candidate is then eventually placed at the TOP-1 output position. This experiment also re-uses the first layer that was obtained in the first experiment; only the higher layers above the first layer have access any of the candidate labels at the center position.

#### Omniscient Neighborhood Knowledge

All experiments described in this section are *trained* with **omniscient neighborhood knowledge**, by which we mean that the TOP-1 candidates returned for nodes *on context rays* are taken to be the true labels that were extracted from the training set for the respective nodes. This means that the network is trained on extracting the true center label from an idealized neighborhood instead of the false neighborhoods that were guessed by one of the models. When validating and evaluating, we of course don't have any knowledge of the true labels for these context nodes, so here we use the baseline candidates as best-effort substitutes.

The main reason for training on idealized neighborhoods is that we'd like the label and neighborhood embeddings to learn the true semantic relationships between the labels, and not some false semantic relationship that was dreamed up by the existing model. The idea is that our neural network uses the semantic information contained in the labels in the neighborhood to predict the best fitting center label, and the most relevant semantic information is the true meaning of the current label assignment. If the label embeddings were trained using the false label assignments from an existing model, the embedded labels would together implicitly convey the wrong intention of the program, even for programs that actually have their correct labels assigned. Thus in this situation with the falsely obtained embeddings, the correct labeling might actually seem to contain contradictory information according to our model, because it is not imperfect like the assignments given by the existing model. We'd like, at least for programs in the training set, for our model to have the property that a perfect labeling of a program is actually the best situation for the model to start making predictions.

Related to this, we want our model to be independent of the actual candidate set used. When we fill context positions with actual candidates while training, our neural network learns to use the underlying prediction pattern of the existing model that generated these candidates to make its own predictions. These patterns are very specific to the source model, and this means our model will degrade in performance when the actual

candidates adhere to a different set of underlying patterns, even if those different patterns *are actually better* than those of the candidates it was trained on. In the end we hope that our model can improve upon the candidate rankings of any existing model, and specifically, that it can improve upon its own generated rankings by feeding the output of one instantiation of our model as input to another instantiation of the same model. The closer the candidates come to the ideal situation, the closer our predictions come to the ideal situation, and then the predictions of a second instantiation of our model would hopefully come even closer. Ideally, we would iterate this process until we reach a fixed point, which would then be our returned solution.

### 6.9.1 Initial Semantic Embedding Classifier

This experiment trains the initial label embeddings. Figure 6.5 graphically shows the information obtained from the radius-2 context rays. The nodes at distance 1 and distance 2 from the center each supply 150 real numbers of information to the context ray from the embedding of the respective labels at these positions. When training, these labels will be the actual labels as there were found in the training set while during validation and evaluation these labels are supplied by the conditional random field baseline model when they are otherwise not available. The edges at distance 1 and distance 2 are always statically known and always supply 150 real numbers worth of information. In total, each context ray delivers 600 reals of information. Table 6.22 shows that each context ray is summarized into 300 reals by a single `tanh`-activated hidden layer, and these individual ray summaries are then averaged via the attention mechanism to obtain a final 300 real vector that represents the complete context.

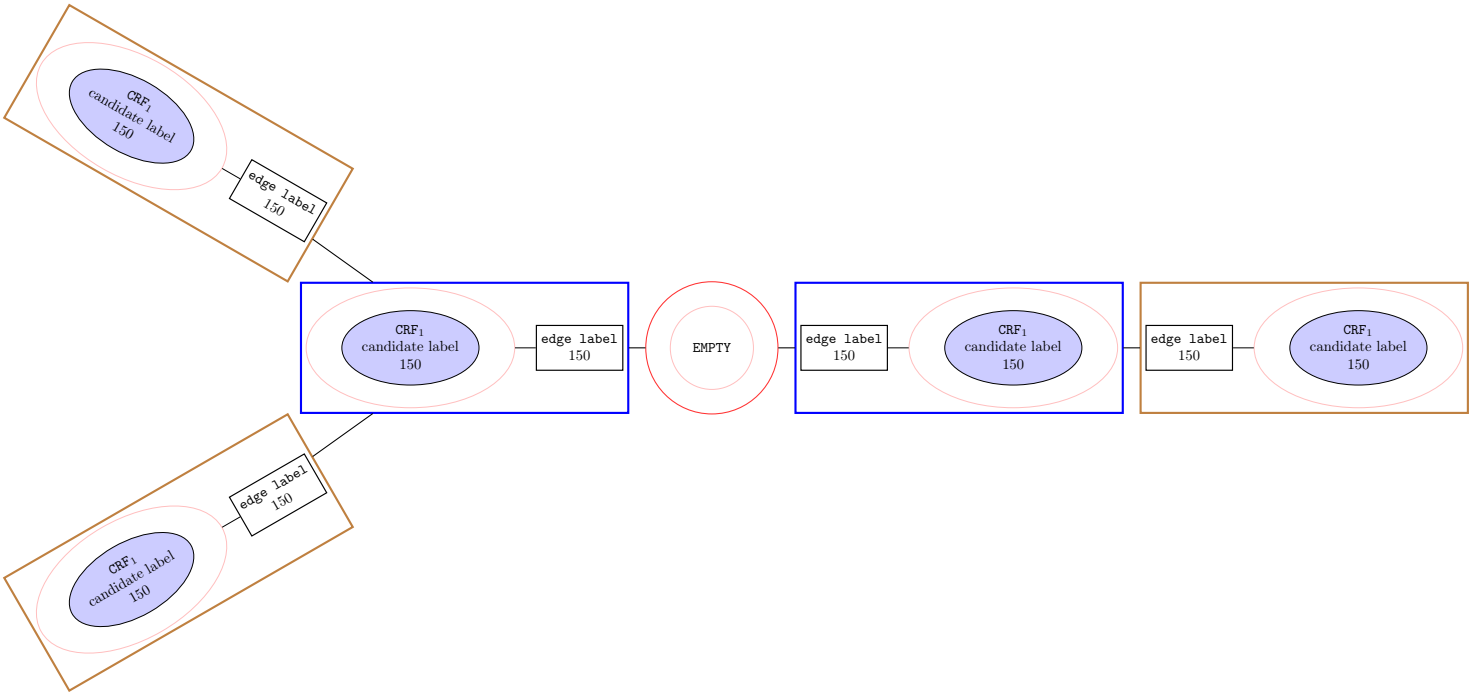


Figure 6.5: Neighborhood layout for layer 0 in the Baseline Extension experiments.

#### Modified Training Objective

Table 6.21 shows that this *summarized context input* represented by 300 reals is transformed by two consecutive stages to yield feature maps of respective dimensions of 300 and 150 reals. In section 6.6.6 we discussed that these last 150 real numbers are the final features used to make the target label predictions via a single linear transformation that puts the final prediction of the layer in the same vector space as the embedding node labels, and it is here where they can subsequently be compared with the predicted vector on similarity via their mutual dot products. Normally, the network is now optimized to directly maximize the similarity of the predicted

vector and the embedding of the label that was given in the training set, but for this first embedding layer we make a slightly different choice.

The problem with predicting from the final 150 outputs of this layer is that there are now three (very wide) layers between the initial embedding and the final predictions. Deep neural networks (i.e. networks having 2 or more hidden layers) are known to be very flexible, being able to fit almost any pattern when the network is made sufficiently wide. For our embedding purposes, we like the information in the embedding vector spaces to be organized as efficiently as possible, so that related concepts occupy similar regions of space. Unfortunately, the flexibility of the following layers doesn't really encourage the neural network to organize the embedded vectors efficiently as possible, because even when similar concepts are placed far apart and are seemingly unorganized, the deep network still has enough capacity to undo these inefficiencies and still come to good predictions for the data it was optimized for. This means the training loss might still become very low while training, even though the network has not learned much about efficiently organizing similar concept. The result is that the network does not generalize very well.

To overcome our network neglecting the initial embedding layer while still allowing it to construct higher level concepts from these embeddings to use in subsequent layers, we've opted for optimizing for both these contradictory objectives at the same time. In addition to trying to predict the center label from the 150 reals delivered by the final stage, we also try to predict the center label directly from the original 300 reals that constitute the *summarized context input* via a single linear transformation from this summary space to the space where the node labels are embedded. Again we use a joint soft-max over the dot products between the predicted vector and each of the embedded node labels in our dictionary to obtain a probability distribution over the target labels, and again we try to maximize the probabilities of the actual labels that were found in the data set. This auxiliary classifier is called the **shortcut** predictor in our framework. The final loss function is now taken to be the average of the original loss function that predicts from the final 150 reals and this latter loss function that predicts directly from the 300 reals from the *summarized context input* column.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| B | - | n/a | n/a | n/a | n/a | **150** | n/a | n/a |
| 0 | 0 | - | - | - | 300 | - | 300 | - |
| | 1 | - | 300 | 300 | - | **150** | **150** | |

Table 6.21: Main Configuration for Initial Semantic Embedding Classifier.

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 ($B_1$) | - | 150 | 150 ($B_1$) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |

Table 6.22: Context Configuration for Initial Semantic Embedding Classifier.

**Results**

We've trained the layer by optimizing the combined loss function over ten epochs. The validation accuracies for both predictors are plotted by epoch in figure 6.6. The `TOP-1`, `TOP-2` and `TOP-3` validation accuracies for the model at epochs 5 and 10 are shown in table 6.23. We see that initially the validation accuracy grows quite rapidly while it starts to stagnate around the fourth epoch. The accuracies of the shortcut classifier are always slightly higher than the accuracies of the classifier that goes through the deep network. In hindsight, we probably should have kept this base embedding shallow from the start, maybe with the averaged context rays directly being delivered as the final feature map. To maintain uniformity across the experiments, we will still always use the 150 feature maps delivered by the final stage when the output of this layer is used as input to a subsequent layer in the experiments below, even though these seem to give slightly worse predictions.

Unfortunately, we've started using this layer as the base layer in our followup experiments already at the end of the fifth epoch, so these experiments all use a base layer that is not optimized as well as it could have been; we only found out that there were a few more accuracy points to be gained at a much later stage when it was already too late to re-run all these experiments with a better optimized base layer. Another thing to note is that these *validation* accuracies were accidentally measured with suboptimal candidate set (we used the *locally* optimized baseline predictions instead of the globally optimized ones, see section 6.2.2). This means that the validation accuracies shown here are slightly lower than the validation accuracies shown for later experiments. No candidates were used while training (we have *omniscient neighborhood information* enabled), so the presence of these wrong candidates does not influence the obtained model.

| layer | epochs | TOP-1 | TOP-2 | TOP-3 |
|---|---|---|---|---|
| 0 | 5 | 46.67 | 54.06 | 57.68 |
| 0 | 10 | 49.18 | 56.41 | 59.80 |
| shortcut-0 | 5 | 47.77 | 55.06 | 58.55 |
| shortcut-0 | 10 | 49.53 | 56.51 | 59.90 |

Table 6.23: Accuracy results for the Initial Semantic Embedding Classifier on the validation set (around 500k predictions total) after 5 and after 10 epochs.

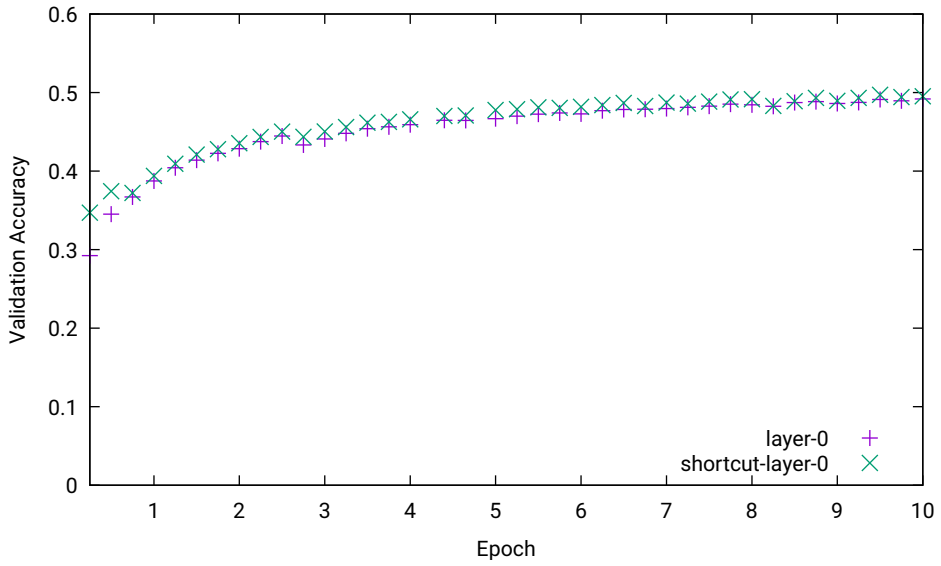


Figure 6.6: Validation accuracies during the initial embedding phase, measured on a validation set of 5000 programs with approximately 500k predicted labels. The shortcut graph directly predicts from the averaged context rays, bypassing the higher stages of the layer.

### 6.9.2 Augmented Classifier

For this embedding we build upon the base layer trained in the previous experiment at the end of the fifth epoch. Table 6.24 shows the main configuration for the layers used by this experiment. Layer 0 has the same configuration parameters as the previously discussed baseline layer, while there are three layers added on top that each consume some of the feature maps generated by the final stages of some of the previous layers. Layers 1 and 2 are similar in that their first stage only consumes the *summarized context input* while their second stage combines the output from the first stage with the feature maps delivered by the last stage of the previous layer. Layer 3 breaks this pattern by having the first stage consume both the *summarized context input* and the delivered feature maps from the final stages of all of the previous layers. Table 6.25 shows that all layers above the base layer only source from the contexts at the first stage, where they use the baseline candidates whenever a requested node still needs to have its value predicted. Again, these candidates are only used when

validating and evaluating; while training, all layers have omniscient neighborhood knowledge enabled (see the discussion in the introduction of this section). Figure 6.7 graphically shows the information that is available at each position for layer 1 and 2, while figure 6.8 shows what information is available at each position for the third layer.

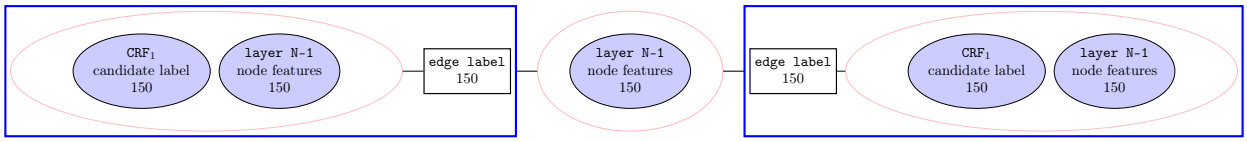


Figure 6.7: Neighborhood layout for layer $N \in \{1, 2\}$ in the Augmented Classifier experiment.

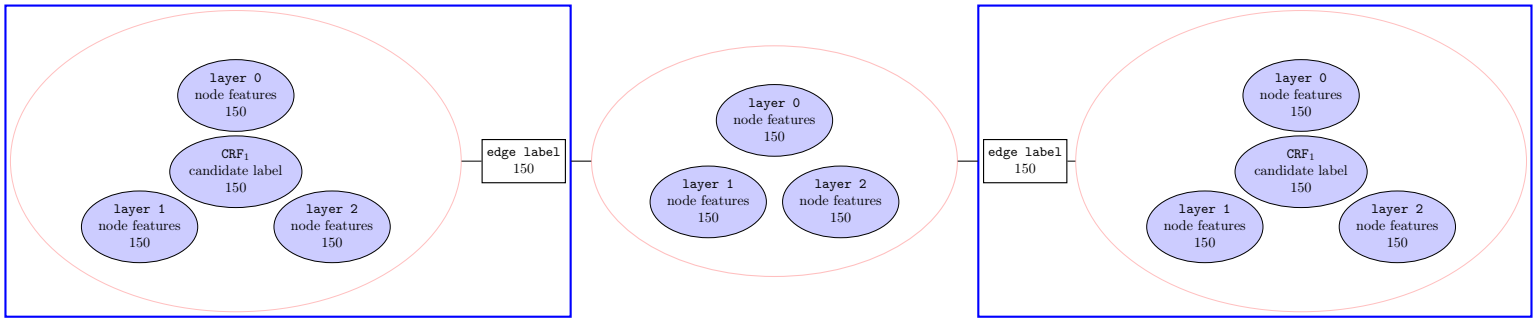


Figure 6.8: Neighborhood layout for layer 3 in the Augmented Classifier experiment.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| B | - | n/a | n/a | n/a | n/a | **150** | n/a | n/a |
| 0 | 0 | - | - | - | 300 | - | 300 | - |
| | 1 | - | 300 | 300 | - | **150** | **150** | |
| 1 | 0 | - | - | - | 300 | - | 300 | 0 |
| | 1 | - | 150 300 | 450 | - | **150** | **150** | |
| 2 | 0 | - | - | - | 225 | - | 225 | - |
| | 1 | - | 150 225 | 375 | - | **150** | **150** | |
| 3 | 0 | - | 150 150 150 | 450 | 400 | - | 425 | - |
| | 1 | - | 425 | - | - | - | 300 | |
| | 2 | - | 300 | - | - | **150** | **150** | |

Table 6.24: Main Configuration for Augmented Classifier.

**Results**

We've shown some performance measurements of the various classifiers used in this experiment in table 6.26. The base layer 0 of course shows the same (up to random fluctuations) performance as it did during the first experiment. From the results for layer 1 we see directly that adding the feature maps from the base layer greatly improves the prediction performance of the layer over all performance measures by at least 10 percent. Unfortunately, the addition of layer 2 above layer 1 does not show a similar jump in performance, although it still manages to improve most measures by slightly less than 1 percent. Layer 3 was configured to consume the final stage feature maps from all layers before it, with the hope that these layers all offered useful different perspective on the situation. Unfortunately, this increase in information does not easily translate into an increase in accuracy: the final performance of layer 3 after 5 epochs is slightly lower than it was for layer 2 after the

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 ($B_1$) | - | 150 | 150 ($B_1$) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 1 | 0 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 2 | 0 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 225 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 3 | 0 | 50 | 150 | 150 ($B_1$) | 150 150 150 | - | - | - | 750 | 400 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| | 2 | 0 | - | - | - | - | - | - | - | - |

Table 6.25: Context Configuration for Augmented Classifier.

same number of epochs. Unfortunately we stopped training each of the layers after 5 epochs and as such we currently do not know whether layer 3 would have eventually caught up or even surpassed the other 2 layers.

Besides the four main classifiers corresponding to the main layers, we've also combined our layer classifiers by averaging over their predicted probability distributions (see section 6.7.1) and we've also combined this averaged classifier with the baseline classifier using the combination heuristic (see section 6.7.2). These are shown in the table under the layers respectively named `A` and `C`. Unfortunately, the averaged classifier, while being more accurate than any of the numbered layer classifiers, does not manage to come close to the performance of the baseline `B`, with it still lacking around 4 percentage points in accuracy. On the other hand, the heuristically combined classifier does actually manage to improve upon the baseline ranking in all six measured performance criteria. The precision, recall, $F_1$ and `TOP-1` accuracies are marginally improved by about a quarter percentage points each, but the `TOP-2` and `TOP-3` accuracies each show improvements of around 1 percent. Tables A.1, A.2 and A.3 show pairwise comparisons of the various classification layers, where for each pair of layers we've recorded how many times either of the layers made a prediction in respectively the `TOP-1`, `TOP-2` and `TOP-3` that was strictly better in rank than the prediction made by the other classifier. These tables also record the p-value of these results under the null hypothesis that these win-lose and lose-win events should occur with a probability of 0.5 when both classifiers perform equally well. These tables show that the obtained results are very unlikely under this null hypothesis at a significance level of 0.005, and thus that the shown accuracy improvements are all statistically significant.

| layer | epochs | precision | recall | $F_1$ | TOP-1 | TOP-2 | TOP-3 | time | notes |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | 50.06 | 49.82 | 49.83 | 48.43 | 55.30 | 58.61 | $\sim$ 2.5 days | initial embedding, radius = 2 |
| 1 | 5 | 62.82 | 62.65 | 62.64 | 61.57 | 66.76 | 68.99 | $\sim$ 2 days | [feature maps 0] |
| 2 | 5 | 63.67 | 63.46 | 63.48 | 62.47 | 67.41 | 69.54 | $\sim$ 2.5 days | [feature maps 1] |
| 3 | 5 | 63.23 | 62.97 | 63.01 | 62.03 | 67.04 | 69.18 | $\sim$ 3.5 days | [feature maps 0, 1, 2] |
| A | 20 | 64.07 | 63.84 | 63.87 | 62.88 | 67.79 | 69.92 | $\sim$ 10.5 days | averaged over layers 0, 1, 2, 3 |
| B | 22 | 68.89 | 68.82 | 68.82 | 68.38 | 71.91 | 72.98 | $\sim$ 5 days | conditional random field |
| C | - | **69.06** | **68.98** | **68.98** | **68.54** | **72.79** | **74.07** | - days | heuristically combined A and B |

Table 6.26: Accuracy results for Augmented Classifier on the evaluation set (around 2 million predictions total).

### 6.9.3 Refinement Classifier

With the previous experiment we tested whether our convolutional architecture could aggregate enough information from the candidate-filled neighborhoods to arrive at a classifier that could beat the prediction performance of the baseline model. Here each of the layers had access to the `TOP-1` candidate for any context position that still needed to be predicted, but it did not directly have access to any of the baseline predictions for the actual position currently being predicted.

Unfortunately, none of our main layer classifiers could directly improve upon the baseline model, although an ad-hoc combination of the baseline classifier with the averaged super classifier did manage to improve the

TOP-K performances in a statistically significant way. For this experiment we want to see what happens when we let each layer take the TOP-3 baseline candidates for the node that is currently being predicted as additional input features. Hopefully, the model can be taught to pick the correct answer whenever its given the TOP-3 baseline candidates where this correct answer is already present. A comparison of the TOP-K accuracy score measurements for the baseline model shows that the correct answer was placed at the 2nd or 3rd position in around 4 percent of the cases, which means that there is around 4 percent to be gained by just re-organizing within the TOP-3.

Figure 6.9 graphically shows the available features for each of the three higher layers in this experiment. Table 6.28 shows the data flow between the various layers while table 6.29 shows the information extracted from the context positions for each of the layers. Again we use the same base layer 0 as we obtained from our first initial embedding experiment. For this experiment we've configured each layer above the base layer to read from the context at both the first and the second stage. Each of these stage calculates its own attention vector and is thus allowed decide on its own on which of the context rays to put its focus on.

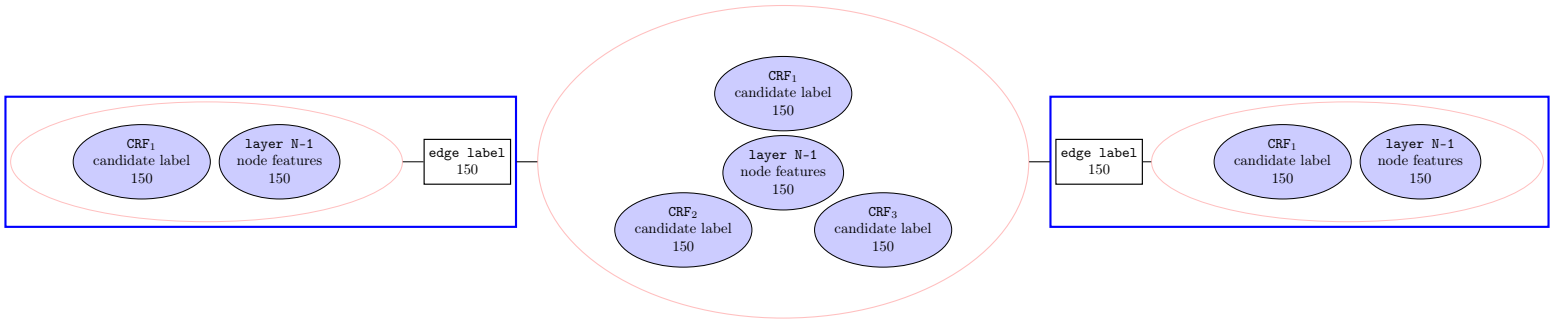


Figure 6.9: Neighborhood layout for layers $N \in \{1, 2, 3\}$ in the Refinement Classifier experiment.

### Candidate Shuffling

The main purpose of this experiment is to obtain a classifier that is able to recognize and then utilize the presence of the correct label from a lineup of labels where the target label may or may not actually be present. When we actually use the classifier, we first query the baseline model for the 3 most likely candidates, whose concatenated embeddings are then fed as input features to the neural network. Hopefully the network will use the knowledge it gained on the training set to pass through the information related to the candidate that is the most consistent with the surrounding context, while blocking candidates that are less inconsistent with the context. To train the classifier, we've generated candidates for each point of prediction in our training set by generating the candidates for one part of the training set by using a baseline model trained on the complementary part (see section 6.2.3). Generating candidates this way (as opposed to generating them using the full baseline model trained on the complete training set) ensures that the candidate distribution is not 'too good' due to the candidates being generated for the same dataset that the generating model was trained on. Of course, the baseline model is already quite good at predicting the correct label, with the baseline attaining TOP-1 prediction accuracies of around 70 percent. Unfortunately, the high TOP-1 accuracy of these candidate predictions presents a small problem for training our model.

If we would train our model directly on the baseline candidates in their original baseline ranking, our neural model might get the idea that the correct candidate is always the candidate ranked highest by the baseline model. This strategy gives the correct answer in about 70 percent of the cases and doesn't even require any contextual knowledge. Obviously the model learning this strategy is not desirable. What we actually want is that the model learns come to a decision by comparing each of the three candidates with its context, and selecting the candidate most consistent with this context. To encourage the model to analyze the information at all three candidate positions, we've decided to randomly permute the candidates in the training set a bit so that the actual target label will not be at the TOP-1 position as often as they would have been if we had always kept the baseline rankings. This way the model sees the correct label randomly jump between the first, second and third ranks, which forces it to actually examine the contents of the labels instead of relying on just the rank. Table 6.27 shows the initial probabilities we choose for picking a candidate from the 5 highest ranked

candidates. Candidates are drawn without replacement until all 5 ranks are filled. After a candidate is picked it is removed from the table and all remaining probabilities are renormalized.

| **position** | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **probability** | 0.50 | 0.25 | 0.15 | 0.05 | 0.05 |

Table 6.27: Initial candidate probabilities for when shuffling the baseline candidate lists for the Refinement Classifier.

Candidates are shuffled on-the-fly when reading the pre-processed, candidate-augmented training set. An advantage of this on-the-fly candidate shuffling is that our (post-shuffling) training set is now never quite the same over multiple epochs, with each epoch giving rise to different candidate shufflings for the same data points. Unfortunately we did not get around to investigate the effects of disabling the random shuffling of candidates.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| B | - | n/a | n/a | n/a | n/a | **150** | n/a | n/a |
| 0 | 0 | - | - | - | 300 | - | 300 | - |
| | 1 | - | 300 | 300 | - | **150** | **150** | |
| 1 | 0 | 150 ($B_1$) \| 150 ($B_2$) \| 150 ($B_3$) | 150 | 600 | 200 | - | 400 | |
| | 1 | - | 400 | 400 | 200 | - | 300 | - |
| | 2 | - | 300 | 300 | - | **150** | **150** | |
| 2 | 0 | 150 ($B_1$) \| 150 ($B_2$) \| 150 ($B_3$) | 150 | 600 | 200 | - | 400 | |
| | 1 | - | 400 | 400 | 200 | - | 300 | - |
| | 2 | - | 300 | 300 | - | **150** | **150** | |
| 3 | 0 | 150 ($B_1$) \| 150 ($B_2$) \| 150 ($B_3$) | 150 | 600 | 200 | - | 400 | |
| | 1 | - | 400 | 400 | 200 | - | 300 | - |
| | 2 | - | 300 | 300 | - | **150** | **150** | |

Table 6.28: Main Configuration for Refinement Classifier.

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 ($B_1$) | - | 150 | 150 ($B_1$) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 1 | 0 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 1 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 2 | 0 | - | - | - | - | - | - | - | - |
| 2 | 0 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 1 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 2 | 0 | - | - | - | - | - | - | - | - |
| 3 | 0 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 1 | 50 | 150 | 150 ($B_1$) | 150 | - | - | - | 450 | 200 |
| | 2 | 0 | - | - | - | - | - | - | - | - |

Table 6.29: Context Configuration for Refinement Classifier.

**Results**

Table 6.30 shows the evaluation performances for this experiment. We immediately see that the precision, recall and $F_1$ scores for the classifier obtained by averaging of the individual layer classifiers now are *higher* (albeit

marginally) than those of the baseline model, even though its raw TOP-1 accuracy is still a bit lower. This means that our (averaged) model performs worse than the baseline model at correctly pinpointing the exact subtoken sequences, but overall it predicts fewer of the subtokens incorrectly (precision) while it also retrieves more of the expected subtokens (recall) than the baseline model. Tables A.4, A.5 and A.6 again show the pair-wise performances of each of the classifiers used within this experiment.

Of interest here is that while the TOP-1 accuracy of the baseline classifier is still a bit higher than the TOP-1 performance of the classifier obtained by averaging over the predictions made by all four layers, the difference in performance is *no longer significant*. Looking at the TOP-2 and TOP-3 accuracies, we see that our averaged classifier now manages to improve upon the baseline in these categories by at least half a percent. It seems that at least our embeddings are of an high enough quality for the model to be able to differentiate between the various labels with enough confidence to at least match and to improve upon the accuracies of the baseline classifier in respectively the TOP-1 and the TOP-2 and TOP-3 categories.

The heuristic combination classifier again manages to (significantly) improve upon the baseline in all TOP-K categories, with a slightly bigger margin in the TOP-1 and TOP-3 categories and a slightly worse margin in the TOP-2 category compared to the previous experiment were no candidates were used for the center position.

Even though the baseline and the averaged classifiers lie very close to each other with respect to the TOP-1 accuracy, table A.4 shows that there is actually quite a bit of disagreement between the two classifiers. Maybe a deeper analysis of the dominating sets for the respective classifiers can shed more light on how to combine both viewpoints to obtain an even better combination classifier.

Table 6.10 shows the validation accuracy for each of the layers with respect to the number of epochs trained. We see that the accuracy of layer 1 already surpasses the best accuracy obtained by our framework in the previous experiment in the first epoch. The accuracy of layer 2 is a bit higher than the accuracy of layer 1 after 6 epochs (which is when layer 1 stopped training). From epoch 6 until epoch 15 there seem to be only marginal improvements in validation accuracy for layer 2. Layer 3 matches but never improves upon the accuracy of layer 2 and seems to be a bit useless on its own, although it might still have contributed something to the accuracy of the averaged classifier. Unfortunately, the validation accuracies for the first 4 epochs of training layer 3 were lost and are not shown in this graph.

| layer | epochs | precision | recall | $F_1$ | TOP-1 | TOP-2 | TOP-3 | time | notes |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | 50.11 | 49.87 | 49.87 | 48.48 | 55.36 | 58.67 | $\sim$ 2.5 days | initial embedding, radius = 2 |
| 1 | 6 | 68.30 | 68.22 | 68.20 | 67.47 | 72.10 | 73.80 | $\sim$ 3 days | [feature maps 0] |
| 2 | 15 | 68.84 | 68.72 | 68.72 | 68.07 | 72.25 | 73.79 | $\sim$ 9 days | [feature maps 1] |
| 3 | 10 | 68.71 | 68.56 | 68.58 | 67.90 | 72.16 | 73.70 | $\sim$ 7 days | [feature maps 2] |
| A | 20 | **69.14** | 69.01 | 69.02 | 68.37 | 72.47 | 74.08 | $\sim$ 21.5 days | averaged over layers 0, 1, 2, 3 |
| B | 22 | 68.89 | 68.82 | 68.82 | 68.38 | 71.91 | 72.98 | $\sim$ 5 days | conditional random field |
| C | - | 69.12 | **69.05** | **69.05** | **68.61** | **72.72** | **74.10** | - days | heuristically combined A and B |

Table 6.30: Accuracy results for Refinement Classifier on the evaluation set (around 2 million predictions total).

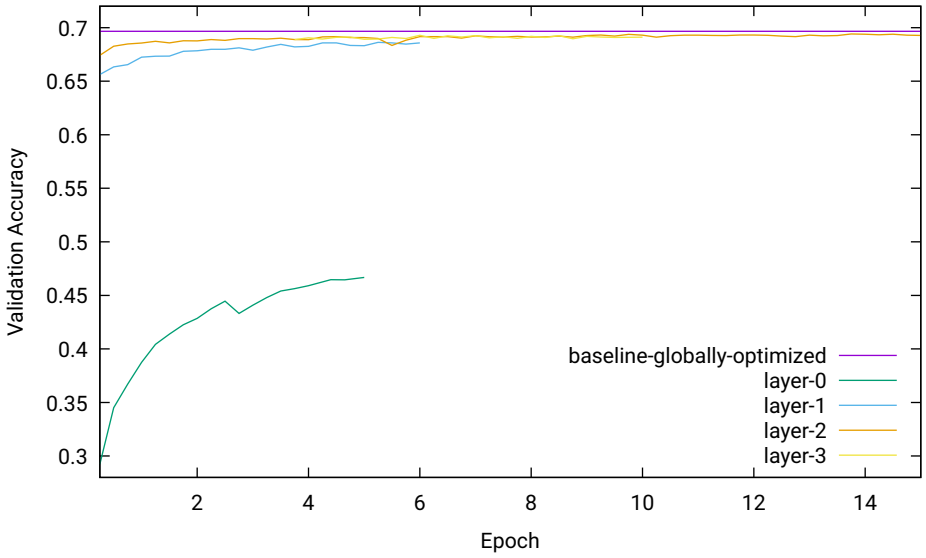


Figure 6.10: Validation accuracies for the Refinement Classifier, measured on a validation set of 5000 programs with approximately 500k predicted labels.

## 6.10 Standalone Experiments

The experiments in the previous section were meant to show some of the qualities of our framework when used for the purpose of extending an existing model by using its candidate predictions for bootstrapping our own predictions. Of course, having these baseline predictions is never strictly necessary, as we can always fill in missing information by using the dummy `MISSING` placeholder. The experiments in this section are designed to probe how well our framework copes when the candidate suggestions have been removed from the information pool.

### 6.10.1 Initial Standalone Embedding Classifier

The initial embedding base layer that was used by all experiments in the previous section was trained to predict center node labels from complete neighborhood contexts. While training, this layer was forced to develop a semantically rich embedding of edge and node labels such that these became excellent features for predicting the center label. For this experiment, we would like to train a standalone base layer that is capable of working with neighborhood contexts where no candidate labels are available for the nodes that still need to be predicted. Even though the contexts now have less information contained in them, the semantic relations established by the base layer from the previous section are still valid, and for our standalone base layer we'd like to reuse these as much as possible.

We start the construction of our our standalone base layer by creating an exact copy of the previous base layer as it was at the end of training the 5th epoch. We then start training this layer for an additional 5 epochs, but now we remove all candidate labels from the training set so that this layer is now forced to learn to predict from partial neighborhoods. This results in a layer that is described by the information shown in figure 6.11 and the tables 6.31 and 6.32. The only difference between these figures and the corresponding figures in the previous section is that here all baseline sourcings (`B`) have been replaced with a missing sourcing (`M`).

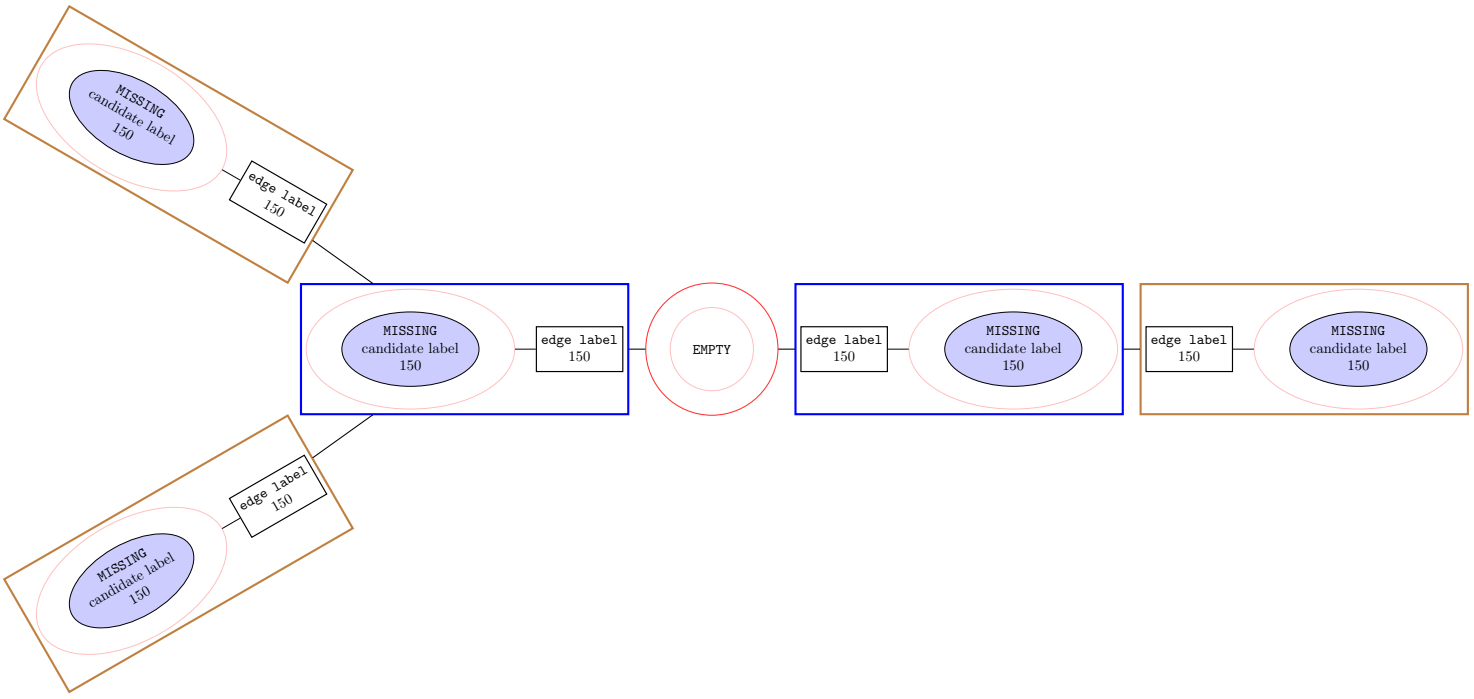


Figure 6.11: Neighborhood layout for layer 0 in the Standalone Experiments.

#### Results

Figure 6.12 shows the validation accuracies of this layer plotted together with the validation accuracies of the previous base layer. We see that the validation accuracies of this standalone base layer are initially higher than

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | - | - | - | 300 | - | 300 | Initial Semantic |
| | 1 | - | 300 | 300 | - | **150** | **150** | Embedding |

Table 6.31: Main Configuration for the Standalone Embedding Classifier.

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 (M) | - | 150 | 150 (M) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |

Table 6.32: Context Configuration for the Standalone Embedding Classifier.

the validation accuracies of the previous base layer during the first epoch. This of course is due to the fact that this layer already starts with its embedding vocabularies fully optimized. After the first epoch, this advantage subsides and after 5 epochs the final validation accuracy of this standalone base layer is quite a bit lower than the validation accuracies of the previous base layer after 5 epochs, which is understandable because this layer has less information to work with. The evaluation results of this base layer are not shown here, but they are present in the tables displaying the results of the experiments shown below that build upon this standalone base layer.

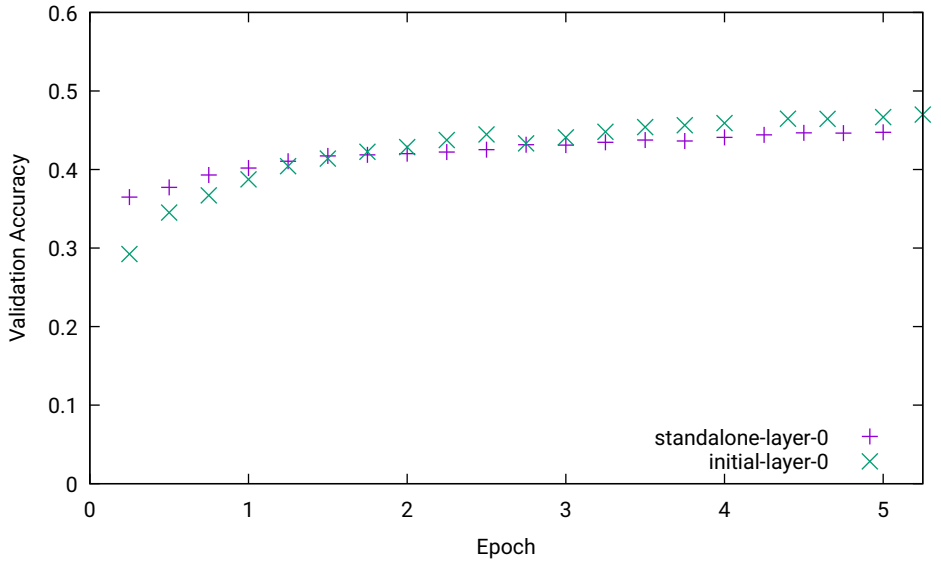


Figure 6.12: Validation accuracies during the standalone embedding phase, measured on a validation set of 5000 programs with approximately 500k predicted labels.

### 6.10.2 Standalone Classifier

With this experiment we build upon the standalone base layer described in the previous subsection. The configuration and neural network topologies are exactly the same as those of the first experiment ('Augmented Classifier') of the previous section. The only difference here in this experiment is that the base layer 0 does not have any candidate suggestions and that all subsequent layers use the candidates generated by their respective predecessor layers instead of the baseline candidates. The available information at the various positions for layers 1 and 2 is shown in figure 6.13, while the same information for layer 3 is shown in figure 6.14. The

adjusted layer data plumbing is shown in tables 6.33 and 6.34.

All layers are initialized with the state of the embedding dictionaries at the end of the previous layer. Layers 2 and 3 have all parameters initialized randomly according to the prescribed default initialization strategies for their respective activation functions (see section 6.3.5). Layer 1 is the exception in that it inherits all parameters from layer 0 whenever these are available in layer 0. This means that the final linear transformation from the 150 result feature maps to the node label space, as well as most of the context and attention parameters are also inherited from the previous layer. The only parameters that are not inherited are those related to the additional feature maps that are added to the context, and these are initialized according to their usual initialization strategies.

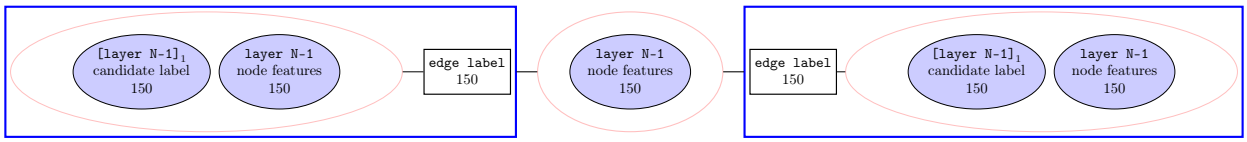


Figure 6.13: Neighborhood layout for layer $N \in \{1, 2\}$ in the Standalone Classifier experiment.

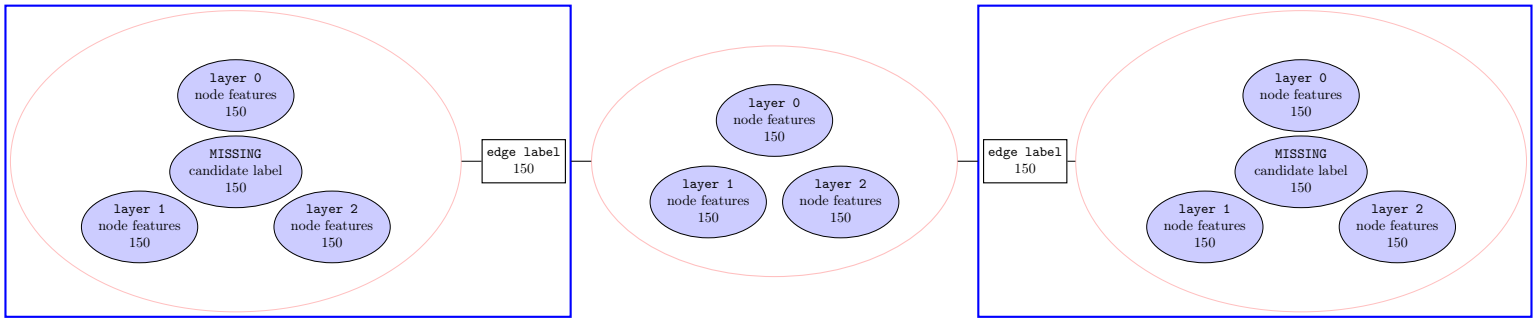


Figure 6.14: Neighborhood layout for layer 3 in the Standalone Classifier experiment.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | - | - | - | 300 | - | 300 | Initial Semantic Embedding |
| | 1 | - | 300 | 300 | - | **150** | **150** | |
| 1 | 0 | - | - | - | 300 | - | 300 | 0 |
| | 1 | - | 150 300 | 450 | - | **150** | **150** | |
| 2 | 0 | - | - | - | 225 | - | 225 | - |
| | 1 | - | 150 225 | 375 | - | **150** | **150** | |
| 3 | 0 | - | 150 150 150 | 450 | 400 | - | 425 | - |
| | 1 | - | 425 | - | - | - | 300 | |
| | 2 | - | 300 | - | - | **150** | **150** | |

Table 6.33: Main Configuration for Standalone Classifier.

### Results

Table 6.35 shows the evaluation accuracies of the various layers in the standalone experiment. We see here that none of the layers come even close in performance to the baseline model in any of the performance metrics. Layer 1 offers a substantial improvement upon the base layer 0 by utilizing the feature maps generated from the latter layer. Layer 2 still managed to improve a bit upon the performance of layer 1, but this increase is not very substantial. Just like in the previous experiment section, layer 3 was designed to see what happens when a layer has access to all final feature maps of all previous layers. Again, just like the corresponding experiment

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 (M) | - | 150 | 150 (M) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 1 | 0 | 50 | 150 | 150 (0) | 150 | - | - | - | 450 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 2 | 0 | 50 | 150 | 150 (1) | 150 | - | - | - | 450 | 225 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 3 | 0 | 50 | 150 | 150 (2) | 150 150 150 | - | - | - | 750 | 400 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| | 2 | 0 | - | - | - | - | - | - | - | - |

Table 6.34: Context Configuration for Standalone Classifier.

in the previous section, this jump in information did not allow the network to squeeze out more performance in the 5 epochs it was given, although it might still be possible that this layer might squeeze out some additional performance when it is given more epochs to properly learn. Figure 6.15 shows the validation accuracies of the various layers plotted against epoch number. The baseline graph shown is the *locally optimized* one from section 6.2.2, which is not nearly as good as the *globally optimized* one (not shown here). Our multi-layer standalone architecture manages to compete with the trivial inference option of the baseline model, while not coming even close to the performance of the actual globally optimized baseline model. Tables A.7, A.8 and A.9 show some of the pair-wise `TOP-K` performances of the various classifiers used in this experiment. The upshot of this experiment is that apparently our architecture is capable enough to correctly predict the target label from a total of about 80k labels in about 60 percent of the cases, which is quite a bit better than mere random guessing.

| layer | epochs | precision | recall | $F_1$ | TOP-1 | TOP-2 | TOP-3 | time | notes |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | 46.55 | 46.24 | 46.27 | 44.85 | 51.84 | 55.20 | $\sim$ 5 days | initial embedding, radius = 2 |
| 1 | 5 | 58.45 | 58.24 | 58.23 | 56.89 | 62.75 | 65.40 | $\sim$ 2 days | [feature maps 0] |
| 2 | 5 | 59.68 | 59.42 | 59.43 | 58.15 | 63.83 | 66.37 | $\sim$ 2.5 days | [feature maps 1] |
| 3 | 5 | 59.40 | 59.15 | 59.16 | 57.84 | 63.64 | 66.26 | $\sim$ 3.5 days | [feature maps 0, 1, 2] |
| A | 20 | 59.91 | 59.64 | 59.66 | 58.38 | 63.93 | 66.44 | $\sim$ 13 days | averaged over layers 0, 1, 2, 3 |
| B | 22 | **68.89** | **68.82** | **68.82** | **68.38** | **71.91** | **72.98** | - days | conditional random field |

Table 6.35: Accuracy results for Standalone Classifier on the evaluation set (around 2 million predictions total).

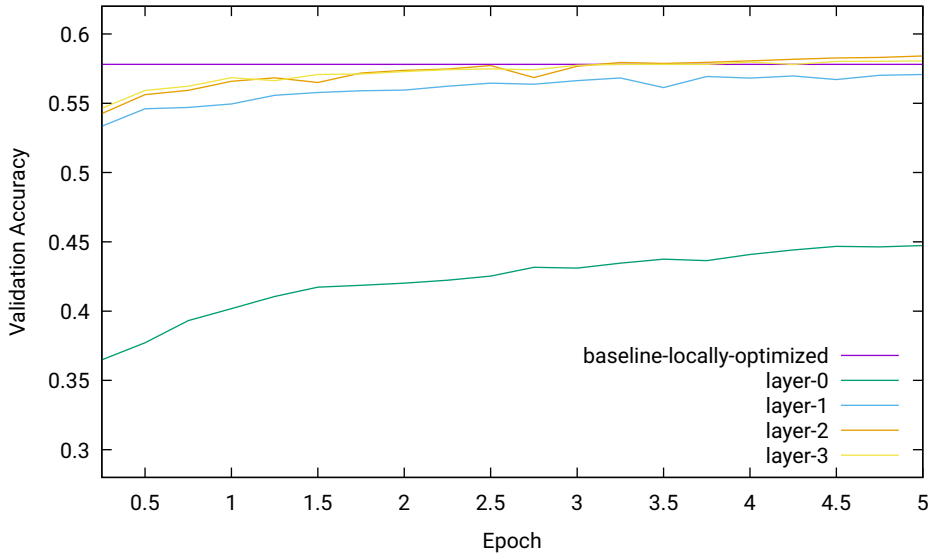


Figure 6.15: Validation accuracies for the Standalone Classifier, measured on a validation set of 5000 programs with approximately 500k predicted labels.

### 6.10.3 Centerless Classifiers

The previous experiment was one attempt at showing the possibilities of using our architecture as a standalone model. This previous experiment was configured (see tables 6.33 and 6.34) to allow each layer to have access to the feature maps generated by the previous layer at both the center and the context positions. Allowing a layer to have access to the generated feature maps of the center location might allow the layer to become a bit lazy: directly passing through the feature maps from the previous layer while ignoring the context would allow the layer to attain exactly the same accuracy as the layer before it without it having to perform any actual additional work. For this experiment we try to see what happens if we remove access to the center feature maps from each layer. Now each higher layer has to base its predictions solely on the information in the context rays, just like base layer 0. The only difference now is that the context rays for the higher layers additionally have the feature maps from the predecessor layer available to them. The main information availability for the layers 1, 2 and 3 is visualized in figure 6.16 while tables 6.36 and 6.37 show how the various layers are interconnected and how information is extracted from the context rays.

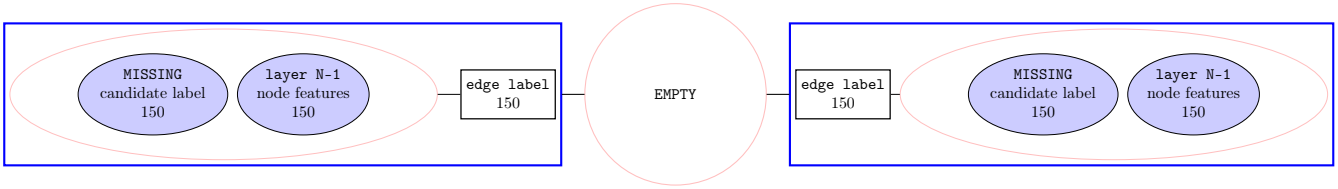


Figure 6.16: Neighborhood layout for layers $N \in \{1, 2, 3\}$ in the Centerless Classifiers experiment.

| layer | stage | center candidate labels | center features | total center input | summarized context input | output labels | output features | inherited layer parameters |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | - | - | - | 300 | - | 300 | Initial Semantic Embedding |
| | 1 | - | 300 | 300 | - | **150** | **150** | |
| 1 | 0 | - | 150 | 150 | 200 | - | 100 | - |
| | 1 | - | 100 | 100 | - | **150** | **150** | |
| 2 | 0 | - | 150 | 150 | 200 | - | 100 | - |
| | 1 | - | 100 | 100 | - | **150** | **150** | |
| 3 | 0 | - | 150 | 150 | 200 | - | 100 | - |
| | 1 | - | 100 | 100 | - | **150** | **150** | |

Table 6.36: Main Configuration for the Centerless Classifiers.

| layer | stage | context pieces | edge 1 labels | node 1 labels | node 1 features | edge 2 labels | node 2 labels | node 2 features | raw context input | summarized context input |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 200 | 150 | 150 (M) | - | 150 | 150 (M) | - | 600 | 300 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 1 | 0 | 50 | 150 | 150 (M) | 150 | - | - | - | 450 | 200 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 2 | 0 | 50 | 150 | 150 (M) | 150 | - | - | - | 450 | 200 |
| | 1 | 0 | - | - | - | - | - | - | - | - |
| 3 | 0 | 50 | 150 | 150 (M) | 150 | - | - | - | 450 | 200 |
| | 1 | 0 | - | - | - | - | - | - | - | - |

Table 6.37: Context Configuration for Centerless Classifier.

**Results**

Table 6.38 shows the measured performances in the usual categories for this experiment. We see that the performances of the higher layers are all about the same and that they are all quite a bit lower than the results of our previous standalone experiment. For this experiment we didn't record any information on the time it took to train each layer because these experiments were performed on a different machine than the other experiments. Tables A.10, A.11 and A.12 show the pair-wise comparisons of respectively the `TOP-1`, `TOP-2` and the `TOP-3` accuracies.

| layer | epochs | precision | recall | $F_1$ | TOP-1 | TOP-2 | TOP-3 | notes |
|---|---|---|---|---|---|---|---|---|
| 0 | 5 | 46.52 | 46.21 | 46.24 | 44.82 | 51.82 | 55.19 | initial embedding, radius = 2 |
| 1 | 5 | 54.53 | 54.27 | 54.25 | 52.71 | 59.42 | 62.51 | [feature maps 0] |
| 2 | 5 | 54.94 | 54.62 | 54.64 | 53.17 | 59.77 | 62.86 | [feature maps 1] |
| 3 | 5 | 54.33 | 53.97 | 54.01 | 52.50 | 59.18 | 62.27 | [feature maps 2] |
| A | 20 | 56.93 | 56.57 | 56.61 | 55.17 | 61.82 | 64.81 | averaged over layers 0, 1, 2, 3 |
| B | 22 | **68.89** | **68.82** | **68.82** | **68.38** | **71.91** | **72.98** | conditional random field |

Table 6.38: Accuracy results for the Centerless Classifiers on the evaluation set (around 2 million predictions total).

## 6.11 Pipelining Experiments

Our framework is designed such that each concrete instantiation of the framework operates as a function that consumes a stream of possibly candidate-augmented graphs and that outputs another stream of graphs that are augmented with the predictions made by the final layer of the instantiation. Conceptually, it does not matter whether the input of such a framework instantiation is the conditional random field baseline classifier or any other candidate-augmenting graph producer. In particular, it is possible for the output of one instantiation of our framework to be consumed as input to another. Our framework actually allows our experiments to be chained this way at will, for example by first running the standalone classifier stack to generate candidates from scratch without any help from the baseline conditional random field, and by then improving these candidates using the refinement classifier stack. Unfortunately, this combination delivered results that were a lot worse than the initial outputs of the standalone stack, and so the dream of obtaining increasingly better solutions by iterating a refinement stack until a fixed point is reached was shattered. In hindsight, this result seems unsurprising, given that the stack is trained to find the best label among only the best 3 candidates, and this is not often the case when the candidates are generated by a badly performing classifier. Perhaps the results would be better if the refinement stack took into account a (much) larger number of candidates than the 3 candidates it is currently trained for.

# Chapter 7

# Related Work

## 7.1 Identifier Inference

### 7.1.1 Statistical Deobfuscation of Android Applications

Bichsel, Raychev, Tsankov, and Vechev, 2016 adapt the conditional random field techniques first developed in (Raychev et al., 2015) to work on obfuscated Java programs. The base techniques used are the same, but the kind of information they model in their knowledge graphs is extended to the *typed* Java domain. While the original framework (which worked on untyped JavaScript programs) was designed to predict only names for local variables, the present framework explicitly does not attempt to make predictions for local variables, but instead makes predictions for all other identifier types *which were not declared in an external API*, i.e. package names, class names, method names etc. The reason for this is that their framework deobfuscates on the level of APK packages, and the original local variable names were never included in the pre-obfuscated packages to begin with.

Working in the typed Java environment, their framework exploits the available static type information by making it available through additional *known nodes* in the knowledge graphs. Their reported performance of an overall prediction accuracy of around 80% is higher than the performance of 63% reported by Raychev et al., 2015 on the `VarNaming` problem in the untyped JavaScript setting.

### 7.1.2 Recovering Clear, Natural Identifiers from Obfuscated JS Names

Vasilescu, Casalnuovo, and Devanbu, 2017 use an off-the-shelf *statistical machine translation* tool called `Moses`, which was originally designed for NLP purposes. The tool works line-by-line and doesn't use any source code specific features. For each identifier they generate candidates for every line where the identifier occurs, using the rest of the tokens on the respective line as context. Identifiers are ranked by the model and the best one is chosen. They go through great lengths to rank the results returned by the tool and to choose an overall assignment which preserves the original program semantics (e.g. no duplicate names within the same scope, which `Moses` does not naturally deal with).

Even though the approach seems to discard quite a lot of structure inherent to source code, it manages to surpass the performance of Raychev et al., 2015 (Raychev et al., 2015), especially in situations where `jsNice` seems to perform badly. The tool they built is dubbed `Autonym`, but they also construct a tool called `jsNaughty` which augments their `Autonym` tool with the predictions made by `JSNice`, hoping to achieve the best of both worlds. They seem to succeed at this, reporting a mean prediction accuracy *per file* which is higher than the reported accuracies of both `jsNice` and `Autonym`.

The `Autonym` tool itself seems to be specifically trained to recover the original variable names for source files which were obfuscated by `UglifyJS` (i.e. it learns the obfuscation algorithm), which would limit the usefulness of the tool in more general situations where there are no original identifier names to begin.

### 7.1.3 Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts

Bavishi, Pradel, and Sen, 2018 use recurrent neural network to embed identifiers in vector space together with their usage contexts. For each usage of an identifier $v$, they collect the $q$ preceding and the $q$ following tokens in the source stream and put them into one-hot encoded context vectors $c \in \{0,1\}^k$ of size $k = \mathit{VocabSize} \cdot 2 \cdot q$. An auto-encoder network is trained to create embeddings for the contexts $c$, where the encoder uses a single layer of recurrent LSTM units to consume the one-hot encoded input into a dense representation. The decoder consists of a single layer of LSTM units, outputting reconstructed contexts $\tilde{c} \in [0,1]^k$ via a softmax layer. The final network is trained to minimize the negative log-loss over the probability $\Pr(\tilde{c} \mid c) := c^T \cdot \tilde{c}$. Afterwards some processing of the results is done to make sure the renamings preserve original program semantics (e.g. no overlapping variable names within the same scope etc).

Given these usage context embeddings, they train a second recurrent LSTM network which takes the first $l$ usages of the variable $v$ and calculates a softmax distribution which is compared with a one-hot encoding of a vocabulary of potential output labels, again using the negative log-loss function.

They report comparable accuracies to `jsNice` and `jsNaughty` while being much more efficient to train. They also mention that `jsNaughty` only achieves good performance when it is allowed to take an unrealistically long time for inference ($> 10$ minutes).

### 7.1.4 Learning to Represent Programs with Graphs

Allamanis, Brockschmidt, and Khademi, 2017 describe how Gated Graph Neural networks can be used in the context of variable identifier naming. Names are predicted for *a single identifier* $Y$ only, using the rest of the program as context. Programs $P$ are represented by knowledge graphs $G^P = (V^P, E^P)$ which are obtained by methods similar to, but not quite the same, as those described in section 4.1. An important difference is that $V^P$ contains a node $Y^k$ for each occurrence $k$ of the variable variable $Y$ within the program whereas in our knowledge graph they are all condensed into a single node. In their paper, Allamanis et al., 2017 also introduce the `VarMisuse` problem which tries to find out whether the variable used at a given location is actually the correct variable to use there, given a list all variables which are in scope at that position.

Each node $Z_i$ has an associated state vector $H_i$ containing its current state. Messages $M_i^r := \mathit{createMessage}^r_{\vec{\alpha}^r}(H_i)$ are calculated from this state which are sent to all nodes $Z_j$ connected to $Z_i$ by the relation $r$. In turn, each node $Z_i$ receives messages $M_j^r$ which were generated by neighboring nodes $Z_j$ which are connected via the relation $r$. Node $Z_i$ aggregates the messages $M_j^r$ thus received into a final summary $\tilde{M}_i := \mathit{aggregateNeighborhood}(\{M_j^r \mid (Z_i, r, Z_j) \in E^P\})$, where the `aggregateNeighborhood` function is a pointwise averaging of the individual messages. Gated recurrent units are used to combine the current state $H_i$ and the aggregated neighborhood information $\tilde{M}_i$ into the successor state $h'_i := \mathit{GRU}_{\vec{\gamma}}(\tilde{M}_i, H_i)$. This propagation of information is repeated for a fixed number of time steps and the final value of the state vector $H_i$ is used as a feature vector to make predictions for the node $Y_i$. The parameters $\vec{\alpha}^r$ and $\vec{\gamma}$ for respectively the message functions and the GRU are determined at training time.

Initially the states $H^k$ of nodes $Y^k \in V^P$ associated to $Y$ get initialized by the embedding of a special `UNKNOWN` placeholder feature, while all other other nodes $Z_i$ get initialized (presumably) by a `word2vec` embedding of their respective tokenized names. The information propagation process described in the previous paragraph is then iterated for the fixed number of time steps, resulting in a feature vector $H^k$ for each location $Y^k$ which summarize the usage of the variable $Y$ at location $k$. The final feature vector calculated for $Y$ is now obtained by averaging the feature vectors obtained at the nodes $Y^k$. This feature vector is then used by another GRU to generate the tokens for the name predicted for identifier $Y$.

The reported accuracy for the `VarNaming` problem on unseen data is 44.0% using their own C# data set, which seems substantially lower than the accuracy reported by Raychev et al., 2015 on a JavaScript data set.

### 7.1.5 Learning Natural Coding Conventions

Allamanis et al., 2015 present `Naturalize`, a framework for predicting variable names using an n-gram model. They treat each identifier as an isolated prediction task, assuming the rest of the identifiers have all been correctly named. Predictions are only made when a certain confidence threshold has been exceeded; a satisfactory comparison of their results which our baseline CRF model could not be made.

### 7.1.6 Suggesting Accurate Method and Class Names

Allamanis et al., 2015 predict variable, method and class names using a neural network model. The predictions made are at the level of subtokens, i.e. a descriptive identifier name is predicted by iteratively generating the individual words $n_i$ (subtokens) which together compose as the identifier's description. The authors construct a log-bilinear probabilistic model where the log-probability $\log \Pr(n_i \mid \vec{c}, n_{i-1}, n_{i-2}, \ldots, n_1)$ for the current token $n_i$, conditioned upon the previously generated tokens $n_{i-1}$, $n_{i-2}$, $\ldots$, $n_1$ and the contextual code sequence $\vec{c} := (c_1, c_2, \ldots, c_k)$, is a bilinear function of respectively the continuous features associated to the current token $n_i$ and the continuous features associated to both the previously predicted subtokens $n_{i-1}$, $\ldots$, $n_1$ and the code context $\vec{c}$ surrounding the identifier which is currently being predicted. Their method does not make any effort for attaining mutual consistency between the predicted identifier names when multiple names are predicted. Predictions are only made when the model has attained a certain (configurable) confidence level on the result; the results presented are not directly comparable to our baseline model.

## 7.2 Embedding Techniques

The following papers describe how various discrete entities can be embedded into some continuous d-dimensional vector space.

### 7.2.1 Translating Embeddings for Modeling Multi-relational Data

Bordes, Usunier, Garcia-Durán, Weston, and Yakhnenko, 2013 show how directed knowledge graphs (see section 4.1) with mainly one-to-one relations can be embedded in some finite dimensional vector space such that the embedded entities approximately respect the head-to-tail arithmetic: for a relation $a \xrightarrow{r} b$ we obtain an embedding $\tilde{a}$, $\tilde{r}$, $\tilde{b} \in \mathbb{R}^d$ of the respective components such that $\tilde{b} \sim \tilde{a} + \tilde{r}$. Using this arithmetic, a relation triple $p \xrightarrow{r} q$ with one of the three components missing can be completed by solving for the unknown within the vector space, i.e. by finding the entity or relation which after embedding best fits the unknown slot.

Their approach doesn't work well with one-to-many and many-to-one relations because the vector space embedding forces two entities to occupy the same location whenever the other entity component and the relationship are shared. This means that unrelated entities will almost surely be thought of as equal as soon as more complex relationships enter to knowledge graph.

### 7.2.2 Knowledge Graph Embedding by Translating on Hyperplanes

Wang, Zhang, Feng, and Chen, 2014 deal with the problem of relations which are not one-to-one (e.g. many-to-one or one-to-many) in knowledge graphs. They improve upon the work of Bordes et al., 2013 by associating to each embedded relation a relation-specific hyperplane and by forcing the `head`-and-`tail` arithmetic to happen on the projections to the hyperplane of their respective embeddings, instead of on the embeddings themselves. This allows entities which are related to multiple other entities to get a relation-specific space to perform the arithmetic in, which prevents the degenerate situation where two different tails must occupy the same location as soon as they are part of a triple in which they share the same head and relation.

### 7.2.3 Representation Learning of Knowledge Graphs with Entity Descriptions

Xie, Liu, Jia, Luan, and Sun, 2016 extend the approach taken by Bordes et al., 2013 of embedding knowledge graphs in $\mathbb{R}^d$ with a complementary embedding of textual descriptions associated to the various entities. This embedding (termed DKRL) is learned in such a way such that both the embedding of the entity as well as the embedding of the accompanying description are encouraged to occupy the same location in $\mathbb{R}^d$. This allows the model to work with entities that were not available during model construction, as long as they have an accompanying textual description. The embedding of this textual description can be used as a placeholder for the embedding of the entity itself and it can be obtained by a generic pre-trained word embedding model (`word2vec`).

### 7.2.4 Learning Graph-Level Representations with Gated Recurrent Neural Networks

Jin and JáJá, 2018 have shown how recurrent neural networks can be used to create vector space embeddings of (possibly large) graphs given the embeddings of the individual nodes and their respective neighborhoods. The nodes are put into a sequence, and the sequence of embeddings of nodes together with their neighborhoods is fed into a recurrent neural network which then calculates a summary for this sequence which represents the whole graph. Putting the nodes into a specific ordered sequence is required for the recurrent neural network to function, but the precise order of this sequence greatly influences the final result, and so determining a good order is crucial for this whole graph embedding to function properly. In their paper, Jin and JáJá, 2018 describe how the Gumbal-Softmax distribution can be used to randomly sample a sequence of nodes in a way which still allows the network to be trained by using standard gradient descent based techniques. The random sequence sampling can be interpreted as a random walk around the graph, with the randomness in the node sequence helping to prevent any order-biases to creep in which can negatively influence the final result. Their work shows that this embedding technique compares favorably to competing graph embedding techniques in the task of predicting a label with classifies the whole graph.

## 7.3 Attention Mechanisms

The papers listed here describe different attention mechanisms that are used to put emphasis on the relevant parts of the input that is needed to calculate the desired output.

### 7.3.1 Summarizing Source Code using a Neural Attention Model

Iyer, Konstas, Cheung, and Zettlemoyer, 2016 use an attention mechanism to predict natural language summaries for code fragments. The code fragment $C$ is split into its token sequence $\vec{c} := (c_1, c_2, \ldots, c_k)$.

The goal is to predict the next word $n_i$ of the summary given the vector of previously predicted words $\vec{n}_{i-1} := (n_{i-1}, n_{i-2}, \ldots, n_1)$ and the token sequence $\vec{c}$. This task is modeled by calculating a conditional probability distribution $\Pr(n_i \mid \vec{c}, \vec{n}_{i-1})$ of the next word to be predicted for the summary given the words already predicted. The result is chosen via a beam search to be the word sequence $\hat{n}$ that maximizes the joint probability over the individual words.

The probability $\Pr(n_i \mid \vec{c}, \vec{n}_{i-1})$ of the current word $n_i$, is calculated via a weighted average $\hat{c}_i$ of embedding vectors $\tilde{c}_j \in \mathbb{R}^d$ of the tokens $c_j$ in the sequence $\vec{c}$. This ensures that each word predicted pays the most attention to the tokens in the fragment that are most important for that word, given the words already predicted previously in the summary. We have

$$\hat{c}_i := \sum_j \mathbb{A}(\tilde{c}_j, \vec{h}_i) \cdot \tilde{c}_j \tag{7.1}$$

where the weights $\mathbb{A}(\tilde{c}_j, \vec{h}_i) \in \mathbb{R}$ are proportional to the dot product between the embedded token $\tilde{c}_j$ and the current hidden state $\vec{h}_i$ of a LSTM unit. This attention mechanism differs from ours in that their attention is *dynamic*, being taken with respect to the current state vector of the LSTM unit, while ours is *static*, being taking against a parameter vector which is global to the neural network.

The words in their natural language summaries might serve as good variable names when adequately concatenated when they are applied to the neighborhood triple embeddings that are described in this paper.

### 7.3.2 A Convolutional Attention Network for Extreme Summarization of Source Code

Allamanis, Peng, and Sutton, 2016 use a convolutional neural network for the task of predicting natural language summaries $N$ for code fragments $C$. The fragment is split into its token sequence $\vec{c} := (c_1, c_2, \ldots, c_k)$.

Again, the goal is to predict the next word $n_i$ of the summary given the vector of previously predicted words $\vec{n}_{i-1} := (n_{i-1}, n_{i-2}, \ldots, n_1)$ and the token sequence $\vec{c}$. This task is modeled by calculating a conditional probability distribution $\Pr(n_i \mid \vec{c}, \vec{n}_{i-1})$ of the next word to be predicted for the summary given the words already predicted.

The probability $\Pr(n_i \mid \vec{c}, \vec{n}_{i-1})$ of the current word $n_i$, is calculated via a weighted average $\hat{c}_i$ of embedding vectors $\tilde{c}_j \in \mathbb{R}^d$ of the tokens $c_j$ in the sequence $\vec{c}$. This ensures that each word predicted pays the most attention to the tokens in the fragment that are most important for that word, given the words already predicted previously in the summary. We have

$$\hat{c}_i := \sum_j \mathbb{A}_{\vec{\phi}}(\vec{c}, \vec{h}_i, j) \cdot \tilde{c}_j \tag{7.2}$$

where the weight $\mathbb{A}_{\vec{\phi}}(\vec{c}, \vec{h}_i, j) \in \mathbb{R}$ used at the $j$th token is calculated via a stack of convolutional layers. For each position $j$, each convolution layer calculated a new attention feature vector $F_j$ that combines the neighborhood information surrounding the token obtained in the previous layer. The last layer of these feature vectors, $\hat{F}_j$, is combined with the current state $\vec{h}_i$ of a GRU unit to obtain at each position an attention feature vector $A_{i,j}$ that combines the prediction history of the previously predicted words $\vec{n}_{i-1}$ with the attention information obtained via the convolution over the source tokens. In a final convolutional layer the vectors $A_{i,j}$ are combined to obtain for each position $j$ the weight $\mathbb{A}_{\vec{\phi}}(\vec{c}, \vec{h}_i, j)$ which indicates the importance of the embedding $\tilde{c}_j$ of the $j$th token.

At each time step, the current hidden state $\vec{h}_i$ of the GRU unit is obtained from the previous hidden state $\vec{h}_{i-1}$ and the embedding $\tilde{a}_{i-1}$ of the last predicted word via $\vec{h}_i := GRU(\tilde{a}_{i-1}, \vec{h}_{i-1})$. The final probabilities for the word $n_i$ given $\vec{n}_{i-1}$ and $\vec{c}$ are calculated via a softmax on $\hat{c}^i$ via a linear layer.

Just like the previous paper, the words obtained from these summaries might serve as good variable names when this mechanism is applied to the triple embeddings in a neighborhood surrounding the variable in the knowledge graph. Each subtoken predicted would get its own attention view on the neighborhood, which differs from the single global attention function described in our current work.

### 7.3.3 Graph2Seq: Graph to Sequence Learning with Attention-based Neural Networks

Xu, Wu, Wang, Feng, and Sheinin, 2018 use recurrent neural networks to generate token sequences for graph inputs. For a given graph $G = (V, E)$ with vector features attached to each node $Z_j \in V$, the goal is to generate a sequence of tokens $n_i$. They do this by modeling a conditional probability distribution $\Pr(n_i \mid G, \vec{n}_{i-1})$ where the probability of generating a token $n_i$ is conditioned on the vector $\vec{n}_{i-1} := (n_{i-1}, n_{i-2}, \ldots, n_1)$ of tokens generated previously.

They describe how the information in node neighborhoods can be aggregated to obtain a neighborhood summary vector $\tilde{c}_j \in \mathbb{R}^d$ for each node $Z_j$. Then, for the prediction of a token $n_i$, a context vector $\hat{c}_i$ is calculated by averaging over the neighborhood summary vectors $\tilde{c}_j$ using attention weights:

$$\hat{c}_i := \sum_j \mathbb{A}_{\vec{\phi}}(\tilde{c}_j, \vec{h}_i) \cdot \tilde{c}_j \tag{7.3}$$

The attention weights $\mathbb{A}_{\vec{\phi}}(\tilde{c}_j, \vec{h}_i)$ used for the predicted token $n_i$ are calculated by a separate neural neural network (via a soft-max layer) which is trained in parallel with the other parameters. The context vector $\hat{c}_i$ is used in combination with the current hidden state $h_i$ of an LSTM unit and the embedding $\tilde{n}_{i-1}$ of the previously predicted token $n_{i-1}$ to calculate the probabilities for the current token $n_i$.

This attention mechanism differs from ours in that we use a simple attention *vector* $a$ to obtain weights for each embedded triple $\vec{c}_j$ by simply calculating the plain dot product between $\vec{c}_j$ and $a$ (and performing a subsequent soft-max operation), while their weights are determined via a possibly arbitrary complex neural network. We've chosen the simpler approach from (Alon et al., 2019) because we like its simplicity.

### 7.3.4 Learning Programs from Noisy Data

Raychev et al., 2016 do generative learning, i.e. they try to model the distribution of code fragments so that they can generate adequate code completions. This is not what we're interested in: we take the code as a static entity where only semantic information in the form of identifiers needs to be inferred. They were kind enough to supply their 150k JavaScript code corpus.

# Chapter 8

# Conclusion

## 8.1 Research Questions

With this project, we tried to answer the following research questions:

1. Can we (significantly) improve upon the conditional random field model by using a complementary (convolutional) neural network which uses higher order neighborhoods for increased information utilization?

2. Can we get our model to synergize with the original model, i.e. can the combination of two models overcome each other's weaknesses for a better overall result?

3. Can we do all this while maintaining the real-time inference property of the original conditional random field framework?

Our first experiment ('Refinement') showed us that we can answer the first research question in a positive way: by combining our convolutional neural classifier with the baseline conditional random field model, we obtain a classifier that is superior to both the baseline classifier and our own classifier. Improvement in `TOP-1` accuracy is marginal but statistically significant, while the `TOP-2` and `TOP-3` accuracies have seen substantial improvements of over 1.5%.

Besides refining an existing model, the architecture we've designed is also capable to function as a standalone model. When used as such, we managed to achieve prediction accuracies of around 59 percent. This is much lower than the 69 percent accuracy achieved by the fully, globally optimized, conditional random field model, but it is very much comparable to the 58 percent accuracy achieved by the locally optimized version of the baseline classifier that predicts its labels from assuming ideal neighborhoods. It would seem that our predictive capabilities in this regard are not as globally informed as we would have liked, as our multi-layer, information aggregating framework manages to squeeze out precisely as much predictive performance as the simple local optimization maximization strategy of the local inference version of the conditional random field framework. Also, the (global) conditional random field model already achieves similar accuracies as our best standalone model when trained on a dataset of only around 5000 programs, while our framework needs a full 10+ epochs over a dataset of 100k programs to achieve similar results. All in all it seems that our model does not offer many advantages when used only for classification purposes. On the other hand, an interesting side-effect of our architecture is the continuous semantic embedding of small graph neighborhoods in some finite dimensional real vector space. This embedding might contain information which has utility beyond the direct prediction of missing labels in our current setting, for example in topic segmentation etc. It remains to be seen whether there are any domains where our standalone architecture has a clear advantage over use of conditional random fields as per (Raychev et al., 2015), either for direct predictive purposes or by the utilization of its neighborhood embeddings.

Unfortunately we were unable to start work on the second research question. Our experiments have shown (see A.2) that there is quite a bit of disagreement between our classifiers and the baseline conditional random field model. Our original hypothesis which led to this research question was that the baseline model gets stuck in local optima during inference (see section 1.5.3). This means that it is unable to find a solution which is globally optimal according to the model, even though it would be able to confirm the superiority of such an

optimal solution if it was magically handed such a solution. We still believe that an exchange of information between our model and the baseline conditional random field model might allow the baseline model to break free of its current local optimum in quite a few situations, allowing it to continue its search and eventually reach a better (but probably still local) optimum. Iterating this process might result in an even better final solution.

Concerning the third research question: we can confirm that running our whole prediction pipeline does not take more than a second of running time on average. This means that the real-time inference property of the original conditional random field model is maintained, and that the integration of our refinement model into an integrated development environment would still result in a feature that can be used on-the-fly while developing without adding any significant delays that might preclude it from using used.

## 8.2 Future Work

### 8.2.1 Enhanced Combination Heuristic

Currently the baseline classifier and our classifiers are combined via a very simple heuristic that uses a single threshold value to decide whether the suggestions returned by the baseline model should be swapped. Something more clever is probably possible. One obvious approach would be to train a small (logistic) neural network that uses the scores returned by each of the classifiers to obtain a better decision procedure on which pairs of candidates should be swapped to obtain a better final candidate ranking.

### 8.2.2 Utilizing Conditional Random Field Weights as Features

In our current (refinement) experiments we only use the raw `TOP-3` candidates returned by the conditional random field model as input to our neural network, without any consideration of the scores the former model already assigned to these labels when seen in their current contexts. Of course, these scores are very informative, and they probably should already have been used as extra features given to our model. Unfortunately, our insight as to their utility came a bit too late for us to pursue this any further, so this remains something to be done.

### 8.2.3 Bottleneck Feature Maps

Currently we train the feed-forwards neural network used in each convolutional layer to predict the center label from its neighborhood context, and then we choose the feature maps generated by the final *stage* of this feed-forward neural network to be the designated output for the node at this convolutional layer. Another possibility would be to take the feature maps generated at any of the intermediate stages as the final output. Currently it might be the case that the feature maps at the final stage are too much focussed on being good predictors for the center label while throwing away neighborhood information that is not directly relevant at this prediction task but which might become relevant when the features would be used as input to any subsequent layers. If we use any of the intermediate layers, we might intercept the information flow at a point where it is not yet committed to the final prediction task, which might result in features that are more generally useful.

### 8.2.4 More Sophisticated Training Objectives

Our current framework trains each layer separately with the training objective of predicting the label for the center node of a neighborhood directly from all other information contained in the neighborhood. This objective forces the layer to generate features which are good direct predictors for the center label, but unfortunately this might not be a good long term strategy. Features which are not directly useful in predicting the final label will never be generated, even though they might indirectly become very useful when used in a more complex hierarchy. A different training objective might allow a layer to generated a more diverse set of features, instead of forcing it to be greedy and only judge features by their direct utility. One such an alternative approach might be to train each layer as an *(variational) autoencoder*, where the training objective is to accurately reconstruct the input neighborhood from the generated features. This would force the network to preserve as much information of the original neighborhood as possible, instead of only preserving information that is directly relevant for predicting the center label. Our current architecture would be extended by a reverse network which takes as input the final feature map of a layer and outputs something which can be compared to the original input neighborhood. The training objective would then be to let this combination minimize (according to some yet to

be decided metric) the difference between the original neighborhood and the reconstructed neighborhood. How this comparison between the initial and final neighborhoods is to be done is still an open research question, given that these neighborhoods are all varying in size and that there is no definitive ordering on the various pieces of information.

### 8.2.5 Abstract Syntax Tree Path Features

We currently use the feature set first introduced in (Raychev et al., 2015), but since then the authors of (Alon et al., 2018) have managed to obtain even better results using the same base model by using an extended feature set (i.e., a different set of features used as edges in the knowledge graphs), obtained from the paths between variables in the abstract syntax tree of the original programs. They managed to improve upon the original accuracies by a few percent. It would be interesting to see whether our model is also able to improve upon the results of this extended baseline model.

### 8.2.6 Languages with Sophisticated Naming Conventions

Our current experiments have been performed on a JavaScript dataset, where the conditional random field already achieved relatively high accuracies of around 70%. We'd say it's plausible that the untyped nature of JavaScript doesn't really encourage descriptive variable names, which would mean that the variety of names which actually occur in practice is limited. This might explain why symbolic approaches like CRFs achieve such high performance. In strongly typed languages such as Java, most method names are very long and specific (e.g. `findPrivateCustomerInPreciouslyGenericDatabase`, which greatly increases the number of unique names in the dataset. The abundance of such very specific names of course makes it much harder to predict the correct name when there are many very similar words in the training vocabulary which only differ only slightly at just a few subtokens. As we've seen, our Refinement Classifier already manages to have better precision and recall scores than the baseline conditional random field classifier, even though its absolute `TOP-1` accuracy is still lagging a bit behind. This gives us an indication that it is quite capable of getting the general gist of a prediction correctly (e.g. `findCustomerInDatabase` or even `findCustomer`), even though it still struggles a bit with pinpointing the exact token sequence. It would be nice to investigate how our model compares in precision and recall scores with the conditional random field model when it is applied in to strongly typed programs where there are many small variations of the same basic label.

### 8.2.7 More Representative Performance Measurements

We already mentioned in the previous item that our model manages to improve upon the baseline model in precision and recall scores, even though it struggles with improving upon the baseline model at raw TOP-1 accuracy. A more sophisticated analysis of the predictions might shed more light on the situation. For example, we could compare the union of the expected subtokens over a connected component with the union of generated tokens over the same connected component. This way a noun subtoken like `customer` or `employee` is only taken into account once when it is consistently wrongly suggested or missing over multiple predictions. This might give a better indication on model's ability to correctly predict the generic features of the required label when the precise prediction is incorrect.

### 8.2.8 Subtoken Level Enhancements

Our classifier generates a number of suggestions which might all be variations of the same idea. For example, if the TOP-4 candidates (in returned order) are `drawRectangle`, `findCustomer`, `findEmployee` and `findManager`, we might draw the conclusion that the result is likely to contain the subtoken `find`. Then some search within the predictions made for neighboring nodes might reveal that multiple suggestions independently mention the subtoken `employee`, which would make the `findEmployee` prediction the most likely, even though it was not returned as the top suggestion by our own classifier.

### 8.2.9 Deeper Networks

Increasing the number of (framework) *layers* beyond 3 became infeasible on our hardware, but increasing the number of *stages* (i.e. the neural network layers within a framework layer) did not seem to drastically increase

the time needed to train a single layer. Our framework does not have absurdly high accuracies on the training set yet; it seems like it's not overfitting. Deeper networks (in the sense of having more stages per layer) might be worthwhile to look at.

### 8.2.10 Larger/Better/Same Dataset

We currently use the publicly available dataset that was first used by Raychev et al., 2016 in their own experiments. Unfortunately, this dataset is only half the size as the dataset that Raychev et al., 2015 originally used in their conditional random field experiments. The performance measurements we made whilst repeating their experiments on our chosen dataset show a substantial increase in accuracy of around 5% of the (almost identically configured) model trained on our dataset compared to the results reported in (Raychev et al., 2015). This increase in performance might be an indication that our dataset is either less diverse or of a worse quality compared to theirs (ours might have more code duplication between the training and evaluation sets), or that we just did something wrong in our performance assessment. A first step towards answering why our results differ so much from the results reported in the original experiments might be to repeat our experimental procedures on their exact dataset.

# Bibliography


Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 281–293). FSE 2014. Hong Kong, China: ACM. doi:10.1145/2635868.2635883

Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2015). Suggesting accurate method and class names. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 38–49). ESEC/FSE 2015. Bergamo, Italy: ACM. doi:10.1145/2786805.2786849

Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to represent programs with graphs. *CoRR*, *abs/1711.00740*. arXiv: 1711.00740. Retrieved from http://arxiv.org/abs/1711.00740

Allamanis, M., Peng, H., & Sutton, C. (2016). A convolutional attention network for extreme summarization of source code. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning* (Vol. 48, pp. 2091–2100). Proceedings of Machine Learning Research. New York, New York, USA: PMLR. Retrieved from http://proceedings.mlr.press/v48/allamanis16.html

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2018). A general path-based representation for predicting program properties. In *Proceedings of the 39th acm sigplan conference on programming language design and implementation* (pp. 404–419). PLDI 2018. Philadelphia, PA, USA: ACM. doi:10.1145/3192366.3192412

Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang. 3*(POPL), 40:1–40:29. doi:10.1145/3290353

Bavishi, R., Pradel, M., & Sen, K. (2018). Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *CoRR*, *abs/1809.05193*. arXiv: 1809.05193. Retrieved from http://arxiv.org/abs/1809.05193

Bichsel, B., Raychev, V., Tsankov, P., & Vechev, M. (2016). Statistical deobfuscation of android applications. In *Proceedings of the 2016 acm sigsac conference on computer and communications security* (pp. 343–355). CCS '16. Vienna, Austria: ACM. doi:10.1145/2976749.2978422

Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., & Sharif, B. (2013). The impact of identifier style on effort and comprehension. *Empirical Softw. Engg. 18*(2), 219–276. doi:10.1007/s10664-012-9201-4

Bordes, A., Usunier, N., Garcia-Durán, A., Weston, J., & Yakhnenko, O. (2013). Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th international conference on neural information processing systems - volume 2* (pp. 2787–2795). NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2999792.2999923

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh & M. Titterington (Eds.), *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (Vol. 9, pp. 249–256). Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR. Retrieved from http://proceedings.mlr.press/v9/glorot10a.html

Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 2073–2083). Berlin, Germany: Association for Computational Linguistics. doi:10.18653/v1/P16-1195

Jin, Y. & JáJá, J. F. (2018). Learning graph-level representations with gated recurrent neural networks. *CoRR*, *abs/1805.07683*. arXiv: 1805.07683. Retrieved from http://arxiv.org/abs/1805.07683

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv: 1412.6980 [cs.LG]

Lafferty, J. D., McCallum, A., & Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the eighteenth international conference on machine learning* (pp. 282–289). ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=645530.655813

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006). What's in a name? a study of identifiers. In *14th ieee international conference on program comprehension (icpc'06)* (pp. 3–12). doi:10.1109/ICPC.2006.51

Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th annual psychology of programming interest group workshop* (Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop). Retrieved from https://www.microsoft.com/en-us/research/publication/cognitive-perspectives-on-the-role-of-naming-in-computer-programs/

Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. (2013). Efficient estimation of word representations in vector space. Retrieved from http://arxiv.org/abs/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th international conference on neural information processing systems - volume 2* (pp. 3111–3119). NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2999792.2999959

Raychev, V., Bielik, P., Vechev, M., & Krause, A. (2016). Learning programs from noisy data. In *Proceedings of the 43rd annual acm sigplan-sigact symposium on principles of programming languages* (pp. 761–774). POPL '16. St. Petersburg, FL, USA: ACM. doi:10.1145/2837614.2837671

Raychev, V., Vechev, M., & Krause, A. (2015). Predicting program properties from "big code". *SIGPLAN Not. 50*(1), 111–124. doi:10.1145/2775051.2677009

Sonoda, S. & Murata, N. (2015). Neural network with unbounded activation functions is universal approximator. doi:10.1016/j.acha.2015.12.005

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* (pp. 683–693). ESEC/FSE 2017. Paderborn, Germany: ACM. doi:10.1145/3106237.3106289

Wang, Z., Zhang, J., Feng, J., & Chen, Z. (2014). Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the twenty-eighth aaai conference on artificial intelligence* (pp. 1112–1119). AAAI'14. Québec City, Québec, Canada: AAAI Press. Retrieved from http://dl.acm.org/citation.cfm?id=2893873.2894046

Xie, R., Liu, Z., Jia, J., Luan, H., & Sun, M. (2016). Representation learning of knowledge graphs with entity descriptions. In *Proceedings of the thirtieth aaai conference on artificial intelligence* (pp. 2659–2665). AAAI'16. Phoenix, Arizona: AAAI Press. Retrieved from http://dl.acm.org/citation.cfm?id=3016100.3016273

Xu, K., Wu, L., Wang, Z., Feng, Y., & Sheinin, V. (2018). Graph2seq: Graph to sequence learning with attention-based neural networks. *CoRR*, *abs/1804.00823*. arXiv: 1804.00823. Retrieved from http://arxiv.org/abs/1804.00823

# Appendix A

# Experimental Results

## A.1 Augmented Classifiers

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 4466 | 1428 | +3038 | +0.16% | 0.0000 |
| C | A | 177434 | 69726 | +107708 | +5.65% | 0.0000 |
| C | 3 | 186093 | 62153 | +123940 | +6.50% | 0.0000 |
| C | 2 | 182480 | 66857 | +115623 | +6.07% | 0.0000 |
| C | 1 | 205095 | 72319 | +132776 | +6.97% | 0.0000 |
| C | 0 | 439631 | 56333 | +383298 | +20.11% | 0.0000 |
| B | A | 178862 | 74192 | +104670 | +5.49% | 0.0000 |
| B | 3 | 187436 | 66534 | +120902 | +6.34% | 0.0000 |
| B | 2 | 183818 | 71233 | +112585 | +5.91% | 0.0000 |
| B | 1 | 206492 | 76754 | +129738 | +6.81% | 0.0000 |
| B | 0 | 440611 | 60351 | +380260 | +19.95% | 0.0000 |
| A | 3 | 41066 | 24834 | +16232 | +0.85% | 0.0000 |
| A | 2 | 30473 | 22558 | +7915 | +0.42% | 0.0000 |
| A | 1 | 45071 | 20003 | +25068 | +1.32% | 0.0000 |
| A | 0 | 297482 | 21892 | +275590 | +14.46% | 0.0000 |
| 3 | 2 | 36624 | 44941 | -8317 | -0.44% | 0.0000 |
| 3 | 1 | 63659 | 54823 | +8836 | +0.46% | 0.0000 |
| 3 | 0 | 300885 | 41527 | +259358 | +13.61% | 0.0000 |
| 2 | 1 | 57483 | 40330 | +17153 | +0.90% | 0.0000 |
| 2 | 0 | 306951 | 39276 | +267675 | +14.04% | 0.0000 |
| 1 | 0 | 278074 | 27552 | +250522 | +13.14% | 0.0000 |

Table A.1: Pairwise TOP-1 comparison of the Augmented Classifier. The (X) dominates-columns indicate the number of times classifier X made a prediction which was correct while the prediction made by the other classifier classifier was incorrect.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 31315 | 12499 | +18816 | +0.99% | 0.0000 |
| C | A | 195917 | 86077 | +109840 | +5.76% | 0.0000 |
| C | 3 | 209592 | 77911 | +131681 | +6.91% | 0.0000 |
| C | 2 | 203668 | 82812 | +120856 | +6.34% | 0.0000 |
| C | 1 | 223956 | 88725 | +135231 | +7.09% | 0.0000 |
| C | 0 | 470525 | 69272 | +401253 | +21.05% | 0.0000 |
| B | A | 200402 | 96675 | +103727 | +5.44% | 0.0000 |
| B | 3 | 211852 | 87986 | +123866 | +6.50% | 0.0000 |
| B | 2 | 206622 | 92787 | +113835 | +5.97% | 0.0000 |
| B | 1 | 227864 | 98535 | +129329 | +6.79% | 0.0000 |
| B | 0 | 470275 | 77869 | +392406 | +20.59% | 0.0000 |
| A | 3 | 63726 | 43750 | +19976 | +1.05% | 0.0000 |
| A | 2 | 50315 | 40962 | +9353 | +0.49% | 0.0000 |
| A | 1 | 69056 | 36438 | +32618 | +1.71% | 0.0000 |
| A | 0 | 345518 | 33374 | +312144 | +16.38% | 0.0000 |
| 3 | 2 | 55725 | 67181 | -11456 | -0.60% | 0.0000 |
| 3 | 1 | 88923 | 78297 | +10626 | +0.56% | 0.0000 |
| 3 | 0 | 350533 | 57868 | +292665 | +15.35% | 0.0000 |
| 2 | 1 | 82660 | 59948 | +22712 | +1.19% | 0.0000 |
| 2 | 0 | 356646 | 54918 | +301728 | +15.83% | 0.0000 |
| 1 | 0 | 330498 | 40938 | +289560 | +15.19% | 0.0000 |

Table A.2: Pairwise TOP-2 comparison of the Augmented Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-2 prediction that was strictly better than prediction made by the other classifier.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 46264 | 17512 | +28752 | +1.51% | 0.0000 |
| C | A | 205916 | 94133 | +111783 | +5.86% | 0.0000 |
| C | 3 | 220560 | 85599 | +134961 | +7.08% | 0.0000 |
| C | 2 | 214154 | 90841 | +123313 | +6.47% | 0.0000 |
| C | 1 | 233642 | 96958 | +136684 | +7.17% | 0.0000 |
| C | 0 | 482537 | 76311 | +406226 | +21.31% | 0.0000 |
| B | A | 208627 | 108494 | +100133 | +5.25% | 0.0000 |
| B | 3 | 220953 | 98869 | +122084 | +6.41% | 0.0000 |
| B | 2 | 215372 | 104084 | +111288 | +5.84% | 0.0000 |
| B | 1 | 236046 | 109648 | +126398 | +6.63% | 0.0000 |
| B | 0 | 480108 | 87807 | +392301 | +20.58% | 0.0000 |
| A | 3 | 77315 | 56064 | +21251 | +1.11% | 0.0000 |
| A | 2 | 62518 | 53292 | +9226 | +0.48% | 0.0000 |
| A | 1 | 83277 | 47404 | +35873 | +1.88% | 0.0000 |
| A | 0 | 369160 | 40409 | +328751 | +17.25% | 0.0000 |
| 3 | 2 | 67549 | 80968 | -13419 | -0.70% | 0.0000 |
| 3 | 1 | 103044 | 91979 | +11065 | +0.58% | 0.0000 |
| 3 | 0 | 374190 | 67384 | +306806 | +16.10% | 0.0000 |
| 2 | 1 | 97423 | 72031 | +25392 | +1.33% | 0.0000 |
| 2 | 0 | 380296 | 63928 | +316368 | +16.60% | 0.0000 |
| 1 | 0 | 356050 | 49113 | +306937 | +16.10% | 0.0000 |

Table A.3: Pairwise TOP-3 comparison of the Augmented Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-3 prediction that was strictly better than prediction made by the other classifier.

## A.2 Refinement Classifiers

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 5813 | 1441 | +4372 | +0.23% | 0.0000 |
| C | A | 64694 | 60140 | +4554 | +0.24% | 0.0000 |
| C | 3 | 68716 | 55235 | +13481 | +0.71% | 0.0000 |
| C | 2 | 65186 | 54930 | +10256 | +0.54% | 0.0000 |
| C | 1 | 84696 | 63057 | +21639 | +1.14% | 0.0000 |
| C | 0 | 439297 | 55672 | +383625 | +20.13% | 0.0000 |
| B | A | 66135 | 65953 | +182 | +0.01% | 0.6185 |
| B | 3 | 70070 | 60961 | +9109 | +0.48% | 0.0000 |
| B | 2 | 66578 | 60694 | +5884 | +0.31% | 0.0000 |
| B | 1 | 86095 | 68828 | +17267 | +0.91% | 0.0000 |
| B | 0 | 439716 | 60463 | +379253 | +19.90% | 0.0000 |
| A | 3 | 27153 | 18226 | +8927 | +0.47% | 0.0000 |
| A | 2 | 19130 | 13428 | +5702 | +0.30% | 0.0000 |
| A | 1 | 34448 | 17363 | +17085 | +0.90% | 0.0000 |
| A | 0 | 401103 | 22032 | +379071 | +19.89% | 0.0000 |
| 3 | 2 | 23537 | 26763 | -3226 | -0.17% | 0.0000 |
| 3 | 1 | 49054 | 40896 | +8158 | +0.43% | 0.0000 |
| 3 | 0 | 406397 | 36253 | +370144 | +19.42% | 0.0000 |
| 2 | 1 | 43499 | 32116 | +11383 | +0.60% | 0.0000 |
| 2 | 0 | 404143 | 30774 | +373369 | +19.59% | 0.0000 |
| 1 | 0 | 383897 | 21911 | +361986 | +18.99% | 0.0000 |

Table A.4: Pairwise TOP-1 comparison of the Refinement Classifiers. The (X) dominates-columns indicate the number of times classifier X made a prediction which was correct while the prediction made by the other classifier was incorrect.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 25777 | 7310 | +18467 | +0.97% | 0.0000 |
| C | A | 80180 | 79432 | +748 | +0.04% | 0.0615 |
| C | 3 | 85877 | 72360 | +13517 | +0.71% | 0.0000 |
| C | 2 | 82133 | 72502 | +9631 | +0.51% | 0.0000 |
| C | 1 | 99234 | 80897 | +18337 | +0.96% | 0.0000 |
| C | 0 | 472092 | 69594 | +402498 | +21.12% | 0.0000 |
| B | A | 81091 | 91316 | -10225 | -0.54% | 0.0000 |
| B | 3 | 84944 | 83179 | +1765 | +0.09% | 0.0000 |
| B | 2 | 81337 | 83552 | -2215 | -0.12% | 0.0000 |
| B | 1 | 99558 | 91664 | +7894 | +0.41% | 0.0000 |
| B | 0 | 469359 | 78077 | +391282 | +20.53% | 0.0000 |
| A | 3 | 44478 | 34875 | +9603 | +0.50% | 0.0000 |
| A | 2 | 35098 | 28135 | +6963 | +0.37% | 0.0000 |
| A | 1 | 50584 | 32703 | +17881 | +0.94% | 0.0000 |
| A | 0 | 442635 | 31499 | +411136 | +21.57% | 0.0000 |
| 3 | 2 | 35654 | 39757 | -4103 | -0.22% | 0.0000 |
| 3 | 1 | 66079 | 58032 | +8047 | +0.42% | 0.0000 |
| 3 | 0 | 452488 | 48926 | +403562 | +21.17% | 0.0000 |
| 2 | 1 | 59558 | 47534 | +12024 | +0.63% | 0.0000 |
| 2 | 0 | 450634 | 42705 | +407929 | +21.40% | 0.0000 |
| 1 | 0 | 438776 | 31674 | +407102 | +21.36% | 0.0000 |

Table A.5: Pairwise TOP-2 comparison of the Refinement Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-2 prediction that was strictly better than prediction made by the other classifier.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| C | B | 41687 | 10663 | +31024 | +1.63% | 0.0000 |
| C | A | 89666 | 88522 | +1144 | +0.06% | 0.0068 |
| C | 3 | 96745 | 81315 | +15430 | +0.81% | 0.0000 |
| C | 2 | 92790 | 81481 | +11309 | +0.59% | 0.0000 |
| C | 1 | 108449 | 90284 | +18165 | +0.95% | 0.0000 |
| C | 0 | 483519 | 76819 | +406700 | +21.34% | 0.0000 |
| B | A | 87526 | 104508 | -16982 | -0.89% | 0.0000 |
| B | 3 | 91545 | 95081 | -3536 | -0.19% | 0.0000 |
| B | 2 | 87711 | 95707 | -7996 | -0.42% | 0.0000 |
| B | 1 | 105434 | 104032 | +1402 | +0.07% | 0.0022 |
| B | 0 | 479104 | 88081 | +391023 | +20.51% | 0.0000 |
| A | 3 | 54386 | 44613 | +9773 | +0.51% | 0.0000 |
| A | 2 | 44707 | 37229 | +7478 | +0.39% | 0.0000 |
| A | 1 | 59941 | 42199 | +17742 | +0.93% | 0.0000 |
| A | 0 | 459664 | 37021 | +422643 | +22.17% | 0.0000 |
| 3 | 2 | 43231 | 47914 | -4683 | -0.25% | 0.0000 |
| 3 | 1 | 75043 | 68113 | +6930 | +0.36% | 0.0000 |
| 3 | 0 | 470105 | 56221 | +413884 | +21.71% | 0.0000 |
| 2 | 1 | 68258 | 56920 | +11338 | +0.59% | 0.0000 |
| 2 | 0 | 468513 | 49701 | +418812 | +21.97% | 0.0000 |
| 1 | 0 | 460170 | 37640 | +422530 | +22.17% | 0.0000 |

Table A.6: Pairwise TOP-3 comparison of the Refinement Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-3 prediction that was strictly better than prediction made by the other classifier.

## A.3 Standalone Classifiers

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 33320 | 22933 | +10387 | +0.54% | 0.0000 |
| A | 2 | 31528 | 27101 | +4427 | +0.23% | 0.0000 |
| A | 1 | 48312 | 19937 | +28375 | +1.49% | 0.0000 |
| A | 0 | 282249 | 24318 | +257931 | +13.53% | 0.0000 |
| 3 | 2 | 40968 | 46928 | -5960 | -0.31% | 0.0000 |
| 3 | 1 | 61221 | 43233 | +17988 | +0.94% | 0.0000 |
| 3 | 0 | 285459 | 37915 | +247544 | +12.99% | 0.0000 |
| 2 | 1 | 65350 | 41401 | +23949 | +1.26% | 0.0000 |
| 2 | 0 | 293050 | 39546 | +253504 | +13.30% | 0.0000 |
| 1 | 0 | 266964 | 37408 | +229556 | +12.04% | 0.0000 |

Table A.7: Pairwise TOP-1 comparison of the Standalone Classifiers. The (X) dominates-columns indicate the number of times classifier X made a prediction which was correct while the prediction made by the other classifier classifier was incorrect.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 53987 | 41824 | +12163 | +0.64% | 0.0000 |
| A | 2 | 51070 | 48159 | +2911 | +0.15% | 0.0000 |
| A | 1 | 73989 | 35581 | +38408 | +2.02% | 0.0000 |
| A | 0 | 331993 | 37524 | +294469 | +15.45% | 0.0000 |
| 3 | 2 | 62640 | 70846 | -8206 | -0.43% | 0.0000 |
| 3 | 1 | 90068 | 65124 | +24944 | +1.31% | 0.0000 |
| 3 | 0 | 339176 | 55281 | +283895 | +14.89% | 0.0000 |
| 2 | 1 | 94523 | 61868 | +32655 | +1.71% | 0.0000 |
| 2 | 0 | 346179 | 57710 | +288469 | +15.13% | 0.0000 |
| 1 | 0 | 320118 | 54473 | +265645 | +13.94% | 0.0000 |

Table A.8: Pairwise TOP-2 comparison of the Standalone Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-2 prediction that was strictly better than prediction made by the other classifier.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 67688 | 56058 | +11630 | +0.61% | 0.0000 |
| A | 2 | 63987 | 63229 | +758 | +0.04% | 0.0338 |
| A | 1 | 90630 | 47112 | +43518 | +2.28% | 0.0000 |
| A | 0 | 358722 | 46360 | +312362 | +16.39% | 0.0000 |
| 3 | 2 | 77050 | 86192 | -9142 | -0.48% | 0.0000 |
| 3 | 1 | 108520 | 79171 | +29349 | +1.54% | 0.0000 |
| 3 | 0 | 367850 | 65836 | +302014 | +15.84% | 0.0000 |
| 2 | 1 | 112673 | 75073 | +37600 | +1.97% | 0.0000 |
| 2 | 0 | 373541 | 68658 | +304883 | +16.00% | 0.0000 |
| 1 | 0 | 348532 | 65409 | +283123 | +14.85% | 0.0000 |

Table A.9: Pairwise TOP-3 comparison of the Standalone Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-3 prediction that was strictly better than prediction made by the other classifier.

## A.4 Centerless Classifiers

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 75320 | 24341 | +50979 | +2.67% | 0.0000 |
| A | 2 | 61517 | 23329 | +38188 | +2.00% | 0.0000 |
| A | 1 | 73574 | 26639 | +46935 | +2.46% | 0.0000 |
| A | 0 | 260565 | 63169 | +197396 | +10.36% | 0.0000 |
| 3 | 2 | 56350 | 69141 | -12791 | -0.67% | 0.0000 |
| 3 | 1 | 70260 | 74304 | -4044 | -0.21% | 0.0000 |
| 3 | 0 | 255055 | 108635 | +146420 | +7.68% | 0.0000 |
| 2 | 1 | 71458 | 62711 | +8747 | +0.46% | 0.0000 |
| 2 | 0 | 258467 | 99259 | +159208 | +8.35% | 0.0000 |
| 1 | 0 | 256961 | 106500 | +150461 | +7.89% | 0.0000 |

Table A.10: Pairwise TOP-1 comparison of the Centerless Classifiers. The (X) dominates-columns indicate the number of times classifier X made a prediction which was correct while the prediction made by the other classifier classifier was incorrect.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 119959 | 44678 | +75281 | +3.95% | 0.0000 |
| A | 2 | 102446 | 43725 | +58721 | +3.08% | 0.0000 |
| A | 1 | 116242 | 46995 | +69247 | +3.63% | 0.0000 |
| A | 0 | 306760 | 88285 | +218475 | +11.46% | 0.0000 |
| 3 | 2 | 85444 | 103235 | -17791 | -0.93% | 0.0000 |
| 3 | 1 | 102836 | 108231 | -5395 | -0.28% | 0.0000 |
| 3 | 0 | 305416 | 141436 | +163980 | +8.60% | 0.0000 |
| 2 | 1 | 104705 | 92797 | +11908 | +0.62% | 0.0000 |
| 2 | 0 | 309333 | 130129 | +179204 | +9.40% | 0.0000 |
| 1 | 0 | 307020 | 138459 | +168561 | +8.84% | 0.0000 |

Table A.11: Pairwise TOP-2 comparison of the Centerless Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-2 prediction that was strictly better than prediction made by the other classifier.

| (1) | (2) | (1) dominates | (2) dominates | difference | relative difference | p-value |
|---|---|---|---|---|---|---|
| A | 3 | 145569 | 58914 | +86655 | +4.55% | 0.0000 |
| A | 2 | 126215 | 58362 | +67853 | +3.56% | 0.0000 |
| A | 1 | 140950 | 61724 | +79226 | +4.16% | 0.0000 |
| A | 0 | 331674 | 103185 | +228489 | +11.99% | 0.0000 |
| 3 | 2 | 103846 | 124722 | -20876 | -1.10% | 0.0000 |
| 3 | 1 | 122837 | 129353 | -6516 | -0.34% | 0.0000 |
| 3 | 0 | 331455 | 159836 | +171619 | +9.00% | 0.0000 |
| 2 | 1 | 125548 | 111886 | +13662 | +0.72% | 0.0000 |
| 2 | 0 | 335984 | 147311 | +188673 | +9.90% | 0.0000 |
| 1 | 0 | 333279 | 156267 | +177012 | +9.29% | 0.0000 |

Table A.12: Pairwise TOP-3 comparison of the Centerless Classifiers. The (X) dominates-columns indicate the number of times classifier X made a TOP-3 prediction that was strictly better than prediction made by the other classifier.