Method Call Argument Completion using Deep Neural Regression

Terry van Walen
student@terryvanwalen.nl

August 24, 2018, 40 pages

Academic supervisors: dr. C.U. Grelck & dr. M.W. van Someren
Host organisation: Info Support B.V., http://infosupport.com
Host supervisor: W. Meints

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Code completion is extensively used in IDE's. While there has been extensive research into the field of code completion, we identify an unexplored gap. In this thesis we investigate the automatic recommendation of a basic variable to an argument of a method call. We define the set of candidates to recommend as all visible type-compatible variables. To determine which candidate should be recommended, we first investigate how code prior to a method call argument can influence a completion. We then identify 45 code features and train a deep neural network to determine how these code features influence the candidate‘s likelihood of being the correct argument. After sorting the candidates based on this likelihood value, we recommend the most likely candidate.

We compare our approach to the state-of-the-art, a rule-based algorithm implemented in the Parc tool created by Asaduzzaman et al. [ARMS15]. The comparison shows that we outperform Parc, in the percentage of correct recommendations, in 88.7% of tested open source projects. On average our approach recommends 84.9% of arguments correctly while Parc recommends 81.3% correctly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code completion is extensively used in IDE's [MKF06]. It reduces the amount of typing [PLM15] and *"speeds up the process of writing code by reducing typos or other programming errors, and frees the developer from remembering every detail."* [ARMS15]. A well known form of code completion, that has been researched extensively, is the completion of method calls [BMM09, NNN+12, ARSH14, RVY14, PLM15, NHC+16]. Interestingly, however, the research into the completion of their respective arguments (method call arguments) has been limited. As far as we know only two papers investigate the completion of arguments directly [ZYZ+12, ARMS15] and one paper discusses it indirectly [LLS+16].

This is interesting because the benefits of method call completion are also mostly applicable to method call argument completion. It still reduces typing, reduces typos and frees developers from remembering every detail in the same way method call completion does. Features, used in method call argument completion, have also been shown to reduce the number of programming errors or coding mistakes by automatically detecting relevant anomalies [PG11].

There is also an added benefit to also research method call argument completion. When both forms of code completion reach a certain performance they can be combined to recommend a complete method call including its arguments at once.

1.1 Previous work

For the Java programming language, method call argument completion has already been implemented in well known IDE's (Eclipse, Netbeans, IntelliJ). At first the completion simply consisted of a list of accessible type-matched variables. This was later expanded so that likely arguments were located more to the top.

However, one of the major challenges of argument completion is that the amount of type-compatible candidates can be many. Arguments can also take the form of method calls, cast expressions, literals or any expression that results in a type-compatible value.

Zhang et al. [ZYZ+12] were in 2012, to the best of our knowledge, the first to publish about the completion of arguments. Using their tool Precise, they can recommend types of arguments that could not be recommended before including cast (`(int) var`) and certain literal (`4|"a"`) expressions. Precise recommends method call arguments based on previously collected argument usage patterns. Using four contextual features they capture the context of the code prior to the location of the argument. This context (usage pattern) is then used to determine the contextual similarity of this usage pattern to those in the dataset. Using a kNN (k-Nearest Neighbour) algorithm the best matching usage patterns in the dataset are found. Subsequently the argument linked to this usage pattern is recommended.

In 2015 Asaduzzaman et al. [ARMS15] presented a newer approach to this problem called Parc. Parc uses a similar method to Precise but can recommend even more types of arguments. In total, seventeen types of arguments were identified by Asaduzzaman et al. [ARMS15] and Parc supports eleven of them.

Asaduzzaman et al. [ARMS15] analyzed three code projects to discover the distribution of these argument types over all of the projects arguments. They found that 98% of arguments fall in one of the eleven argument types that are supported by Parc. This leaves 2% of arguments that can not be recommended by Parc because their argument type is not supported. Adding support for any or all of the remaining types can, therefore, only lead to an overall performance increase of 2%. On the other hand, the analysis also shows that around 40% of all arguments are basic variables. This number is also supported by research of Zhang et al [ZYZ+12]. This means that any improvement in the recommendation of basic variables as arguments does have a significant effect on the overall performance.

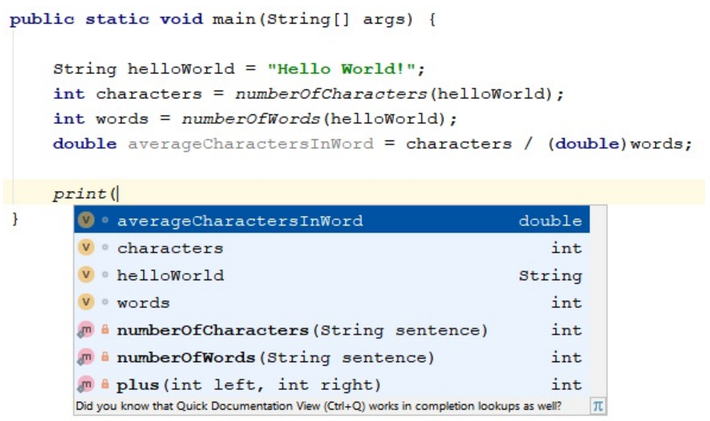
Precise delegates the recommendation of basic variables to the Eclipse JDT[1]. However, Asaduzzaman et al. [ARMS15] did investigate how basic variables should be recommended. They first determined which rules the Eclipse JDT uses to recommend basic variables as arguments. They then manually investigated code in which basic variables were used as arguments. They found that developers tend to initialize or assign new values to variables just before they use them for a method call argument. Therefore, if the algorithm sorts all candidates according to the distance to their point of initialization, it will increase the number of correct recommendations.

Despite the addition and improvement that Asaduzzaman et al. [ARMS15] made in regards to the recommendation of basic variables as arguments, we expect that there is still an unexplored gap to be filled.

1.2 Proposed approach

In this thesis we continue on the work by Asaduzzaman et al. [ARMS15]. We explore more of these coding patterns and the features, like the initialization distance, in the code that could be used to detect them. Using these features it is expected that more correct recommendations can be made.

However, adding features also necessitates modifying how they influence the recommendation. Using a strict rule-based system like the one in Parc is, for our purposes, not the best approach. Instead we propose to use a form of regression using deep neural networks. A neural network is useful because it takes care of calculating not only how each feature should influence the recommendations but also how these features should interact with each other to do that.



```
public static void main(String[] args) {

    String helloWorld = "Hello World!";
    int characters = numberOfCharacters(helloWorld);
    int words = numberOfWords(helloWorld);
    double averageCharactersInWord = characters / (double)words;

    print(|
}
```

```
averageCharactersInWord                    double
characters                                 int
helloWorld                                 String
words                                      int
numberOfCharacters(String sentence)        int
numberOfWords(String sentence)             int
plus(int left, int right)                  int
Did you know that Quick Documentation View (Ctrl+Q) works in completion lookups as well?  π
```

Figure 1.1: Example request for argument recommendations in IntelliJ.
The simple name variable candidates are: 'averageCharactersInWord', 'characters', 'helloWorld' and 'words'.

The example in figure 1.1 is used to illustrate a request for a recommendation. For every candidate argument (all accessible type-compliant variables; 'averageCharactersInWord', 'characters', 'helloWorld' and 'words') a set of features is determined. Examples are the distance of the argu-

[1]Java development tools that includes a recommendation engine

ment location to the line of code where the candidate is declared or initialized. Another example is the similarity of the candidate's name to the formal parameter name of the function or method. All these features are then used as the input for the deep neural network (Figure 1.2). Using deep neural regression a value for each candidate is calculated. This value represents the likelihood that the candidate should be used in the context in which it is requested. The likelihood values of all candidates are compared and they are sorted accordingly. The top candidate or candidates can then be recommended.

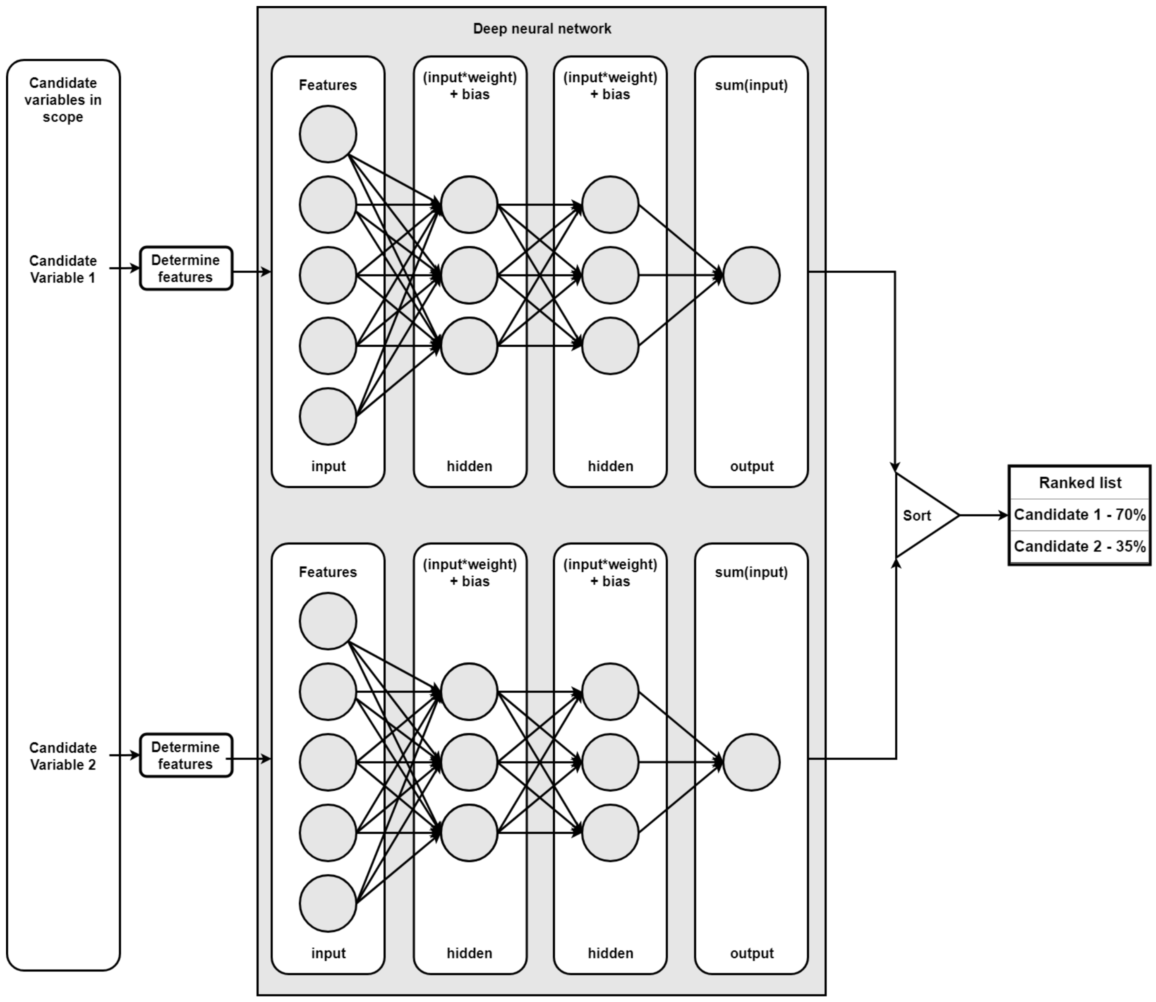


Figure 1.2: Illustration of proposed method using a deep neural regression network. For every candidate, features are collected. These features are used as input for the deep neural network. Regression is used to calculate a value depicting the likelihood that the candidate is the actual argument. The values are then compared and sorted resulting in a ranked list of most likely candidates.

1.3 Research method

The proposed approach leads to one overall research question which can be subdivided into three sub questions.

RQ1: Can we improve method call argument completion of basic variables using deep neural regression?

SQ1: Which code features contribute to the recommendation of the correct candidate variable?

SQ2: In what way should the deep neural regression network be applied to improve the percentage of correct recommendations?

SQ3: How does the proposed approach compare to previous research?

To answer SQ1, existing systems like Eclipse JDT, Parc and literature are explored to find likely features. Second, continuing the efforts of Asaduzzaman et al. [ARMS15] code surrounding method call arguments of the basic variable type are manually reviewed for more likely features. Third, based on Parc and our own approach we investigate code surrounding arguments that were incorrectly recommended. Finally, these collected features are tested for their actual influence on the recommendations and if they are beneficial to include or should be eliminated from the model.

To answer SQ2, the hyperparameters with which the network is initialized will be reviewed and tweaked. We investigate the effect of the networks batch size, layer configuration and if candidates of different arguments should be weighted differently or equally.

To answer SQ3, 142 open source Java projects are collected. From these projects all method call arguments and their respective candidates are collected. Using both a replicated form of the algorithm used in Parc and our own approach, recommendations are generated for all these arguments. The percentage of correct recommendations of both methods is then compared and evaluated.

1.4 Outline

In Chapter 2 we discuss the background and context to our thesis. In Chapter 3 we discuss how the features were identified and we demonstrate our motivation behind them. In Chapter 4 we evaluate these features and the hyper parameters of the deep neural regression model. In Chapter 5 we compare our results to the state of the art. Finally we summarize and discuss our results in Chapter 6.

Chapter 2

Background and Context

2.1 Types of arguments

To determine the number of basic variables that were used as method call arguments, Asaduzzaman et al. [ARMS15] analysed three subject systems. They were JEdit[1], ArgoUML[2] and JHotDraw[3]. In these projects all method calls were identified that targeted Swing or AWT libraries. The arguments of all these method calls were then collected to determine their expression type. Asaduzzaman et al. [ARMS15] found that between 36% and 41% of all arguments are of the basic variable type.

Zhang et al. [ZYZ+12] use a similar approach for three other subject systems. They were Eclipse 3.6.2, JBoss 5.0, and Tomcat 7.0. For Precise only method calls targeting the SWT framework[4] have been investigated. Their results show that between 38% and 47% of arguments are of the basic variable type.

2.2 Features from literature

The Parc tool by Asaduzzaman et al. [ARMS15] tries to determine the best match by ranking all type-compliant accessible variables. These *candidates* are ranked according to the following set of rules, whereby each rule is more significant than the rules below it.

- Locally declared candidates have precedence over field candidates.
- Field candidates have precedence over inherited field candidates.
- Candidates with a longer case insensitive substring match to the formal parameter name have precedence.
- Unused candidates have precedence.
- Candidates that are declared, initialized or assigned a new value closer to the method call argument have precedence.

Liu et al. [LLS+16] propose that the similarity between candidate names and formal parameter names (of the targeted method) can effectively be used to pair candidates to an argument. They base their work on code analysis and find another way to calculate the similarity compared to the feature used by Parc. They calculate lexical similarity (Equation 2.2) using the subterms of each name instead of the individual characters. The subterms of a variable name are defined as the individual parts separated by capitalization (camelCase) or underscores. A variable with the name `fieldLength` or `field_length` will, therefore, be decomposed to `field` and `length`.

[1]http://sourceforge.net/projects/jedit/
[2]http://argouml.tigris.org/
[3]http://sourceforge.net/projects/jhotdraw/
[4]http://www.eclipse.org/swt/

The process to calculate the lexical similarity is as follows; Let `C` be the name of the candidate variable and let `F` be the name of the formal parameter. After decomposing the names, there are two sequences of terms (Equation 2.1).

$$\begin{aligned} C &= (c_1, c_2 ... c_m) \\ F &= (f_1, f_2 ... f_n) \end{aligned} \tag{2.1}$$

To calculate the similarity between these sequences of terms the Longest Sequence of Common Terms is calculated. Using `C` and `F`, the LSCT is the longest subsequence of `C` where each term in the subsequence appears in `F` (Listing 2.1).

```
Define the function LSCT(C,F) as:
  Initialize subLSCT to 0
  Initialize longestLSCT to 0

  For each term in C do:
      If F contains term do:
          Add 1 to subLSCT
      End of if

      If F does not contain term do:
          Set longestLSCT to the maximum of subLSCT and longestLSCT
          Set subLSCT to 0
      End of if
  End of for

  Set longestLSCT to the maximum of subLSCT and longestLSCT

  Return longestLSCT
```

Listing 2.1: Pseudocode of LSCT algorithm.

The LSCT of `C` and `F` is not equal to the LSCT of `F` and `C`, therefore, the sum of both is taken. The final value is then divided by the combined amount of terms of both names to get the lexical similarity (Equation 2.2). Examples of names and their lexical similarity are provided in table 2.1.

$$\texttt{Lexical Similarity} = \frac{LSCT(C,F) + LSCT(F,C)}{|C| + |F|} \tag{2.2}$$

C	C subterms	F	F subterms	Lexical Similarity
length	length	inputLength	input, length	$\frac{1+1}{1+2} = \frac{2}{3}$
field_length	field, length	fieldLength	field, length	$\frac{2+2}{2+2} = \frac{1}{1}$
thisVariableName	this, variable, name	thisNameVariable	this, name, variable	$\frac{3+3}{3+3} = \frac{1}{1}$
variableThisName	variable, this, name	thisOtherVariable	this, other, variable	$\frac{2+1}{3+3} = \frac{1}{2}$
variableThisName	variable, this, name	thisVariableOther	this, variable, other	$\frac{2+2}{3+3} = \frac{2}{3}$
ASTNode	ast, node	node	node	$\frac{1+1}{2+1} = \frac{2}{3}$

Table 2.1: Examples of calculating the Lexical Similarity for a name `C` and a name `F`.

Based on this paper the following feature is derived:

- A candidate with a higher lexical similarity to its formal parameter has precedence.

One important aspect to note about this feature is that is requires knowledge about which method is targeted with a method call. In the Java language this is, however, not always clear because methods can be overloaded. However, following previous research [ARMS15, LLS+16] we assume knowledge about the targeted method is available to minimize complexity.

2.3 Deep neural regression

For a neural network there are some settings that can be adjusted. These settings are called hyperparameters. These hyper-parameters influence how, how fast, how well and how long a model is trained. From the start it is hard to predict which hyper-parameters will give the best result. A few of these hyperparameters are now discussed and how they will likely influence the model.

Layers

Neural networks consist of layers, the input layer consists of all feature values collected in the data collection step. The input layers are connected to a sequence of hidden layers of a certain depth and width. The depth is the amount of hidden layers and the width is the amount of neurons each layer contains. Finally the output layer is the final and last layer. In a regression network the end layer consists of one layer that takes the sum of all values from the previous layer. The main question is with which width and depth the network will produce the best results.

Batch size

The batch size is the number of candidates that are passed to the network before the network updates the weights. In theory a higher batch size will lower the models ability to generalize [KMN+16] and a smaller batch size will take a longer amount of time to train.

Epochs

The amount of epochs is automatically determined by a process that monitors the loss of the validation set. The validation set is a small part of the training set that is not used to train the model, but is set aside to monitor how well the model performs. When the loss does not improve on the validation set for a certain amount of epochs the training is terminated and the best model, until that moment, is chosen.

Weight initialization

Weights communicate to the network to what respect a training sample should impact how the model is modified in between batches.

Chapter 3

Feature discovery

Research has already shown that source code has some structural, syntactic, contextual and semantic attributes that influence which variables are more likely to be used as method call arguments. Source code, just like natural language, has a *"surprising amount of regularity"* and is more repetitive than natural languages [HBG+]. This repetitiveness in code and how it is written opens up the possibility to learn from past cases.

In other research, Asaduzzaman et al. [ARMS15] showed that source code is locally specific to method call arguments. This means that there is a correlation between which candidate variable is used and the tokens prior to the method call. In other words, the code prior to the method call reflects which argument(s) will be used.

In this chapter more ideas and coding patterns are explored. Features are proposed to reflect the existence of these patterns in the code prior to the method call argument. Being aware of these patterns can help the deep neural network make a more informed recommendation.

3.1 Analysis of Parc features

Based on the algorithm for Parc (Section 2.2) the following features are derived:

- The candidate is:
  - isLocal: a locally declared variable.
  - isField: a field variable.
  - isInheritedField: an inherited field variable.
  - usedInMethodCall: used in a prior method call as an argument.
- lexicalSimilarityParc: A measure of lexical similarity between the candidate name and the name of the formal parameter. This is expressed as the number of characters in the longest case insensitive substring match.
- distanceToDeclaration: The number of declared candidates between where this candidate is declared and the method call argument.
- distanceToInitialization: The number of candidates that have been initialized[1] between where this candidate was last initialized and the method call argument.

Continuing on this list, it is first concluded that Parc does not seem to make a distinction between local variables and the method parameters. It could be, however, that there does exist a distinction between these two and therefore another feature is introduced alongside isLocal, isField and isInheritedField:

- isParameter: Candidate is a parameter of the parent method.

[1] Initialized is used for both a variable that is initialized as a variable that is assigned a new value

Based on the research by Liu et al. [LLS+16] the feature LexicalSimilarity is also investigated. This feature, similar to lexicalSimilarityParc used in Parc, measures the lexical similarity between the name of the candidate variable and the name of the targeted formal parameter. In section 2.2 the LexicalSimilarity feature is discussed based on the research by Liu et al. [LLS+16].

- lexicalSimilarity: The lexical similarity as discussed in section 2.2 using equation 2.2.

However, other methods to compare the candidate name to the formal parameter name can also be used. We investigate both a stronger and weaker form of lexical similarity.

- lexicalSimilarityStrictOrder: The lexicalSimilarityStrictOrder is almost the same as the lexicalSimilarity except that where lexicalSimilarity accepts a corresponding term everywhere in the matching name it will only accept a corresponding term if it is in the same relative order. For example: `fieldLength` and `lengthField` have a lexicalSimilarityStrictOrder value of $\frac{1+1}{2+2} = \frac{1}{2}$ but a lexicalSimilarity of $\frac{2+2}{2+2} = \frac{1}{1}$ (Equation 2.2).
- lexicalSimilarityCommonTerms: The ratio of common terms. The lexicalSimilarityCommonTerms is a more loose version of lexical similarity. The corresponding terms of both names are counted and the lowest count is multiplied by two and then divided by the number of terms in total. For example: `variableThisName` and `thisOtherVariable` have a lexicalSimilarityCommonTerms of $\frac{2*2}{3+3} = \frac{2}{3}$ [2] and a lexical similarity of $\frac{2+1}{3+3} = \frac{1}{2}$ (Equation 2.2).

The Parc feature usedInMethodCall is true if the candidate has already been used as an argument of a method call prior to this method call argument. However, a candidate can also be used in any of the following ways.

- Candidate is used in:
  - usedInVariableDeclaration: the initializer part of a variable declaration.
  - usedInAssignExpression: the value part of an assign expression.
  - usedInArrayAccessExpression: the index of an array access expression.
  - usedInForEachStatement: the iterable variable of a foreach statement.
  - usedInObjectCreationExpression: an argument to an object creation expression.
  - usedInExplicitConstructorCall: an argument to an explicit constructor call statement.

3.2 Code patterns

In this section code examples of interesting coding patterns are used to illustrate the need for specific features.

We find a common coding pattern that is often misclassified in the `remainder` method of the `UnsignedInts` package of the Google Guava library. The first method call in this method is `toLong`. It accepts one argument, in this case, `dividend`. However it has actually two candidates: `dividend` and `divisor`. Both are formal parameters of the parent method and `dividend` is declared before `divisor`. Since both are unused, Parc would recommend the candidate that was declared or initialized closest, in this case that would be the candidate `divisor`. However, as seen in the example (Listing 3.2), not `divisor` is used but `dividend`.

An explanation for this behaviour is that when variables are declared without being used immediately it is because the developer wants to use them in the order of declaration. An example of this is when a method has multiple formal parameters. This type of misclassification happens often and therefore we want to detect this scenario.

To include this scenario in the model we add the following features so that the model could potentially learn to detect this scenario.

[2] (variable, this) matches (this, variable)

```java
public final class UnsignedInts {
  ...
  public static int remainder(int dividend, int divisor) {
    return (int) (toLong(dividend) % toLong(divisor));
  }
  ...
}
```
https://github.com/google/guava/blob/master/guava/src/com/google/common/primitives/UnsignedInts.java

Listing 3.2: Code example: Candidates used in order of declaration or initialization instead of the inverse order.

- DISTANCETODECLARATIONSPECIAL: The DISTANCETODECLARATION but in a different order: First are all local variables in order of declaration, then all field variables and then all inherited field variables.
- UNUSEDLOCALCANDIDATES: The number of unused local candidates. More unused candidates could indicate that another distance feature should be used.
- UNUSEDLOCALCANDIDATESINROW: The number of local candidates that are unused in sequence (in the order of the DISTANCETOINITIALIZATION feature). In the example. (Listing 3.2) the value would be two. Both parameters are unused at the point of predicting the first method call argument.

From the improved PARC algorithm we derived features concerning if the candidate was used in specific ways. At that point an usage in the predicate of an `if` statement was not discussed. We expect that if a variable is used in the predicate it does not negatively impact the likelihood that that variable will be used again. The reason for this is that comparing a variable to something else is in most instances not the goal in and of itself but a way to establish something about that variable. Therefore, it could also be the case that when a variable is used in a predicate, it could actually increase the likelihood that that variable is used within that `if` block. In the example from the `QuantilesAlgorithm` package (Listing 3.3) all `swap` method calls indeed use the variables used in the predicate of the parent `if` statement of the `swap` method call. Most interesting is the last `if` statement. In this predicate not all three variables (`array`, `from`, `to`) are used. Only `array` and `from` are used and indeed only those two are used in the `swap` method call.

```java
static double select(int k, double[] array) {
    ...
    int from = 0;
    int to = array.length - 1;
      ...
        if (array[from] > array[to]) {
          swap(array, from, to);
        }
        if (array[from + 1] > array[to]) {
          swap(array, from + 1, to);
        }
        if (array[from] > array[from + 1]) {
          swap(array, from, from + 1);
        }
      ...
  }
```
https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/math/QuantilesAlgorithm.java

Listing 3.3: Code example: Candidate used in the current predicate of the parent if statement.

To include this pattern in our model the following two features are proposed:

- inIfStatement: Method call argument is within an `if` statement.
- usedInCurrentIfPredicate: Candidate is used in the predicate of the parent `if` statement.

A special case of the above idea was found in a discussion we had with the manager of the knowledge centre of our hosting organization: Gert Jan Timmerman (G. J. Timmerman, personal communication, June 13, 2018). This case can also be found in the `createCacheBuilder` method in the `CacheBuilderFactory` package of the Google Guava library (Listing 3.4). The method call `builder.concurrencyLevel` is wrapped in an `if` statement where one of its candidates is compared to `not null`. Comparing to `not null` in `Java` can be done to determine if the variable exists. Testing if the variable exists, we expect, is done because the developer wants to use that variable but is unsure if it exists. The same pattern is apparent in the other two method calls in this method.

```
private CacheBuilder<Object, Object> createCacheBuilder(
    Integer concurrencyLevel,
    Integer initialCapacity,
    Integer maximumSize,
    ...
    ) {

  CacheBuilder<Object, Object> builder = CacheBuilder.newBuilder();
  if (concurrencyLevel != null) {
    builder.concurrencyLevel(concurrencyLevel);
  }
  if (initialCapacity != null) {
    builder.initialCapacity(initialCapacity);
  }
  if (maximumSize != null) {
    builder.maximumSize(maximumSize);
  }
  ...
}
```

https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/cache/CacheBuilderFactory.java

Listing 3.4: Code example: Candidate compared to null in predicate of parent `if` statement

To include this scenario in the model we add the following features so that the model can potentially learn to detect this scenario.

- comparedToNotNullInIfPredicate: Is the candidate compared to `not null` in the predicate of the parent `if` statement.
- comparedToNullInIfPredicate: Is the candidate compared to `null` in the predicate of the parent `if` statement.
- inIfBlock: The method call argument is within an `if` block.
- inElseBlock: The method call argument is within an `else` block.

In Parc the candidate is negatively affected if is has been used in an earlier method call. However, it is questionable if this is the case when not only the candidate but the whole method, including the candidate, has been used before. In listing 3.5 an example is shown that illustrates this point. The method call `getComments` is called twice in this example. Using our collected features the argument of the first instance can be solved by looking at the initialization distance. For the argument of the second method call this is not the case. As can be seen in the example, there are now four candidates (excluding field and inherited field candidates). The candidate `paginationRequest` (actual argument) is initialized farthest away, all candidates have been used before and there is no similarity between one of the candidates names and the formal parameter name. However, in this case the same combination of method call and argument has been used before in this method. How effective this feature is could in theory be subject to the position of the argument in the method call or the number of arguments. To cover these scenarios three features are introduced.

- positionOfArgument: Starting at zero and from left to right the arguments position in the method call is established.
- numberOfArguments: Number of arguments in the method call.
- usedInMethodCallCombination: The same method call is called before using the same candidate name as an argument at the current position. The important aspect of this feature is that it only takes the name of the candidate into account, not if it is actually the same candidate. This could be useful in cases like listing 3.6.

```java

public void addArticleComment() throws Exception {
    ...
    JSONObject paginationRequest = Requests.buildPaginationRequest("1/10/20");
    JSONObject result = commentQueryService.getComments(paginationRequest);
    ...
    final JSONObject requestJSONObject = new JSONObject();
    ...
    final JSONObject addResult =
      commentMgmtService.addArticleComment(requestJSONObject);
    ...
    result = commentQueryService.getComments(paginationRequest);
    ...
}
```

https://github.com/guoguibing/librec/blob/3.0.0-beta/core/src/main/java/net/librec/recommender/cf/rating/FMALSRecommender.java

Listing 3.5: Code example: method call and candidate combination already used

```java
static final void blackboxTestRecordWithValues(...) throws Exception
  {
    ...
    for (int i = 0; i < values.length; i++) {
      final int pos = permutation1[i];
      rec.setField(pos, values[pos]);
    }
    ...
    for (int i = 0; i < values.length; i++) {
      final int pos = permutation1[i];
      rec.setField(pos, values[pos]);
    }
  ...
}
```

https://github.com/stratosphere/stratosphere/blob/master/stratosphere-core/src/test/java/eu/stratosphere/types/RecordTest.java

Listing 3.6: Code example: Same method call with same argument but the candidate is only the same in name

Beside these collected features, most directed at the candidate in question, it could also be helpful for the training to provide some more contextual features for the method call. One basic question is where does the method call reside relative to other program constructs.

- Method call is within
    - inMethod: a method declaration.
    - inConstructor: a constructor declaration.
    - inEnum: an enum declaration.
    - inForEach: a foreach statement.
    - inFor: a for statement.
    - inDo: a do statement.
    - inWhile: a while statement.
    - inTry: a try statement.
    - inSwitch: a switch statement.
    - inAssign: an assign expression.
    - inVariable: a variable declaration.

Then there are four more features that might impact the prediction score by giving more context about the code and the candidate.

- isPrimitive: Candidate is a primitive.
- numberOfCandidates: The number of candidates.
- scopeDistance: The distance in scope.
- parentCallableSize: The declaration size of the parent method or constructor in lines. When the method call argument does not have a callable parent, it takes the size of the whole class.

3.3 Identified features

- comparedToNotNullInIfPredicate
- comparedToNullInIfPredicate
- distanceToDeclaration
- distanceToDeclarationSpecial
- distanceToInitialization
- inAssign
- inConstructor
- inDo
- inEnum
- inForEach
- inFor
- inElseBlock
- inIfBlock
- inIfStatement
- inMethod
- inSwitch

- inTry
- inVariable
- inWhile
- isLocal
- isParameter
- isPrimitive
- isField
- isInheritedField
- lexicalSimilarityParc
- lexicalSimilarity
- lexicalSimilarityCommonTerms
- lexicalSimilarityStrictOrder
- parentCallableSize
- positionOfArgument
- numberOfArguments
- numberOfCandidates
- scopeDistance
- unusedLocalCandidates
- unusedLocalCandidatesInRow
- usedInArrayAccessExpression
- usedInAssignExpression
- usedInCurrentIfPredicate
- usedInExplicitConstructorCall
- usedInForEachStatement
- usedInIfPredicate
- usedInMethodCall
- usedInObjectCreationExpression
- usedInMethodCallCombination
- usedInVariableDeclaration

Chapter 4

Evaluation

In this chapter the performance of the deep neural regression approach is evaluated. To test how well our approach performs, we apply it to all method call arguments, whose actual argument is a basic variable, from 142 open source projects (Appendix A). In our dataset, 36% of all method call arguments are of the basic variable type, which is in line with the 36% to 47% found by [ZYZ+12, ARMS15] (Section 2.1).

To evaluate how our approach performs on these method call arguments, the precision (Equation 4.1) is measured. The higher the precision, the better our approach performs.

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{4.1}$$

In this equation, the recommendations are considered relevant if the top recommendation is equal to the actual argument. When the precision of the deep neural regression approach is given it will be referred to as the **prediction score**.

The deep neural regression method has certain hyperparameters that influence how well it will perform (Section 2.3). The features selected in section 3.1 will also impact the prediction score. In this chapter, these features and hyperparameters are therefore evaluated. The goal is to find a combination of features and hyperparameters that result in more correct recommendations.

4.1 Approach

First, using some preliminary tests a base variation of features and hyperparameters is established. All subsequent models are based on this model. It uses all features identified in section 3.1. The hyperparameters of the model are set to a batch size of 1024, three hidden layers of respectively 2025, 45 and 45 nodes (layer configuration 7) and the candidates are all weighted the same. Second, the individual features and hyperparameters are modified one at a time. By changing these model variables slightly the effect of these individual parts on the prediction score can be compared. Third, the best of all variations is used to determine the final prediction score of the deep neural regression method.

The basic approach to determine the prediction score of a variation consists of five steps. First, all projects (Appendix A) are listed. Second, the projects are randomly divided over two groups of equal size, a training group and a testing group. Third, both groups are divided again into two sets resulting in four separate sets. Each set is a subcollection of projects from the initial list. Fourth, for every set in the training group a model is trained on the candidates of that set. There are two training groups, therefore, two separate models are trained. Fifth, both models are separately used to suggest candidates for both the first and second testing set. For each argument in the testing set the actual argument is compared to the models suggested candidate. Resulting in a prediction score for each of the four combinations (Equation 4.1).

However, if nothing is changed and step four and five are repeated, prediction scores will slightly change. This is caused by how neural network models work and specifically how they are initialized using random numbers. Removing this randomness, by fixing these numbers, does result in equal prediction scores for the same configuration, training set and testing set. This, however, does not remove the underlying variation. It only fixes the prediction score to one specific variation. Therefore, it does not provide any information to how the change in hyperparameters or features impacts the prediction score.

Therefore, step four and five are repeated three times. Averaging the prediction scores of each run into a final prediction score including the accompanying standard deviation. However, comparing two numbers is different from comparing two means. Comparing the means of two prediction scores is only valid when the means are statistically different from each other. If two means are not statistically different from each other, their difference does not provide any information. To do this, the students two-sample, two-tailed heteroscedastic t-test is used. This test provides a confidence value, or p-value, as output that indicates how confident we can be that the two means are indeed different. If the confidence of this test is below 0.05 the null hypothesis is rejected. In other words, the two means can be assumed to be statistically different.

Splitting all projects into two groups and then splitting each group into two sets results in four different sets of projects. These four sets are used in most of what is discussed in this chapter. When an experiment deviates from this set-up it is explicitly conveyed. Table 4.1 shows basic information about the four sets. In the subsequent text each set is referenced according to the group name and set number displayed in this table[1].

Data set	Training group		Testing group	
	Set 1	Set 2	Set 1	Set 2
Name	Training1	Training2	Testing1	Testing2
Projects included	36	36	36	35
Arguments	90.408	127.951	127.568	167.438
Candidates	444.259	805.877	569.357	862.376
Guess score	20,4 %	15,9 %	22,4 %	19,4 %
Prediction score average (n=9)	-	-	87,3 %	82,9 %

Table 4.1: Context for the used training and testing sets.
The prediction score for each training and testing set combination is the average of 9 runs. To get the prediction score average for each testing set, the two training sets that were applied to that testing set are averaged

4.2 Hyperparameters

As discussed in section 2.3, tweaking the hyperparameters of the neural network model could increase the prediction score. Three hyperparameters are tested. The configuration (width and height) of the hidden layers, the batch size and if the candidates should be weighted differently based on the amount of candidates per argument. The activation and loss function are not tested and are set to the ReLu function and the mean squared error (MSE) respectively.

To determine which configuration of layers performs best we first establish 14 different configurations.

f = number of input features. The layers are separated by a comma.

- Model 1: f
- Model 2: f^2
- Model 3: f^3
- Model 4: f, f

[1]Example: Set 1 of the training group is named training1

- Model 5: f, f, f
- Model 6: f, f, f, f
- Model 7: f^2, f
- Model 8: f^2, f, f
- Model 9: f^2, f^2
- Model 10: f^2, f^2, f, f
- Model 11: $f^2, f^2, f, f/2$
- Model 12: $f^2, f^2, f, f/2, f/4$
- Model 13: f^3, f
- Model 14: f^3, f, f
- Model 15[2]: f^3, f^3

These configurations are not exhaustive but represent certain aspects we want to test. Specifically, the height of the first hidden layer relative to the input layer, the height of the last layer and the width of the network. The number of nodes is always relative to the number of input features because this allows for specific scenarios. If the number of nodes is equal to the number of input features, every feature can be combined with one or more other features into a new node. With the second power of the number of input features, every input feature can be combined with every other input feature into a separate node. The third power is tested as an experiment whether even more nodes could be beneficial.

Changing the configuration of the hidden layers (Table 4.2) does not have a statistically relevant impact[3] on the prediction score for 10 out of 14 different configurations (Table 4.2). From these 10 different configuration, configuration 7 is chosen. Configuration 7 has a comparatively low amount of nodes and connections which results in comparatively lower training time. The average standard deviation of the scores in configuration 7 is also low compared to the other configurations with a low amount of nodes.

Name	Hidden layers	Testing1		Testing2	
		Training1	Training2	Training1	Training2
7	2025, 45	86.6% ± 0.2	88.0% ± 0.0	82.7% ± 0.1	83.1% ± 0.1
3	91125	86.7% ± 0.3	87.9% ± 0.2	82.7% ± 0.1	83.0% ± 0.0
14	91125, 45, 45	86.4% ± 0.2	87.9% ± 0.3	82.6% ± 0.1	83.2% ± 0.2
12	2025, 2025, 45, 22, 11	86.6% ± 0.4	87.7% ± 0.3	82.6% ± 0.6	83.0% ± 0.2
8	2025, 45, 45	86.5% ± 0.4	88.0% ± 0.2	82.2% ± 0.4	83.1% ± 0.1
10	2025, 2025, 45, 45	86.7% ± 0.2	87.8% ± 0.5	82.5% ± 0.3	82.9% ± 0.2
9	2025, 2025	86.3% ± 0.4	87.8% ± 0.3	82.5% ± 0.2	83.0% ± 0.1
2	2025	86.6% ± 0.4	87.7% ± 0.1	82.5% ± 0.2	82.8% ± 0.2
11	2025, 2025, 45, 22	85.9% ± 0.4	87.9% ± 0.1	82.3% ± 0.2	82.9% ± 0.1
6	45, 45, 45, 45	85.8% ± 0.7	87.7% ± 0.3	81.5% ± 0.4	82.5% ± 0.2
5	45, 45, 45	86.1% ± 0.6	86.8% ± 0.9	82.0% ± 0.4	82.2% ± 0.6
4	45, 45	85.7% ± 0.6	87.1% ± 0.5	81.3% ± 0.4	81.8% ± 0.4
1	45	84.1% ± 2.4	86.6% ± 0.8	79.5% ± 2.6	81.5% ± 0.7
13	91125, 45	73.7% ± 0.0	78.4% ± 6.5	69.5% ± 0.0	73.9% ± 6.2

Table 4.2: Prediction scores of variations in the structure of hidden layers (Section 2.3) (n=3). If the scores between two layers are equal, they both have the same position score (ranking) for that combination of training and test set. Cells in red are statistically different (Students T-Test, two-sample, two-tailed, heteroscedastic) from the average prediction score of configuration 7.

[2]Server did not have enough GPU memory to do this operation

[3]Using the students T-Test (two-sample, two-tailed, heteroscedastic) compared to prediction scores of batch size 1024

For the different batch sizes (Table 4.3) a similar pattern is present. Changing the batch size does not impact the prediction score in a statistically significant way (Table 4.3) for most sizes. Only the batch sizes 32, 64 and 8 are significantly different[4] in more than one combination. Therefore, based on the data available nothing can be said about which of the other batch sizes will result in better prediction scores. In section 2.3 it is discussed that higher batch sizes result in quicker training. Therefore, the batch size of 1024 is chosen.

Batch size	Test set 1		Test set 2	
	Train set 1	Train set 2	Train set 1	Train set 2
1024	86.9% ± 0.7	88.0% ± 0.2	82.8% ± 0.2	83.1% ± 0.0
512	86.9% ± 0.3	87.9% ± 0.2	82.8% ± 0.1	82.8% ± 0.3
128	86.6% ± 0.7	88.0% ± 0.1	82.6% ± 0.2	82.8% ± 0.1
16	86.8% ± 0.5	87.8% ± 0.1	82.4% ± 0.5	82.7% ± 0.3
256	86.5% ± 0.6	87.7% ± 0.3	82.4% ± 0.5	82.9% ± 0.2
32	86.4% ± 0.1	87.7% ± 0.2	82.5% ± 0.1	82.6% ± 0.2
64	86.4% ± 0.6	87.5% ± 0.2	82.2% ± 0.2	82.7% ± 0.1
8	86.5% ± 0.1	87.4% ± 0.4	82.3% ± 0.2	82.3% ± 0.3

Table 4.3: Prediction scores of different batch sizes (n=3).
If the scores between two layers are equal, they both have the same position score (ranking) for that combination of training and test set. Cells in red are statistically different (Students T-Test, two-sample, two-tailed, heteroscedastic) from the average prediction score of batch size 1024.

The last hyper parameter that we test is the weight initialization of the individual candidates. Weights communicate to the network to what respect collections of candidates should impact how the model is modified in between epochs.

In the dataset used in this study, every argument has a certain amount of candidates. Some arguments have few candidates and some have more than twenty. Because all candidates are evaluated individually, arguments with few candidates could be underrepresented and arguments with a lot of candidates could be overrepresented. To counteract this, the model can weigh the candidates differently. The candidates of arguments, that have few candidates, are weighed more (**few candidates more**) in this scenario.

However, an argument can also be made that if an argument has more candidates, it is also more complex. Which will mean that the model should weigh each candidate the same or these candidates more. Another possibility is that certain kinds of candidates will always be found among a large set of candidates. An example could be inherited field or field variables. This can lead to the model not being able to learn how to recommend candidates of these types well. Therefore candidates of this type should be weighed more (**many candidates more**).

Another reason to weigh the candidates is that for every argument only one candidate is the actual argument. Therefore, there are more positive training examples than negative ones. This will, however, be true for both the training set (with which the model is trained) and the testing set (which is used to simulate giving the recommendations).

To evaluate how we should set this hyperparameter a different approach is chosen than for the other hyperparameters. In the first two tests we noticed that it was hard to determine a significant difference between two batch sizes or layer structures. Therefore, we try a different approach for this hyperparameter. Instead of the two training and two testing sets used before, we divide all projects over one training and one testing set. We also shuffle the projects after every three runs to get a new configuration of projects in the training and testing set each time. We call every new configuration of projects a project set. In table 4.4 we see that initializing the candidates with weights does not impact the prediction score significantly. It does not matter if we weigh some candidates more or less. Therefore, no weight initialization is used.

[4]Students t-test (two-sample, two-tailed, heteroscedastic) compared to prediction scores of batch size 1024

Project set	No weight initialization	Few candidates more	Many candidates more
1	$84,8\% \pm\ 0,3$	$84,9\% \pm\ 0,3$	$85,0\% \pm\ 0,5$
2	$84,3\% \pm\ 0,2$	$84,2\% \pm\ 0,5$	$84,1\% \pm\ 0,2$
3	$86,1\% \pm\ 0,2$	$85,8\% \pm\ 0,0$	$86,2\% \pm\ 0,5$
4	$84,5\% \pm\ 0,4$	$84,3\% \pm\ 0,5$	$84,0\% \pm\ 0,1$
5	$85,9\% \pm\ 0,3$	$85,9\% \pm\ 0,5$	$85,9\% \pm\ 0,1$
6	$83,9\% \pm\ 0,2$	$84,1\% \pm\ 0,3$	$83,8\% \pm\ 0,1$
7	$84,1\% \pm\ 0,3$	$84,7\% \pm\ 0,1$	$84,1\% \pm\ 0,8$
8	$85,4\% \pm\ 0,2$	$85,3\% \pm\ 0,1$	$85,1\% \pm\ 0,1$
9	$85,2\% \pm\ 0,3$	$85,2\% \pm\ 0,4$	$85,2\% \pm\ 0,1$
10	$84,7\% \pm\ 0,1$	$84,7\% \pm\ 0,3$	$84,7\% \pm\ 0,0$
11	$84,1\% \pm\ 0,2$	$84,3\% \pm\ 0,3$	$84,1\% \pm\ 0,2$
12	$85,2\% \pm\ 0,1$	$85,7\% \pm\ 0,2$	$85,3\% \pm\ 0,2$
13	$83,7\% \pm\ 0,2$	$84,3\% \pm\ 0,3$	$83,9\% \pm\ 0,2$
14	$85,0\% \pm\ 0,2$	$85,5\% \pm\ 0,3$	$85,0\% \pm\ 0,4$
15	$84,2\% \pm\ 0,2$	$83,8\% \pm\ 0,1$	$83,8\% \pm\ 0,3$
16	$86,7\% \pm\ 0,1$	$86,6\% \pm\ 0,3$	$86,7\% \pm\ 0,2$
17	$84,2\% \pm\ 0,1$	$84,0\% \pm\ 0,3$	$84,2\% \pm\ 0,1$
18	$86,7\% \pm\ 0,2$	$86,2\% \pm\ 0,4$	$86,4\% \pm\ 0,2$
19	$84,8\% \pm\ 0,7$	$85,2\% \pm\ 0,2$	$84,1\% \pm\ 1,3$
20	$84,9\% \pm\ 0,3$	*	*
Average	$84,9\% \pm\ 0,3$	$85,0\% \pm\ 0,3$	$84,8\% \pm\ 0,3$

Table 4.4: Prediction scores of different weight initializations (n=3).
All candidates of an argument are weighted according to the number of candidates that argument has. **Few candidates more** weights smaller candidate sets more and **Many candidates more** weights larger candidate sets more.
** Early termination, no data available.*

4.3 Features

To understand which features impact the final prediction scores two methods are proposed. The first method concerns a simple statistical analysis of the candidates in the dataset. For both test sets and for every feature we determine how many candidates possess that feature and how many actual arguments have that feature. Some features are also combined with another feature to measure their combined effect. The second method concerns the actual deep neural regression model. For each feature a model is trained with all features except for that feature. The resulting change in prediction score is used to determine the impact of the feature on the total prediction score. Combinations of features are also tested. In particular the feature combinations that overlap in some respects.

In table 4.5 for both test sets it is shown what percentage of candidates possess a certain feature. They also show what percentage of those candidates, that possess the feature, are the actual argument and what percentage of the remaining candidates are the actual argument. For example in the set testing1 (Table 4.5) 29.99% of candidates are inherited fields. Of those 29.99% of candidates, only 5,96% of them are the actual argument used. Of the remaining 70.01% of candidates 29,45% are used as the actual argument. Therefore, if the only available information about the candidate is that it is an inherited field, then that candidate is not likely to be the actual argument. The same principle holds in reverse. If it is known that a candidate is NOT an inherited field, it increases the chance of it being the actual argument.

There is a point were it will be beneficial to increase or decrease the likelihood of the candidate being the actual argument. This point is equal to the guess score (Table 4.1) of the respective test set, taking into account slight variations in data between sets of projects. If the percentage is above the guess score it is beneficial. If not, it is detrimental.

However, notice that this table does not show the distribution of the features across the candidates of an argument. Therefore, if all candidates for a single argument possess the feature, this still does not help in predicting the correct argument. It, however, does give some insight in the features of the testing sets and the distribution over the aggregated candidates.

Testing1

Feature	Candidate has feature		Candidate does not have feature
		Candidate is actual argument	
isPrimitive	15,50 %	13,75 %	23,99 %
isLocal	21,58 %	43,81 %	16,52 %
isParameter	15,39 %	54,53 %	16,56 %
isField	33,13 %	8,46 %	29,32 %
isInheritedField	29,99 %	5,96 %	29,45 %
usedInMethodCallCombination	6,50 %	48,19 %	20,61 %
comparedToNullInIfPredicate	0,10 %	39,45 %	22,39 %
comparedToNotNullInIfPredicate	0,38 %	70,58 %	22,22 %
usedInIfPredicate	7,32 %	19,32 %	22,65 %
usedInMethodCall	37,48 %	26,62 %	19,88 %
usedInVariableDeclaration	16,20 %	21,43 %	22,59 %
usedInForEachStatement	0,40 %	14,12 %	22,44 %
usedInCurrentIfPredicate	1,75 %	51,68 %	21,88 %
usedInAssignExpression	6,16 %	20,85 %	22,51 %
usedInExplicitConstructorCall	0,10 %	11,36 %	22,42 %
usedInObjectCreationExpression	5,32 %	17,82 %	22,66 %
usedInArrayAccessExpression	0,25 %	19,05 %	22,41 %

Testing2

Feature	Candidate has feature		Candidate does not have feature
		Candidate is actual argument	
isPrimitive	31,45 %	8,02 %	24,64 %
isLocal	22,12 %	38,84 %	13,90 %
isParameter	16,86 %	48,45 %	13,53 %
isField	35,22 %	6,82 %	26,26 %
isInheritedField	25,86 %	0,99 %	25,84 %
usedInMethodCallCombination	7,03 %	44,94 %	17,49 %
ComparedToNullInIfPredicate	0,12 %	41,60 %	19,39 %
ComparedToNotNullInIfPredicate	0,32 %	60,59 %	19,29 %
usedInIfPredicate	8,61 %	16,53 %	19,69 %
usedInMethodCall	37,43 %	20,31 %	18,88 %
usedInVariableDeclaration	17,38 %	17,48 %	19,82 %
usedInForEachStatement	0,31 %	20,98 %	19,41 %
usedInCurrentIfPredicate	1,87 %	49,18 %	18,85 %
usedInAssignExpression	8,41 %	16,47 %	19,69 %
usedInExplicitConstructorCall	0,35 %	2,84 %	19,47 %
usedInObjectCreationExpression	3,78 %	15,87 %	19,56 %
usedInArrayAccessExpression	0,34 %	13,83 %	19,43 %

Table 4.5: Candidate specific features for testing1 and testing2.
For all candidate specific features that can be present or not present it is established in what percentage of candidates they occur (candidate has feature). For these candidates it is tested if they are the actual argument used. For the remaining candidates the same is done.

Parc proposed to replace the distance measurement. Instead of the declaration distance, they proposed to use the distanceToInitialization of a candidate. For the arguments in the testing sets (Table 4.6) it is indeed the case that the closest candidate according to the DISTANCETOINITIALIZATION is more often the actual argument than according to the DISTANCETODECLARATION.

In section 3.2 the DISTANCETODECLARATIONSPECIAL was the third distance value discussed. However, this feature is supposed to work together in the model with the UNUSEDLOCALCANDIDATESINROW feature to get a valuable distance measurement. In this respect however, the same distance feature as intended can be created out of DISTANCETOINITIALIZATION and UNUSEDLOCALCANDIDATESINROW (Equation 4.2).

$$\begin{aligned} \text{Variables} &= \begin{cases} i = \text{distanceToInitialization} \\ u = \text{unusedLocalCandidatesInRow} - 1 \end{cases} \\ \text{distanceToInitializationUnused} &= \begin{cases} u - i, & \text{if } \; i \leqq u \\ i, & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

This distance measurement does not outperform the DISTANCETOINITIALIZATION and DISTANCETODECLARATION features in testing1 but it does in testing2.

The feature SCOPE DISTANCE is less effective than the other distance measurements but it measures something different. In contrast to the other distance measures, candidates can have the same SCOPEDISTANCE. However if the DISTANCETOINITIALIZATION is used as a tie-breaker the top pick is still not better than the DISTANCETOINITIALIZATION on its own. The feature could however still be beneficial, especially in combinations with others, and is not ruled out based on these scores.

Testing1

Feature	First candidate		Remaining candidates
		Candidate is actual argument	
distanceToInitialization	22,41%	76,10%	6,90%
distanceToDeclaration	22,41%	75,60%	7,04%
distanceToInitializationUnused	22,41%	75,39%	7,11%
scopeDistance	19,50%	40,26%	18,08%

Testing2

Feature	First candidate		Remaining candidates
		Candidate is actual argument	
distanceToInitialization	19,42%	70,97%	6,99%
distanceToDeclaration	19,42%	70,71%	7,06%
distanceToInitializationUnused	19,42%	71,87%	6,78%
scopeDistance	22,70%	30,73%	16,09%

Table 4.6: Distance features for testing1 and testing2.
For all distance related features the candidates for each argument are sorted according to the distance feature. It is established what percentage of candidates are first if sorted according to the specific distance feature. Each first candidate is then compared to the actual argument. Each remaining candidate is also compared to the actual argument. The percentage of the candidates that are sorted first and are the actual argument is equal to the prediction score because for every argument only one candidate is first (except for SCOPEDISTANCE).

The similarity between the formal parameter name and the candidate name can be an important indicator that can help predict the correct candidate to use. Four features are proposed (Section 3.1) to cover this aspect, all calculating some form of lexical similarity between the two names. In table 4.7 the performance of each lexical feature is stated. Based on these two test sets the lexical similarity measured according to the LEXICALSIMILARITY, LEXICALSIMILARITYSTRICTORDER and lexicalSimilarityCommonTerms features outperforms the lexicalSimilarityParc feature in the used data set.

Testing1

Feature	A similarity is measured		No similarity	Candidate is best match
		Candidate is actual argument		
lexicalSimilarityParc	25,59%	65,52%	7,58%	84,38%
lexicalSimilarity	23,15%	70,00%	10,59%	84,66%
lexicalSimilarityStrictOrder	23,15%	70,00%	10,59%	84,68%
lexicalSimilarityCommonTerms	23,15%	70,00%	10,59%	84,68%

Testing2

Feature	A similarity is measured		No similarity	Candidate is best match
		Candidate is actual argument		
lexicalSimilarityParc	24,23%	53,85%	8,40%	77,54%
lexicalSimilarity	20,19%	59,93%	11,21%	77,76%
lexicalSimilarityStrictOrder	20,19%	59,93%	11,21%	77,78%
lexicalSimilarityCommonTerms	20,19%	59,93%	11,21%	77,78 %

Table 4.7: Lexical features for testing1 and testing2.
For all features related to lexical similarity, the percentage of candidates that have a similarity with their respective formal parameter are calculated (similarity above 0.0). Of these candidates it is established which are the actual argument used and what percentage of candidates that are not similar in any way are the actual argument used. A candidate is the best match for its argument if the similarity according to the specific feature is highest compared to the other candidates. Note that because of how the features are extracted from the projects, the initial ordering is based on the distanceToInitialization feature, therefore when no similarity is found or the similarities are equal, the distanceToInitialization breaks the tie.

The statistical distribution of features over all the candidates and the actual arguments specifically tells a lot about the importance of the feature. However, for the proposed method of deep neural regression, it is also important to know what the combined effect of the features are. For all features we create and test a model using all but that one feature (Table 4.8). By measuring how the elimination of each feature impacts the prediction score we aim to find how important the feature is to our approach.

In the table, the cells that are colored red indicate that the mean prediction score in that cell is assumed statistically different from the base scores (the prediction score derived using all features). The t-test that is used to determine if two means are statistically different uses a p-value of 0.05. This means that in 5% of cases the test has a false positive. For this table 180 t-tests were executed. That results in 9 possible means which could be misclassified. If this is taken into account the table does not provide much information except that most features do not, in and of itself, have an impact on the prediction score. There are, however, two features that do show a statistical significant difference in all runs. They are methodCallCombinationUsed and unusedLocalCandidatesInRow. Another feature has a difference in three runs (usedInMethodCall). Because of the low statistical proof, only these three features are considered to be proven beneficial to the model using this set-up.

	Testing1		Testing2	
Feature	Training1	Training2	Training1	Training2
All features included (n=9)	86.7% ± 0.5	87.9% ± 0.3	82.7% ± 0.3	83.2% ± 0.2
methodCallCombinationUsed	85.7% ± 0.2	87.4% ± 0.1	80.9% ± 1.1	81.9% ± 0.0
unusedLocalCandidatesInRow	86.0% ± 0.3	87.4% ± 0.3	82.1% ± 0.2	82.0% ± 0.2
usedInMethodCall	86.2% ± 0.2	87.2% ± 0.3	81.9% ± 0.2	81.8% ± 0.2
numberOfArguments	86.3% ± 0.3	87.3% ± 1.0	82.3% ± 0.3	82.3% ± 1.2
positionOfArgument	86.1% ± 0.1	87.6% ± 0.2	82.3% ± 0.3	82.7% ± 0.1
inEnum	86.2% ± 0.3	87.6% ± 0.3	82.3% ± 0.3	82.8% ± 0.5
numberOfCandidates	86.0% ± 0.1	87.5% ± 0.1	82.4% ± 0.1	82.9% ± 0.0
usedInIfPredicate	85.8% ± 1.0	87.9% ± 0.2	81.7% ± 1.8	82.8% ± 0.3
inAssign	86.4% ± 0.3	87.6% ± 0.3	82.4% ± 0.3	82.6% ± 0.4
inVariable	85.8% ± 0.3	87.8% ± 0.4	82.2% ± 0.1	82.9% ± 0.4
inDo	86.2% ± 0.1	87.6% ± 0.2	82.4% ± 0.1	82.9% ± 0.2
isLocal	85.9% ± 0.2	87.7% ± 0.1	82.4% ± 0.1	82.9% ± 0.2
callInElseBlock	86.4% ± 0.3	87.7% ± 0.8	82.3% ± 0.3	82.7% ± 0.9
inElseBlock	86.4% ± 0.5	87.6% ± 0.2	82.5% ± 0.4	82.6% ± 0.2
scopeDistance	86.2% ± 0.1	87.9% ± 0.3	82.3% ± 0.3	82.7% ± 0.4
distanceToInitialization	86.4% ± 0.3	87.5% ± 0.3	82.6% ± 0.2	82.6% ± 0.6
unusedLocalCandidates	86.2% ± 0.1	87.7% ± 0.1	82.4% ± 0.2	82.9% ± 0.1
usedInCurrentIfPredicate	86.4% ± 0.4	87.8% ± 0.3	82.2% ± 0.2	82.7% ± 0.3
lexicalSimilarityCommonTerms	86.2% ± 0.2	87.8% ± 0.3	82.4% ± 0.1	82.8% ± 0.2
lexicalSimilarityStrictOrder	86.0% ± 0.2	87.9% ± 0.2	82.3% ± 0.1	82.9% ± 0.3
usedInExplicitConstructorCall	86.2% ± 0.4	87.8% ± 0.1	82.4% ± 0.4	82.8% ± 0.2
usedInObjectCreationExpression	86.4% ± 0.2	87.6% ± 0.4	82.6% ± 0.2	82.6% ± 0.5
comparedToNotNullInIfPredicate	86.3% ± 0.3	87.5% ± 0.1	82.7% ± 0.3	82.6% ± 0.1
lexicalSimilarityParc	86.3% ± 0.4	87.8% ± 0.3	82.1% ± 0.2	82.9% ± 0.4
isField	86.2% ± 0.1	87.7% ± 0.3	82.5% ± 0.3	82.9% ± 0.3
inForEach	86.3% ± 0.2	87.6% ± 0.4	82.7% ± 0.0	82.6% ± 0.6
inVariable	86.5% ± 0.4	87.6% ± 0.3	82.6% ± 0.2	82.5% ± 0.5
isPrimitive	86.4% ± 0.2	87.7% ± 0.2	82.6% ± 0.2	82.6% ± 0.2
isParameter	86.1% ± 0.2	87.9% ± 0.1	82.3% ± 0.1	83.0% ± 0.1
isInheritedField	86.4% ± 0.7	87.8% ± 0.1	81.8% ± 1.2	83.0% ± 0.3
distanceToDeclaration	86.8% ± 0.1	87.4% ± 1.1	82.7% ± 0.1	82.1% ± 1.5
inFor	86.4% ± 0.2	87.7% ± 0.0	82.5% ± 0.2	82.9% ± 0.1
inForEach	86.2% ± 0.3	87.9% ± 0.3	82.6% ± 0.1	82.8% ± 0.4
inSwitch	86.6% ± 0.4	87.7% ± 0.4	82.5% ± 0.1	82.7% ± 0.4
usedInArrayAccessExpression	86.5% ± 0.4	87.7% ± 0.4	82.6% ± 0.1	82.7% ± 0.4
usedInAssignExpression	86.2% ± 0.2	87.9% ± 0.3	82.5% ± 0.2	83.0% ± 0.2
inWhile	86.5% ± 0.2	87.8% ± 0.3	82.5% ± 0.2	82.8% ± 0.3
lexicalSimilarity	86.1% ± 0.1	87.9% ± 0.1	82.6% ± 0.2	83.0% ± 0.1
inTry	86.3% ± 0.3	88.1% ± 0.2	82.3% ± 0.2	83.1% ± 0.1
parentCallableSize	86.4% ± 0.2	88.0% ± 0.1	82.4% ± 0.2	83.1% ± 0.2
distanceToDeclarationSpecial	87.5% ± 0.4	87.9% ± 0.6	82.9% ± 0.1	82.5% ± 0.7
inMethod	86.5% ± 0.3	87.8% ± 0.4	82.6% ± 0.0	83.2% ± 0.3
inConstructor	86.5% ± 0.3	88.0% ± 0.0	82.6% ± 0.1	83.1% ± 0.0
comparedToNullInIfPredicate	86.7% ± 0.4	88.0% ± 0.2	82.5% ± 0.3	83.2% ± 0.1
inIfStatement	86.6% ± 0.1	88.0% ± 0.1	82.6% ± 0.4	83.1% ± 0.1

Table 4.8: Impact of elimination of features on prediction score.
Features ordered on how much eliminating them from the model impacts the prediction score. All prediction scores are an average of running the training model and predicting the arguments three times (n=3) for their respective training and testing set. Cells in red indicate that the mean prediction score is statistically different from the base prediction score according to a students t-test.

Almost all features do not significantly impact the prediction score if eliminated. However, this can also be attributed to the fact that some features overlap in some respects. To counteract this effect combinations of features are removed (Table 4.9 and 4.10).

Removing all features that calculate a kind of lexical similarity (lexicalSimilarityStrictOrder, lexicalSimilarity, lexicalSimilarityCommonTerms and lexicalSimilarityParc) between the formal parameter name and the candidate name has the most impact (Table 4.9). Removing the distance measurements does not provide a conclusive result. There is no clear statistical benefit or disadvantage (Table 4.10).

Features removed	Testing1		Testing2	
	Training1	Training2	Training1	Training2
All features included (n=9)	86.7% ± 0.5	87.9% ± 0.3	82.7% ± 0.3	83.2% ± 0.2
Keep none	81.4% ± 0.1	81.5% ± 0.3	78.7% ± 0.2	78.8% ± 0.6
Keep only lexicalSimilarityParc	85.7% ± 0.3	86.5% ± 0.5	81.5% ± 0.1	81.5% ± 0.65
Keep only lexicalSimilarityCommonTerms	86.1% ± 0.5	86.5% ± 0.5	82.1% ± 0.2	82.2% ± 0.7
Keep only lexicalSimilarity	86.1% ± 0.3	87.5% ± 0.5	82.1% ± 0.1	82.8% ± 0.4
Keep only lexicalSimilarityStrictOrder	86.2% ± 0.0	87.1% ± 0.6	82.2% ± 0.3	82.6% ± 0.2

Table 4.9: Eliminating combinations of lexical similarity features.
The lexical similarity features consist of lexicalSimilarityStrictOrder, lexicalSimilarity, lexicalSimilarityCommonTerms and lexicalSimilarityParc. In this table, all these features are removed from the model except for one. The model is also run without any of these features.

Features removed	Testing1		Testing2	
	Training1	Training2	Training1	Training2
All features included (n=9)	86.7% ± 0.5	87.9% ± 0.3	82.7% ± 0.3	83.2% ± 0.2
Keep none	87.3% ± 0.2	87.6% ± 0.2	81.7% ± 0.2	81.6% ± 0.2
Keep only scopeDistance	87.5% ± 0.2	87.7% ± 0.1	82.3% ± 0.1	81.8% ± 0.1
Keep only distanceToDeclaration	87.5% ± 0.3	87.8% ± 0.1	82.8% ± 0.1	82.6% ± 0.1
Keep only distanceToInitialization	87.6% ± 0.3	88.0% ± 0.2	82.7% ± 0.4	82.7% ± 0.2

Table 4.10: Eliminating combinations of distance features.
The distance features consist of distanceToInitialization, distanceToDeclaration, distanceToDeclarationSpecial and scopeDistance. In this table, all these features are removed from the model except for one. The model is also run without any of these features.

Chapter 5

Comparison to previous work

In this chapter we compare our approach to previous work. We, however, do not compare our method directly to Precise because Precise does not natively support the recommendation of basic variables. It delegates this to the Eclipse IDE. The only difference between the algorithm used in the Eclipse IDE and Parc is how they measure distance [ARMS15]. The Eclipse IDE uses the distanceToDeclaration feature while Parc uses the distanceToInitialization feature. Based on table 4.6 we derive that, in our dataset too, the distanceToInitialization feature is indeed the better choice. However, in table 4.10 we do not see a significant difference between the two features. Based on this information we do not expect the Eclipse IDE algorithm to perform better than Parc. Therefore, we only compare our approach to Parc directly.

To compare our approach to that of Parc [ARMS15], we change the method of evaluation. In the original method all projects were divided into four sets. Therefore, every prediction score was based on only 1 set for training and 1 set for testing. In total every prediction score was, therefore, based on only 50% of available projects. In this part however, all projects are used. For every calculated prediction score 50% of projects will be used to train a model and the other 50% will be used to test that model and calculate the prediction score. For every run the projects are first shuffled to randomize their order and then divided into the training and testing set according to that order.

We also introduce the **Parc score**, which represents the precision (Equation 4.1) of the Parc tool.

5.1 Parc

To compare our approach to that of Parc the prediction score and Parc score are calculated using the same training and testing sets. All projects have been shuffled 20 times to generate 20 different training and 20 different testing sets. For each training set a corresponding testing set exists that together contain all projects. Based on these 20 sets the average prediction score is 84.9% $\pm$ 0.3 and the average Parc score is 81.3% (Table 5.1). The deep neural regression approach results therefore in an improvement over Parc of 3.6 $\pm$ 0.3 percent point (pp) or 4.4% on this dataset.

Project set	Prediction score	Parc Score	Difference (pp)
1	84.8% ± 0.3	81.6%	3.3 pp
2	84.3% ± 0.2	80.5%	3.8 pp
3	86.1% ± 0.2	83.4%	2.7 pp
4	84.5% ± 0.4	81.6%	2.9 pp
5	85.9% ± 0.3	82.5%	3.4 pp
6	83.9% ± 0.2	79.3%	4.6 pp
7	84.1% ± 0.3	81.0%	3.1 pp
8	85.4% ± 0.2	81.3%	4.1 pp
9	85.2% ± 0.3	81.0%	4.2 pp
10	84.7% ± 0.1	80.7%	4.0 pp
11	84.1% ± 0.2	80.9%	3.3 pp
12	85.2% ± 0.1	81.3%	3.9 pp
13	83.7% ± 0.2	80.1%	3.5 pp
14	85.0% ± 0.2	82.2%	2.9 pp
15	84.2% ± 0.2	81.2%	3.1 pp
16	86.7% ± 0.1	82.9%	3.7 pp
17	84.2% ± 0.1	80.2%	4.0 pp
18	86.7% ± 0.2	83.1%	3.6 pp
19	84.8% ± 0.7	81.4%	3.5 pp
20	84.9% ± 0.3	80.8%	4.2 pp
Average	84.9% ± 0.3	81.3%	3.6 ± 0.3 pp

Table 5.1: Prediction score (n=3), Parc score and difference.
For randomly shuffled sets of projects. The prediction score is calculated based on all arguments.

The prediction scores in table 5.1 are calculated based on the aggregated set of all arguments from all projects. However, if the model is applied to the individual projects, in a test set, 88.7% of projects have the same prediction score or benefit from the new approach over Parc (Figure 5.2). In total an average gain of 2.6 percent point is realized on the individual projects of this test set. However, if the size of the individual projects, expressed in the amount of arguments they contain, is taking into account, then the average gain is 4.1 percent point (pp). This indicates that arguments in bigger projects are better to predict than smaller projects in the data used[1]. Eleven of the twelve projects that show no improvement or a negative improvement indeed contain less arguments than the average and median amount of arguments.

[1]A project is always used for the training set or the testing set, never both.

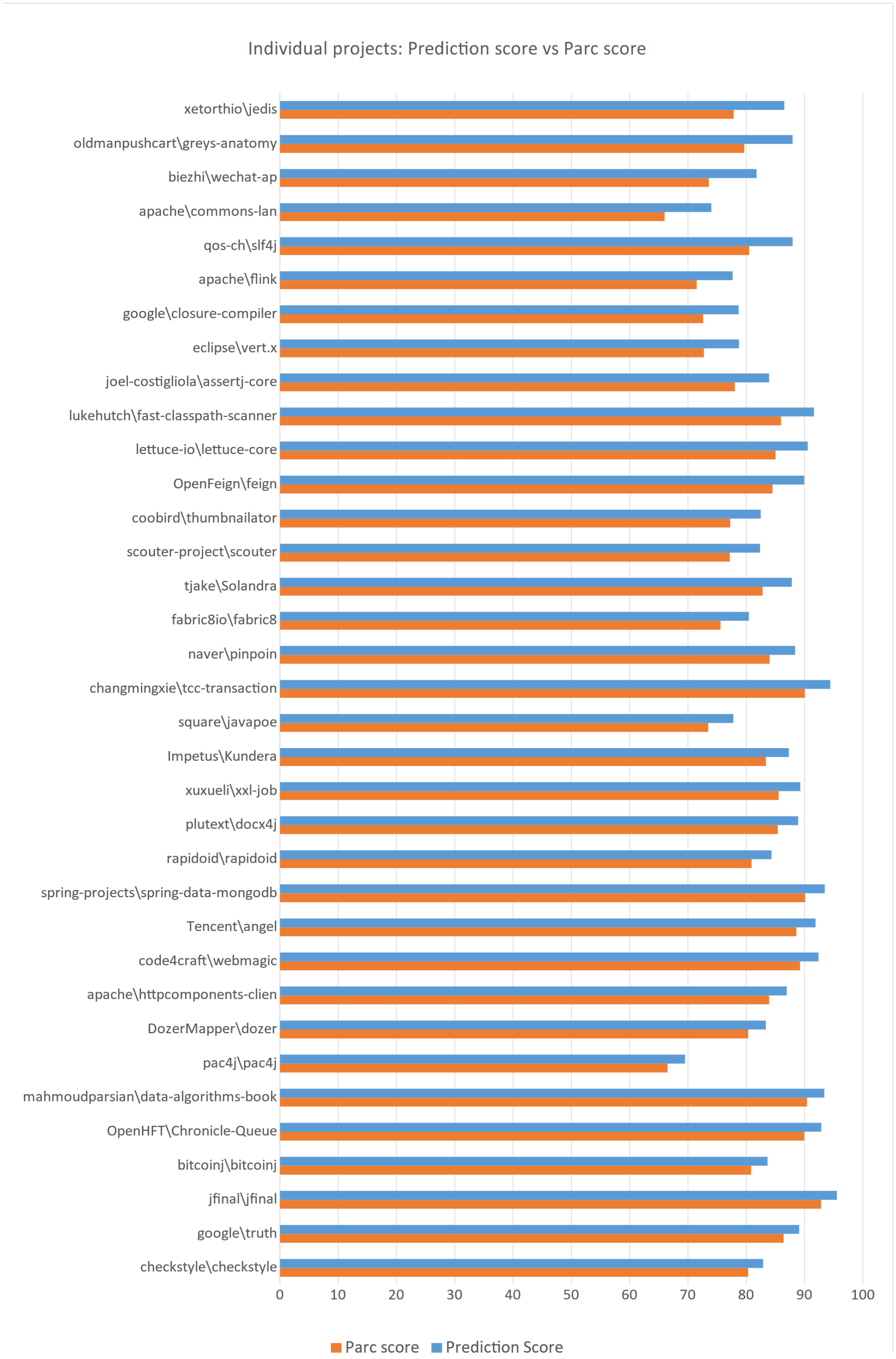


Figure 5.1: Prediction score and Parc score for individual projects (1 of 2).

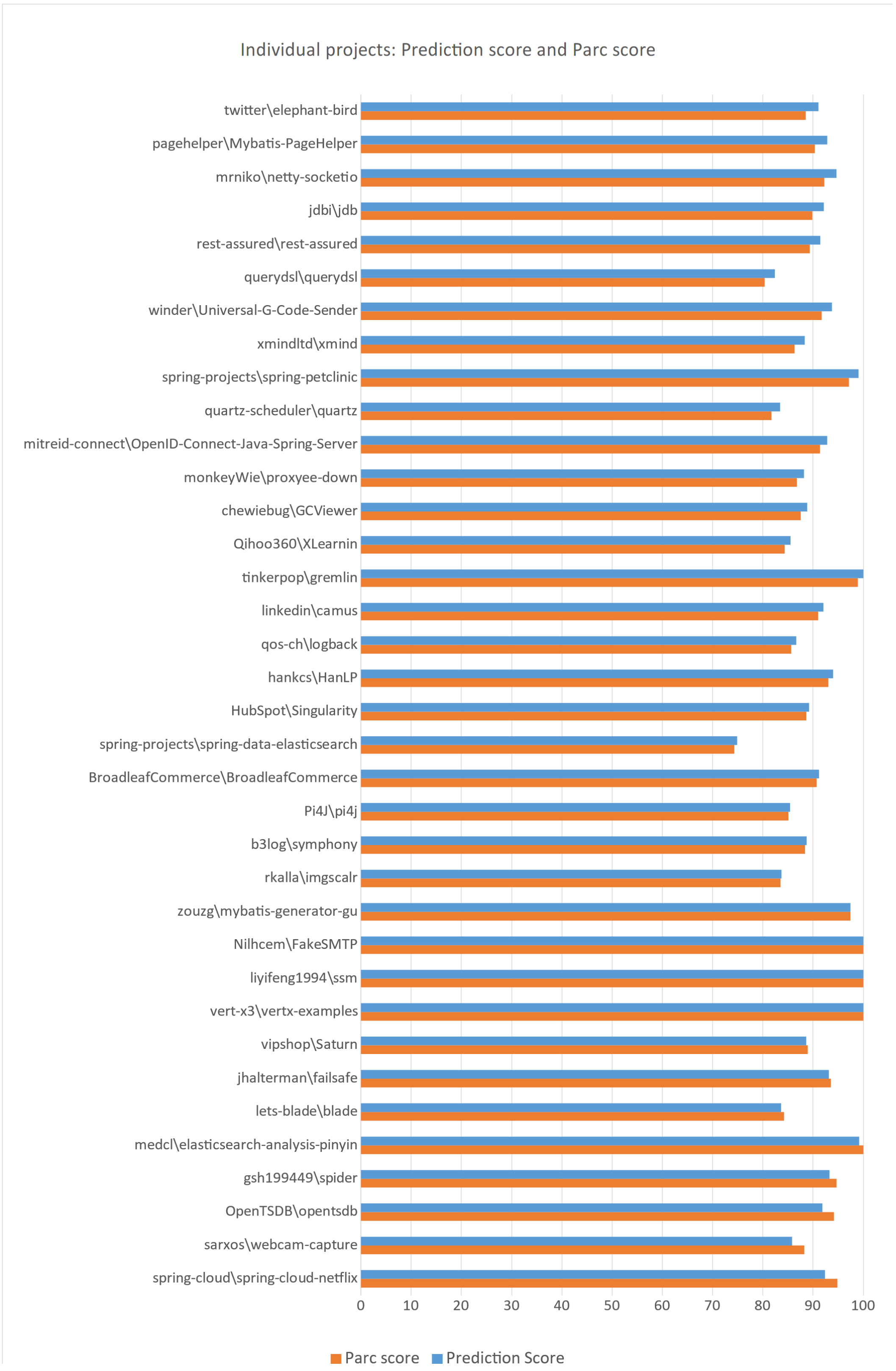


Figure 5.2: Prediction score and PARC score for individual projects (2 of 2).

5.2 Limiting features

In this chapter the features are discussed that can be removed without making an impact on the actual precision achieved. For most features however, no conclusive evidence is found that indicates that they do not provide a positive outcome. Only the overlapping features for distance (declarationDistance) and lexical similarity (parcLexicalSimilarity, lexicalSimilarity, commonTermRatio) were found to conclusively not be beneficial to the prediction score. There is also no evidence that the feature PARENTCALLABLESIZE does provide a beneficial factor (Table 4.8). Removing these features in the model indeed does not change the final results (Table 5.2).

If all features that are not used in the PARC algorithm are removed than we see a definite drop in performance (Table 5.2). Compared to the precision of PARC itself the score achieved using deep neural regression is 2 percent point lower for this instance. In other words, the deep neural regression method does not outperform the PARC algorithm using only the features used in PARC.

Project set	Prediction score with all features	Prediction score with limited features	Prediction score with PARC features
1	$84,8\% \pm 0,3$	$84,5\% \pm 0,2$	$78,6\% \pm 0,1$
2	$84,3\% \pm 0,2$	$84,6\% \pm 0,2$	$78,9\% \pm 0,1$
3	$86,1\% \pm 0,2$	$86,5\% \pm 0,3$	$81,4\% \pm 0,5$
4	$84,5\% \pm 0,4$	$84,3\% \pm 0,2$	$79,7\% \pm 0,2$
5	$85,9\% \pm 0,3$	$85,6\% \pm 0,3$	$80,1\% \pm 0,1$
6	$83,9\% \pm 0,2$	$83,6\% \pm 0,0$	$77,1\% \pm 0,0$
7	$84,1\% \pm 0,3$	$84,6\% \pm 0,3$	$79,1\% \pm 0,3$
8	$85,4\% \pm 0,2$	$84,7\% \pm 0,0$	$78,9\% \pm 0,1$
9	$85,2\% \pm 0,3$	$85,1\% \pm 0,1$	$79,3\% \pm 0,3$
10	$84,7\% \pm 0,1$	$84,3\% \pm 0,1$	$77,3\% \pm 0,1$
11	$84,1\% \pm 0,2$	$84,8\% \pm 0,2$	$79,2\% \pm 0,2$
12	$85,2\% \pm 0,1$	$84,9\% \pm 0,1$	$79,0\% \pm 0,3$
13	$83,7\% \pm 0,2$	$84,5\% \pm 0,1$	$78,0\% \pm 0,4$
14	$85,0\% \pm 0,2$	$85,9\% \pm 0,1$	$79,7\% \pm 0,6$
15	$84,2\% \pm 0,2$	$84,7\% \pm 0,2$	$79,2\% \pm 0,1$
16	$86,7\% \pm 0,1$	$87,1\% \pm 0,2$	$79,9\% \pm 0,3$
17	$84,2\% \pm 0,1$	$83,8\% \pm 0,1$	$78,0\% \pm 0,3$
18	$86,7\% \pm 0,2$	$85,8\% \pm 0,3$	$80,0\% \pm 0,0$
19	$84,8\% \pm 0,7$	$85,0\% \pm 0,1$	$79,1\% \pm 0,1$
20	$84,9\% \pm 0,3$	$85,1\% \pm 0,2$	$78.4\% \pm 0,6$*
Average	$84,9\% \pm 0,3$	$85,0\% \pm 0,2$	$79,0\% \pm 0,3$

Table 5.2: Prediction scores for all features, limited features and PARC features. The limited features contain all features except for: DISTANCETODECLARATION, LEXICALSIMILARITYPARC, LEXICALSIMILARITY, LEXICALSIMILARITYCOMMONTERMS and PARENTCALLABLESIZE. The PARC features consist of ISPARAMETER, ISLOCAL, ISFIELD, ISINHERITEDFIELD, LEXICALSIMILARITYPARC, USEDINMETHODCALL and DISTANCETOINITIALIZATION.
** Early termination, average is taken over two runs instead of three.*

5.3 Allowing multiple recommendations

In all evaluations we have focused only on the top-1 recommendation (Section 4) provided by the method used. This is largely done because in many cases there is only room for one argument recommendation of the basic variable type. If there is room to recommend more variables we can modify the precision equation 4.1. A top-n recommendation is, in this section, relevant when one of the first n recommendations is the actual argument. In figure 5.3 the prediction score and PARC score are shown for the top-1, top-2 and top-3 recommendations. The top-2 and top-3 recommendations show the same trend as the top-1 recommendation compared to the PARC score. From this it can be concluded that the deep neural regression method does not only improve the top suggestion but it is also beneficial if more recommendations are requested. There is also another interesting pattern in that the top-2 recommendations of the deep neural regression method are quite close to the top-3 Parc score. Meaning that almost the same precision can be achieved with two recommendations using the deep neural regression method as with three using Parc.

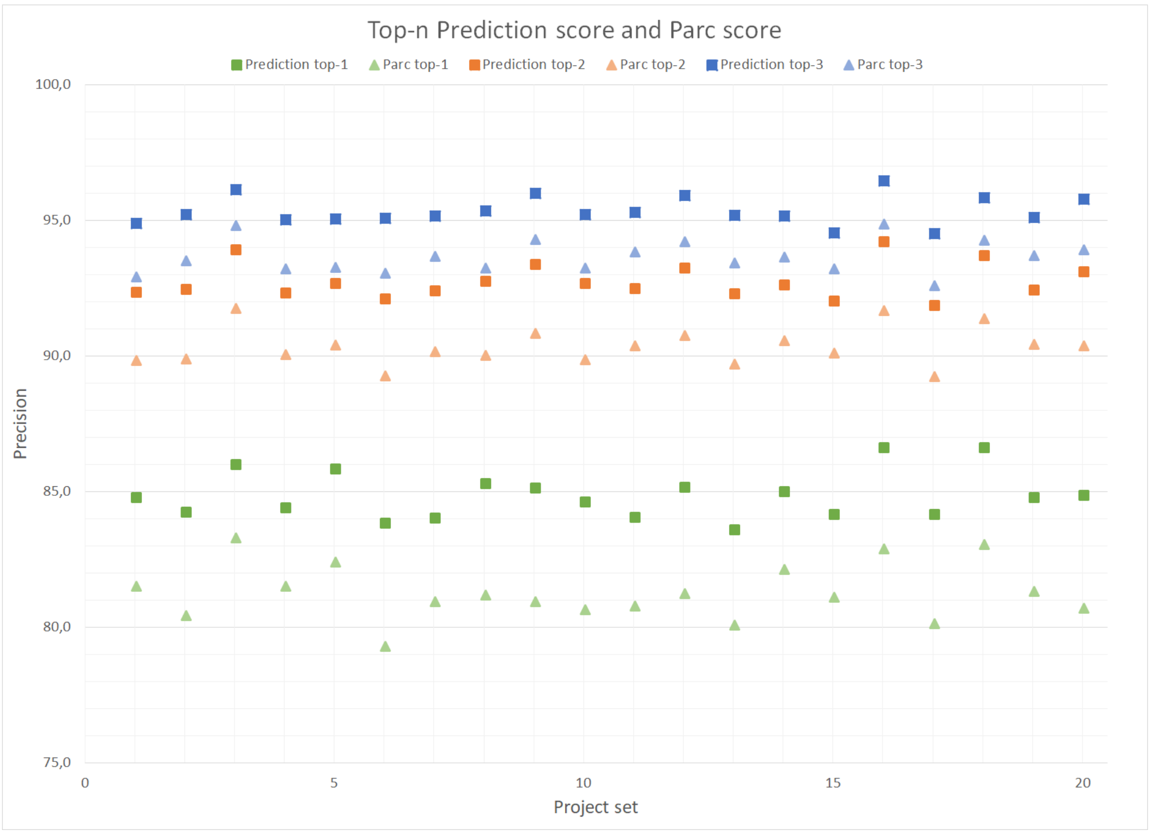


Figure 5.3: Prediction score and Parc score for top-n recommendations.
Showing what the prediction and Parc score would be if 1, 2 and 3 recommendations could be provided.

5.4 Runtime performance

The usability of a recommendation is also dependent on how fast recommendations can be given. According to Proksch et al. [PLM15] a recommendation should be provided within 100 milliseconds to be useful to the developer. Based on a test with 20 projects (Table 5.3) we establish that collecting all features for every candidate of one argument takes on average 2 milliseconds. The recommendations are generated separately in this test and this takes around 33 seconds for all 46170 arguments, which is negligible per argument. Therefore, our approach is well within the maximum of 100 milliseconds.

Projects	Arguments	Seconds
NanoHttpd__nanohttpd	602	15
NLPchina__elasticsearch-sql	1397	27
OpenTSDB__opentsdb	708	17
Pi4J__pi4j	2804	44
xetorthio__jedis	6906	108
allure-framework__allure1	467	14
apache__rocketmq	8432	116
apache__storm	4172	79
biezhi__wechat-ap	210	10
brettwooldridge__HikariCP	492	13
brianfrankcooper__YCSB	528	10
bytedeco__javacpp	1776	27
cglib__cglib	866	12
changmingxie__tcc-transaction	280	5
codingXiaxw__seckill	64	3
coobird__thumbnailator	3011	21
DozerMapper__dozer	3644	50
dropwizard__dropwizard	1877	165
dropwizard__metrics	1434	48
FasterXML__jackson-core	6500	90
Total	46170	874

Table 5.3: Time taken to collect all candidates, and their features, for all method call arguments in a project.

Chapter 6

Conclusions

This work continues on the research by Zhang et al. [ZYZ+12] and Asaduzzaman et al. [ARMS15] to improve the recommendation of method call arguments. We determined that the highest performance increase could be attained by improving the completion of one specific type of argument, instead of adding support for more types of arguments. Therefore, we focussed on the completion of basic variables, which represent around 40% of all method call arguments [ZYZ+12, ARMS15].

By analysing code prior to method call arguments we determined features in the code which should impact argument completion. 45 possible features were found. We then demonstrate that a deep neural regression approach is useful to infer the most likely candidate, for the argument, based on these features.

6.1 Discussion

This study identifies certain threats to its validity. The data that is used throughout the study is based on popular open source projects found on Github (Appendix A). The fact that the projects are popular could indicate that they are well supported and that the code has been refactored. This could mean that the code used is more structured and names of variables are better chosen than live code being written on the fly.

The manner in which the arguments and their candidates were extracted from the open source projects is also of importance. For the extraction a third party open source library was used that was still in development. Although the library performed well, there were some problems. In some cases we could not link the correct method to a method call. For some methods we also could not solve the names of their formal parameters. Adding to this there were also some cases in which generics could not be resolved. In all theses cases the arguments have been ignored.

Another threat to the validity is the use of the students t-test in the evaluation. There are two problems identified regarding this test. The first is the low number of samples that is used to calculate the average prediction scores. Although there does not exist a minimum amount of samples to compare means using the students t-test, it is generally assumed that a higher number of samples provides more robust results. Secondly, using the t-test for multiple comparisons significantly lowers the validity because every test has a 5% change of giving an incorrect result. To counteract this, the t-test is only used in experiments to compare prediction scores to one base score and not between all scores. Secondly, the results of the t-test are only used for conclusions when they provide the same results for both testing sets in the evaluation.

The data and the features that were used in this study were focussed on the completion of method call arguments in the Java programming language. However, we assume that this method could be used in most other general-purpose object-oriented programming languages (Python, C#.net, VB.net). The reasoning for this is that most of these languages show the same characteristics that are caught in the selected features found in chapter three. However, this does not mean it will be as successful in these other languages. One big obstacle is statically versus dynamically typed languages. In a dynamically typed language we cannot filter the list of candidates based on type-compliancy, therefore, we are left with more candidates. Another obstacle could be how and if late binding and dynamic dispatch are implemented. However, in this study we have assumed that the method being targeted by a method call is always known.

6.2 Conclusion

In this work we have explored a deep neural regression approach to the completion of method call arguments of the basic variable type. We focused on understanding which coding features contributed to the correct recommendation of candidate variables **(SQ1)**, how the deep neural regression network should be applied to improve the percentage of correct recommendations **(SQ2)** and how our proposed approach compares to previous research **(SQ3)**.

Based on the evaluation we conclude that the features we have added are beneficial to the performance **(SQ1)**. We identified four features that significantly affect the performance:

- `usedInMethodCallCombination`: if the combination of method call and argument was used prior to where the argument is requested.
- `unusedLocalCandidatesInRow`: the amount of unused candidates in sequence from where the argument is requested.
- `usedInMethodCall`: if the candidate in question has been used in another method call.
- `lexicalSimilarity`: the lexical similarity between the name of the candidate and the formal parameter name.

Based on the results regarding the variations, on batch size and hidden layer structure and weight initialization of the deep neural regression network we do not see a significant impact on the performance of our approach **(SQ2)**. Therefore, the variations resulting in faster training and calculation times have been chosen.

We compare the results of our approach to that of the rule-based algorithm used in Parc, which represents the state of the art in this field of research.

To compare our approach to Parc we randomly selected 71 out of 142 open source projects. To make predictions for all method call arguments in these 71 projects, we trained our deep neural network on the remaining 71 projects. Repeating these steps 20 times with a new selection of projects ever time, we average the prediction scores to find the final prediction score. We find that our approach outperforms PARC consistently and predicts on average 84.9% of method call arguments correctly to Parc's 81.3% **(SQ3)**. On the individual project level, based on one of these selections of 71 projects, we show that our approach is on par with or outperforms Parc in 88.7% of projects.

Based on our evaluation we conclude that we can improve method call argument completion of basic variables using deep neural regression **(RQ1)**.

6.3 Future work

In future work we would like to incorporate our approach into an IDE plugin that can be used by developers. Secondly, we would like to explore more features. In the current approach all type-compliant variables are treated equally. However, we could argue that in Java there are certain degrees of type-compliancy. Does the candidate type exactly match the requested type or is it a sub-type of this type or a sub-type of that sub-type? A feature could be added that would reflect this information. In the study we also noticed the importance of the lexical similarity between the name of the candidate and the name of the formal parameter name. This similarity is measured using the exact strings used for both names. However, it would be interesting to see if developers use synonyms of the formal parameter name in their naming of candidates, especially if the method call targets a method from an external library or framework. Synonyms can be found using standard dictionaries. These list of synonyms can then be extended or limited to words used in the accompanying text of the code and overall project. Accompanying text could provide enough information for this purpose because *"In order to manage and understand the complexity of their programs, software developers embed important, useful information in test suites, error messages, manuals, variable names, code comments, and specifications"* [Ern17].

In this study we have recommended method call arguments for projects with an unknown amount of developers. It will be interesting to see how well this method performs on personal projects written by only one developer. The method could also be personalized by training the model on a combination of the project and the code of an individual developer.

In this study a project has always been used to either train the model or recommend arguments, never both. However, it could be beneficial on larger existing projects to train the model on the existing code repository as well. This could enhance the method by learning the local rules of the project.

It can also be investigated if this method works for other types of arguments. A likely candidate to begin with is the field access expression (`object.field`).

Acknowledgment

We would like to sincerely thank our direct supervisors dr. Clemens Grelck, dr. Maarten van Someren and Willem Meints for their excellent guidance and great feedback. Secondly, we want to thank Bart Bijl and all others at Info Support B.V. for their support, stimulating conversations and insight. Consecutively, we want to thank Info Support B.V. for this opportunity, the support they provided and for the stimulating work environment with even more helpful people and the infrastructure necessary to effectively work on this thesis.

We would also very much like to thank Asaduzzaman et al. for the generously provided source code to the Parc tool.

Bibliography

[ARMS15] Muhammad Asaduzzaman, Chanchal K Roy, Samiul Monir, and Kevin A Schneider. Exploring API method parameter recommendations. In Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 271–280. IEEE, 2015.

[ARSH14] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. CSCC: Simple, efficient, context sensitive code completion. In Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 71–80. IEEE, 2014.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 213–222. ACM, 2009.

[Ern17] Michael D Ernst. Natural language is a programming language: Applying natural language processing to software development. In Proceedings of the LIPIcs-Leibniz International Conference in Informatics, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[HBG+] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. Communications of the ACM.

[KMN+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. CoRR, abs/1609.04836, 2016. URL: `http://arxiv.org/abs/1609.04836`, `arXiv:1609.04836`.

[LLS+16] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In Proceedings of the 38th International Conference on Software Engineering (ICSE), pages 1063–1073. IEEE/ACM, 2016.

[MKF06] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the Eclipse IDE? IEEE Software, 23(4):76–83, 2006.

[NHC+16] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. API code recommendation using statistical learning from fine-grained changes. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pages 511–522. ACM, 2016.

[NNN+12] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In Proceedings of the 34th International Conference on Software Engineering (ICSE), pages 69–79. IEEE, 2012.

[PG11] Michael Pradel and Thomas R Gross. Detecting anomalies in the order of equally-typed method arguments. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA), pages 232–242. ACM, 2011.

[PLM15] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with Bayesian networks. ACM Transactions on Software Engineering and Methodology (TOSEM), 25(1):3, 2015.

[RVY14] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In ACM Sigplan Notices, volume 49, pages 419–428. ACM, 2014.

[ZYZ^{+}12] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical API usage. In Proceedings of the 34th International Conference on Software Engineering (ICSE), pages 826–836. IEEE, 2012.

Appendix A

Open source java projects that were used

- Activiti/Activiti.git
- alibaba/fastjson.git
- allure-framework/allure1.git
- Alluxio/alluxio.git
- antlr/antlr4.git
- apache/commons-lan.git
- apache/flink.git
- apache/flume.git
- apache/httpcomponents-clien.git
- apache/rocketmq.git
- apache/storm.git
- asciidocfx/AsciidocFX.git
- ata4/disunity.git
- AxonFramework/AxonFramework.git
- b3log/solo.git
- b3log/symphony.git
- biezhi/wechat-ap.git
- bitcoinj/bitcoinj.git
- brettwooldridge/HikariCP.git
- brianfrankcooper/YCSB.git
- BroadleafCommerce/BroadleafCommerce.git
- bytedeco/javacpp.git
- caoxinyu/RedisClien.git
- cglib/cglib.git
- changmingxie/tcc-transaction.git
- checkstyle/checkstyle.git
- chewiebug/GCViewer.git
- clojure/clojure.git
- cloudera/flume.git
- code4craft/webmagic.git
- codingXiaxw/seckill.git
- coobird/thumbnailator.git
- deeplearning4j/dl4j-examples.git
- DozerMapper/dozer.git
- dropwizard/dropwizard.git
- dropwizard/metrics.git
- ebean-orm/ebean.git
- eclipse/smarthome.git
- eclipse/vert.x.git
- elasticjob/elastic-job-lite.git
- fabric8io/fabric8.git
- FasterXML/jackson-core.git
- google/auto.git
- google/closure-compiler.git
- google/error-prone.git
- google/truth.git
- Graylog2/graylog2-server.git
- gsh199449/spider.git
- guoguibing/librec.git
- hankcs/HanLP.git
- haraldk/TwelveMonkeys.git
- hcoles/pites.git
- hs-web/hsweb-framework.git
- HubSpot/Singularity.git
- Impetus/Kundera.git
- jamesdbloom/mockserver.git
- jdbi/jdb.git
- jfinal/jfinal.git
- jhalterman/failsafe.git
- jhipster/jhipster-sample-app.git
- jmxtrans/jmxtrans.git
- joel-costigliola/assertj-core.git

- JpressProjects/jpress.git
- junit-team/junit4.git
- kanwangzjm/funiture.git
- kbastani/spring-cloud-microservice-example.git
- kevinsawicki/http-reques.git
- kevin-wayne/algs4.git
- kiegroup/optaplanner.git
- knightliao/disconf.git
- lets-blade/blade.git
- lettuce-io/lettuce-core.git
- liaohuqiu/android-Ultra-Pull-To-Refresh.git
- linkedin/camus.git
- liyifeng1994/ssm.git
- lukehutch/fast-classpath-scanner.git
- mahmoudparsian/data-algorithms-book.git
- medcl/elasticsearch-analysis-ik.git
- medcl/elasticsearch-analysis-pinyin.git
- mitreid-connect/OpenID-Connect-Java-Spring-Server.git
- monkeyWie/proxyee-down.git
- mrniko/netty-socketio.git
- NanoHttpd/nanohttpd.git
- naver/pinpoin.git
- Nilhcem/FakeSMTP.git
- ninjaframework/ninja.git
- NLPchina/ansj_se.git
- NLPchina/elasticsearch-sql.git
- NLPchina/nlp-lan.git
- oldmanpushcart/greys-anatomy.git
- OpenFeign/feign.git
- OpenHFT/Chronicle-Map.git
- OpenHFT/Chronicle-Queue.git
- opentripplanner/OpenTripPlanner.git
- OpenTSDB/opentsdb.git
- openzipkin/zipkin.git
- pac4j/pac4j.git
- pagehelper/Mybatis-PageHelper.git
- Pi4J/pi4j.git
- plantuml/plantuml.git
- plutext/docx4j.git
- proxyee-down-org/proxyee-down.git
- Qihoo360/XLearnin.git
- qos-ch/logback.git
- qos-ch/slf4j.git
- quartz-scheduler/quartz.git
- querydsl/querydsl.git
- rakam-io/rakam.git
- rampatra/jbo.git
- rapidoid/rapidoid.git
- redisson/redisson.git
- rest-assured/rest-assured.git
- richardwilly98/elasticsearch-river-mongodb.git
- rkalla/imgscalr.git
- sarxos/webcam-capture.git
- scouter-project/scouter.git
- scribejava/scribejava.git
- shardingjdbc/sharding-jdbc.git
- socketio/socket.io-client-java.git
- spring-cloud/spring-cloud-netflix.git
- spring-projects/spring-data-elasticsearch.git
- spring-projects/spring-data-mongodb.git
- spring-projects/spring-petclinic.git
- sqshq/PiggyMetrics.git
- square/javapoe.git
- square/keywhiz.git
- square/okio.git
- Tencent/angel.git
- tinkerpop/gremlin.git
- tjake/Solandra.git
- TooTallNate/Java-WebSocke.git
- traccar/traccar.git
- twitter/ambrose.git
- twitter/elephant-bird.git
- vert-x3/vertx-examples.git
- vipshop/Saturn.git
- winder/Universal-G-Code-Sender.git
- xetorthio/jedis.git
- xmindltd/xmind.git
- xuhuisheng/lemon.git
- xuxueli/xxl-job.git
- zouzg/mybatis-generator-gu.git

The projects were downloaded on the 13th of June 2018. The selection of projects was based on the following requirements:

- The project is a Java project

  - This will be limited even further by also filtering on projects that do not contain the word Arduino or Android in the maven file to limit the projects as much to pure java projects.

- The project is build with Maven
  - Useful because dependencies can be automatically acquired.
  - Automatically establish if the project compiles.
- The project compiles
- The size of the projects is between 10MB and 500MB
  - Due to limited time and disk space.
- Projects are sorted based on the amount of times they were starred
  - The assumption is that starred projects are used more and are generally of a better quality