DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Context-Sensitive Code Completion

Improving Predictions with Genetic Algorithms

MARCUS ORDING

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

ABSTRACT

Within the area of context-sensitive code completion there is a need for accurate predictive models in order to provide useful code completion predictions. The traditional method for optimizing the performance of code completion systems is to empirically evaluate the effect of each system parameter individually and fine-tune the parameters.

This thesis presents a genetic algorithm that can optimize the system parameters with a degree-of-freedom equal to the number of parameters to optimize. The study evaluates the effect of the optimized parameters on the prediction quality of the studied code completion system. Previous evaluation of the reference code completion system is also extended to include model size and inference speed.

The results of the study shows that the genetic algorithm is able to improve the prediction quality of the studied code completion system. Compared with the reference system, the enhanced system is able to recognize 1 in 10 additional previously unseen code patterns. This increase in prediction quality does not significantly impact the system performance, as the inference speed remains less than 1 ms for both systems.

Keywords Context-sensitive code completion · Genetic algorithms · k-fold cross-validation

ABSTRAKT

Inom området kontextkänslig kodkomplettering finns det ett behov av precisa förutsägande modeller för att kunna föreslå användbara kodkompletteringar. Den traditionella metoden för att optimera prestanda hos kodkompletteringssystem är att empiriskt utvärdera effekten av varje systemparameter individuellt och finjustera parametrarna.

Det här arbetet presenterar en genetisk algoritm som kan optimera systemparametrarna med en frihetsgrad som är lika stor som antalet parametrar att optimera. Studien utvärderar effekten av de optimerade parametrarna på det studerade kodkompletteringssystemets prediktiva kvalitet. Tidigare utvärdering av referenssystemet utökades genom att även inkludera modellstorlek och slutledningstid.

Resultaten av studien visar att den genetiska algoritmen kan förbättra den prediktiva kvaliteten för det studerade kodkompletteringssystemet. Jämfört med referenssystemet så lyckas det förbättrade systemet korrekt känna igen 1 av 10 ytterligare kodmönster som tidigare varit osedda. Förbättringen av prediktiv kvalietet har inte en signifikant inverkan på systemet, då slutledningstiden förblir mindre än 1 ms för båda systemen.

Nyckelord Kontextkänslig kodkomplettering · Genetiska algoritmer · k-delad korsvalidering

Table of Contents

1 Introduction

Software developers often need to have updated knowledge about multiple languages and frameworks, forcing them to rely on different development tools. One such tool is the code completion feature of Integrated Development Environments (IDEs). Murphy et al. (2006) found that code completion in the Eclipse IDE can reduce coding efforts and provide insights into Application Programming Interfaces (APIs) by displaying variables and methods applicable within a scope. Similar findings were discussed in relation to the Visual Studio IDE by Amann et al. (2016).

Traditional code completion systems are based on determining the static type of variables, and in the completion process suggesting all actions applicable to that type (Bruch et al., 2009). Efforts to improve the relevance and number of predictions have often consisted of prefix matching, something that only benefits a developer already familiar with an API (Proksch et al., 2015). However, recent systems take a less naive approach. These intelligent code completion systems are able to take surrounding context of the current edit location into consideration (Proksch et al., 2015). With this information, a list of only the relevant predictions can be presented. The process behind context-sensitive code completion starts with extracting a context describing the current edit location. This context is then matched against source code repositories from which the relevant actions can be proposed.

Proksch et al. (2015) found that the relevance of intelligent code completion predictions are highly dependent on the context information, available code repositories and the model from which predictions are extracted. This thesis will explore how the negative impact of these can be reduced to provide more relevant predictions.

1.1 Background

One already applied framework for introducing programming standards is coding conventions, a set of best practices applied at a language- and programming paradigm level, sometimes modified within a single project or business (Sommerville, 2010, p. 52). These support the design of concise, easily maintainable and stable code that every developer involved can understand.

Intelligent code completion systems can exploit the fact that code often follows strict structural rules. For example, one common guideline is to keep the scope of variables as small as possible. This practice can lead to some parts of the code being clustered, where related object instantiations and method invocations appear on adjacent lines in the source code. Ordering might also be important within these clusters.

Predictive modeling is commonly used in modern intelligent code completion systems, such as those presented in the work by Bruch et al. (2009) and Proksch et al. (2015). These predictive models are trained to recognize patterns in code and are then used to predict how likely a certain prediction is in a context.

1.2 Problem

Predictive models are highly dependent on the data available, but also on the training process. Particularly, the problem of recognizing previously unseen patterns is difficult to solve. If the patterns contain too much information, there is a risk of losing generalizability. On the other hand, predictions become imprecise if the patterns ignore too much of the information.

Current methods of training code completion systems can possibly be improved in order to reduce prediction errors. This requires that the amount of information considered in a pattern is optimized. The question raised is then: how should the training of predictive models be focused in order to increase prediction quality?

1.3 Purpose

This thesis will aim to answer the question: to what extent is it possible to improve predictions of existing state-of-the-art code completion systems with a genetic algorithm?

1.4 Goal

The goal of this project is to enhance an existing code completion system with a genetic algorithm, and to deliver a research prototype as a proof-of-concept. Existing evaluation methods of code completion systems will be extended to more closely model the environment in which code completion is typically applied.

1.5 Benefits, ethics and sustainability

Benefits

Researchers working on code completion would benefit the most of the contributions of this work. These contributions could possibly be applied in other research fields where predictive models are used as well. Developers and maintainers in many software development companies could also benefit from this work. The contributions of this work has the potential to simplify and speedup the coding process, thereby increasing productivity.

Ethics

The major ethical concern regarding this work is the influence of other researchers work. There is a need to give the researchers credit for their ideas and contributions, but also how their research has influenced this work. It is also important that their work is represented accurately, and that any modifications are clearly communicated. On the other hand, it is equally as important to critically evaluate the validity of the related research and how it may affect this work.

Since the research is conducted outside of any contracts or non-disclosure agreements, with no industry stakeholder, such secrecy does not need to be accounted for. In addition to this, there are no safety critical aspects to be considered. What must be observed however is that any licensing of development tools and software projects supports the use cases of this work.

Sustainability

One common approach to sustainable development is the three-pillar approach, decomposing the often overlapping goals of sustainable development into economical, environmental, and social aims (Hilty and Hercheui, 2010, p. 229). The research presented in this thesis will improve sustainable development in the following ways.

- *Economical* : While this thesis does not provide any answers through a user study, it is possible that developers' productivity can be increased by improving the intelligence of existing code completion systems. Bruch et al. (2009) has conducted such a user study with developers that shows this correlation. An increase in productivity could lead to economical growth, even without consuming additional resources, thus contributing to sustainable development.
- *Environmental* : The research artefacts of this work will be published in their entirety as open-source, thus contributing to environmental sustainability by eliminating the need for future researchers to replicate the development efforts.
- *Social* : Since an intelligent code completion system benefits inexperienced developers the most, it is possible that social equity could be increased in the field of software development. Increased social equity could also promote diversity by making software development available to a wider audience.

1.6 Methodology

Methods of inquiry in conducting scientific research can be divided into qualitative and quantitative approaches. Such a crude division can be rather misleading, however, as most studies will naturally fall somewhere on a continuum between these extremes (Newman and Benz, 1998, p. 3). The methodology applied in this thesis is a complementary combination of both quantitative and qualitative methods. Ultimately, the research hypotheses is answered by a qualitative study of the intelligence of a reference code completion system and the same system trained with the new predictive model training scheme. The two candidates in the study only differ in the way they have been trained, which means the reason for any differences in intelligence is identified and understood. However, the intelligence of the two candidates is measured quantitatively.

Furthermore, the reader should be aware that while the experiments can be repeated, the results are not guaranteed to be replicable. There is a random factor introduced by the genetic algorithm used in the training of the code completion system that makes the experiments non-deterministic.

1.7 Delimitations

This thesis is confined in its scope by the following delimitations:

- *Objective*: The research conducted intends to improve the prediction quality of a single code completion system. The studied system is CSCC, proposed by Asaduzzaman et al. (2014). A genetic algorithm will be used to optimize the parameters of the CSCC system.
- *Variables of study*: The evaluation of the code completion systems in this research focuses on three variables: prediction quality, model size and inference speed. These variables are elaborated on in Chapter 5.1.
- *Perspectives*: This thesis studies code completion in relation to C#, as this particular perspective on code completion is mostly unexplored. It should be noted, however, that object-oriented languages are common subjects of research on code completion.
- *Population*: There are eight code repositories chosen as a source of usage contexts. Chapter 5.2 describes the choice of these particular code repositories in more detail.

No IDE plug-in based on the algorithm will be delivered, thus it will not be possible to evaluate the algorithm qualitatively in a real environment. This delimits the study in that it will not be possible to find a correlation between the intelligence of the code completion

system and the productivity of a developer. It will also not be possible to determine whether developers can actually perceive any difference between the studied systems.

1.8 Outline

The reader is first introduced to the more technical areas of the thesis. Chapter 2, Previous work on code completion, outlines the current knowledge including substantive findings, as well as theoretical and methodological contributions of previous work on code completion. Chapter 3, Parameter optimization with genetic algorithms, introduces the background of genetic algorithms and their application in parameter optimization.

After the introductory theory follows two chapters explaining the process behind this work. Chapter 4, Development process, focuses on the publishing of the source code as well as the choice of code analysis service and genetic algorithm library. Chapter 5, Research methods, goes into more detail about how the research was approached. It is also discussed how the validity of the results is ensured.

In Chapter 6, Implementation of GeneCSCC, the actualization of the proposed code completion system is detailed.

The results of the evaluation are presented in Chapter 7, Results. Following this is a discussion about these findings in Chapter 8, Discussion.

Finally, Chapter 9, Conclusions and Future work, recaps how the stated problem was tackled. The major accomplishments and results are highlighted, and the section finishes with relating the research to the world outside of academia as well as providing some future directions.

2 Previous work on code completion

This chapter presents a review of related research on state-of-the-art code completion systems and discusses possible extensions of the research.

2.1 Review of theoretical and methodological contributions

Bruch et al. (2009) noted that the code completion system featured in Eclipse lacked the intelligence necessary to support developers, particularly in cases when the classes define diverse functionality or deep inheritance hierarchies. As a possible solution to this, three different systems were presented, all based on statistics gathered from existing code repositories.

Frequency-based code completion

FreqCCS was based on the relative frequencies of a methods use in the example code, where a large number of occurrences yielded a higher score. The simple rationale behind this was that popular methods are more likely to be applicable in the future as well. One can immediately identify some shortcomings of this reasoning. For one, there is no guarantee that the examined code is representative of a class in the current context. Secondly, many methods might receive similar scores accounting for the entire code base. However, the frequent use of a method might be confined to a particular context, and this system is not able to infer such information.

Association rule-based code completion

ArCCS applied the machine learning technique *association rule mining* to find useful associations between code patterns. Such a problem corresponds to finding all rules $A \rightarrow B$ that associate one set of items with another set. A correspondence on this form can be formulated as A *implies* B *with some confidence level*. In ArCCS, association rules were mined from code repositories at a variable-scoped level. Code completion then worked by extracting the surrounding context of a variable and using the context to determine the rule to select.

Modified k-nearest neighbor code completion

BMNCCS was a modification the k-Nearest Neighbor machine learning algorithm. This algorithm bases its predictions on previous experience (training), taking into consideration some observations about the current context and storing it as a feature vector. Contexts

are determined per call site, where each context consists of the methods that have already been invoked on the receiver variable as well as the enclosing method of the call site. When proposing a set of possible code completions, the algorithm identifies method calls to be recommended based on their frequencies in the selected nearest neighbors.

Based on the overall prediction quality, as defined in Section Evaluating the predictive quality of code completion systems, the authors findings were that FreqCCS performed the worst, whereas the other systems were relatively equal. A slight edge was given to BMNCCS, which was also the system that became the basis for Eclipse Code Recommenders.

Pattern-based Bayesian network code completion

Based on the findings by Bruch et al. (2009), Proksch et al. (2015) identified that model sizes and inference speed did not scale well with additional input data, whereas prediction quality increased with available data. To solve this, they applied a probabilistic graph model called Bayesian networks to BMNCCS. Bayesian networks are able to model a context from limited knowledge, and then infers relationships based on probabilities. By only considering these dependencies and ignoring any other, the model size can be greatly reduced.

Proksch et al. (2015) found that model sizes could be reduced with up to 90%, while only incurring a small reduction in prediction quality. In fact, the inference speed of their solution was shown to scale logarithmically, while BMNCCS scaled linearly. However, this only had significant impact for input sizes larger than 15,000 usage patterns, an amount that was not observed in any of the tested frameworks. The major benefit of using clustering would instead seem to be that the threshold for probabilistic relationships could be configured to allow for reduced model sizes while still maintaining high prediction quality.

Proksch et al. (2015) also noted that the current version of Eclipse Code Recommenders had implemented a similar approach while their own research was ongoing (Recommenders).

Token-based similarity code completion

Based on the findings by Bruch et al. (2009), Asaduzzaman et al. (2014) identified a different way to represent the context information. The context consisted of the method invocations, target type in variable assignments and the language keywords found in the lines prior to the considered invocation. Asaduzzaman et al. (2014) implemented a code completion system based on this definition of a context called ***CSCC***.

CSCC works in three steps. The first step mines usage contexts from existing code repositories and builds a model of the contexts. An inverted index structure indexed by the type of the

invocation call site is used to represent the model. The second step extracts a list of candidate code completions by using a coarse similarity metric to filter those candidates that fall below the similarity threshold. The third step further filters the list of candidates, but the similarity metric used is more fine-grained than that in the second step.

There are multiple parameters of the CSCC system that affects the predicted code completions. As such, there is a direct and easily observable correlation between the parameter value and the resulting code completion, This makes the CSCC system an ideal candidate in this study, a decision that is further elaborated on in Section Discussion of this chapter.

Evaluating the predictive quality of code completion systems

There already exists an established way of evaluating information retrieval systems such as search engines (Beitzel, 2006), used by Bruch et al. (2009), Asaduzzaman et al. (2014) and Proksch et al. (2015).

Recall can be defined as the ratio of relevant predictions that the system made (#hits) to the total number of relevant predictions in a context (#expected).

$$recall = \frac{\#hits}{\#expected} \tag{1}$$

Precision can be defined as the ratio of relevant predictions made (#hits) to the total number of predictions made (#proposed).

$$precision = \frac{\#hits}{\#proposed} \tag{2}$$

The overall prediction quality combines recall and precision into the F_1-measure. It is the harmonic mean of recall and precision, where both are equally weighted. F_1 is based on the effectiveness measure established by Rijsbergen (1978).

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3}$$

Discussion

Of the reviewed code completion systems, CSCC is the target of this study. This choice is partly forced by limitations of time, as both BMNCCS and PBN are too complex to cover within the scope of this study. Asaduzzaman et al. (2014) also concluded that CSCC

improved the prediction quality over BMNCCS, an improvement comparable with that of PBN over BMNCCS. Even though this study is not concerned with how accurate the reference system is, it might make sense to enhance the more accurate system if the system should be integrated into Visual Studio in the future. However, future research should not exclude BMNCCS or PBN as candidates due to this reason.

From what has been observed in the examined literature, it seems that the choice of training data is key in producing a useful model. What none of the previous research has applied yet is pre-processing of gathered patterns. If the training data could be validated for some key factors before they are fed into the training phase, it is possible that the model sizes could be reduced and prediction quality increase by not considering unnecessary patterns. Related to this is how the test cases are determined. The reviewed research is based on randomly removing a method call from the code, and then evaluating what the tested system proposes. A more structured approach could potentially be used to gain more representative statistics of actual use cases.

The reviewed research is all based on code completion for Java, specifically the Eclipse IDE. Since similar research is yet to be done for C# and the Visual Studio IDE, focusing on these provides this study with a unique perspective. The current state-of-the-art code completion systems for C# are those implemented in ReSharper and CodeRush, none of which are open-source so their internal implementation will not be possible to study. However, of interest to this study is that ReSharper does not leverage Roslyn (Gorohovsky, 2014), whereas CodeRush has transitioned towards it (Miller, 2014).

3 Parameter optimization with genetic algorithms

This chapter introduces genetic algorithms. The rest of this chapter is structured as follows. Chapter 3.1 gives a brief background on genetic algorithms. Chapter 3.2 introduces the terminology of genetic algorithms, as well as their their application in this work. Chapter 3.3 presents some limitations of genetic algorithms. Finally, Chapter 3.4 present some alternatives for solving the optimization problem of this study.

3.1 Background

Genetic algorithms pertain to the field of Artificial Intelligence. Based on the evolutionary process, a genetic algorithm has the ability to learn and adapt in the same sense as natural selection changes in a response to its environment (Mitchell, 1998, p. 2). A genetic algorithm is a meta-heuristic, a problem-independent heuristic able to search for approximate solutions to a problem. The intention of the genetic algorithm is typically to solve a global optimization problem by moving away from local minima or maxima (Said et al., 2014). It is this effectiveness in solving optimization problems, especially feature selection and weighting tasks, that is of interest to the problem approached in this thesis.

3.2 Genetic algorithms in this study

Refer to Figure 1 for a basic overview of the genetic algorithm used in this work. The genetic algorithm seeks to approximate the best combination of the three parameters of the studied code completion system that are subject to optimization.

A genetic algorithm starts with a population of individuals called *chromosomes* that encode possible solutions to the problem. The chromosome consists of a set of variables, each known as a *gene*. These genes represent a parameter of the problem being solved. The encoding relies on the problem being solved (Aggarwal et al., 2014), and for the particular problem in this study, *value encoding* is used. Value encoding represents each gene in a way that is natural for the parameter it encodes. Specifically, the chromosomes in this work are encoded as a mix of real-valued numbers and natural numbers.

The *population* of candidate solutions evolves towards better solutions through an iterative process, with each new population being called a *generation*. Evolution is guided by a fitness function that corresponds to a value of an objective function in the given optimization problem. In this case it is the prediction quality of the code completion system with parameter

values as specified by the chromosome being evaluated. Those individuals with greater *fitness* are stochastically selected and their genome is transferred to the next generation.

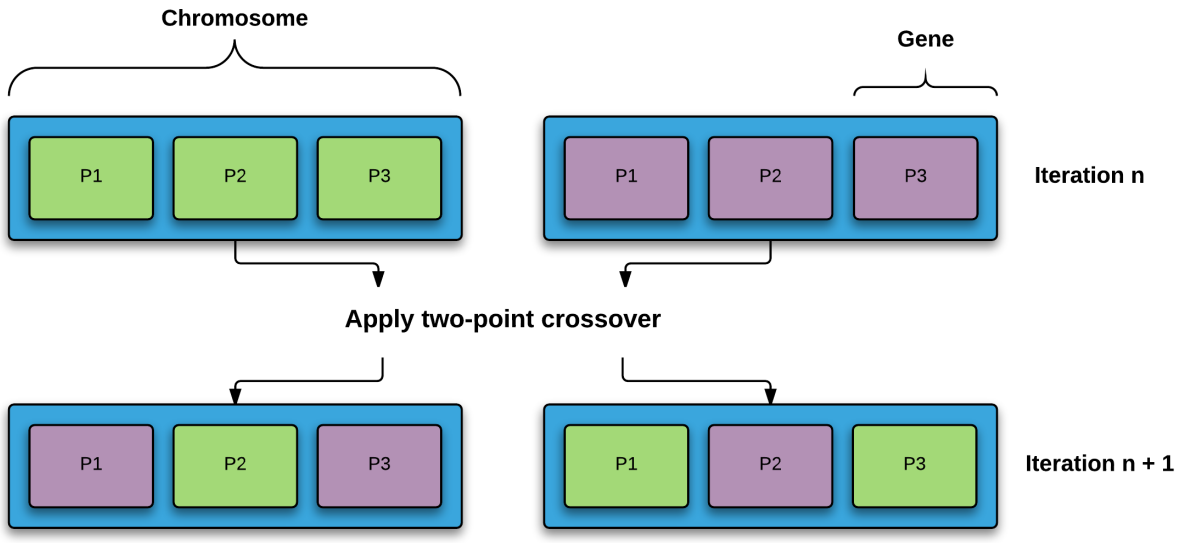

Figure 1: Conceptual design of the genetic algorithm used in this work.

Initialization

Each generation consists of a pre-determined, variable number of individuals. The initial population is a randomly generated set of permutations of the original chromosome subject to optimization. This chromosome is encoded with the values of the parameters used in the *CSCC* system, or alternatively, a previously found fittest solution.

Selection

Selection is the process of choosing the fittest individuals from a population. A naive approach to this selection can consist of evaluating the fitness function for each individual and then normalizing the values. Normalization is achieved by dividing the fitness score of each individual with the sum of the fitness values of all individuals. Each normalized value then lies between 0 and 1, and the sum total of all values is 1. This serves the purpose of scaling each generation so that they are comparable. If this step would be left out, it would be impossible to tell how close to the global optima the solutions are. The described process

is called fitness proportionate selection, but is seldom used in practice due to not exhibiting constant selection pressure (Bäck, 1994), meaning there is a tendency to always transfer the best solution to the next generation. This would affect the genetic diversity.

There are multiple ways of selecting more than one individual. Depending on the difference in fitness these methods may give unfair advantages to certain individuals. Since the combination of a low-fitness and high-fitness chromosome might result in a chromosome with higher fitness than either one individually, it is important to give all chromosomes sufficient time to reproduce (Blickle and Thiele, 1995). However, doing so could also result in slower convergence, making it an important trade-off to consider.

Due to the limited population size and generation count used in this work, slow convergence is not an option. For this reason, *elitism* is used as the selection mechanism. *Elitism* involves copying a small proportion of the fittest candidates to the next generation. Candidate solutions that are preserved remain eligible for selection as parents when breeding the remainder of the next generation.

Genetic operators

Generating the next generation of chromosomes requires the combination of two genetic operators: *crossover* and *mutation*. Crossover is the reproductive process where two individual chromosomes are selected from a population and segments of their genes are recombined, resulting in two new chromosomes. Mutation is the process of randomly changing some genes, possibly resulting in new chromosomes that no crossover method would create. In the mathematical sense, crossover will converge towards local optima, whereas mutation allows for finding optima closer to the global optima. The probability of mutation determines how many individuals will mutate. If the probability is set too low, the solutions might not converge towards the global optima. On the other hand, if it is set too high, the stability of the genome decreases and the genetic algorithm turns into random search.

Termination

The evolutionary process is repeated until some condition is met. A condition could be any one or a combination of the following (in this particular work):

- When the minimum criteria for optimization is met
- When a pre-determined number of generations has been created
- When the fitness of the highest ranking chromosome has plateaued

As a way of reducing the time it takes to find an optimal solution, and to avoid any one solution to model a particular problem too closely, termination occurs on either of the conditions.

3.3 Limitations

Whitley (2001) has presented some of the limitations inherent to genetic algorithms. One of the most prohibitive aspects in relation to this work is the fitness function. Repeated evaluation of a complex fitness function will limit the amount of iterations and therefore the achievable optimization level. It should also be noted that the diversity among individuals is crucial. Given a selection function that skews too much towards the fittest individuals, the search may end up converging towards local optima or arbitrary points. Since the genetic algorithm in this work selects according to *elitism*, results could possibly be affected in that the found solution is not globally optimal.

3.4 Alternatives

Random search

Random search is a numerical optimization method. The algorithm starts with choosing a random solution to the problem being solved. A second solution is then chosen at random and the fitness value of the two solutions are evaluated and compared. If the second solution is better, the algorithm moves to that solution and continues evaluation. While the implementation is simple, the fitness function takes too long to evaluate in the case of this work for random search to be viable.

Simulated annealing

Simulated annealing is a probabilistic meta-heuristic similar to genetic algorithms in many ways, with two important differences. First, there is only a population of 1 being maintained. Due to the population size, there is also no concept of crossover. Since there is no crossover, simulated annealing is limited in its narrow view of the entire solution space. In addition to this, genetic algorithms are inherently parallelizable. However, there is no reason that prevents simulated annealing from being a viable option to the optimization problem in this work.

4 Development process

This chapter first motivates the choice of publishing platform and licensing of the source code produced. The chapter continues with the introduction of the compiler platform used for source code analysis. Finally, the chapter ends with the introduction of the genetic algorithm library used in this work.

4.1 Source code version control and publishing

All source code produced and research artefacts used as part of this work will be published on GitHub under the MIT license (GitHub, a). A link to the GitHub repository is provided in Appendix A. An open publishing platform will ensure transparency in the development process. By using a permissive license, as opposed to a copy-left license such as GNU General Public License v3.0. (GitHub, b), the barrier to entry is lowered. This opens the research up to reproduction and extension of the results, a desirable goal that strengthens the research validity. It could be argued that this will hinder commercial applications of the research, or allow others to benefit from it. As a counter-point to this, the potential for recognition in the scientific community through citations and collaboration could be valued greater. In the end, the decision is made by considering the potential application of the research in the overall development community. There seems to be few open-source code completion systems available, making it more attractive to develop it as such to differentiate it from the current market.

4.2 Compiler services

Roslyn

Roslyn is an open-source compiler platform for C#, released under the Apache 2.0 license (Roslyn, a). What separates Roslyn from the standard compiler is its transparency. Traditionally, compilers are black boxes which are fed code and produces a compiled assembly (Roslyn, b). What Roslyn does is provide Compiler-as-a-Service, where the internal compiler API is exposed through the Roslyn API. This allows transparent access to syntactic and semantic information about the code, and even supports modifying the code and recompiling it. Since code analysis is a large part of developing a code completion system this is an essential tool. The choice of Roslyn is further supported by the fact that it can easily integrate with Visual Studio, which means the developed code completion system can replace the standard one with less effort.

When Roslyn parses the source code during compilation, it creates a concrete syntax tree. This is a representation of every syntactic structure found in the source code, including grammatical constructs (non-terminals of the language grammar), lexical tokens (terminals of the language grammar) as well as whitespace and other language-agnostic structures. The syntax tree is an immutable snapshot of the compiled source code, but the way Roslyn re-utilizes underlying nodes makes modification particularly fast.

In order to better understand how Roslyn represents compiled code, consider the simple code snippet in Figure 2. It consists of a class declaration ***C***, containing one method declaration ***M*** that takes one argument ***x***.

```
public class C
{
    public void M(int x)
    {
    }
}
```

Figure 2: A simple C# code snippet.

In Figure 3, the syntax tree generated from the code in Figure 2 is presented. Roslyn represents non-terminals as SyntaxNodes, terminals as SyntaxTokens and everything else as SyntaxTrivia. To be more specific, SyntaxNodes are the declarations, statements, clauses and expressions of the language. SyntaxTokens are the keywords, identifiers, literals and punctuation.

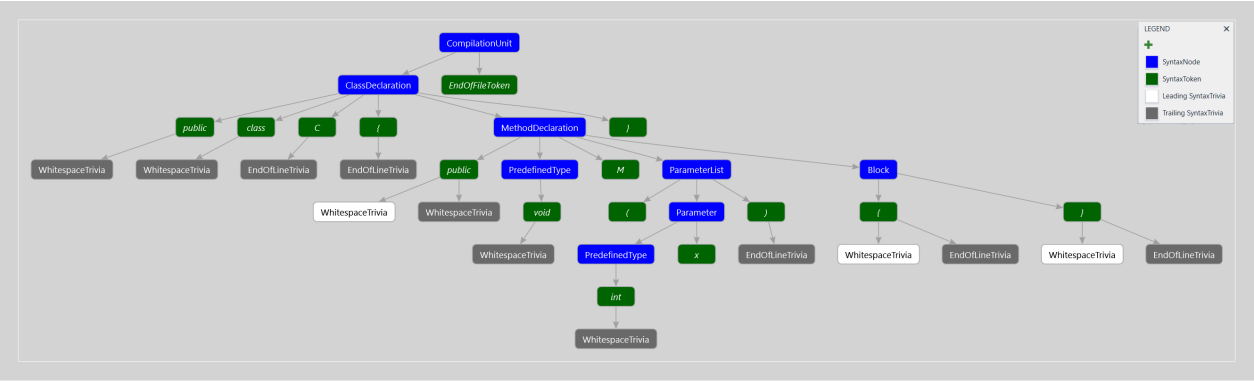

Figure 3: The syntax tree generated by Roslyn for the code in Figure 2.

Alternatives

As the closest alternative to Roslyn stands NRefactory. The only major difference in functionality is that it is not a complete compiler. Rather, NRefactory only handles C# code and does not generate Common Intermediate Language (CIL) code (NRefactory, b). Neither of these affect this work, at least not in its current scope.

The choice of Roslyn over NRefactory is instead motivated by other factors. Roslyn is developed internally at Microsoft, which means it is more likely to have support for new versions of the language where the parser must be rewritten. An internal Microsoft team is also more likely to have intricate knowledge of the compiler. In addition to this, the NRefactory development team discussed the future of their project and noted that it did not make sense to maintain it when Roslyn fills the compiler-service product segment (NRefactory, a).

4.3 Genetic algorithm libraries

GeneticSharp

GeneticSharp is an open-source Genetic Algorithm library for C#, released under the MIT license (Giacomelli). It has an extensible interface that allows for most, if not all, functionality to be implemented from scratch via interfaces or leveraged by extending base classes. Classes and interfaces also use the same terminology that has already been established, which makes the translation from theory to implementation much more clear. Since the GitHub repository has an explanation of included features, and also contains some example usages, there is little need to reiterate it here.

Alternatives

There were two additional libraries considered before GeneticSharp was chosen: ***AForge.NET Genetic Algorithms Library*** (AForge.NET) and ***Genetic Algorithm Framework*** (Newcombe). While no in-depth analysis has been carried out, all three options seem to be comparable with respect to functionality and public interface. The internal implementations might differ substantially, however, something that might affect performance and quality of results.

GeneticSharp has the most permissive license, whereas the other alternatives are both published under GNU Lesser General Public License v3.0.. This was the deciding factor, as the MIT license is much easier to comply with, reducing the risk of legal pitfalls.

One factor that could have shifted the decision in favor of AForge.Genetic is that the AForge.NET framework consists of multiple libraries related to Artificial Intelligence, such

as AForge.Neuro (neural networks) and AForge.MachineLearning (machine learning). Since these are not needed in this work it was not factored into the decision. It is likely that there are other options for these libraries that would need to be considered as well.

5 Research methods

This chapter presents the research methods of the study, continuing from what was established in Chapter 1.6. It is structured as follows. Chapter 5.1 presents the variables that are studied. Chapter 5.2 describes the code repositories that make up the research population. Chapter 5.3 provides a detailed description of how the results are obtained. Chapter 5.4 describes the data collection method. Chapter 5.5 describes the data analysis method. Chapter 5.6 describes how the quality of results is ensured. Chapter 5.7 describes the experimental setup.

5.1 Variables of study

There are three variables that will be considered in this study: prediction quality, model size and inference speed.

Prediction quality

Refer back to Chapter 2.1, Section Evaluating the predictive quality of code completion systems, for the original declaration of the following equations.

$$recall = \frac{\#hits}{\#expected} \quad \text{(1 revisited)}$$

$$precision = \frac{\#hits}{\#proposed} \quad \text{(2 revisited)}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad \text{(3 revisited)}$$

Model size

While it is possible to determine memory requirements of objects empirically, doing so is very imprecise. Factors that impact the observed memory are the version of the Common Language Runtime (CLR), whether the memory is managed by the Garbage Collector and the implementation details of the data structure. Since the goal is to compare memory usage between models, only a comparable measurement is necessary as long as the memory usage is deterministic.

For the code completions system CSCC and its optimized counterpart GeneCSCC described in Chapter 6, model size will be estimated as the number of usage contexts contained in the model multiplied with the size of each usage context in bytes. Only the memory usage of the model in main memory will be considered. The size of memory that the persistently stored model occupies on the physical drive will not be evaluated as part of this study, as the persisted model is serialized.

Inference speed

Proksch et al. (2015) have defined five context-dependent calculations that are necessary in each invocation of code completion:

- 1. Extracting the context information surrounding the current edit location
- 2. Constructing the query
- 3. Loading the appropriate model into memory
- 4. Inferring the relevant predictions
- 5. Displaying the predictions

These steps all contribute to the experienced response time of code completion systems. However, only step 4 is relevant to this work as the studied systems will not be integrated into Visual Studio. How the code completion systems infer the relevant predictions is different between the studied systems as the system parameters are not the same. It is this difference that is of interest to this study. Furthermore, the inference speed will be assumed to be dependent on the number of contexts in the model. As such, the scalability of the systems when the number of contexts in the model increases will be studied as well.

The *Stopwatch* class of C# can provide accurate timing measurements in clock ticks or milliseconds (Microsoft, c). Ticks can be converted to nanoseconds, but nanosecond accuracy will not be not necessary as humans are unlikely to perceive such fine-grained timing measurements. Inference speed measurements will be given in milliseconds instead. Proksch et al. (2015) suggests that the accuracy of the results can be improved with two steps. The tests in this study will be designed in the same way. First, the total elapsed time of each query to the code completion system will be measured. The measured time will then be divided by the total number of queries to give the average inference speed. As a model with few contexts typically return the proposals within a millisecond, the second step will be to measure at least 3,000 queries. Finally, the tests will be repeated three times and an average of the inference speed will be taken over these measurements to reduce the potential impact of interfering processes on the results.

5.2 Code repositories

All tests in this study will target class libraries found in the the .NET Framework, specifically those libraries included in the *System* namespace as described in the .NET Reference Source (Microsoft, b). The reason for this choice is simple; it should be far easier to find a large corpus of usage contexts due to the prevalence of the .NET core libraries in any project. This quality is desirable since the only qualifying condition of a code repository needs to be its size, and not if the code base targets a specific API.

In the study, eight source code repositories will be included. *Mono*, by far the largest one, is a software platform based on the .NET Framework. The .NET Framework (version 4.6.1) itself will also be included, as well as *CoreFX*, the library implementation of the .NET cross-platform counterpart. There will also be *Roslyn*, the .NET compiler platform. *SignalR*, a library that adds real-time web functionality, will be included to possibly diversify the corpus of usage contexts. In addition, there will be three smaller libraries included as well: the unit testing framework *NLog*; Microsoft's data access technology, *Entity Framework*; a JSON framework, *Json.NET*.

5.3 Methodology

A purely experimental methodology will be employed in finding the cause-and-effect relationship between the parameters that will be optimized by the genetic algorithm, described in Chapter 6.4, and the measurable effect that optimization has on prediction quality. These correlations will be used to verify the hypothesis and draw conclusions.

Model evaluation

A method for evaluating predictive models is k-fold cross-validation, where the goal is to evaluate the generalizability of the results. k-fold cross-validation is an extension of the *holdout* cross-validation method, where a data set is divided into two parts, a training set and a validation set. In k-fold cross-validation, the data is divided into k subsets, referred to as a $fold$. For each iteration, one of the k subsets is used as the validation set and the other $k-1$ subsets are merged to form the training set. When evaluating with k-fold cross-validation, an average prediction quality is taken over the k iterations. Averaging reduces the variance, with the disadvantage of increased computation time. In particular, classes with few usage contexts exhibit high variance because they occur infrequently in the validation set.

An alternative to k-fold cross-validation is *stratified* cross-validation, where each *class label* is distributed equally among the folds. In this case, the class label is the class on which an invocation of a usage context occurs. Stratification should in theory reduce bias and variance as the mean response value is approximately the same in each fold. However, in the case when a class label has fewer occurrences than the number of folds, stratified cross-validation would be optimistically biased towards that class. Bias is introduced since the class label would always need to occur in the validation set but would be missing from some training sets. Since a large amount of data will be available in this study, stratification should not have to be considered.

James et al. (2013, p. 181-184) states that choosing the value of k in k-fold cross-validation is not obvious and is highly influenced by the *bias-variance* trade-off. When k is set low the bias in the error estimate of the model is minimized as the validation set is used fewer times in the training phase. However, this is a poor choice due to the overlap between training folds, which leads to high variance. According to Witten et al. (2011, p. 153), $k = 10$ is often used in practice, as was the case in the research reviewed in Chapter 2.1. Thus, this will also be the case in this study.

A conceptual description of the 10-fold cross-validation method that will be used in this study is described in Figure 4. The entire test data set is first partitioned into ten folds at random. In each iteration, the i-th fold is chosen as the validation fold. The validation fold consists of the usage contexts that are used to query the code completion system. Meanwhile, the $k-1$ other folds form the training folds with usage contexts that are used to populate the model. Each cross-validation iteration produces a prediction quality measurement. The final prediction quality is calculated by taking the average over the k iterations.

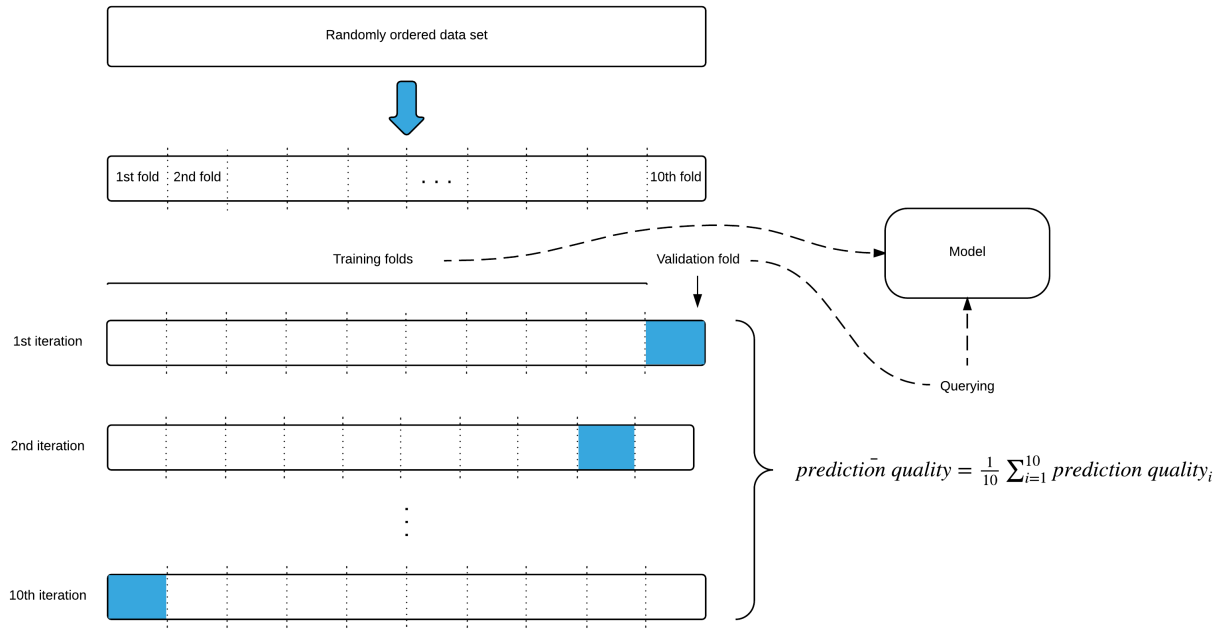

Figure 4: Conceptual diagram of the 10-fold cross-validation applied in this study. Adaptation of a picture from (Raschka).

5.4 Data collection

Data collection methods are structured processes for collecting research data. Since few studies can feasibly gather data covering every aspect of a research field, sampling must be used. This raises a number of practical issues, mainly what sampling technique is used and the size of samples. These choices impact the quality of the collected data, and drawing conclusions from faulty data could present an inaccurate view.

This study will be focused on quantitative analysis. In the absence of existing data sets, experiments will be performed to gather the necessary data. At first, the code repositories presented in Chapter 5.2 will be analyzed for usage contexts. Due to limitations of space and time constraints, only contexts of public method invocations will be included. Furthermore, as described in Chapter 5.2, only classes found in the *System* namespace and all sub-namespaces of *System* will be included.

Value of the data

Any sampling technique involves a trade-off between breadth and depth of data, a trade-off that will affect the generalizability of the theory and its ability to be applied to specific populations (Frechtling, 2010, p. 52).

In order to provide valuable data for the code completion systems in this study, a sufficiently large and diverse corpus of usage contexts must first be established. One way of increasing the amount of valuable data would be to focus on large source code repositories with well-established coding practices. Chapter 5.2 goes more in-depth as to which repositories will be considered in this study. Collecting usage contexts from multiple repositories should provide a more generalized data set, but there will be other biases to overcome as well.

Due to the specific coding practices inherent to each code base, usage contexts collected from a particular repository could be highly similar, leading to a positive bias for contexts gathered from the same repository. Proksch et al. (2015) observed such a bias in their own research, a bias they reduced by evaluating usage contexts from several repositories against usage contexts not gathered from those repositories. Designing the tests in such a way might introduce a negative bias instead, so this study will not focus on this part of the test design. Any bias, be it positive or negative, will be present in all prediction quality measurements to an equal extent.

5.5 Data analysis

The data analysis method used in this study will be *statistical*. A simple approach will be taken to the analysis of prediction quality and inference speed. These variables will be analyzed by taking the arithmetic mean over several iterations. Since the prediction quality will be evaluated by 10-fold cross-validation, repeated for ten different data sets, the usual pitfall of the arithmetic mean when dealing with outliers in the data should not be an issue as the variance will be very smooth with the chosen evaluation method. Furthermore, the F-measure will be used to analyze the accuracy of the tests. For the inference speed, the arithmetic mean will be taken over three separate runs, each based on 3,000 measurements. However, the variance within each run will not be addressed statistically. The model size variable will be measured in absolute numbers and will not be subject to statistical analysis.

5.6 Quality assurance

Internal validity

- *Prediction quality*: The variables affecting prediction quality are the optimized parameters. Since several parameters will be optimized co-dependently, each variable's effect will be difficult to distinguish and ascertain. However, the genetic algorithm's effect will be measurable.
- *Model size*: There is only one variable affecting the model size; whether or not duplicate usage contexts are allowed in the model. Any differences in model size will be possible to attribute to this variable.
- *Inference speed*: As discussed in Chapter 5.1, efforts will be taken to reduce any outside interference in the timing measurements. The only variable will be model size.

External validity

In statistics and machine learning, the ability of a supervised learning algorithm to generalize beyond a training set is limited by the *bias-variance* trade-off. Bias and variance can be decomposed into two sources of error, and the problem consists of minimizing these (Giovanni and Elder, 2010, p. 22-25).

- Bias error is the result of faulty assumptions in the learning algorithm. As a result, the relevant relationships between features and expected output can be lost, a problem known as *under-fitting*.
- Variance error is the result of sensitivity to small fluctuations in a training set, causing the model to reflect random noise rather than the intended output. This is also known as *over-fitting*.

One way of reducing bias is by making the model more complex. However, a more complex model would increase the variance, so an evaluation method that can control the variance is necessary. k-fold cross-validation will be applied to address these issues.

As discussed in Chapter 5.4, the possibility of intra-project bias is recognized but will not be addressed in this study.

Reliability

Reliability will be improved by using k-fold cross-validation, given a sufficiently large sample of unordered data, which will be randomly divided to reduce bias. k-fold cross-validation

increases the probability that the data is representative of the data-generating process, a process which in this case is influenced by potentially biased developers writing code. A higher number of k will also used, which will smooth the variance and therefore reduce the noise in the results.

Replicability

Besides the measures taken to ensure reliability, all tests will be performed multiple times. The entire process will be repeated, meaning that the data will be collected in the same way, that the same data analysis will be performed, and that the same conclusions will be arrived at.

Ethics

Cognitive bias is difficult to identify from inside a study, yet it presents a real risk in every step from presenting the hypothesis to drawing the conclusion. In an effort to reduce such bias in this study, the data collection and analysis plan will be presented in its entirety before any results are gathered, and even before an actual implementation of the proposed algorithm exists.

Transparency is key in validating the research by allowing the reproduction of results. Since the code repositories that are used in producing the results are likely to evolve over time, a static snapshot of these repositories will be provided, along with the source code produced in this work. This allows for other researchers to compare results between different versions of code repositories. Researchers will also have an exact reference implementation to compare against.

5.7 Experimental setup

Experiments will be run on a system with the following specification:

- Operating system: Windows 10 Pro 64-bit (Version 1511, OS Build 10586.318)
- Processor: Intel(R) Core(TM) i7-3630QM CPU @ 2.4 GHz
- Memory: 8 GB RAM (SO-DIMM DDR3 1600 MHz)

All experiments will also run within Visual Studio Enterprise 2015. Due to the large number of usage contexts, the tests need to run with *gcAllowVeryLargeObjects* enabled and specifically target a 64-bit runtime environment (Microsoft, a).

6 Implementation of GeneCSCC

This chapter outlines the implementation of *GeneCSCC* (pronounced *genesis*), the enhanced system based on CSCC. Chapter 6.1 explains how source code repositories are analyzed for code examples. Chapter 6.2 describes how the model is represented and built. Chapter 6.3 details the process of how code completion calculates the predictions. Chapter 6.4 describes the optimization process resulting in GeneCSCC. Finally, Chapter 6.5 specifies how the final set of proposed recommendations is determined.

6.1 Static analysis of source code repositories

In the initial step, existing code repositories are mined for contexts in which method calls of the targeted API appear. The data mining process is all handled by Roslyn. First, each source code file is compiled after which the generated syntax tree is traversed. A context is specifically defined as the four lines prior to a method call, hereafter referred to as the *extended context*, as well as the line where the method call appears, hereafter referred to as the *local context*. Only lines within the surrounding method declaration are of interest. There are three kinds of information collected: (1) The fully qualified name of any methods (2) Any C# keywords except access specifiers, which are not valid within method declarations (3) In assignment declarations, the type of the assigned variable is included.

Every token that is not a grammatical construct is ignored when considering the extended context. This includes blank lines, comment lines and lines containing only curly braces. For the local context, only the context before the target method call is considered. In both contexts, multi-line method calls are treated as one line.

In order to further illustrate what extended and local context include, consider the code in Figure 5. Any code outside of the method is excluded since it is ignored.

```
1 public void MinedMethod()
2 {
3     var now = DateTime.Now;
4     now.ToString(CultureInfo.InvariantCulture);
5
6     var day = now.ToLocalTime().Day;
7 }
```

Figure 5: C# code example mined for context.

The context information that is gathered for each relevant line is as follows:

- Line 3:
  Extended context - Empty, this is the first line in the method declaration
  Local context - *System.DateTime*, from the assignment to the variable *now*
  Invocation - *System.DateTime.Now*
- Line 4 (1):
  Extended context - *System.DateTime* and *System.DateTime.Now* from Line 3
  Local context - Empty
  Invocation - *System.DateTime.ToString*(*System.IFormatProvider*)
- Line 4 (2):
  Extended context - Same as Line 4 (1)
  Local context - *System.DateTime.ToString*(*System.IFormatProvider*)
  Invocation - *System.Globalization.CultureInfo*
- Line 6 (1):
  Extended context - *System.DateTime* and *System.DateTime.Now* from Line 3
  System.DateTime.ToString(*System.IFormatProvider*) and *System.Globalization.CultureInfo* from Line 4
  Local context - *int*, from the assignment to the variable *day*
  Invocation - *System.DateTime.ToLocalTime*()
- Line 6 (2):
  Extended context - Same as Line 6 (1)
  Local context - Same as Line 6 (1), with the addition of *System.DateTime.ToLocalTime*() from the previous invocation
  Invocation - *System.DateTime.Day*

6.2 Model for context-sensitive code completion

The model is represented as an inverted index structure, where the type of the invocation call site serves as the index to a collection of usage contexts. Looking back at Figure 5 and the succeeding explanation of how the contexts are extracted, the inverted index would look as follows:

- System.DateTime: Line 3, Line 4 (1), Line 6 (1), Line 6 (2)
- System.Globalization: Line 4 (2)

The CSCC system maintains every usage context in the model. Since the current implementation of CSCC does not use the frequency of a specific context for any purpose, any duplicates are filtered by GeneCSCC.

6.3 Finding candidate code completions

Code completion is invoked either when typing the *new* keyword and calling a constructor, or when typing a dot (.) after a declared variable or the type name if the invocation is static. The algorithm then extracts the extended and local contexts for the invocation. From this information, a list of candidate completions are extracted from previous usage contexts by matching them against the current context. Specifically, the type name of the receiver object (either a declared variable, a variable in an assignment or the previous invocation as part of method chaining) is used as an index into the model to retrieve the relevant usage contexts. This becomes the *base candidate list*.

As the base candidate list could potentially consist of thousands of contexts, there is a need to reduce these to the most likely candidates. This process consists of two steps. Step one uses the *simhash* technique to reduce the list to those that are determined to be most similar. Simhashes are described more in-depth in Appendix B, but in essence a simhash is a compressed binary representation of a data set. Each token in a context is first concatenated and a simhash is generated. The base candidate list is then sorted by descending order of simhash similarity by comparing each context in the base candidate list with the current context. Comparison is done in constant time by calculating the Hamming distance between the contexts' simhash values. The simhash of the extended context is used in the similarity comparison, unless the similarity of the local contexts exceed a predefined threshold value. In that case, the local context takes precedence. CSCC sets the threshold to 1, whereas GeneCSCC optimized this parameter as described in Chapter 6.4, getting a value of 0.77. From the sorted list, the k closest candidates are chosen. CSCC sets k to 200, whereas GeneCSCC optimized this parameter as described in Chapter 6.4, getting a value of 326. The list of k closest candidates will be referred to as the *refined candidate list*.

From the refined candidate list, the most likely candidates will be presented to the user. First, the list is sorted in descending order by taking the normalized Longest Common Subsequence distance as a similarity measure between the token sequence of the extended contexts (see Equation 4).

$$\text{normalized Longest Common Subsequence} = 2 \cdot \frac{LCS(A,B))}{|A| + |B|} \tag{4}$$

In the case of ties, the Levenshtein distance between the token sequences of the local contexts is normalized and used as a similarity measure (see Equation 5). Levenshtein distance denotes the number of deletions, insertions and substitutions required to transform one sequence into another. Compared with the Longest Common Subsequence, the Levenshtein distance is more fine-grained.

$$\text{Levenshtein similarity} = 1 - \frac{\text{Levenshtein distance}(A,B))}{Max(|A|,|B|)} \tag{5}$$

Finally, the sorted list is reduced by removing those candidate solutions that drop below a predefined threshold value. CSCC sets the threshold to 0.3, whereas GeneCSCC optimized this parameter as described in Chapter 6.4, getting a value of 0.65.

The simhash values are generated from the hash function that is defined for C# strings. It is possible that there are more efficient or better suited hash functions with better distribution, but using the default implementation removes the need to separately implement and test other hash functions. Due to the hash function only generating 32-bit values, all simhash values will be 32 bits. CSCS on the other hand uses the *Jenkin* hash function and 64-bit hash values, generating larger simhash values possibly representing the data more accurately. However, no increase in prediction quality was observed when comparing the two options.

6.4 Training with the genetic algorithm

In the research on context-sensitive code completion by Asaduzzaman et al. (2014), multiple variables were chosen on the basis of empirical tests. These variables are therefore subject to optimization. The values that were chosen for the variables are:

- Lower threshold for simhash similarity based on extended context or local context:

$$similarity = \begin{cases} similarity(local\ context), & \text{if } \frac{similarity(local\ context)}{similarity(extended\ context)} > \\ & threshold \\ similarity(extended\ context), & \text{otherwise} \end{cases}$$

- Maximum size of refined candidate list: 200
- Lower threshold for filtering of the refined candidate list based on normalized Longest Common Subsequence and Levenshtein distance: 0.3

Instead of choosing these empirically, a genetic algorithm is used in this work to find an approximate best combination. The chromosome fitness is determined by 10-fold cross-

validation. Each round uses the 9 training folds to construct a model of usage contexts. The validation fold is then used to query the model. Equation 6 describes the relationship between fitness and prediction quality. Each validation round has a prediction quality measure, the F_1-measure. The fitness is taken as the average of the 10 rounds.

$$fitness = \frac{1}{k}\sum_{i=1}^{k} \text{prediction quality}_i, \text{ where } k = 10 \tag{6}$$

Cross-validation is used here to reduce over-fitting, which in this case would mean that the chosen parameters would represent the model too closely. Initial tests with this scheme generated a new partition of ten folds for each chromosome. Although cross-validation produces generalizable results, generating new folds did affect the ability to reliably compare chromosomes within a generation. The tests where chromosomes within each generation instead used the same folds were comparable, however the evaluation was far too slow. An attempt to reduce the number of generations and increase population size was made, but due to the low number of generations the solutions showed poor convergence. As a result of these tests, the same 10-fold partition is used in each generation and for each chromosome.

An additional step is taken to ensure that the chosen parameters form a generalizable solution. First, the 100 classes with the most usage contexts are chosen from the entire collection of mined contexts. The entries are then partitioned into 10 equally large partitions, with ten classes in each partition. The genetic algorithm described in Figure 6 runs for each of the ten partitions separately. When the genetic algorithm has terminated for each partition, the fittest solution for each partition is extracted. Each solution is then evaluated against each of the ten partitions and the solution with the best average prediction quality is taken as the optimal solution. The reason behind this process is again to reduce over-fitting. It allows for using large amounts of data, which normally risks leading to over-fitting, by breaking the data down into smaller partitions and taking an average instead.

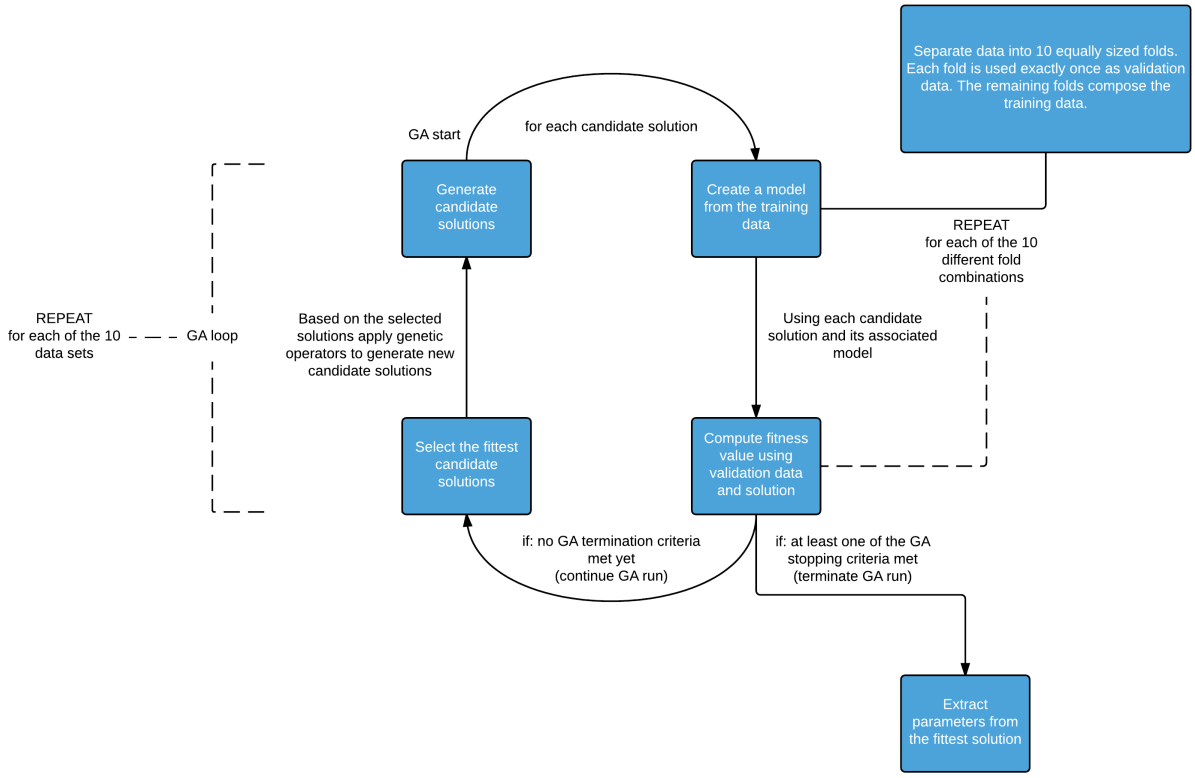

Figure 6: Conceptual work flow for determining optimal parameters.

6.5 Proposing a list of recommendations

In this final step, the list of completion proposals is presented to the user. Since the refined candidate list could potentially contain duplicates these are first removed. After this, the three most likely proposals are presented while the rest are given in alphabetical order. It is however possible to configure how many predictions should be presented.

7 Results

In this chapter, the findings of the study are presented in the following order: (1) prediction quality, (2) model size and (3) inference speed.

7.1 Prediction quality

Figures 7 and 8 summarizes the results of the evaluation as described in Chapter 5.3. The figures shows the precision, recall and F_1 measures as described in Chapter 5.1. Both the CSCC and GeneCSCC systems use optimized parameters with respect to the F_1-measure. CSCC was optimized empirically by Asaduzzaman et al. (2014). GeneCSCC has been optimized with a genetic algorithm as part of this work, as described in Chapter 6.4.

Figure 7 shows the performance of the top prediction returned by the code completion systems. The precision measure shows that the prediction is correct in 28 percent of the cases for CSCC. For GeneCSCC, the precision has increased to 36 percent. Both the CSCC and GeneCSCC systems manage to make a prediction in all cases, as indicated by the recall measure. The F_1-measure shows that the optimization with a genetic algorithm has increased the prediction quality by 9 percentage points.

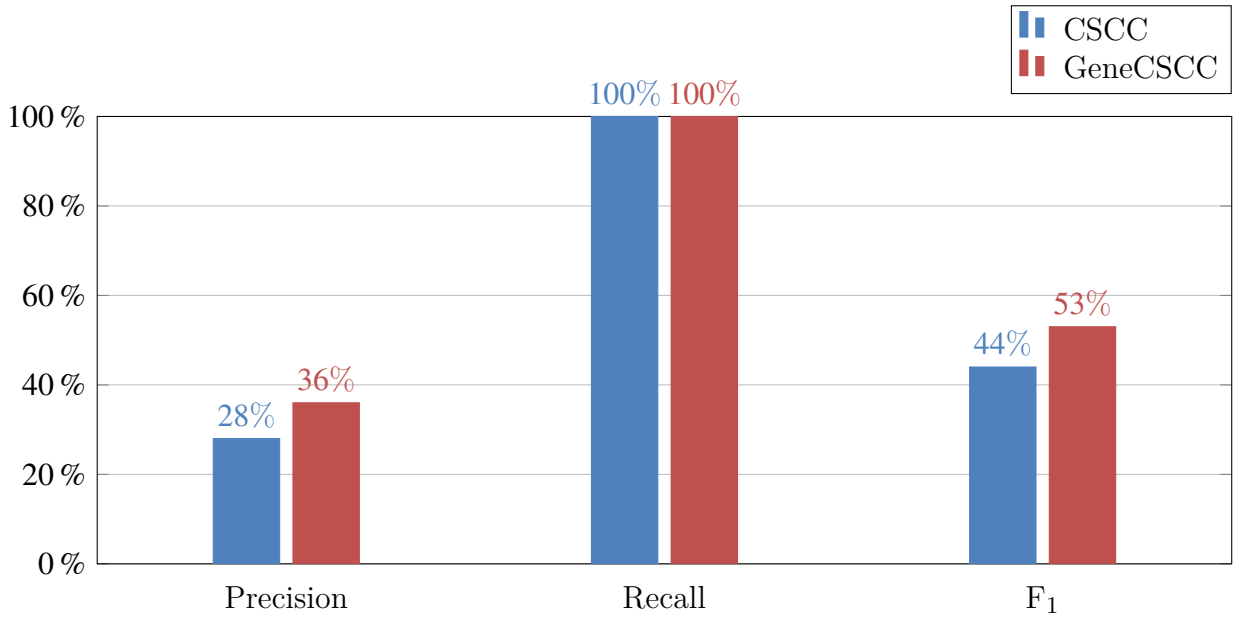

Figure 7: Measured accuracy of the top prediction returned by the systems.

Figure 8 shows the performance of the top three predictions returned by the code completion system. The precision measure shows that one of the three predictions is correct in half of the cases for CSCC. For GeneCSCC, the precision has increased to 57 percent. Both the CSCC and GeneCSCC systems are expected to return three predictions for each code completion query. As indicated by the recall measure, the GeneCSCC system filters predictions somewhat more aggressively, as it returns 1 percentage point fewer predictions than the CSCC system. The F_1-measure shows that the optimization with a genetic algorithm has increased the prediction quality by 6 percentage points.

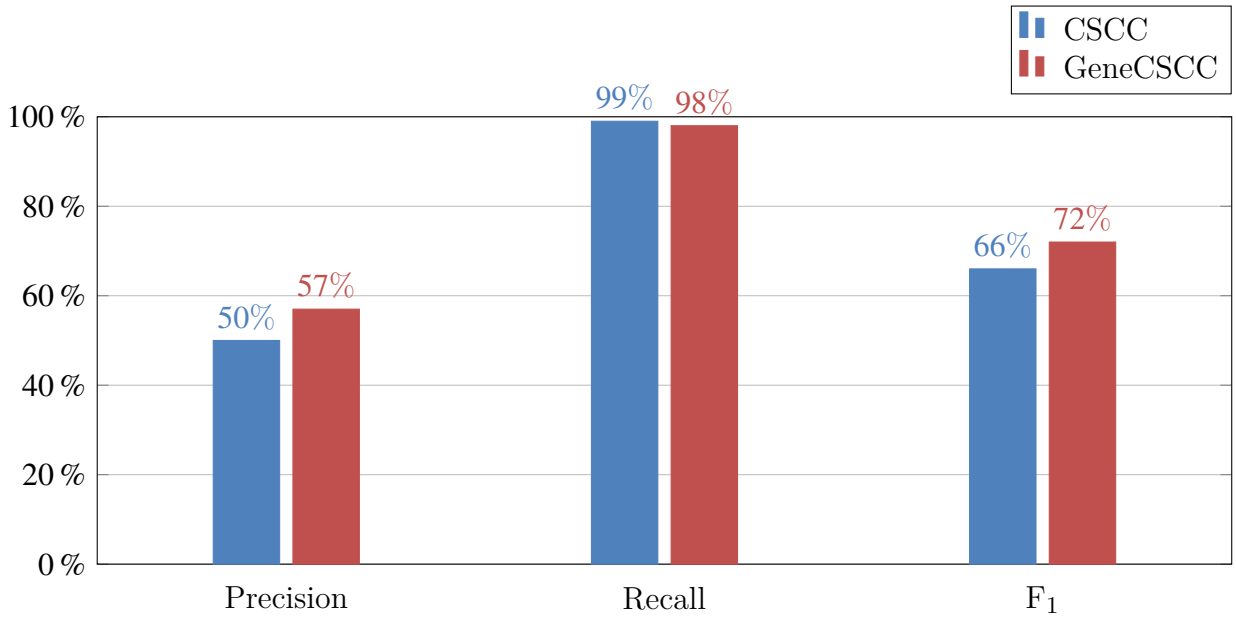

Figure 8: Measured accuracy of the top three predictions returned by the systems.

Comparing the results of Figure 7 and 8 shows some discrepancies with the results found by Asaduzzaman et al. (2014) in their evaluation of the CSCC system. In fact, the CSCC system performs about 30 percentage points worse with respect to the precision in this study.

7.2 Model size

Figure 9 shows the number of usage contexts gathered per source code repository, as described in Chapter 6.1. The difference between the CSCC and the GeneCSCC systems is that all duplicate contexts are removed in the GeneCSCC model. As such, the blue bar represents the entire corpus of usage contexts, whereas the red bar shows the same corpus with all duplicate usage contexts removed. It then becomes apparent that there is between a third to a half of all contexts that are duplicates of another context for all libraries. This is consistent with the findings of Hindle et al. (2012) that software is repeated and predictable, something that both CSCC and GeneCSCC exploit to make predictions for code completion.

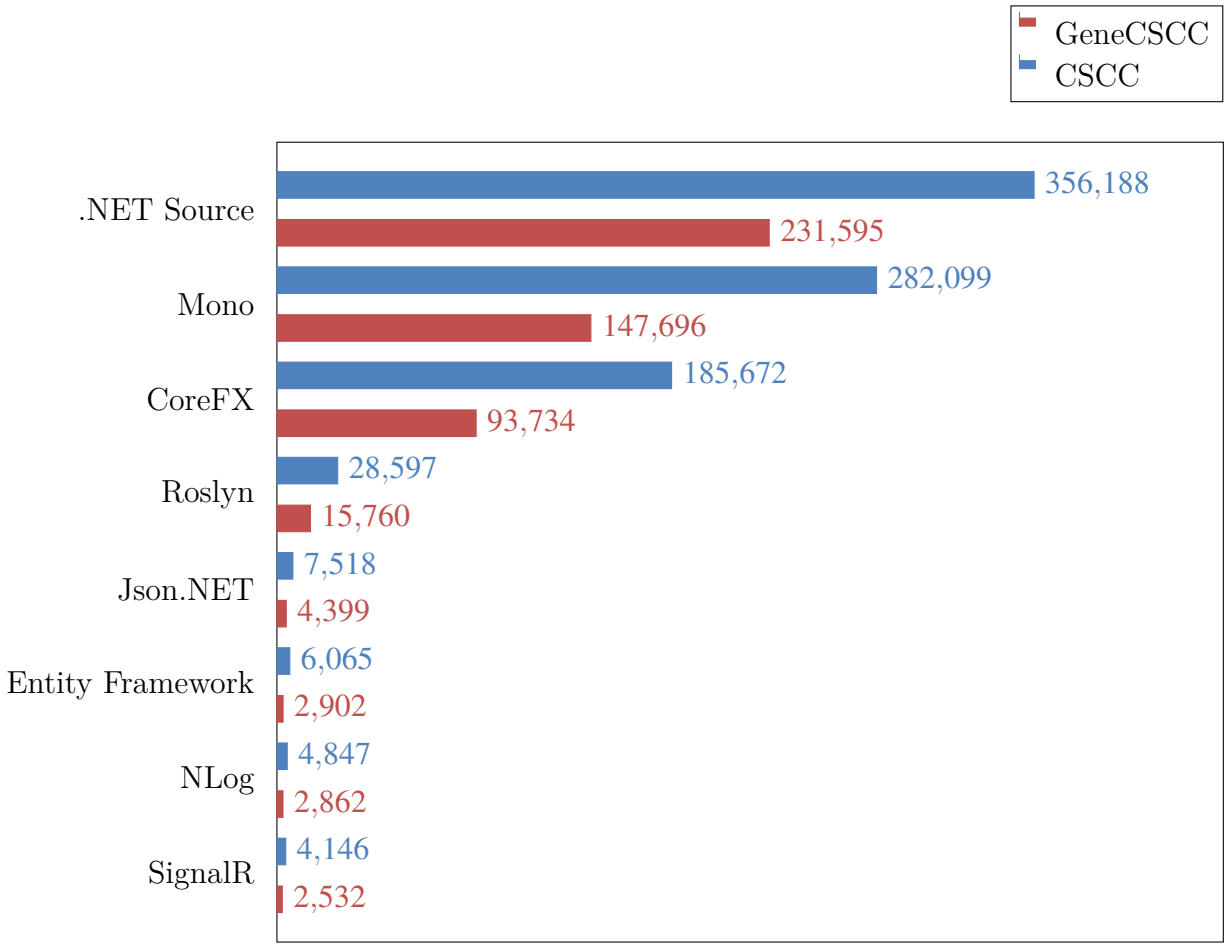

Figure 9: Number of usage contexts in the model generated from each code repository.

Figure 10 shows the in-memory size of the models generated from each library, as described in Chapter 5.1. All measurements are given in megabytes. Despite the large number of duplicate usage contexts that are eliminated when generating the model for GeneCSCC, there is only a very slight reduction in memory footprint. The most likely explanation for this is that the eliminated contexts contain few tokens, with a higher likelihood of these simpler contexts having duplicates.

Another observation is the very large model sizes. The average size for a context from the .NET Source is roughly 2,000 bytes. For the most frequently used types, the number of contexts exceed 10,000. Keeping the usage contexts of only one of these types then requires 20 megabytes of memory. While very few types are associated with over 10,000 contexts, the memory issue is still a valid concern.

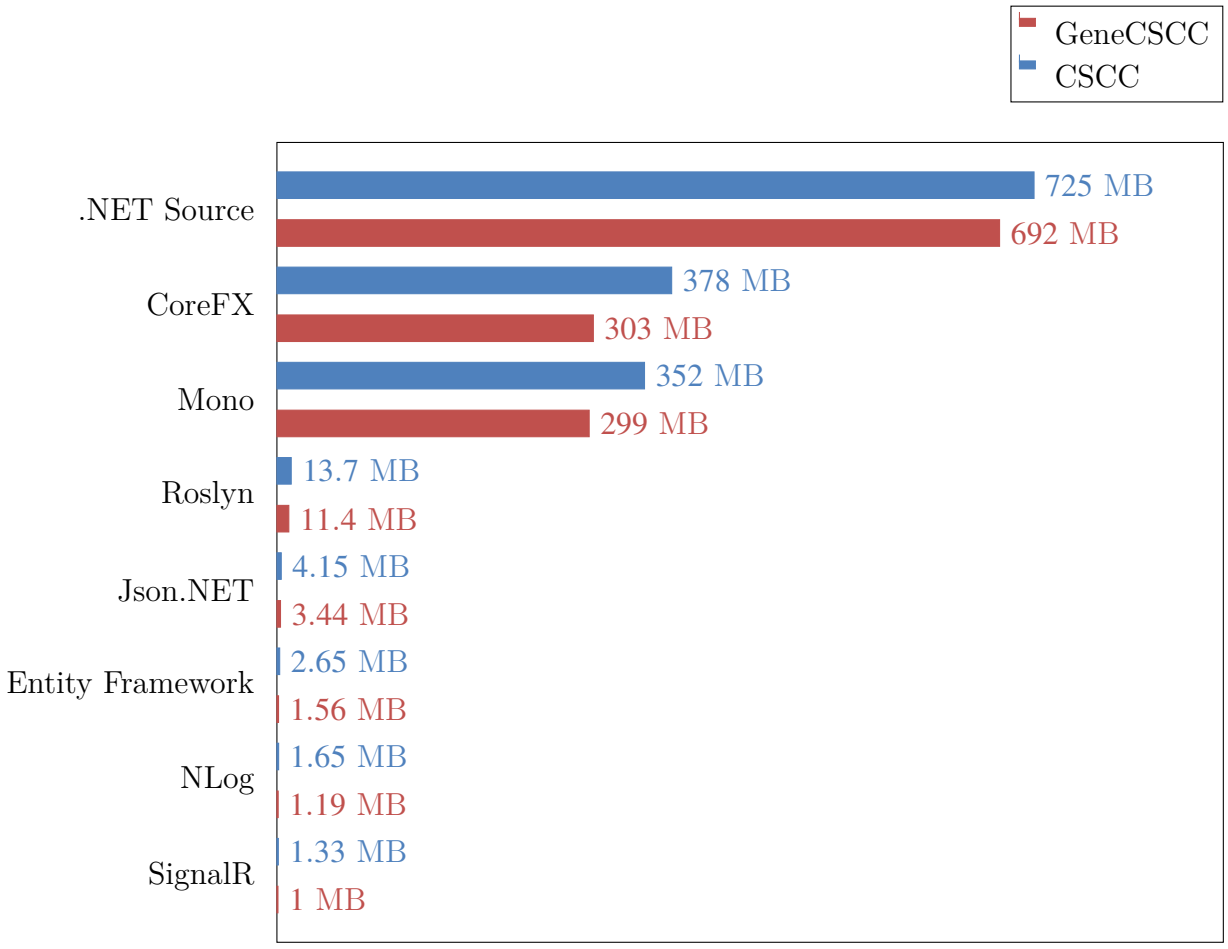

Figure 10: Total size of the model generated from each code repository.

7.3 Inference speed

The inference speed of both CSCC and GeneCSCC was evaluated as described in Chapter 5.1. A somewhat unexpected outcome was found, as the inference speed was found to be below 1 ms per query even for model sizes up to 500,000 usage contexts. Nielsen (1994) states that any code completion system should respond to queries within 100 ms to avoid breaking the developer's workflow. Since model sizes of 500,000 usage contexts present limitations such as reaching the limit of the physical address space, which introduces paging noise, it does not make sense to evaluate the scalability of the systems further. The only possible conclusions to be made are that the code completion systems are able to adequately respond to queries for any reasonable model size, and that the optimized parameters of GeneCSCC does not reduce performance.

8 Discussion

The purpose of this study was to answer the question: to what extent is it possible to improve predictions of existing state-of-the-art code completion systems with a genetic algorithm? It has been shown that the input parameters to the code completion system CSCC can be optimized to provide more accurate predictions for previously unseen patterns.

- *Prediction quality*: The results in Chapter 7.1 indicate that 1 out of 10 additional previously unseen patterns can be correctly recognized with the optimized parameters. Comparing the parameters of the CSCC and GeneCSCC systems, defined in Chapter 6.3, shows that the improvement is a result of putting more importance on the extended context. The comparison also shows that the predictions in the refined candidate list can be filtered heavily without reducing prediction quality, but this finding has no practical use as of now.
- *Model size*: The results in Chapter 7.2 shows that while the number of usage contexts can be reduced with between a third to a half for the GeneCSCC system compared with the CSCC system, this does not translate to a significant reduction in memory footprint of the model. This discrepancy is due to contexts with complex usage patterns, and consequently larger memory footprint, having few duplicate occurrences.
- *Inference speed*: Chapter 7.3 shows that even though the results of evaluating the inference speed were unanticipated, it can be shown that the optimization of CSCC with a genetic algorithm does not reduce the usability of the code completion system since the inference speed remains the same between CSCC and GeneCSCC.

Differences in results from previous research

It is not exactly clear what causes the discrepancy in prediction quality identified in Chapter 7.1, but it is likely a combination of three factors:

- 1. The larger breadth of classes targeted in this study.
- 2. The broader use of these classes which makes it difficult for the code completion system to recognize the correct pattern.
- 3. The fact that this study only evaluates against previously unseen usage contexts.

Limitations and weaknesses of the study

This study is limited to evaluating the code completion systems against classes in the .NET Framework. Whether the usage of these types is similarly complex compared with the usage of other frameworks is not known. It might be shown that the results do not generalize at all, or that similar results can be achieved by modifying the parameters of GeneCSCC.

The usage contexts in this study are combined from several source code repositories. There are large discrepancies in the number of usage contexts contributed from each repository, which could lead to a positive bias towards the contexts of a particular repository. The impact of this bias is unknown, however, both the CSCC and GeneCSCC systems are evaluated against the same data. Any bias should be present in both systems.

It is also unknown whether combining usage contexts from multiple source code repositories impacts the prediction quality. Future studies might show that combining the contexts introduces noise that makes it more difficult for the code completion system to find the best predictions. However, a predictive system that exactly models a certain environment might not have many practical applications.

Implications of the findings

The main finding of this study is that the prediction quality of a code completion system can be increased by enhancing the system with a genetic algorithm. It is concluded that careful consideration of the context information is one important aspect. Previous research has consisted of finding different approaches to represent the context information. Asaduzzaman et al. (2014) and Proksch et al. (2015) make a concerted effort to analyze the impact of adding additional tokens to the context. However, no consideration is taken on how to optimally use the context information when it is defined. This thesis contributes to the existing literature by studying this particular optimization problem.

9 Conclusions and Future work

In this chapter, the major contributions of the work presented in this thesis are put in a larger context. The second part of this chapter discusses some directions for future work to extend the work presented in this thesis.

9.1 Conclusions

This study set out to explore how predictive models in code completion systems are trained and has identified where the noise in the considered context information comes from. In particular, the disadvantage of predictive systems in recognizing unseen patterns is reduced. Existing literature is mainly focused on how to represent the context information and does not consider the impact of the model training on the prediction quality.

As a result of the new training scheme, the quality of predictions can be increased without losing generalizability. Application of the new training scheme could possibly be applied to any code completion systems that trains a predictive model, making it a candidate for improving existing systems as well as in future research.

9.2 Future work

The work presented in this thesis is an enhancement of an existing system by Asaduzzaman et al. (2014). As such, it assumes that the choice of context information and lines of context is valid and instead focuses on optimizing the system parameters. Additional study should be done, however, to examine how prediction quality changes based on the amount of context information for the optimized parameters. Furthermore, the degree of freedom could be increased, allowing both the system parameters and the amount of context information to change independently during training with the genetic algorithm.

The code completion system introduced in this work does not use clustering, such as PBN which is based on Bayesian networks. Future work could first compare the two code completion systems as they are, and then implement clustering for this work to see how prediction quality, model size and inference speed is impacted. More sophisticated machine learning techniques could also be used, ones that remove outliers from the data set to reduce noise in the model.

Processing the input data before training the code completion system could be a consideration in future work. The optimization algorithm could analyze the impact of particular training data on prediction quality, model size and inference speed. Input data could then be filtered

based on its impact. Another idea is to analyze each context and remove the code tokens that are not relevant for that particular usage.

As this work is simply a research prototype, there is still work left in integrating it into the Visual Studio IDE. Since the code completion system should ultimately be used by developers, a study should be conducted with developers to examine the viability of the code completion system compared with existing solutions. The study could potentially identify more nuanced areas of improvement that the prediction quality measurement can not find.

For a different approach to code completion, it is suggested that collaborative data gathering could be used. Specifically, a code completion system that gathers usage information from all developers working on a project. The usage information would then reflect the coding standards established within the project. This code completion system could get feedback from developers when they either choose a prediction because it was correct or when they reject it because it was incorrect.

References

AForge.NET. Genetic algorithms library. http://www.aforgenet.com/framework/features/genetic_algorithms.html. Accessed: 2016-04-14 16:52:15.

S. Aggarwal, R. M. Garg, and D. P. Goswami. A review paper on different encoding schemes used in genetic algorithms. International Journal of Advanced Computer Science and Applications, 4(1):596–600, 2014.

S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of visual studio usage in practice. In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16), 2016.

M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. Cscc: Simple, efficient, context sensitive code completion. In International Conference on Software Maintenance and Evolution (ICSME), pages 71–80, 2014.

S. M. Beitzel. On understanding and classifying web queries. PhD thesis, Illinois Institute of Technology, May 2006.

T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report, 11(2), 1995.

A. Z. Broder. Identifying and filtering near-duplicate documents. In Proc. FUN, 1998.

M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In Proc. FSE, pages 213–222, 2009.

T. Bäck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In Proc. 1st IEEE Conf. on Evolutionary Computation, pages 57–62, 1994.

M. S. Charikar. Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing (STOC '02), pages 380–388, 2002.

A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In WWW, May 2007.

J. Frechtling. In The 2010 User-Friendly Handbook for Project Evaluation. National Science Foundation, 2010. .

D. Giacomelli. Geneticsharp. https://github.com/giacomelli/GeneticSharp. Accessed: 2016-04-14 18:59:06.

S. Giovanni and J. F. Elder. In Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions. Morgan & Claypool, 2010. .

GitHub. Mit license. http://choosealicense.com/licenses/mit/, a. Accessed: 2016-04-29 13:14:39.

GitHub. Licenses. http://choosealicense.com/licenses/, b. Accessed: 2016-04-29 13:08:45.

J. Gorohovsky. Resharper and roslyn: Q&a. https://blog.jetbrains.com/dotnet/2014/04/10/resharper-and-roslyn-qa/, 2014. Accessed: 2016-04-22 18:20:43.

L. M. Hilty and M. D. Hercheui. Ict and sustainable development. In What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience. Springer, 2010. .

A. Hindle, E. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. In Proc. ICSE, pages 837–847, 2012.

G. James, D. Witten, T. Hastie, and R. Tibshirani. In An Introduction to Statistical Learning with Applications in R. Springer, 1st edition, 2013. .

Microsoft. <gcallowverylargeobjects> element. https://msdn.microsoft.com/en-us/library/hh285054%28v=vs.110%29.aspx, a. Accessed: 2016-05-23 17:16:35.

Microsoft. Reference source. http://referencesource.microsoft.com/, b. Accessed: 2016-05-10 14:57:45.

Microsoft. Stopwatch class (system.diagnostics). https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch%28v=vs.110%29.aspx, c. Accessed: 2016-05-23 13:53:47.

M. Miller. Is there a roslyn-based coderush in your future? https://community.devexpress.com/blogs/markmiller/archive/2014/04/16/is-there-a-roslyn-based-coderush-in-your-future.aspx, 2014. Accessed: 2016-04-22 18:23:39.

M. Mitchell. In An Introduction To Genetic Algorithms. MIT Press, 5th edition, 1998. .

G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? IEEE Software, 23(4):76 – 83, 2006.

J. Newcombe. The genetic algorithm framework for .net. http://johnnewcombe.net/gaf. Accessed: 2016-04-14 16:22:41.

I. Newman and C. R. Benz. Qualitative-quantitative research: A false dichotomy. In Qualitative-quantitative Research Methodology: Exploring the Interactive Continuum. Southern Illinois University Press, 1998. .

J. Nielsen. In Usability Engineering. Academic Press, 1994. .

NRefactory. Future of nrefactory. https://github.com/icsharpcode/NRefactory/issues/394, a. Accessed: 2016-04-18 18:01:58.

NRefactory. Nrefactory. https://github.com/icsharpcode/NRefactory, b. Accessed: 2016-04-18 17:30:12.

S. Proksch, J. Lerch, and M. Mezini. Intelligent code completion with bayesian networks. ACM Transactions on Software Engineering and Methodology, 25(1), 2015.

S. Raschka. Machine learning faq. http://sebastianraschka.com/faq/docs/evaluate-a-model.html. Accessed: 2016-05-23 17:56:14.

C. Recommenders. Jayes | code recommenders. http://www.eclipse.org/recommenders/jayes/. Accessed: 2016-04-09 23:39:27.

C. J. v. Rijsbergen. In Information Retrieval. Butterworths, 2nd edition, 1978. .

Roslyn. Roslyn. https://github.com/dotnet/roslyn, a. Accessed: 2016-04-14 19:42:09.

Roslyn. Roslyn introduction. https://github.com/dotnet/roslyn/wiki/RoslynOverview, b. Accessed: 2016-04-18 13:35:32.

G. A. E.-N. A. Said, A. M. Mahmoud, and E.-S. M. El-Horbaty. A comparative study of meta-heuristic algorithms for solving quadratic assignment problem. International Journal of Advanced Computer Science and Applications, 5(1), 2014.

I. Sommerville. In Software Engineering. Pearson, 9th edition, 2010. .

D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. Information and Software Technology, 43(14):817–831, 2001.

I. H. Witten, F. Eibe, and M. A. Hall. In Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers, 3rd edition, 2011. .

Appendices

A GitHub repository

https://github.com/godtopus/GeneCSCC

B Locality-sensitive hashing

Locality-sensitive hashing is a scheme for reducing the complexity of comparing the similarity of large datasets. The similarity problem can be formulated as in Equation 7:

$$similarity(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{7}$$

where the part $|A \cap B|$ requires $|A|^{|B|}$ equality comparisons. This number is potentially prohibitively large, especially for strings where the complexity further increases with the string length k to $k \cdot |A|^{|B|}$.

Locality-sensitive hashing schemes compresses the datasets via hashing and maps these to different buckets, with a high probability of similar datasets occurring in the same bucket. This is achieved by hashing the datasets in such a way that the probability of collisions is maximized. The scheme is often used in identification and filtering of near-duplicate documents as employed by AltaVista (Broder, 1998), but also has applications in data clustering and nearest neighbor search, such as in Google News (Das et al., 2007).

B.1 Simhash

Simhash is an algorithmic implementation of locality-sensitive hashing, developed by Charikar (2002). It works by reducing each dataset to a *fingerprint* via a hash function. A fingerprint can not be generated by taking the entire dataset through the hash function, as the output would not represent the text very well. This is due to the intrinsic quality of a hash function to reduce collisions and therefore small changes in the dataset would induce large changes to the fingerprint. Instead, simhash generates the fingerprint through the following steps:

- 1. Define a fingerprint size, preferably 32 or 64 bits as these can be represented by primitive types
- 2. Create an array V[ ] of the same size as the fingerprint, filled with zeros
- 3. For each element in the dataset, create a hash using some hash function with fixed-size output
- 4. For each hash, and for each bit i in this hash:
  - If the bit is 0, add 1 to V[i]
  - If the bit is 1, subtract 1 from V[i]
- 5. For each bit j of the fingerprint:

- If V[j] >= 0, set V[j] = 1
- If V[j] < 0, set V[j] = 0

When the bits in the hashes are mapped to the array in step 4, a histogram is formed. This histogram describes the occurrence of a certain feature (a bit being set or not) in all the elements in the data set. Given that a hash function produces the same output for the same input, and widely different outputs for those inputs that differ even slightly, it is probabilistically likely that two data sets are similar when their histogram signatures match.

Since the fingerprint is binary-encoded, the operation $\cap$ in Equation 7 is equivalent with an XOR operation for the fingerprints. With the knowledge that the number of set bits after an XOR operation is equal to the Hamming distance of the binary sequences, the difference can be expressed as in Equation 8:

$$difference(A,B) = \frac{Hamming\ distance(simhash(A),\ simhash(B))}{fingerprint\ size} \tag{8}$$