Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

An Approach to Rank Program Transformations Based on Machine Learning

José Aldo Silva da Costa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Gustavo Araújo Soares, and Rohit Gheyi
(Orientadores)

Campina Grande, Paraíba, Brasil
©José Aldo Silva da Costa, 01/02/2019

C837a Costa, José Aldo Silva da.
An approach to rank program transformations based on machine learning / José Aldo Silva da Costa. – Campina Grande, 2019.
93 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.
"Orientação: Prof. Dr. Gustavo Araújo Soares, Prof. Dr. Rohit Gheyi ".
Referências.

1. Transformações de programas. 2. Ranqueamento. 3. Aprendizado de máquina. I. Soares, Gustavo Araújo. II. Gheyi, Rohit. III. Título.

CDU 004.42(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA MARIA ANTONIA DE SOUSA CRB 15/398

"AN APPROACH TO RANK PROGRAM TRANSFORMATIONS BASED ON MACHINE LEARNING"

JOSÉ ALDO SILVA DA COSTA

DISSERTAÇÃO APROVADA EM 13/02/2019

GUSTAVO ARAÚJO SOARES, Dr., MICROSOFT
Orientador(a)

ROHIT GHEYI, Dr., UFCG
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

LEOPOLDO MOTTA TEIXEIRA, Dr., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

À medida que o software evolui, desenvolvedores realizam edições repetitivas ao adicionarem features e corrigirem bugs. Técnicas de Programação-por-Exemplo (PbE) automatizam edições repetitivas inferindo transformações a partir dos exemplos. No entanto, exemplos são ambíguos e limitados, uma vez que os usuários desejam fornecer um número mínimo deles (de preferência 1). Assim, técnicas de PbE precisam ranquear as transformações inferidas selecionando aquelas que melhor se ajustam ao interesse do usuário. Abordagens comuns de ranqueamento favorecem transformações mais simples ou menores, ou atribuem pesos às suas características específicas, ou *features*. No entanto, o peso ideal de cada *feature* varia de acordo com o domínio do problema e encontrar esses pesos requer esforço manual e conhecimento específico. Propomos uma abordagem baseada em Aprendizado de Máquina (ML) para reduzir o esforço manual em encontrar pesos para funções de ranqueamento eficientes, que ranqueiam a transformação desejada usando o mínimo de exemplos. Nossa abordagem compreende a) banco de dados de treinamento/teste, b) extração de características, c) treino e teste de modelo, e d) instanciação das funções. Também investigamos o efeito de exemplos negativos na eficiência das abordagens de ranqueamento, bem como a acurácia das transformações nas primeiras 10 posições. Comparamos cinco abordagens: a) Máquina de vetores de suporte (SVM), b) Regressão logística (LR), c) Redes neurais (NN), d) Especialista-Humano (HE) e e) Pesos aleatórios (RW). Nós as avaliamos em 28 cenários de cinco projetos em C# do GitHub usando a técnica REFAZER que aprende múltiplas transformações a partir de exemplos. Medimos a eficiência das abordagens contando os exemplos necessários para colocar a transformação correta na primeira posição, usando exemplos negativos para evitar transformações que editam locais desnecessários. Como resultado, o LR apresentou um desempenho similar em relação ao HE, com médias de 1,67 e 1,64, respectivamente. Comparado a RW, LR provê uma diferença estatística, com p-valor < 0.05. Quanto à efetividade, LR é similar a HE com Precisão e NDCG de 0,5 e superior a RW com 0,2. Portanto, a abordagem baseada em ML pode ser tão eficiente quanto HE, enquanto reduz o esforço manual em encontrar pesos para criar funções de ranqueamento dos projetistas de ferramentas PbE.

Abstract

As software evolves, developers perform repetitive edits while adding features and fixing bugs. Programming-by-Example (PbE) techniques automate repetitive edits by inferring transformations from examples. However, examples are ambiguous and limited, since users want to provide a minimum of them (preferably 1). Thus, PbE techniques need to rank the inferred transformations to select the ones that best fit the user intent. Common ranking approaches favor the simplest or the shortest transformations, or they assign weights to their specific characteristics, or *features*. However, the ideal weight of each feature varies according to the problem domain and finding these weights requires manual effort and specific knowledge. We propose a Machine Learning (ML) based approach to reduce the manual effort in finding the weights for efficient ranking functions, which rank the desired transformation using the minimum number of examples. Our approach comprehends a) training/testing database, b) feature extraction, c) model training and testing, and d) ranking instantiation. We also investigate the effect of *negative* examples on the ranking approaches efficiency, as well as the accuracy of the top-10 rank positions. We compare five approaches: a) Support Vector Machine (SVM), b) Logistic Regression (LR), c) Neural Networks (NN), d) Human-Expert (HE), and e) Random Weights (RW). We evaluate them in 28 scenarios of five C# projects from GitHub using REFAZER technique that learns multiple transformations from examples. We measure the approaches' efficiency by counting the examples required to put the correct transformation in the first position, adding negative examples to prevent transformations that edit unneeded locations. As a result, LR presented a similar efficiency compared to HE, with example means of 1.67 and 1.64, respectively. Compared to RW, LR provides a statistical difference, with p-value < 0.05. Concerning the effectiveness, LR is similar to HE with both Precision and NDCG of 0.5 and superior to RW with 0.2. Therefore, the ML-based ranking approach can be as efficient as HE, while reducing the manual effort in finding weights to build ranking functions of PbE tool's designers.

Agradecimentos

A Deus pela força e amparo em momentos difíceis dessa jornada, meu refúgio e minha fortaleza, auxílio sempre presente na adversidade;
Aos meus pais José Vicente da Costa Neto e Maria das Neves Silva da Costa, aos meus irmãos José Júnior Silva da Costa de Leidiane Silva da Costa, e a todos os familiares;
Aos irmãos da igreja que congrego pelo apoio e orações;
Aos meus orientadores, Gustavo Soares e Rohit Gheyi, pelas excelentes contribuições na minha formação como pesquisador e orientações para que esse trabalho se tornasse possível;
A Reudismam Rolim, que também me acompanhou de perto durante o mestrado e contribuiu com orientações valiosas e pela paciência;
Aos professores Tiago Massoni e Leandro Balby pelas sugestões e contribuições no trabalho;
A meus colegas do SPG, *Software Productivity Group*.
Aos professores e funcionários da Coordenação de Pós-graduação em Informática (COPIN) e do Departamento de Sistemas e Computação (DSC);
A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio e suporte financeiro fornecidos a este trabalho.

Contents

1 Introduction **1**

1.1 Problem 2

1.2 Motivating Example 2

1.3 Solution 5

1.4 Evaluation 6

1.5 Conclusions 7

1.6 Summary of contributions 7

1.7 Organization 7

2 Background **8**

2.1 Software Evolution 8

2.1.1 Repetitive Edits 10

2.2 Program Transformation 10

2.3 Program-by-Example 11

2.3.1 Ranking Problem in Program Transformations 15

2.3.2 Domain-Specific Language 16

2.3.3 Ranking Functions 17

2.3.4 Human-Expert Ranking Approach 20

2.4 Machine Learning 21

2.4.1 Supervised Learning 23

2.4.2 Learning Algorithms 28

2.5 Learning Ranking Function for Ranking Problems 33

3 State of the art **35**
3.1 Motivation . . . 35
3.1.1 Research Questions . . . 36
3.2 Methodology . . . 37
3.2.1 Search *String* . . . 38
3.2.2 Databases . . . 38
3.2.3 Study Selection . . . 39
3.2.4 Information Extraction . . . 40
3.2.5 Execution . . . 40
3.3 Results . . . 42
3.3.1 Selected Studies . . . 43
3.4 Discussion . . . 46
3.4.1 Programming by Example approaches . . . 46
3.4.2 Study limitations . . . 48
3.5 Conclusion Remarks . . . 49
3.6 Answers to the Research Questions . . . 50

4 A Machine Learning Based Ranking Approach **51**
4.1 Overview . . . 51
4.2 Training/Testing Data Generation . . . 52
4.3 Feature Extraction . . . 54
4.4 Model Training and Testing . . . 55
4.5 Ranking Instantiation . . . 57

5 Evaluation **60**
5.1 Definition . . . 60
5.2 Planning of the experiment . . . 61
5.2.1 Benchmark . . . 61
5.2.2 Instrumentation . . . 62
5.2.3 Training Database Generation . . . 64
5.2.4 Training Models . . . 65
5.2.5 Using Negative Examples . . . 66

5.2.6 Ranking Evaluation 67
5.3 Results 69
5.4 Discussion 71
5.4.1 Characteristics of the domain 71
5.4.2 Characteristics of the edits 73
5.4.3 Efficiency of ranking approaches 74
5.4.4 Generalization versus specialization 75
5.4.5 Effectiveness of ranking approaches 76
5.4.6 Characteristics of ML-based ranking approaches 77
5.5 Threats to Validity 77
5.6 Answers to the Research Questions 79

6 Related Work **80**
6.1 Human-expert based ranking approaches 80
6.2 Machine Learning based ranking approaches 82

7 Conclusions **84**

A Training Models Parameters **93**

List of Figures

1.1 Examples of two edits applied by the developer 3

2.1 Components of the PbE architecture 13
2.2 Example of a DSL for *string* transformation 14
2.3 Data for the e-mail automation problem 14
2.4 A core DSL for describing AST transformations in REFAZER 17
2.5 Binary classification problem 25
2.6 Regression problem 25
2.7 Fitting of function and complexity in a regression problem 26
2.8 Confusion Matrix 27
2.9 Logistic sigmoid function 29
2.10 SVM function 30
2.11 Neural Networks structure 31
2.12 Activation functions 32

3.1 Number of studies for each event. 42
3.2 Number of studies by year. 42
3.3 Number of studies by application domain and ranking approach. 43

4.1 Workflow of the proposed ML-based ranking approach. The inputs consist of a PbE tool, edit examples, and ML algorithms. Step 1 comprehends the generation of training and testing data, Step 2 consists of feature selection and extraction, and Step 3 consists of training and testing models. The output of the ML approach consists of a model, or a function, which is instantiated in a PbE tool in Step 4. 52

4.2 Before and after version of code 53
4.3 A program in the DSL generated consistent with input-output examples . . 56
4.4 Example of feature vector for a program in the DSL 56
4.5 Cross-validation strategy for estimating the performance of the model . . . 57
4.6 Ranking score computation process 58

List of Tables

3.1 Studies for key terms and other spellings 37
3.2 Synonyms and other spellings 38
3.3 Study selection criteria 39
3.4 Number of papers retrieved, selected before snowballing, and selected after performing snowballing 41
3.5 Summarizing the results 45

5.1 Edit scenarios characterization. Project = name of the project; Scenario = edit scenario that represents a specific task; Locations = number of edit locations; Identical = whether the edit is the same for all the location; Granularity = whether one or more lines were changed; Scope = scope of the edit; Role = whether the scenario was used for training/testing the model or for evaluating the distinct ranking approaches 63
5.2 Dataset of labeled transformations. Project = name of the project; Scenario ID = ID of the edit scenario; Learned transformations = number of learned transformations; Correct = number of transformations labeled as correct; Incorrect = number of transformations labeled as incorrect 65
5.3 Performance of models predictions 66
5.4 The "FPG" represents treatments where the highest number of examples available were given but were not enough to rank the correct transformation on the first position. The symbol "-" represents the scenarios where approaches with only positive examples were able to put the correct transformation in the first position 68

5.5 Comparison of the number of examples required by ranking approaches. N/A represents the absence of the value due to memory errors and other types of issues 69
5.6 Comparison of the number of examples required by distinct random approaches. N/A represents the absence of the value for memory errors and types of similar issues 70
5.7 Comparison of the p-values for the distinct approaches 71
5.8 Comparison of the NDCG and Precision metrics for the top-10 positions in ranking approaches 72
5.9 Comparison of the p-values for the distinct approaches 72

Chapter 1

Introduction

During software evolution, developers often perform code edits when adding features and fixing existing bugs. Some of these edits are similar, sharing the same structure but involve different expressions, and repetitive, occurring in multiple locations throughout the entire source code. Searching for each of these locations to be edited and manually applying all necessary edits becomes a tedious, time-consuming, and error-prone task.

Integrated Development Environments (IDEs) and static analyzers are tools that support developers in automating repetitive code edits. However, these tools are based on pre-defined classes of transformations and their scope is limited to a subset of common transformations applied by developers. To extend these classes of transformations, the developer would need a specific set of skills. On the other hand, Programming-by-Example (PbE) techniques appear as an alternative solution to automate repetitive edits [13]. PbE techniques learn new classes of transformations based on input-output examples of edits applied by users using inductive reasoning.

PbE techniques have been used in a wide range of applications, being the most predominant ones the string and program transformation domains [14]. An example of such PbE techniques for program transformation domain is REFAZER [7], which receives examples of edited locations and generalizes those edits into classes of transformations that can be applied to other similar unseen locations.

1.1 Problem

Examples are ambiguous and constitute an incomplete form of specification, which often results in a large set of inferred transformations, reaching an amount of 10^{20} in some cases [44]. However, some of these transformations satisfy the provided examples but are undesired by the developer, requiring more examples to filter them out [44]. Nevertheless, users want to provide only a few examples (preferably 1) [59] and still get the correct transformation (i.e., the one that produces the user intent when applied to unseen similar locations). It happens because providing many examples implies it would be better to perform the edit by themselves, thus the key challenge here is to learn an efficient ranking function, which ranks the correct transformation in the first position with the minimum number of examples. To this end, common ranking approaches favor the simplest or the shortest transformations [30; 11], or they assign weights to specific characteristics, or *features* [7; 26; 16], such as size, simplicity, frequency, and generality [7; 30; 11; 26; 16; 14], using only positive examples. However, the ideal weight of each feature varies according to the problem domain and finding these weights requires manual effort and specific knowledge.

Manually finding the ideal weight of each feature follows a trial and error approach, which is cumbersome because the number of different combinations grows exponentially as features are added. For instance, to build a function with three features, trying integer weights in a range of 1 to 100, the number of distinct combinations reaches up to 100^3, since the ranking function has to capture the relevance of the feature considering its relationship with the others features, making this task even more challenging. Even though this effort can be reduced by using the domain knowledge of an expert, the manual effort of the expert is necessary and a trial and error approach is used to test the ideal weights that can be used to make the functions efficient, ranking with minimum number of examples.

1.2 Motivating Example

We start by presenting a repetitive edit that occurs at the Roslyn repository, the Microsoft's library for compilation and code analysis for C# [35]. Consider the two following edits for a given task in Figure 1.1. This figure presents two examples of code edits to perform

a repetitive task. In this task, the developer intends to replace the `"=="` operator with an invocation of a new method `IsKind` on the left-hand side expression, passing the right-hand side expression as the new method argument. Thus, 26 locations in the source code share the same structure and need to be modified. A learned transformation from these two examples can be used to automatically edit the other 24 locations in the source code.

Figure 1.1: Examples of two edits applied by the developer

```
1 - if (m.CSharpKind() == modifier)
2 + if (m.IsKind(modifier))
3
4 - if (trivia.CSharpKind() == SyntaxKind.None)
5 + if (trivia.IsKind(SyntaxKind.None))
```

To perform the edits intended by the developer, a PbE technique, such as REFAZER, generates a set of program transformations consistent (i.e., produces the output example for the given input example, but not necessarily for unseen inputs) with the given examples. Some of these program transformations that are consistent with the given two examples of edits in Figure 1.1 can be described as follows:

- p_1: replace the "==" operator with an invocation of a new method `IsKind`, passing the right-hand side *expression* as the method's argument, where an *identifier* precedes `CSharpKind`, inside an `if` statement as context.
- p_2: replace the "==" operator with an invocation of a new method `IsKind`, passing the right-hand side *expression* as the method's argument, where other *expression* precedes `CSharpKind`, disregarding the `if` statement as context.
- p_3: replace the "==" operator with an invocation of a new method `IsKind`, passing the right-hand side *expression* as the method's argument, where an *expression* precedes the "==" operator, disregarding the `if` statement as context.

The learned transformations differ in terms of *generalization* and *specialization* capacity. For instance, p_1 can be applied only to locations that share similar structure but an *identifier* (i.e., a name given to an entity) precedes `CSharpKind()`, and the whole operation must be inside an `if` context, which makes it a specialized transformation. A more generalized

transformation, p_2, can be applied to locations where an *expression* (i.e., combination of one or more entities, such as function calls) precedes `CSharpKind()`, disregarding the `if` statement as context. Another still more generalized transformation, p_3 can be applied to locations where an *expression* precedes an equals operator, disregarding the `if` statement as context. Even if the edit examples have *identifiers* preceding `CSharpKind()`, and `CSharpKind()` is present in both examples, REFAZER infers all sorts of possible transformations that include the node itself, the context around it, and the type of node, which may vary from identifier to expression.

Even though all the shown learned transformations have the ability to produce the specified given outputs for the input-examples in Figure 1.1, not all of these transformations can be considered correct ones, producing the desired edits. For instance, consider the unseen location that shares a similar structure of the examples, such as `if (r.Parent.CSharpKind() == SyntaxKind.WhileStatement)`, which the developer intends to edit. The learned transformations p_1 cannot generalize to match this location, since, in this case, an *expression* and not an *identifier* precedes `CSharpKind()`, which makes p_1 an incorrect transformation. On the other hand, p_2 and p_3 can generalize to match this unseen location, but they also match other locations. For instance, consider another unseen location `a.AsNode() == null`. The developer does not intend to edit this location, but it is matched by the transformation p_3, thus editing incorrectly more locations than needed.

An example-based specification can be based on two types of examples, namely *positive* and *negative* examples. Only positive examples may cause overgeneralized transformations, which tend to edit more locations than needed. In this context, negative examples work as counterexamples providing additional information for preventing overgeneralized transformations. For instance, a negative example is a location such as `a.AsNode() == null`, which specifies where the transformation must not edit. Thus, the negative example filters out all the learned transformations that match this location, which includes p_3, since it generalizes `CSharpKind()` to match the undesired location that contains `AsNode()`.

Thus, having in mind that the set of synthesized transformations from an example-based specification includes correct and incorrect transformations, the challenge of the PbE tool's designers consists of building a ranking function that is able to find the correct transforma-

tion, p_2, and move it to the first position while penalizing incorrect transformations, p_1 and p_3, so that they go down in the rank. Intuitively, we realize that some transformations characteristics are more desirable than others and human-experts use their domain knowledge to favor transformations with these more desirable characteristics to best rank the correct one. For instance, too specific or too general transformations tend to produce undesired edits [7]. The human-expert has to find the weights to favor transformations that are not too specific, but not too general. For instance, consider a simple linear ranking function such as *f(·)* = w_1 × *identifier* + w_2 × *expression* + w_3 × *context*, where *identifier*, *expression* and *context* are characteristics of the transformations. Assigning a high weight to w_1 and w_3 while penalizing w_2 will favor p_1, so that it goes to the top. However, assigning a high weight to w_2 and penalizing w_1 and w_3 will favor p_3. The challenge is to find a balance between specialization and generalization so that p_2 goes to the top.

However, finding the ideal weights to build a ranking function that ranks the correct transformation in the first position is cumbersome. The number of characteristics is usually more numerous than only three such as used in our examples. In addition, the function can be more complex, such as a polynomial one. It requires the human-expert to test a large set of distinct combinations in order to evaluate how useful is the resulting function. The number of combinations grows exponentially as more features are used, which requires a lot of manual effort, is tedious and time consuming to test all possible values assigned to the weights.

1.3 Solution

In this work, we propose a Machine Learning (ML) based approach to automatically build ranking functions. Our solution proposes to reduce the manual effort involved in building efficient ranking functions by assigning to the transformations characteristics automatically computed weights. Our key idea is that efficient ranking functions can rank correct transformations in the first position with the minimum number of examples provided.

Our approach comprehends a) training/testing database generation, b) feature extraction, c) model training and testing, and d) ranking instantiation. In Step 1, we take a set of learned program transformations as the input, and we label them according to their correctness. In

Step 2, we select and extract a set of features to describe correct and incorrect transformation. In Step 3, we build feature vectors, train and test ML models. In Step 4, we instantiate the model weights in a PbE tool ranking system.

1.4 Evaluation

We provide a comparison analysis among five distinct approaches to rank program transformations: Support Vector Machine (SVM), Logistic Regression (LR), Neural Networks (NN), Human-Expert (HE), and Random Weights (RW). We first compare the three ML-based ranking approaches, SVM, LR, and NN, for ranking program transformations, and then we compare them with a HE and RW approaches. To compare these approaches, we instantiated them on a state-of-the-art PbE technique, REFAZER [7], given its capabilities to learn multiple transformations in program transformation domain given examples of edits [7]. We have used edit scenarios from three open-source C# projects from GitHub, namely Roslyn [35], Entity Framework [33], NuGet available in public usage [7]. In addition, we have manually analyzed commits from other two C# projects, namely ShareX [37] and [34] Newtonsoft [36]. In total, we use 43 edit scenarios, from which 15 are used to generate transformations for training ML models, and we evaluate all the approaches in the other 28 edit scenarios. We measure their efficiency in ranking the transformations by comparing the number of examples required to rank the desired transformation in the first position for each edit scenario. We also measure their effectiveness by accessing their accuracy and relevance of the ranked transformations by evaluating how well the ranking approaches put correct transformations in the top-10 using the Normalized Discounted Cumulative Gain (NDCG) and Precision metrics.

An ML-based approach presented a similar performance compared to the HE, however, reducing the manual effort in finding the weights of the ranking function. In terms of efficiency to rank program transformations, LR presented similar results compared to HE, with example means of 1.67 and 1.64, respectively. Compared to RW, LR provides a statistical difference, with p-value < 0.05, which indicates that the weight automatically obtained had an effect in ranking transformations. In terms of effectiveness, LR presented similar results to HE with both Precision and NDCG of 0.5 and superior to RW with 0.2.

1.5 Conclusions

In this work, we propose an ML-based approach for automatically computing the weights to build ranking functions. We evaluate three ML-based ranking approaches for program transformations and compare them with a HE and RW approaches. ML-based approaches are efficient and effective as HE approach while reducing the manual effort in finding the weights, which reduces the manual effort of PbE tool designers to find the ideal weights of transformations features. For end-users, it helps to reduce the manual effort in providing examples to PbE tools, contributing to widespread their usage. Future studies are required to analyze how users can interact with the top ranked solutions to express their preference.

1.6 Summary of contributions

In summary, this study makes the following key contributions:

- An ML-based approach to automatically build ranking functions for program transformations (Chapter 4);
- An empirical study comparing different ML-based ranking approaches to rank transformations (Chapter 5);
- A comparative study involving an ML-based approach with a Human-Expert and Random ranking approaches (Chapter 5).

1.7 Organization

This work is organized as follows: In Chapter 2, we provide a background with relevant concepts for the understanding of this work. In Chapter 3, we present the state of the art on ranking approaches employed by PbE tools dealing with ranking of program transformations. In Chapter 4, we present our approach for ranking program transformations, based on ML techniques to automatically build ranking functions. In Chapter 5, we describe the evaluation of our technique. In Chapter 6, we describe the main related work covering Human-Expert and ML-based ranking approaches. In Chapter 7, we present the conclusion of this work.

Chapter 2

Background

In this chapter, we present the background with relevant concepts for the understanding of our work. In Section 2.1 we present the context of software evolution, emphasizing repetitive edit. In Section 2.2 we describe program transformations. In Section 2.3 we describe the Program-by-Example (PbE) approach, Ranking problem in program transformations, Domain Specific Language (DSL), ranking functions, and human-expert ranking approach. In Section 2.4 we present the context of machine learning, supervised techniques, and main algorithms. Finally, in Section 2.5 we discuss about learning functions in ranking problems.

2.1 Software Evolution

Evolution consists of an essential aspect of the nature of any entity across domains. It relates to changes that occur over time, mainly for adaptations and progressive improvements given a changing environment in the domain [29]. Each domain has their own properties and characteristics to determine whether a phenomenon can be considered an evolution or not. However, we are not interested in discussing the aspects of this topic across domains. Our discussion in this work is limited to how evolution occurs in software engineering domain, more specifically, how software evolves over time.

Software engineering is a discipline that is concerned with the aspects of software production, including techniques that support program specification, design, and evolution [61]. It also can be seen as a discipline involves the aspects of development, operation and maintenance of software [17]. Even though much attention has been given to the development

aspect of the software production, evolution plays a key role in the lifetime of a software. For example, when we compare the costs involved in these two aspects, the evolution costs for a custom software often exceeds the development costs [61], reaching 40% or more [2].

According to the first law of evolution of Lehman [27], software employed in a real-world environment must be continually adapted. If it does not happen, the software will become less and less useful in that environment. The discipline that studies the software changes is called software maintenance. Software maintenance refers to the process of evolving, changing, adapting a software maintaining embedded its assumptions and compatibility valid in changed domains and under new circumstances [28]. Software maintenance consists mainly of changes to repair design defects, add incremental function, or adapt to changes in the use environment or configuration [2]. Usually, the environment in which the software operates has a dynamic nature. Given that dynamism, new needs often arise and designing a software to attend those needs, most of which previously unknown, makes the changes inevitable.

According to IEEE [17], software maintenance can be categorized into the following types:

- *Corrective* - performed to correct faults;
- *Adaptive* - performed to make the software usable in a changed environment;
- *Perfective* - performed to improve the performance, maintainability, or other attributes of the software;
- *Preventive* - performed to prevent problems before they occur.

However, as software evolves and changes are made, its structure will become more and more complex. Thus, the second law of Lehman [28] affirms that this will happen, unless active efforts are made to avoid or reduce it. One of the main strategies adopted to handle that side-effect, reducing the complexity of the software while allowing to be easier to maintain is called *refactoring*. Refactoring can be understood as the process of changing a software to improve its internal structure in such a way that it does not alter the external behavior of the code [8]. Although, the changes that occur in software maintenance are not restricted only to refactorings, but may involve other types modifications in the source code such as adding new functionalities and *bug* fixing activities.

2.1.1 Repetitive Edits

Correcting errors, performing adaptive modifications in the software, implementing requests and suggestions, and reorganizing code are activities that involve a number of edits in the source code. Many of these edits are repetitive, in the sense that, one specific edit can occur in two or more locations in the code base. Since manually applying multiple repetitive edits, specially in a large system, can be tedious, time consuming and error-prone, several tools are proposed to automate them.

Among these tools, we have Integrated Development Environments (IDEs) such as Eclipse [63], Visual Studio [38], and NetBeans [1] offer several functionalities to support developers automating repetitive edits. For instance, these tools include refactoring functions such as *rename*, *extract method*, *move method*, and others. In addition of IDEs, static analyzers such as ReSharper [19], Error Prone [10], and FindBugs [62] automate the analysis of potential errors in the code, finding mistakes and providing fixes.

Even though there are many tools to automate repetitive code edits, many developers still perform edits manually. For instance, the majority of refactorings are performed manually without the help of any tool [55] [41]. The three main reasons for that are *awareness* of an existing tool to apply the refactoring, *opportunity* to use a certain refactoring feature even though the developer knows how to use the feature, and lack of *trust* in the tool, given the concern of introducing errors or having unintended side-effects [41]. Another reason is that many edits cannot be automated by current IDEs and static analyzers. The aforementioned tools, as well as others, rely on predefined classes of program transformations that represent common edits applied by developers. More complex edits would require the developer to have knowledge and specific skills to extend the classes of program transformations.

2.2 Program Transformation

A *program transformation* is the process of mapping a program to another program. It can be understood as a set of rules that should be applied to a program in order to edit it, producing another program. The transformation represents a higher level of abstraction than the concrete edits, which are the actual edit operations in the code performed according to the specification rules in the transformation.

A program is represented in a Abstract Syntax Tree (AST), which is an abstraction of the source code represented as a tree. Thus, the edit operations are performed in an input AST in order to map it to another tree. The elementary tree operations consist basically of *insert*, *delete*, *update*, and *move* [6]. To understand how these operations occur, let us consider an input tree T_1, on which the edit operations are applied, and a target tree T_2, the resulting tree, once operations are performed. These operations can be defined as:

- Insert(a,b,c): insert node a as the cth child node of b of tree T_1;
- Delete (a): delete the node a from the tree T_1;
- Update(a,b): change the value of node a to b in tree T_1;
- Move(a,b,c): move node a with all its children to cth child node of b in tree T_1.

The sequence of edit operations necessary to transform tree T_1 into tree T_2 represents an *edit script*. Usually, the difference between two distinct trees can be represented by more than only one edit script. For instance, the result of *move* operation can be obtained combining *delete* and *insert* operations. In that case, usually, we want the edit script with the minimum number of edits operations.

Many tools include a set of transformations to help developers along the software development and evolution. However, these tools are based on a predefined set of classes of transformations, such as *rename*, since it is not possible to include all the transformations that the developers would like to perform in a tool. Thus, the transformations are limited to common edits applied by the developers and extending these transformations requires specific skills, time and manual effort. Automating the process of extending these transformations in order to support more complex edits would provide a useful solution. In this scenario, one of the approaches to allow abstracting new classes of program transformations automatically is called Program-by-Example (PbE).

2.3 Program-by-Example

Repetitive edits are often *similar*, in the sense that they share the same structure but involve different expressions. For instance, replacing "==" expression, by invocating a method, in

two or more locations with same structure represents an edit, which happens to be similar and repetitive in this case. Similar edits can then be generalized into a transformation, which is an abstraction of the concrete edits, and applied to other locations in the code. However, the generalization process can be cumbersome if it is done by analyzing each similar location and manually implementing a transformation. To obviate this process, a PbE approach aims to infer new classes of transformations from input-output examples of edits performed by developers [11]. Previous and after versions of code edited are used as examples of the desired transformation the user wants to perform. A PbE tool abstracts the edits, generalizes them and applies this generalization to other similar locations in the source code.

PbE tools have been used across distinct domains, supporting users dealing with repetitive edits. The two most common application domains include data transformation/wrangling and code transformations [14]. In data wrangling domain, which we refer to as *string* transformation domain, end-users deal with transforming the raw data format into a more appropriate and easy to visualize format. A large number of end-users have computers nowadays. They want the benefits provided by the computers, but usually they do not have enough skills to write programs even to automate simple tasks. For instance, 99% of end-users do not know how to program, and struggle with repetitive edits [12]. PbE tools arise in this context enabling end-users to specify their desired intent by giving examples of edits. An example of these tools is FlashFill, which supports end-users automating repetitive edits in spreadsheets generating new classes of transformations from input-output examples [11]. Other PbE tools are designed to automate more specific transformations, such as those related to numbers [57] and matrices [66].

In program transformation domain, PbE tools are designed to support developers dealing with edits in the source code. Most common situations in this domain include the application of refactorings, API migration, and feedback generation for programming assignments in education scenario [14]. For instance, PROSPECTOR [30] tool uses a technique for automatically synthesizing code fragments given a query that expresses the input and output examples of the desired code, helping developers. Another tool designed to address two of these common needs is REFAZER [7], which takes input-output examples of edits performed by the developers and infer classes of transformation to automate the repetitive edit. In the education scenario, REFAZER takes examples of edits applied by students, infers classes of

transformations, and then uses the transformations to fix new submissions or provide feedback.

The process of inferring the desired transformation from examples involves three key components, which are the search algorithm, ranking strategy and interaction models [14]. Figure 2.1 depicts the components of the PbE architecture. The search algorithm receives as inputs an example-based intent, a ranking function, and a Domain Specific Language (DSL), and it is able to synthesize a set of transformations consistent with the given examples. This set of transformations is expected to have the correct programs on the top, i.e., the ones with the highest likelihood of being correct on unseen data. The debugging component interacts with the user, presenting the ranked list of programs, usually top-k transformations. The user can refine the specification, whenever the set of transformations does not correspond to her desired intent. When the synthesized transformations correspond to the desired intend, they can be translated into the target language.

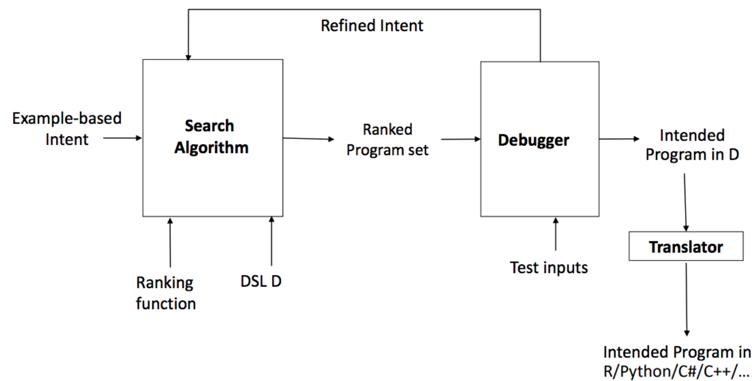


Figure 2.1: Components of the PbE architecture

Since PbE tools usually are designed to solve problems in specific domains, such as automating *string* or programs transformations, the tool designers have to deal with the representation of those domain tasks being specified through examples. The space of possible programs that can be learned can be expressed in a DSL.

The DSL describes the syntax of possible programs abstracted from the given examples in a specific domain. Since the range of program possibilities the user might want to express is extensive, the DSL has to meet some criteria. The DSL has to be expressive enough to represent several common programs that occur in practice and restrict enough to enable

efficient search [13]. Figure 2.2 depicts an example of a DSL used in FlashFill for *string* transformation domain [11]. For instance, `concat` is an operator of the language responsible for the concatenation of two expressions E_1 and E_2. `substr` operator receives as an input a string x and returns the substring between its initial and final position P_1 and P_2. The operator `conststr` constructs a constant string while $String$ is any constant and $Integer$ can take any integer to specify a position, and `pos` returns the kth position of string x that matches the regular expressions R_1 and R_2.

String Expression E:=concat(E_1, E_2) | substr(x, P_1, P_2) | conststr($String$)
Position P:= $Integer$ | pos(x, R_1, R_2, k)

Figure 2.2: Example of a DSL for *string* transformation

For instance, consider Figure 2.3. In column A, it includes undergraduate students' first and second names, and in column B, the names of the first student concatenated to the university domain. The second and third row of column B are the expected outputs, given the first row. Let us suppose that an Excel user wants to create an academic email for all students of the university campus and wants to use FlashFill to automate this task. Thus, the task consists of concatenating the first and second names of the students with the university academic domain. Once the user gives one example of the desired intent based on the first row of the column A, the tool infers a transformation that can be applied to other rows of the same column.

	A	B
1	**Input**	**Output**
2	maria silva	mariasilva@uni.edu.br
3	pedro costa	pedrocosta@uni.edu.b
4	clara torres	claratorres@uni.edu.br

Figure 2.3: Data for the e-mail automation problem

The following string transformations can be generated from the given example:

- p_1: concat(conststr("mariasilva"), conststr("@uni.edu.br"))
- p_2: concat(substr(Input, ϵ, " ", 1), conststr("silva"), conststr("@uni.edu.br"))

- p_3: concat(conststr(“maria”), substr(Input, “ ”,ϵ, -1), conststr(“@uni.edu.br”))
- p_4: concat(substr(Input, ϵ, “ ”, 1), substr(Input, “ ”,ϵ, -1), conststr(“@uni.edu.br”))

The synthesized transformations range from the most specific, generating only "mariasilva@uni.eu.br" for any given input, to the most general one, generalizing the input names. Give that p_4 has the highest probability of satisfying the user intent when generalized to "pedro costa", thus is can be considered the correct one. This transformation receives two strings "maria" and "silva" as input strings, concatenates them, constructs the constant string "@uni.edu.br", and transforms the expressions into mariasilva@uni.edu.br. In this transformation, the first `substr` operator returns the first position within the input string such that the left-hand side of the position matches anything, represented by ϵ, and the right-hand side matches white space. The second `substr` operator returns the last position within the input string such that the left-hand side of the position matches white space and the right-hand side matches anything. The resulting strings of these two `substr` operations are concatenated and added a constant string.

2.3.1 Ranking Problem in Program Transformations

In general, PbE approaches that learn transformations from examples deal with the key challenge associated with the nature of the examples. Examples are under-specified and specifying a desired task by using examples of edits may lead to a number of desired and undesired transformations by the developer. For instance, consider the edit that replaces "==" expression in `if(m.CSharpKind() == modifier)`, by invocation a new method IsKind, passing the right-hand side expression as the new method argument, resulting in `if(m.IsKind(modifier))`. The same edit is applied to `if(trivia.CSharpKind() == SyntaxKind.None)` resulting in `if(trivia.IsKind(SyntaxKind.None))`. The set of learned transformations from these two examples includes transformations that edits less similar locations than needed by the developer, such as one that edits only locations where an *identifier* precedes `CSharpKind()`. Another transformation edits more locations than needed, such as one that edits all locations that have an *expression* preceding the equals operator.

Being concerned that the learned transformation selected by the PbE tool could produce undesired modifications in the source-code, developers could naturally avoid using those tools and opt for manually performing the edits. Thus, the challenge posed to PbE tools in program transformation domain relies on selecting the learned transformation, in a set of learned transformations, that has the highest probability of producing the developer's desired edits on unseen similar locations. In addition, developers do not feel encouraged to provide many examples to specify the desired transformation, since providing many examples implies that it would be more advantageous to perform the edit by themselves.

2.3.2 Domain-Specific Language

Similarly to PbE tools for string domain, in program transformation domain, PbE tools also base on the DSL for describing possible transformations. For instance, let us consider REFAZER. A transformation is defined as a sequence of distinct rewrite rules applied to an *Abstract Syntax Tree* (AST). Each of these rules specifies an *operation* that should be applied to some *locations* in the input AST. In Figure 2.4, we can see REFAZER DSL with the tree edit operators such as *Insert*, *Delete*, and *Update*, a list of processing operators such as *Filter* and *Map*, and pattern-matching operators on trees.

The DSL can be seen as a tree. For each production rule (i.e., line in the DSL), the terminal symbols (i.e., right-hand side in the production rule) are nodes, and the parameter of each of the terminal symbols (i.e., non-terminal symbols) are the children of this node. For instance, the `Transformation` terminal symbol has as children a sequence of rules. A rule specifies an edit *operation* that has to be applied to a specific *location* in the input AST in order to generate a new AST. A location is specified by a *filter*, which analyses all nodes in the input AST looking for a pattern matching. According to Figure 2.4, the hierarchical structure of AST transformations are built upon two main nodes: *ConstNode* (i.e., creates a sub-tree from scratch, without any reference to the nodes in the AST input specification) and *Reference* (i.e., creates a sub-tree making references to the AST input specification.). The *operation* described in the DSL operates over these two nodes to build a new AST.

For instance, in the aforementioned problem in program transformation domain, the transformation edit operations in the DSL can be applied in the locations characterized by the pattern *expression*.`CSharpKind()` == *expression*. The edit operation `Update` is applied

$\langle transformation \rangle$ ::= `Transformation(`$\langle rule \rangle_1, \ldots, \langle rule \rangle_n$`)`

$\langle rule \rangle$ ::= `Map(`$\lambda x \rightarrow \langle operation \rangle, \langle locations \rangle$`)`

$\langle locations \rangle$ ::= `Filter(`$\lambda x \rightarrow$ `Match(`x`,` $\langle match \rangle$`)`,`AllNodes())`

$\langle match \rangle$ ::= `Context(`$\langle pattern \rangle$`,` $\langle path \rangle$`)`

$\langle pattern \rangle$ ::= $\langle token \rangle$ | `Pattern(`$\langle token \rangle$`,` $\langle pattern \rangle_1, \ldots, \langle pattern \rangle_n$`)`

$\langle token \rangle$ ::= `Concrete(`$kind$`,` $value$`)` | `Abstract(`$kind$`)`

$\langle path \rangle$ ::= `Absolute(`s`)` | `Relative(`$\langle token \rangle$`,` k`)`

$\langle operation \rangle$ ::= `Insert(`x`,` $\langle ast \rangle$`,` k`)` | `Delete(`x`,` $\langle ref \rangle$`)`
| `Update(`x`,` $\langle ast \rangle$`)` | `Prepend(`x`,` $\langle ast \rangle$`)`

$\langle ast \rangle$ ::= $\langle const \rangle$ | $\langle ref \rangle$

$\langle const \rangle$ ::= `ConstNode(`$kind$`,` $value$`,` $\langle ast \rangle_1, \ldots, \langle ast \rangle_n$`)`

$\langle ref \rangle$ ::= `Reference(`x`,` $\langle match \rangle$`,` k`)`

Figure 2.4: A core DSL for describing AST transformations in REFAZER

to the matched locations. It edits the selected location by applying the rule that updates the location by applying the *ConstNode*, which creates "`IsKind`" that is not present in the input AST and reuses nodes from the input tree, making reference to it, such as the expression that precedes `CsharpKind()` and the one that succeeds the "==" expression.

This operation updates the selected location x with a fresh call to `IsKind`, performed on the extracted receiver AST from x, and with the extracted right-hand side AST from x as its argument. `Pattern(==, Pattern(., Abstract(expression), Concrete (<call>, "CSharpKind()"), Abstract(expression)).` The transformation applied to this location is characterized by `Update(`x`, ConstNode(.,` r_1`, <call>, "IsKind",` r_2`))`, where r_1 and r_2 make reference to the input.

2.3.3 Ranking Functions

Examples are ambiguous, hence the number of inferred transformations consistent with an example based specification is often huge. Although the inferred transformations are con-

sistent with the given examples, they are distinct and do not equally satisfy the user desired intent. In general, very specific or very general transformations are not desired, as they are likely to respectively produce false negative or false positive edits on unseen programs [7]. For instance, in Figure 2.3, consider the input-output example given "maria silva" to "mariasilva@uni.edu.br". A very specific transformation inferred from this example would be restricted only to input strings "maria" and "silva", treated as constant strings in the specification, and all the other locations following the same constant pattern. However, this specific inferred transformation would not satisfy the user desired intent when editing, which is to generalize the example specification to "pedro" and "costa", and "clara" and "torres". Whenever one example, such as this one, is unable to specify the correct intent, another example would be required by the tool in order to refine the example-based specification producing a more generalized transformation.

Thus, in this context, a challenge to PbE approaches is identifying the correct transformation (i.e., the one that has the highest likelihood of producing the desired edits when generalized to other inputs) in a large set of inferred transformations consistent with the input-output examples. The task of selecting the correct transformation is done by a *ranking function*, which sorts the transformations aiming to put the ones that are more prone to be correct in the top position. In this context, the main goal consists of learning an appropriate ranking function to rank transformations regarding their correctness. In order to be more usable, PbE approaches have to infer the desired transformations and rank them based on the minimal number of examples, preferable one.

Since the set of inferred transformations may involve distinct transformations, the ranking function has to be able to disambiguate them. These transformations involve distinct characteristics, and intuitively, we realize that some of these characteristics are more desirable than others. For example, transformation with more general expressions are more likely to produce the desired intent in the email problem, rather the very specific expressions such as *constants*. One common strategy to build ranking functions relies on using *heuristics*, which uses specific knowledge from an expert in the application domain to favor transformations that hold specific characteristics.

The idea of using heuristics is to assign transformations with specific properties a distinct *score*. Each transformation in the set of synthesized transformations receives a score,

which represents the correctness of the transformation. In other words, the score represents the probability of the transformation producing the programmer's desired intent. The ranking scores are given by a ranking function. The ranking function is built based on specific properties of the grammar and each property receives a *weight*, which is a real number that represents the influence of that particular property on the final score of a given program.

In string transformation domain, FlashFill tool ranking approach is based on a set of heuristics. One general principle adopted, also observed in other data wrangling tools ranking approaches, consists of using the Occam's razor principle. This heuristic principle states that the simplest explanation is usually the correct one [11]. Thus, simpler transformations are preferable and favored, occupying the highest positions in the ranking. For example, considering the example in Figure 2.3, a `subStr` operator is simpler than both `constStr` and `concatenate`. `constStr` involves constant expressions that are less likely to occur in the input string, since they are built from the scratch, thus increasing the chances of producing a very specific edit. `concatenate` is simple if it contains a small number of arguments. However, in `concatenate` constructor, if a long substring match between input and output is observed, it is more probable that the corresponding part of the output was produced by a single substring. In order to select the desired transformation, ranking functions have been built in order to benefit specific transformations. A function in string domain can be expressed as $f(\cdot) = w_1 \times concat + w_2 \times input + w_3 \times substr + w_4 \times conststr + b$. To favor `subStr` over `constStr`, a domain specialist assigns a higher real value to w_3 than w_4. Is implies that the `subStr` is more relevant, thus contributing more to the ranking score of the transformation.

In the program transformation domain, the ranking approach employed by REFAZER tool follows similar heuristics, favoring transformations with specific properties. The ranking favors transformations that reuses nodes from the input AST rather than those ones that construct nodes from the scratch, thus, being less likely to occur in the input AST. Another similar idea concerns to the surrounding context of the edit location. REFAZER ranking favors transformations that select its locations based on the existing context, and when they do exist, the ranking favors the shorter ones. The ranking favors specific transformations by assigning their desired properties higher weights, resulting in distinct scores.

However, building a heuristic based ranking approach may present some disadvantages.

Manually assigning weights to specific transformation properties can be tedious and cumbersome. The ranking designer is required to have domain specific knowledge and understanding of how each property may influence the final score and manually giving a real number to describe that level of influence.

2.3.4 Human-Expert Ranking Approach

Based on the intuition that certain properties of the transformations have greater contribution in the final computed score, the Human-Expert based ranking approach follows some principles or heuristics, such as favoring specific features in a set of features. These specific features receive higher weights, thus moving the transformations with these features to a better position in the ranking.

There are three main heuristics employed in the approach. The first one consists of favoring *Reference* over *ConstNode*, therefore, *Reference* receives a higher weight. According to this principle, a transformation that reuses a node from the input AST has higher probability of producing the desired developer's intent than creating a constant node without any reference to the input nodes. For instance, a transformation that receiving the input `trivia.CShapKind` generalizes it to *expression.*`CShapKind` has higher probability of selecting other similar locations, rather than only locations with constant `trivia.CSharpKind`. Thus, consider the human-expert based approach as a linear function, in which weights are assigned to specific characterizes in order to benefit certain transformations. A function expressed in the form of $f(\cdot)$ = w_1 × *Reference* + w_2 × *ConstNode* + w_3 × *Concrete* + w_4 ×*Abstract* + w_5 × *Context* + ... + *b*. The heuristic used here assigns a higher real value to w_1 than to w_2.

The second principle builds on the idea of favoring patterns that consider surrounding context of a location. Transformations that ignore the context of a location have higher probability of generating more false positives, thus ignoring locations that must be edited. The heuristic used here assigns a high real value to w_5.

The third principle affirms that, whenever a non-empty context appears, it favors the shorter ones, following the Ocam's razor principle. By doing so, the rank prevents favoring transformations that generate false negatives, which means those that edit locations that should be edited. The idea behind this principle, and the previous one, relies on searching

for a balanced solution, favoring transformations that do not ignore locations that must be edited whilst ignore locations that must not be edited.

In addition, *Concrete* patterns receive higher weights over *Abstract* patterns, which favors transformations that match more specific edits. Favoring specific patterns reduces the capacity of generalization of the transformations, restricting the matches to specific locations. However, this heuristic can be useful in situations where the output follows the same pattern in all the locations to be edited, thus reducing the number of required input examples. The heuristic used here assigns higher real values to w_3 than to w_4.

Building ranking functions based on favoring selected features requires specific knowledge and skills of an expert in the program transformation domain. Manually selecting relevant features and assigning distinct weights to them to build a ranking function can be a tedious, time-consuming and error-prone task. In addition, trying all possible values follows a trial and error approach, which, depending on the number of features, becomes unsuitable. Moreover, the complexity of the function, is another aspect to deal in building functions manually. Thus, building a ranking function can be benefited from automated techniques. ML techniques have been applied in several application domains to solve difficult problems [39], thus learning several types of functions, ranging from the easiest to the most complex ones, such as nonlinear functions. In program transformation domain, ML techniques can be used to automatically learn the weights of the features based on experience and build ranking functions that can properly rank transformations.

2.4 Machine Learning

Machine Learning (ML) is the field of study concerned with programs that learn from past experience [50]. Another more detailed view explains that a program is said to learn from experience when it improves a measured performance on a task with experience [39]. For instance, suppose we have a cat learning problem. We want a program to learn whether an image represents a cat or not based on experience. For this problem, the previous experience could consist of a collection of images previously labeled as "cat" and "not cat". This collection of images constitute our *training examples* dataset and each training example is a pair consisting of an input image and the correct label for that input image. The task of an

ML algorithm is to learn the desired model based on the training examples. This model is a mathematical function that receives an unseen image as an argument and outputs a label, which is a prediction of the correct label for that image. For our example, our task is to simply learn a model that assigns a label "cat" or "not cat" to any given image, classifying them in two groups. To evaluate the performance of an ML algorithm we can use some metrics. For this problem, we can use as metric the accuracy, which measures the number of instances correctly classified over the total number of instances.

The range of domain applications of ML techniques is wide, involving robotics and autonomous vehicle control, speech processing and natural language processing (i.e., voice recognition), neuroscience research, applications in computer vision, and more recently, recommendation systems [20]. The later topic, for example, is concerned with predicting the preference of a user for a item, in a list of items, based on experience. The experience represents a collection of records of interactions of the user with items of her preference, and the ML technique learns a function the is able to predict the user's preference for unseen items. Once the ML technique is able to predict the items that have higher probability of matching the user's preference, the technique ranks them and recommends those on the top (i.e., items with highest probabilities of matching user's preference). Some of the main applications of recommended systems are music, movies, books, and product for sale in general.

Learning problems, as well as learning strategies, can be divided in three main categories: supervised learning, reinforcement learning, and unsupervised learning [50]. Each of these categories can be understood as follows:

- Supervised Learning - for each training example, there is an associated output. The goal consists of finding a function that relates the output to the associated input example, aiming to predict outputs for future observations (prediction) or understand the relationship between the output and the observation.

- Reinforcement Learning - the learning process aims to improve some measured performance at some task by being rewarded or penalized. Whenever the action taken goes toward the task's goal, it is rewarded. Whenever it takes a step into another direction other than the goal, it is penalized. The goal consists of predicting which step to take that maximizes the rewards, which eventually leads to an oriented goal.

- Unsupervised Learning - there is no explicit outcome or correct answer associated with the observations. The goal consists of understanding the relationship between the observations and finding patterns. One of the sub-classes of unsupervised learning problems relies on finding a function that can group similar observations in distinct classes, a process called *clustering*.

Even though these three learning strategies are important to understand how application domains have been benefited from their contributions, we do not focus on discussing in details the reinforcement and unsupervised learning in this work. We have briefly expressed how these learning strategies occur but, instead, we focus on supervised leaning, since it plays a more relevant role in the comprehension of this work. PbE has been a promising application domain of supervised strategies, specially to guide the search of possible transformations and rank the correct ones based on their properties. But, before getting into more details about supervised learning, let us consider an important contrast.

It is important to make a distinction between the role of PbE techniques and ML techniques, since these both techniques are learning strategies that can take examples of a task from the user (training examples), learn a function (model) and are able to automate the same task in the future on unseen data (prediction of future behavior) [25]. These techniques can easily be misunderstood; however, they differ in certain aspects. First, ML techniques require as many training examples as possible, even thousands, to be able to learn a reliable function. On the other hand, in order to be useful, PbE techniques must learn a function based on the minimum number of examples, preferable one. Second, functions learned by a PbE are human-readable and, usually, editable programs unlike some ML functions, which are black-box, such as Neural Networks. Nevertheless, ML is better suited for fuzzy/noisy tasks [12].

2.4.1 Supervised Learning

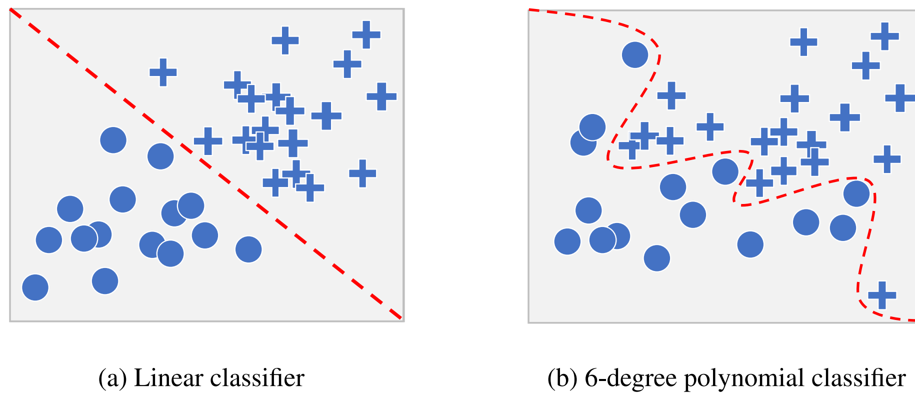
Supervised machine learning is the learning strategy that observes some example input–output pairs and learns a function that maps from input to output [51]. The function has to *generalize* and predict the correct output of new observations based on the training data examples. Generalization is fundamental because the training data represents only a

sample, usually noisy, of the total of the observations in a domain. To access how well the function generalizes and performs on novel observations, the technique uses a set of test examples, usually a fraction of the training examples not included in the training process. If the function is able to predict correctly the outputs for test examples, then the function is said to have a good generalization for novel observations.

There are two common types of output variables for input examples in machine learning problems, which are *quantitative* and *qualitative* or categorical. Quantitative variables are represented by continuous values and are used in distinct contexts, such as predicting one's age, income or the price of products in the market. Qualitative variables are represented by discrete values, categories or classes, instead of continuous numerical values. For example, determining if an email is a spam or not, diagnosing cancer, recognizing an animal's breed, and others. Problems with quantitative responses are referred to as *Regression* problems, while problems with categorical responses are referred to as *Classification* problems [18]. In this sense, the cat learning problem previously mentioned can be seen as a classification problem. Since there are two possible outputs for the images, this problem can be defined as a binary *classification*. The classifier function has to be able to classify the observation into predefined categories of objects, in this case, the two classes "cat" and "not cat".

In Figure 2.5a, we can see a classification problem, two sets of observations being separated by a dotted straight red line, which represents a *linear function* or linear classifier. For this specific example, based on the input-output examples and their properties, the learning strategy finds a simple linear function that is able to completely and successfully classify the observations in two categories. However, more complex problems involve observations that are a much more difficult to separate using a simple straight line. In those cases, the learning strategy requires more complex solutions, such as polynomial functions with high degrees. The less uniformly spread the observations look like, the higher the complexity and the degree of the polynomial function learned. For instance, in Figure 2.5b we see a problem with observations that are not uniformly spread, becoming difficult to separate them in two categories with a straight line. In this case, a 6-degree polynomial function is able to successfully reach that goal.

In Figure 2.6 we see a regression problem, characterized by numerical values or continuous outputs. By regression problem, we mean the problem of predicting a continuous



(a) Linear classifier (b) 6-degree polynomial classifier

Figure 2.5: Binary classification problem

quantity based on input examples quantities. For example, predicting a price of a house based on its size, or number of rooms, or predicting one's income based on years studied. To solve those types of problems, a linear function takes the input examples, such as different sizes of houses along with their prices, and predicts the price of a novel house given its size.

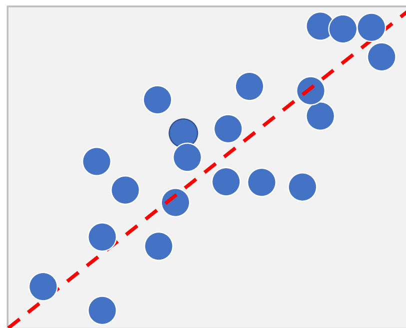
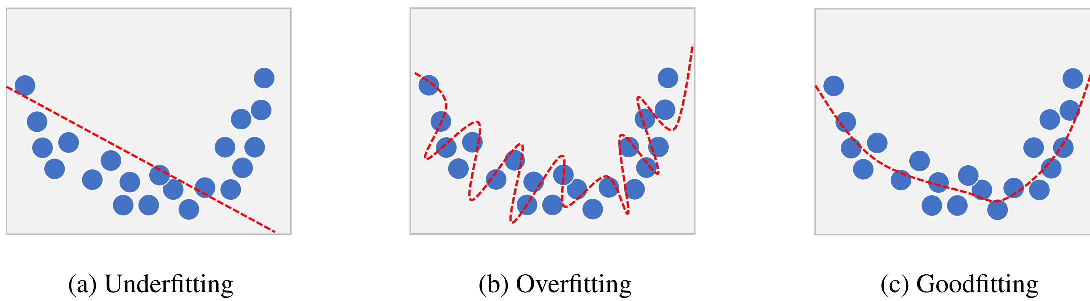


Figure 2.6: Regression problem

The complexity of the function may have an influence on how the function will fit new observations. The goodness of the fit is not determined by how well the function fits the training data but how well it generalizes to new observations. Depending on how the observations are spread on the graph, a too much simple function leads to *underfitting*, as seen in Figure 2.7a. This undesirable phenomenon is characterized by a function that does not capture a great part of the observations in the training data and does not learn its trend. On the other hand, more complex functions lead to *overfitting*, as seen in Figure 2.7b, which is also an undesirable situation, since the function captures basically all the observations,

including the noise and outliers. In other words, the function will be biased to the training data and will not generalize on new observations. The best scenario is seen in Figure 2.7c, which represents a *goodfitting*. In this case, there is a balanced trade-off between complexity and capacity of generalization. The function is complex enough to capture the trend of the observations but flexible enough to be generalized to new observations.



(a) Underfitting (b) Overfitting (c) Goodfitting

Figure 2.7: Fitting of function and complexity in a regression problem

The goodness of the fitting or the *performance* of the function is an important factor to determine how effective the function can be in producing the correct outputs for the observations. To access the performance of a given function, we base on some metrics. To explain each one of these metrics, we will use a confusion matrix. In supervised learning, the confusion matrix allows us to visualize the performance of a function. Figure 2.8 represents a confusion matrix with the classification on the row and the actual class on the column. Let us consider the cat problem again to understand the confusion matrix. *True Positive (TP)* refers to images classified as "cats" that are actually as "cats", *True Negative (TN)* refers to images classified as "not cats" that are actually "not cats", *False Positive (FP)* refer to images classified as "not cats" that are actually "cats", and *False Negative (FN)* refers to images classified as "not cats" that are actually "cats".

The first metric that we will see is the accuracy, which is commonly used across different techniques in supervised learning. The accuracy can be seen as the number of instances correctly classified over all the instances. We show the formal definition of accuracy in (Eq. 2.1) [45], based on the confusion matrix. For our cat problem, the accuracy relates to how frequently "cats" are classified as "cats" and "not cats" are classified as "not cats".

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

Predicted Class

Actual Class	Yes	No
Yes	TP	FN
No	FP	TN

Figure 2.8: Confusion Matrix

Precision evaluates over all instances classified as positives, the ones that are really positives. We show the formal definition of precision at (Eq. 2.2). For our cat problem, precision refers to the proportion of images classified as "cats" that are really images of "cats".

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

Recall evaluates over all instances that are true positives, the ones that the classifier was not able to classify correctly as true positives. We show the formal definition of recall at (Eq. 2.3). For our cat problem, recall refers to proportion of images of "cats" that were really classified as "cats".

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

However, there has to be a trade-off between the metrics precision and recall. In many situations, we have unbalanced classification problems. In situations, such as spam detecting, the number unimportant e-mails may outnumber the number of really important ones, or in diagnosing cancer, the number of healthy people may outnumber the number of sick people. These situations impose a challenge to measuring the performance of classifiers.

In spam-detecting problem, mis-classifications (FP) can have serious implications, for examples, classifying an important e-mail as spam. In another situation, such as cancer diagnosing problem, mis-classification (FN) can have serious implications, for example, classifying a person who has cancer as not having cancer. In the spam-detecting problem, we want a classifier with high precision, classifying as spams only the e-mails the are really spams. While in diagnosing cancer problem, we want a high recall, classifying people as not having cancer only the ones that are really healthy. Let us suppose our model classifies every single e-mail as spam. The precision is high, even though it is not useful since we have many false

positives, and the recall is low. In the cancer diagnosing problem, a model that classifies every single person as not having cancer has high recall, but it is not useful, since we have many false negatives, and the precision is low. F-measure (Eq. 2.4) attempts to provide a solution to the previous trade-off dilemma by computing the harmonic mean of precision and recall, especially in unbalanced classification problems.

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall} \tag{2.4}$$

Regarding the differences in performances, as well as the number of metrics to evaluate them, there is no single learning strategy that provides the best performance over all possible learning problems. This problem is emphasized by the No Free Lunch (NFL) theory [69], which explains that if a learning technique A performs better then B in one class of problems, then A must perform worse than B in other classes of problems. Similarly, there is no standard metric to evaluate the performance of all the distinct learning techniques and each metric has advantages and disadvantages depending on the data, the importance of false positives and false negatives, and whether the number of observations are unbalanced or not.

2.4.2 Learning Algorithms

Machine learning *algorithms* are the learning techniques responsible for solving classification or regression problems by estimating the function that maps input to desired the outputs. Common algorithms in supervised learning include logistic regression, Support Vector Machine (SVM), Neural Networks (NN), random forests, K-Nearest Neighbors and Decision Trees [5]. In this section, we describe the algorithms used in this work.

Logistic Regression

Even though the term "regression" can be used to describe predictive models of continuous output, such as linear regression, in this context, logistic regression concerns with classification models of discrete outputs, more specifically, 0 and 1. The algorithm is characterized by being simple and applied for classification or prediction in problems with two possible outputs, binary classification, such as "cats" or "not cats", "cancer" or "not cancer". As seen in Figure 2.9, the goal consists of finding a sigmoid function, called logistic function, which given an input value, it produces an output between 0 and 1. Note, from Figure 2.9, that

when the module of the input value is large, the output is always 0 or 1. However, when the module of the input value is small, the value oscillates between 0 and 1. Thus, the algorithm uses a threshold, which is commonly 0.5, to decide whether the instance should be classified as 0 or 1. To do so, it compares the output of the sigmoid function with the threshold. If the output is greater than the threshold, the instance is classified as 1, otherwise the instance is classified as 0. In a ranking problem, such as predicting or classifying a transformation regarding its correctness, the output of the sigmoid function can be used to determine how confident the model is in classifying the transformation. The value represents a probability, which can be used to distinguish between transformations.

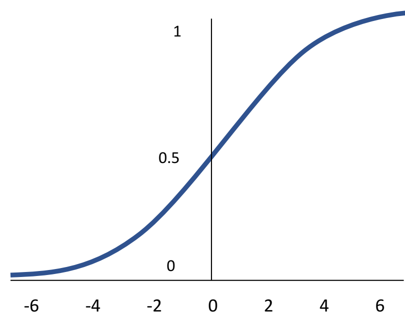


Figure 2.9: Logistic sigmoid function

Support Vector Machine

Support Vector Machine (SVM) learning algorithm is a classifier. It aims to find a function, known as hyperplan, that maximizes the distance between the two classes of observations, as seen in Figure 2.10. The closest observations to the hyperplan are called support vectors and the distance between a support vector and the hyperplan is the margin. The support vectors determine how the hyperplan can separate the two classes while maximizing their margins. SVM algorithms can also deal with observations that are not linearly separable. In the problem of classifying transformations as correct or incorrect, the line the best separates the two classes represents the function to be used. The most determinant observations will be the closest ones to the line, which shape the line. On the other hand, the furthest ones of the line represent the ones the model is most confident in classifying.

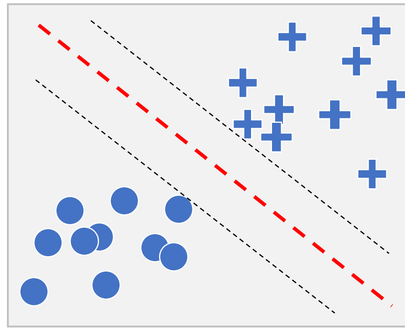


Figure 2.10: SVM function

Neural Networks

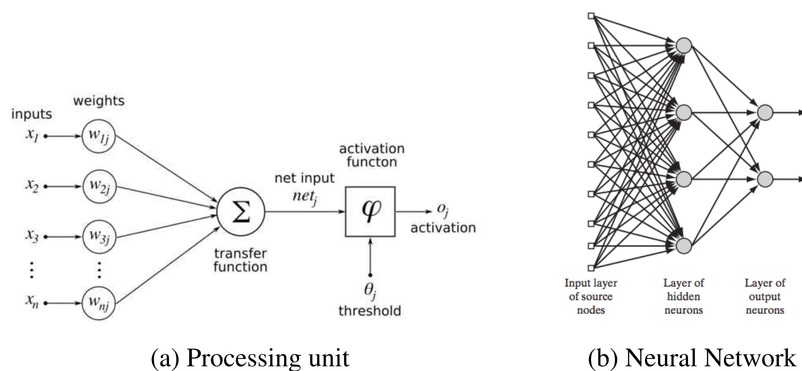
Neural Networks (NN), also called Artificial Neural Networks, have inspiration in the operation of the human brain, which is a network of interconnected neurons that can process information using parallelism. From a computational point of view, NN can be understood as a method of representing functions using networks of computing elements and about methods for learning such representations from examples [51]. Using the analogy of the brain, NN consist of networks of interconnected processing units that can process information.

A neural network can be seen as a graph where each processing unit represents a node in this graph. A neural network has a number of levels, and each level contains a number of nodes. The first level represents the input for the neural network and the last level represents the output, which is the classification of the neural network. For instance, in 2.11b, we have a network with three levels, where the first level contains 10 nodes, the second level contains four nodes, and the last level contains two nodes.

In a neural network, each node represents a function that receives n inputs and produces 0 or 1 as output. Each one of the n inputs has a weight associated with it. In Figure 2.11a, we can see the details of a processing unit. The function associated with a node can be composed of two functions. The transfer function that produces as output a real value, given the inputs and the weights. Then, given this value, the activation function produces as output a value that can be 0 or 1. An example of an activation function is the sigmoid function discussed previously. The output of a function in a node, by its hand, works as the input for all the nodes in the next level that, that by its hand, works as input for the next level. This process continues until the last level, which is the output of the neural network.

Along the learning process, the neural network is modified to improve the classification. In successive iterations, the output of the neural network is compared with the real label of the training example instance. This comparison is used to improve the neural network. For instance, when the output of the neural network differs from the real label, the weights of the network need to be modified in order to improve the neural network to classify this instance correctly. This update goes backward, a process known as backpropagation. First, the weights associated with the output level is updated, then the weight associated with the previous level and so on.

Now that we have a general overview of an neural network, let's discuss how the processing unit works. Figure 2.11a presents a processing unit. The processing unit includes and transfer function, a linear component that computes the sum of the weights of the input's values, represented by Σ. It also includes an activation function, a nonlinear component that modifies the weighted sum in the final activation value, and represented by φ. In Figure 2.11b, we see a neural network with one two layers of processing units. The first layer does not count since there is no computation on it.

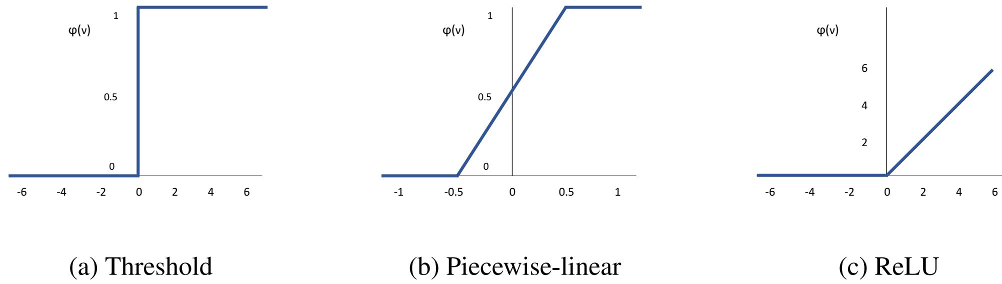


(a) Processing unit (b) Neural Network

Figure 2.11: Neural Networks structure

As activation function, a number of functions can be used. Previously, we have discussed one of these functions, the Sigmoid function. In Figure 2.12 we see three simple types of activation functions given by $\varphi(\upsilon)$. Among these functions, we have:

- *Threshold function*: this can be seen in Figure 2.12a. Given the value of υ, the corre-



(a) Threshold (b) Piecewise-linear (c) ReLU

Figure 2.12: Activation functions

sponding output will be given in terms of the equation (Eq. 2.5).

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{2.5}$$

- *Piecewise-linear function*: this can be seen in Figure 2.12b. Given the value of v, the corresponding output will be given in terms of the equation (Eq. 2.6).

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0.5 \\ v & \text{if } -0.5 < v < 0.5 \\ 0 & \text{if } v < -0.5 \end{cases} \tag{2.6}$$

- *Rectified Linear function (ReLU)*: this can be seen in Figure 2.12c. Given the value of v, the corresponding output will be given in terms of the equation (Eq. 2.7).

$$\varphi(v) = \begin{cases} 0 & \text{if } v < 0 \\ v & \text{if } v \geq 0 \end{cases} \tag{2.7}$$

In classification problems, such as classifying transformations as correct or incorrect based on their characteristics, the first level of the NN receives a set of features with corresponding values, which are then propagated through the next levels until reaching the last level, thus having a class as the output. The connections between the nodes in the levels receive weights, which indicate the strength of the connection and are determinant in computing the final class.

2.5 Learning Ranking Function for Ranking Problems

We have seen a variety of learning problems that can be solved by supervised learning techniques, more specifically, linear regressions and classifiers. Another class of problems consists of ranking problems. The main goal of ranking problems consists of sorting multiple instances according to specific criteria in order to evaluate the most relevant instances. These instances are usually sorted in ascending or descending order according to their assigned scores, which is a measure of their relevance to the problem. Instances with higher scores are ranked in higher positions while instances with lower scores receive lower positions in the rank.

There is a variety of applications of ranking approaches in distinct problem domains. Information retrieval is one of the most popular ranking application domains, involving search engines and recommendation systems. In searching engines, for example, the users want to access the most relevant documents given a search query. The documents are ranked according to certain ranking criteria, such as frequency of key-words, number of links, and others. Each of these criteria are associated with specific weights that represent the importance of that criterion to the document. Performing arithmetic operations over the weights the can result in a final score.

Similar to the previous learning problems, ranking involves learning a function that assigns scores to instances based on their degree of relevance or preference. For example, in recommendation problems, ranking functions aim to compute the preference of a user for an item, in a set of multiple items, based on past interactions with them. In those situations, the number of times the user viewed an item, number of votes the user gave to it or to similar items, the number purchases of similar items, and other interactions are relevant properties that can be used by a function to assign a score.

While the ranking function can be constructed manually, based on the knowledge of an expert in the problem domain, it can also be obtained automatically by means of machine learning. Intuitively, some characteristics of specific items are more desirable than others. For example, documents with high frequency of key-words that match the search query might be more desirable than documents that do not have this property. Thus, manually constructing a function the assigns higher weights to some specific characteristics based

domain knowledge of an expert may provide an appropriate solution.

However, manually assigning weights to characteristics can be tedious, time consuming and error-prone. Another strategy to obtain a ranking function consists of using machine learning algorithms for automatically learning the weights of the characteristics of the instances and constructing a function. In this scenario, traditional ML algorithms, such as linear regressions and classifiers can provide an alternative solution to learn ranking functions. Considering that some of these algorithms can predict the preference of the user for an item based on examples, the prediction usually is estimated as a real valued probability that, given a threshold, can be converted to a class. These probabilities provide a meaningful strategy to ranking instances, providing a list of instances ranked according to their probabilities.

Chapter 3

State of the art

In this chapter, we present the state of the art on ranking approaches employed by Programming-by-Example (PbE) tools dealing with ranking of program transformations. To obtain the state of the art, we employed a systematic review of the literature. In Section 3.1 we present the background, motivation, goal and research questions of our study. In Section 3.2 we present the review methodology including the search *string* formulation, selection of searching databases and criteria for the selection of studies. In Section 3.3 we present the results of the systematic review. In section 3.4 we discuss the results obtained in our state of the art and present the limitations of our study. Finally, in Section 3.5 we present the conclusion remarks.

3.1 Motivation

During software evolution, developers often perform similar repetitive edits when adding features, refactoring code and fixing bugs. These bugs are often recurring and can be fixed by similar edits [42]. Some approaches have been proposed to support developers automating repetitive edits, such as IDEs and static analyzers. Another approach is Programming-by-Example (PbE), which aims to infer a program from input and output examples using inductive reasoning, thus saving time, minimizing the manual effort, and errors. Usually, users are reluctant to provide many examples, otherwise, they are performing the repetitive edits by themselves. Thus, PbE approaches have to infer the desired program from the minimal number of examples, preferable a single one.

However, examples are underspecified and the inferred transformation consistent with an example-based specification are often numerous [44]. Identifying the correct transformation in a large set of transformations poses a ranking problem to PbE approaches. The key challenge is to learn an appropriate ranking function to select the correct transformation in a large set of transformations induced from a minimal number of examples.

To overcome the posed ranking problem, approaches have been proposed. Commonly adopted ranking approaches are based on heuristics favoring certain aspects of the solutions such simplicity, size and others [11; 21; 56]. Instead of relying on heuristics, others employ a Machine Learning (ML) ranking approach to automatically learn a ranking function to rank transformations in string transformation context [59].

The literature presents reviews that gather specific studies on the automation of repetitive code edits to support developers [40]. Kim et al. [22] conduct a comparison of existing PbE approaches in terms of input, output, edit type, and automation capability. However, the study is limited to a predefined set of studies, rather than collected in a systematic way. Also, it does not focus on ranking approaches. To the best of our knowledge, no systematic review was conducted on the exploration of different ranking approaches in the context of transformation transformations.

The main goal of this systematic review of the literature relies on identifying and describing the main approaches used by these tools to deal with ranking of transformations in distinct domains. Formally, we can express our goal in the following sentence:

Goal: *Analyze* ranking approaches *for the purpose of* characterizing them *with respect to* the ranking of program transformations *from the point of* view of tools and techniques *in the context* of Programming by Example.

3.1.1 Research Questions

Once we have expressed the goal of our study, we address our research question. We address the following question:

- **RQ1:** What approaches do Programming-by-Example tools employ to rank multiple learned transformations in different domains?
- **RQ2:** What are the most common features employed by Programming-by-Example

tools to build ranking functions for ranking transformations in different domains?

We systematically gather a set of studies from specific databases and through a searching procedure, we intend to find studies that contribute somehow to answer our RQs. This selection process is described in the Section 3.2.

3.2 Methodology

We start by exploring a set of pre-selected studies with the intent of obtaining a meaningful comprehension about program transformation in PbE. These studies can help us understand the topic, investigate possible research gaps and obtain a reference of how the key terms may change according to the domain of applications. These insights can help us perform a more useful selection procedure.

In this exploring phase, the initial set of papers examined for the extraction of the key terms and synonyms were indicated mostly by specialists. Table 3.1 presents a set of three papers selected in order to explore the topic and obtain the key terms for searching, as well as other synonyms.

Authors	Studies
Singh and Gulwani [59]	Predicting a correct transformation in programming by example
Gulwani, S. [11]	Automating string processing in spreadsheets using input-output examples
Rolim et al. [7]	Learning syntactic program transformations from examples

Table 3.1: Studies for key terms and other spellings

The key terms found in the previously mentioned studies admit synonyms and other spelling as they refer to the same object of study. For example, considering these three studies, the term Programming by Example [7] can also be referred to as Program by Demonstration [11] and example-based program synthesis [59]. In Table 3.2 we present the key terms, synonyms and other spellings obtained from the previously mentioned studies.

Key Terms	Synonyms and Other Spellings
Programming by example	programming by demonstration, program synthesis, example-based programming and learning from examples.
Ranking	sorting and ordering
Approaches	techniques and strategies

Table 3.2: Synonyms and other spellings

3.2.1 Search *String*

Using specific keywords and synonyms found in the initially set of selected papers along with boolean connectors, we constructed our search string.

("programming by example" OR "programming by demonstration" OR "program synthesis" OR "example-based programming" OR "learning from examples") AND (Approach OR technique OR strategy) AND (ranking OR sorting OR ordering)

3.2.2 Databases

In the selection process of the databases, we considered using certain criteria, such as, observing their relevance to the area studied in this work and also considering the opinion of experts regarding to the choice of databases. We decided to concentrate our efforts on databases with the English language, considering that the number of published studies in English are more numerous. Regarding the selection of the databases, since we use a search *string*, it is recommended to use ACM Digital Library - Association for Computing Machinery, IEEE *Xpore* digital library - Institute of Electrical and Electronics Engineers and Scopus [23]. In addition, we also found relevant to use ScienceDirect and Google Scholar, which seem to be common used databases in systematic reviews.

3.2.3 Study Selection

Once we have performed the search procedure over the selected databases, a number of studies are retrieved. The studies are gathered from distinct publication sources and only a subset of these studies may be selected to be included in the review. In order to select these studies, we specify two types of criteria: Inclusion Criteria (IC) and Exclusion Criteria (EC). We accept and discard studies according to the following criteria specified in Table 3.3.

ICs	Criteria	ECs	Criteria
IC1	The study relates to PbE or Program Synthesis technique	EC1	The study does not present a tool, system, approach or technique
IC2	The study describes the ranking approach of the mentioned tool	EC2	The study is duplicated
IC3	The study was indicated by a specialist in the area	EC3	The study is not in English

Table 3.3: Study selection criteria

We analyze each study retrieved by the search procedure according to the following order of priority: first we read the title, then the abstract and keywords, then the introduction and finally the complete article. According to this analysis process, we read the studies observing the priority and judging the studies according to the previously pre-defined ECs. At this stage, we aim to exclude all those studies that have no potential to help us answer our RQ. Studies that meet any of the ECs receive a label of *Rejected*. Once we get all the studies that seem more relevant to us, we then judge them according to the ICs. The studies have to meet IC1 and IC2 at the same time, in order to help us answering our RQ, unless they meet the IC3. These studies receive the label *Accepted*.

The studies cited in the bibliographic references may also be taken into account as additional research resources, if they have not been returned in the research databases, also undergoing analysis according to the ICs and ECs. In these cases, we perform backward and forward snowballing process to find other relevant studies.

3.2.4 Information Extraction

In order to address our RQ1 and RQ2, we extract relevant information from the selected studies. Besides general information such as author, title, year and proceeding, we focus on specific additional information:

- **Tool**: We collect the name of the tool, technique or system (e.g. REFAZER, Prospector, CodeHint)
- **Domain**: We collect the transformation domain in which the tool operates (e.g. Program, String, numbers)
- **Ranking**: We collect the type of ranking approach employed by the tool (e.g. Heuristics, ML)
- **Features**: We collect the main features used to disambiguate between the programs in the ranking approach (e.g. size of the program, context, similarity)

3.2.5 Execution

The total number of studies retrieved, as well as, the number of studies selected after inclusion, exclusion criteria and quality assessment are found in Table 3.4. The total of studies retrieved includes the duplicated ones but when we perform the selection process, we remove them, thus they do not appear in the total number of selected studies. We also indicate how many studies we select after performing backward and forward snowballing according to studies retrieved from each database.

Database	Studies retrieved	Selected	Snowballing	Total Selected
ACM Digital Library	65	8	5	13
IEEE *Xplore*	62	0	0	0
Scopus	39	2	1	3
ScienceDirect	4	0	0	0
Google Scholar	100	2	4	6
Total	270	12	10	22

Table 3.4: Number of papers retrieved, selected before snowballing, and selected after performing snowballing

ACM Digital Library

We performed a search in ACM Digital Library with the predefined search *string* and had 65 resulting papers. We performed the search using the default search field.

IEEE Xplore

We performed an advanced search in IEEE Xplore with the predefined search *string* and had 62 papers resulting papers. We used the command search field in the advanced search options. Initially, we searched for full-text and metadata but the number of resulting papers was numerous. We restricted the search for metadata only, which includes abstract (summary), title, and indexing terms.

Scopus

We performed a search in Scopus with the predefined search string and had 39 resulting papers. We limited the search to abstract (summary), title and key terms.

Science Direct

We performed a search in Science Direct with our search string and the number of resulting papers was around 1,700. Thus, we limited our search to the abstract (summary), title and key terms but the number of relevant returned results decreased to 4 studies. However, none of them had to contribute to our RQs.

Google Scholar

We performed a search in Google Scholar with our search string and the number of resulting paper was too high, around 13,000. Thus, we limited our search to the title, selected the option to sort the results by relevance and restricted patents, citations and the language to

English but still the number of retrieved studies was too high, reaching up to 2,550 studies. Then we decided to analyze only the first 100 results.

3.3 Results

In this section, we summarize the results obtained. In total, we selected 22 studies summarized in Figure 3.1 according to the events.

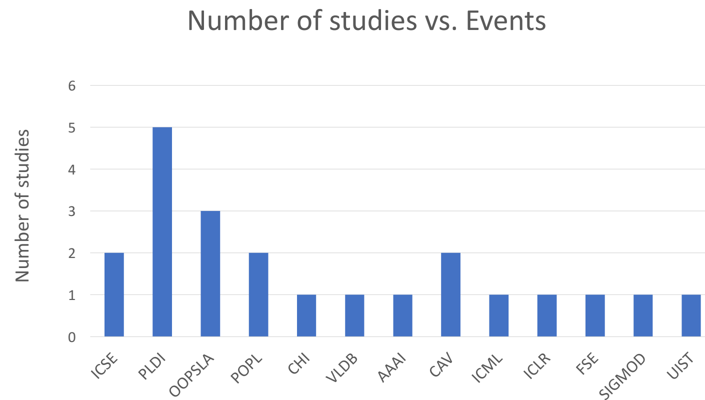


Figure 3.1: Number of studies for each event.

Figure 3.2 depicts the published studies grouped by five years. The majority of studies selected were published in the last five years. Also, there has been an increase on the number of studies when we consider the last fifteen years.

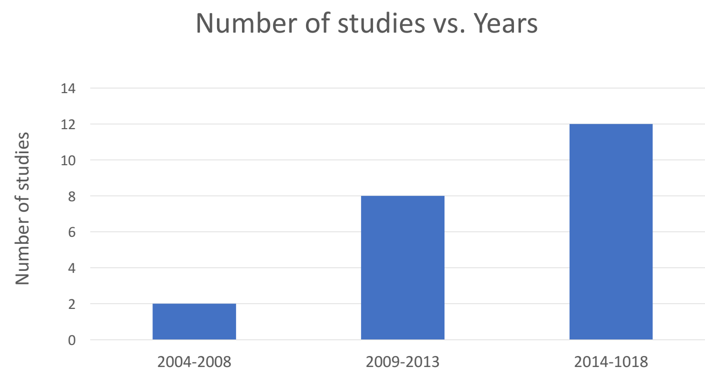


Figure 3.2: Number of studies by year.

In Figure 3.3 we depict the number of studies by application domains. The majority of studies selected are divided between *string* and program transformation domain. We also observe that the heuristic ranking based approach predominates and are employed basically in string and program transformation.

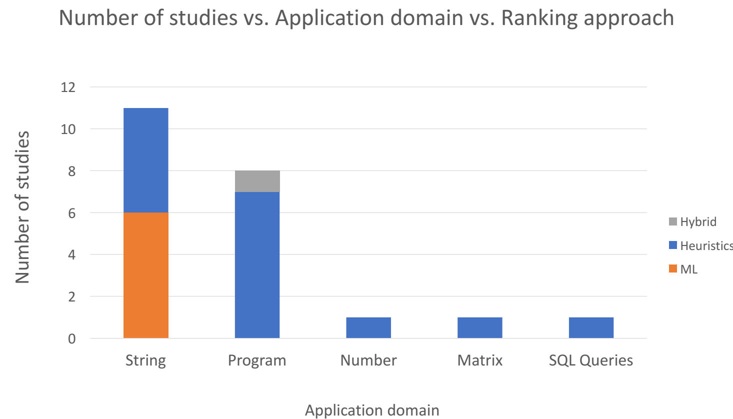


Figure 3.3: Number of studies by application domain and ranking approach.

3.3.1 Selected Studies

After gathering all the selected studies reported in the section of the results, we performed the information extraction process. This process was carefully performed aiming to help us answer our RQ. We focused on the domain of application of the tools, the type of ranking and the criteria used in the ranking approaches for multiple solutions.

Study	Year	Tool	Domain	Ranking	Criteria
[7]	2017	REFAZER	Program	Heuristics	Size of program transformations, presence of context and frequency of occurrences of characteristics (constants, references, nodes, patterns, operations)
[30]	2005	PROSPECTOR	Program	Heuristics	Size (length) and simplicity of code snippets.
[43]	2012	-	Program	Heuristics	Similarity between suggested and known expressions (type distance, depth or size, in-scope static methods, common namespace and same name)

[52]	2006	XSnippett	Program	Heuristics	Size (length) of code snippets, frequency of occurrences of characteristics and context.
[9]	2014	CodeHint	Program	Heuristics	Frequency of certain types, methods, and fields in real-world lines of code.
[11]	2011	FlashFill	*String*	Heuristics	Size of substrings and simplicity (number of arguments and whether arguments are pair-wise)
[21]	2011	Wrangler	*String*	Heuristics	Types of transforms, specification difficulties, frequency, length of selected text, frequency of equivalent transforms and measure of transform complexity.
[56]	2016	BlinkFill	*String*	Heuristics	Context of token sequences with the contexts around them in the transformation.
[68]	2016	FIDEX	*String*	Heuristics	Generality of tokens (a general token has a higher score than a constant token)
[47]	2017	-	*String*	Heuristics	Correspondence between different programs (maximal collection of field-level extractions that align well with one another)
[59]	2015	LearnRank	*String*	ML	Frequency-based features denoting frequencies of patterns in the programs.
[32]	2013	-	*string*	ML	A probability model defined over training data assigns a probability to each program based on characteristics in the structure, such as textual features: whether output is subtring of the input, duplicated lines in the output but not in the input, etc.
[16]	2013	InSynth	Program	Heuristics	Statistical information from a corpus of code, with more frequently occurring declarations having smaller weight.
[25]	2013	SMARTedit	*String*	ML	A probability is assigned to each hypothesis, which are formulated from a demonstration of a task. The user can interact with the most probable hypothesis and choose one.

[46]	2013	RESYNTH	Program	ML/ Heuristics	Refactoring sequences are obtained with a minimization of the edit distance and expression distance from the user edits.
[26]	2017	S3	Program	Heuristics	Syntactic and semantic differences between candidate solutions and original expression in the AST, observing operations, number of occurrences of node types, variables and constants.
[15]	2014	NLyze	*String*	ML	Scores associated with specific rules used to produce the candidate solution and their occurrences, how completely produced expressions cover words in the input.
[70]	2013	STEPS	*String*	ML	A probability model defined over training data assigns a probability to each program based on characteristics in the structure, such as textual features: whether output is substring of the input, duplicated lines in the output but not in the input, etc.
[58]	2012	Excel add-in	Numbers	Heuristics	Gives preference to smaller and simpler solutions, favoring less trailing zeros and whitespaces, unnecessary number truncation and number formatting expressions over rounding transformations
[66]	2017	SCYTHE	SQL Queries	Heuristics	Favors simpler structures and filter predicates, natural predicates more seen in practice and constant coverage.
[67]	2017	SYNGAR	*string*/ matrix	Heuristics	Favors small programs over larger ones. In case of tie, favors use of smaller constants.
[14]	2017	-	*String*	ML	properties such as length of the program, number of operators, etc.

Table 3.5: Summarizing the results

3.4 Discussion

In this section, we discuss the primary studies selected, giving attention to the application domains, the approaches used by different PbE tools and techniques, and the main features along with more details about them.

3.4.1 Programming by Example approaches

Ranking multiple solutions consistent with an example-based specification has been a great barrier to the success of PbE tools. Even though this study explores a specific topic in the area of program synthesis from examples, findings on this topic have a lot to contribute to the development of approaches and tools for this context. Investigating different ranking approaches used by PbE tools in distinct domains may contribute with relevant insights on how to build a more efficient ranking function for the context of program transformations.

The number of studies retrieved in our search confirmed a recent increasing effort in the discussion and proposal of PbE tools and techniques aiming to support different types of users dealing with various tasks. Our interest was to find distinct tools and techniques that employed ranking approaches and investigate the strategies they used to construct proper ranking functions to sort multiple solutions. From this investigation, we aimed to obtain relevant insights on what strategies can be useful to in the construction of efficient ranking approaches. With that in mind, we have filtered all the retrieved studies extracting insights from only 22 selected studies, which seemed to have the quality necessary to help us answering our RQs.

Application domains

The final selected studies did not present a diversity in terms of domain of applications of PbE tools. Regarding these tools, they operate mainly in two domains, being 8 studies in program and 11 in *string* domain. In the program domain, it was observed that there is an effort to support developers dealing with various forms of edits in the source code [7; 30; 52; 9; 16; 43]. Also, there has been initiatives in supporting programming students through program transformation to provide useful feedback in the education scenario, which seemed to be a recent topic investigated in the selected studies [7]. On the other hand, in the *string* transformation domain, the support is mainly given to end users dealing specially with data wrangling tasks, which has been investigated for a much longer time [44; 21; 56; 25; 70]. We also observed that there is an effort to provide support to matrix and numbers transformations, which is slightly different from the application of *string*, considering their nature.

For instance, in matrix transformation domain, SYNGAR [67] operates by transforming matrices from distinct dimensions, such as the ones performed in MATLAB, using examples. In numbers, the attempt is to learn number transformations from examples, using a language specially designed for that purpose, for instance, changing dates of the year or time of events [57].

Ranking approaches

Regarding the ranking approaches employed by the tools, the selected studies presented two main types of strategies adopted, which are based on heuristics and ML concepts. Both heuristics and ML concepts are used in program and *string* transformations presenting positive results according to the studies. The use of human expert knowledge to guide the ranking of the multiple transformations has been investigated by studies dating from earlier years [30; 52], while the application of ML concepts relies on a more recent topic, specially concepts of Neural Networks and probabilistic models [32; 59].

The majority of the ranking approaches observed in the studies relies on specific properties of grammar of domain specific languages used to describes the transformations. Based on the observation that some properties are more desirable that others, these properties are used to disambiguate between multiple programs. We focus our attention on these specific used properties and how they were used to disambiguate between the multiple transformations. A general approach to distinguish between different transformation consists of assigning them a score. This score is computed by assigning weights to specific properties, which according to the approaches reported in the studies, is done manually, based on heuristics, and automatically, generated by ML algorithms. We now discuss some of the main properties used by the reported ranking approaches across different domains: size, frequency, simplicity, context, generality, tree edit distance, and others.

Ranking criteria

Properties such as *size* and *simplicity* are commonly adopted in the greater part of the ranking approaches across different domains reported in the studies [7; 30; 52; 44; 57; 67]. Usually, they relate to the length of the transformation that are measured by using some type of criteria such as the number of tokens or nodes in the AST. As observed, the ranking approaches based on heuristics usually give preference to shorter transformations, following the Ocam's razor principle, i.e., the principle that sates the simpler solutions are more likely to be the desired ones. The studies seem to be categorical when they affirm that shorter solutions are more likely to satisfy the desired intent and encouraging results are reported to support that type of affirma-

tion. Another common property relies in the notion of *frequency*, which may have distinct usages depending on the domain of application. Some studies refer to it as the number of occurrences of a specific property in the transformations and the number of times a certain snippet appears across different locations in the search, in mining code context, for example [7; 9; 59; 26]. *Context* refers to the presence of expressions surrounding a modified location. Usually these expressions represent a pattern used in the matching. The notion of *generality* consists of giving higher or lower weights to more general expressions, which are the ones that match more general sequences of tokens. In this case, it was observed that two studies expressed different views on this heuristic. One of the studies gave preference to more general expressions [68], while the other favored more specific or constant expressions [7]. Even though they diverged in domain and perspective, both of these studies presented potential results making use of this heuristic.

The DSL plays an important role in the generation and ranking of program transformations. The language has to be expressive enough to represent a wide space of tasks in a specific domain and restricted enough to allow efficient search over that space. The language consists of a set of operators and a grammar to describe how to properly make use of the operators. The grammar of different DSLs may be expressed and constructed over distinct operators or properties, considering the characteristics of the application domain. For example, in the *string* transformation domain, properties inherent to characters are relevant, such as substring operations, character positions, concatenation, and others. Some of these properties do not fit in the program transformation domain. When ML models are employed, such as Neural Network, for example, the importance of these properties are not predefined, but given automatically.

The notion of specific properties in the underlying application domain indicates that a ranking based on general properties may be not enough to guarantee an efficient ranking. However, according to results observed, the most general properties previously mentioned seem to have the greatest contribution to the success of a ranking approach across domains. We have focused on the most common and general properties that transcend the domains, and thus can be used in the construction of more general ranking approaches.

3.4.2 Study limitations

We have designed our search *string* to automatically find the maximum number of studies dealing with ranking approaches employed by tools to sort multiple program transformations in different databases. Given the simplicity and generality of the *string*, it is possible that we have missed studies that used different terminologies from the three first collected studies we have based to extract the

search terms. To mitigate that limitation, we have performed snowballing to find additional studies, even though we consider the lack of these studies is due to the *string* formulation process.

We did not include technical reports and graduate thesis assuming that the journals and conference papers have valuable studies to provide a reliable systematic review and state of the art. Even though we did not include those sources, we are aware that they may have relevant, and in some cases, cutting edge material that could contribute with the topic.

The planning and execution of the reviewing process was mostly performed by a single reviewer. The risk of bias in that case is real but to mitigate that limitation, we have tried to discuss any obstacle or unexpected outcome in any of the phases of the reviewing process with a specialist. Also, the first three studies were discussed with an specialist to construct a common understanding and the resulting selected studies were reviewed by the same specialist.

3.5 Conclusion Remarks

PbE tools are proposed to support developers dealing with repetitive edits. Given the underspecified nature of the examples, these tools have to deal with ranking of multiple inferred program transformations. The construction of ranking functions does not consist of a trivial task, and the problem of building an efficient ranking has been handled using different approaches across distinct domains. Our purpose relied on identifying and describing different approaches PbE tools employ to rank multiple learned transformations in different domains.

The state of the art provided us with important insights on the construction of ranking functions. We could observe that the underlying specific application domain has shown to have an influence on the construction of an effective ranking function. Distinct application domains, such as *string* or program transformation domain, involve common and distinct properties. The common properties involved size, simplicity, frequency, context, generality, and edit distance. These properties are desirable across distinct domains, as observed in the studies. Even though many of them have greater contribution than others to the success of a ranking, they are supported by specific properties.

Regarding the ranking approach, two main strategies were observed: heuristics, based on human expert knowledge of the application domain and ML, based on knowledge automatically extracted from the domain. Studies that presented the ranking based on heuristics made explicit the relevance of the properties of size, simplicity and generality, assigning them more weights. However, finding these weights manually using heuristics and domain knowledge is costly and time consuming.

To efficiently construct a ranking approach disregarding the application domain, the most general

properties size, simplicity and generality showed to be necessary. However, we have to consider the characteristics of the application domain, as well as, the DSL for that specific domain, in order to extract other properties to support the most general ones. When considering to manually assign weights or doing so automatically, these general properties are the ones to receive more weights.

3.6 Answers to the Research Questions

Next, we summarize the answers of our research questions.

- RQ_1: What approaches do Programming-by-Example tools employ to rank multiple learned transformations in different domains?

  Regarding the ranking approaches employed by PbE tools, two main strategies were observed: heuristics, favoring specific properties based on the domain knowledge of a human expert, and ML-based, using weights automatically obtained from training examples.

- RQ_2: What are the most common features employed by Programming-by-Example tools to build ranking functions for ranking transformations in different domains?

  The main properties used to build ranking functions for transformation consisted of size, simplicity, frequency, context, generality, and edit distance. However, the influence of each one of them may vary according to the problem domain.

Chapter 4

A Machine Learning Based Ranking Approach

In this chapter, we present our approach for ranking program transformations, based on machine learning techniques to learn ranking functions. In Section 4.1 we present the overview of the steps of the ML approach for program transformations.

4.1 Overview

We propose a supervised ML-based approach to automatically build ranking functions to rank program transformations for PbE techniques.[1] Figure 4.1 presents the workflow of the proposed approach, which comprehends four steps. It receives a PbE tool, examples of edits, and ML algorithms as input. In Step 1, we use an approach to automatically label the transformations learned by the PbE tool from the edit examples. The edit examples include the before and after version of the code, which can be used to judge whether the edits applied by the transformations correspond to what the developer edited. We compare the edits performed by the transformations with the edits applied the developers using a test suite. If the edits are syntactically the same, the transformation is labeled as *correct*. If the transformation edits incorrectly, or edits more or less locations than needed, it is labeled as *incorrect* (Section 4.2). In Step 2, we select 12 features based on the literature [7; 30; 11; 26; 16; 14] and on operators of transformation Domain Specific Language (DSL). The values of the features are the number of their occurrences in the transformations (Section 4.3). In Step 3, using the 10-fold cross-validation method, we train and test three classifiers commonly used in practice, SVM,

[1] https://sites.google.com/view/ml-to-rank-transformations/

LR, and NN (Section 4.4). Finally, in Step 4, we instantiate the model weights in a PbE tool for ranking transformations. We use REFAZER since we are able to change its ranking scheme in order to test other ranking functions (Section 4.5).

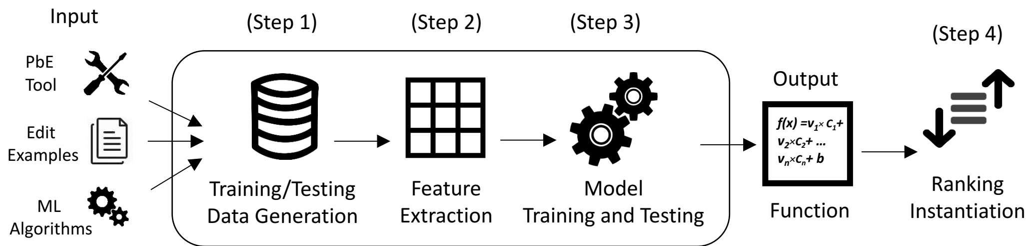


Figure 4.1: Workflow of the proposed ML-based ranking approach. The inputs consist of a PbE tool, edit examples, and ML algorithms. Step 1 comprehends the generation of training and testing data, Step 2 consists of feature selection and extraction, and Step 3 consists of training and testing models. The output of the ML approach consists of a model, or a function, which is instantiated in a PbE tool in Step 4.

4.2 Training/Testing Data Generation

A set of edits applied by the developers that follow a similar structure, thus representing a repetitive task are referred as an edit scenario. Different edits applied by them imply in distinct edit scenarios and these scenarios of edits are used as an input to the approach. We have a benchmark with scenarios of code edits, where each scenario contains a set of locations edited repetitively by developers in some GitHub repositories. These scenarios include the before and after version of the edited locations. In Step 1, we use REFAZER to learn a set of transformations from the examples of edits, which includes correct and incorrect transformations.

In this step, we propose a labeling approach to automatically label the transformations learned. In this approach, each transformation in the set of transformations is applied to the other unseen locations. We compare the edits from the transformation applied to unseen locations with the edits performed by the developers described in our benchmark. If the transformation edits are exactly the same, we label the transformation as *Correct*. On the other hand, if the technique does not locate or does not edit a target location in our benchmark correctly, we label it as *Incorrect*. A transformation is also considered incorrect if it edits more locations than needed (i.e., more locations than those described in our benchmark).

For instance, consider the edit applied the developer in 18 non-identical locations in Entity Framework repository in C# [33] in Figure 4.2. The task here consists of changing the locations with the pattern "*identifier*.`GetModel()`". The edit consists of invoking a new method `VersionedModel()` passing the *identifier*.`GetModel()` as its argument. However, let us consider that a PbE tool has to infer a transformation from only one example, such as the first location (Location 1). Some of the inferred transformations with this example can be described as follows:

```
1  - Owner.GetModel()   // Location 1
2  + VersionedModel(Owner.GetModel())
3    context.GetModel()   // Location 2
…        …
18   Owner.GetModel()   // Location 18
```

Figure 4.2: Before and after version of code

- p_1: change the locations that follow the pattern `Owner.GetModel()`, invoking a new method `VersionedModel()` passing `Owner.GetModel()` as its argument.
- p_2: change the locations that follow the pattern *identifier*.`GetModel()`, invoking a new method `VersionedModel()` passing *identifier*.`GetModel()` as its argument.
- p_3: change the locations that follow the pattern *identifier.expression*, invoking a new method `VersionedModel()` passing *identifier.expression* as its argument.

Each of these transformations inferred from the edit example (Location 1) gets applied to all the other 17 locations of the source code and we test the outputs produced, comparing them with the ones in our benchmark. The comparison is given by test cases, which are constructed based on the edits applied by the developers. To be considered correct, it has to produce the correct output specified by the developer. Whenever it fails to produce the desired output for any of the locations, it is labeled as incorrect. In addition, if it produces the correct outputs for all 17 locations, but also edits additional unneeded locations, is it considered incorrect. In this scenario, any transformation that generalizes the function call `GetModel()` to any other function call, such as `Owner.GetValidationErrors()`, edits more locations than needed, thus being labeled as incorrect.

4.3 Feature Extraction

In Step 2, we select a set characteristics of the transformations, a process known as feature selection. The features are selected based on the literature, that points out some relevant characteristics such as size, context, generality, etc., and on operators of transformation DSL. Intuitively, we realize that certain properties of the programs are more desirable than others. For instance, depending on the applications domain or the problem, a more specific transformation could be more desirable, and, in other situations, a more general is more desirable to perform the edits desired by the developer. Context is also another aspect of the transformation that can be desirable, when the developer intends to edit only locations inside a specific context.

We propose a set of features based on some of these desired properties, being the majority of them pointed out in the literature [7; 30; 11; 26; 16; 14]. We observe the DSL operators as relevant candidates for the feature selection process. To select the features, we pick the grammar properties assumed to have relevant information to construct the function. Our set of features cover the majority of the DSL operators. This strategy has been used in literature, which selects properties of the grammar to help in the ranking functions [59; 11]. These features vary according to the problem domain. Even though some problem domains share common properties, such as size, this process cannot be automated, since distinct DSLs have their own particularities. Thus, to evaluate the approach in program transformation domain using REFAZER to generate the transformations, we select the following 12 features based on its DSL. It is worth mentioning that, in this step, the feature selection process may include more or less features, or even different ones, and their relevance will be tested in other additional steps. The more features selected the more options to be evaluated in terms of relevance.

- *ConstNode* represents the nodes that create a sub-tree from scratch, without any reference to the nodes in the input specification. The nodes are created only using expressions found in the AST output specification.

- *Reference* represents the nodes that create a sub-tree making references to the input specification. The nodes are created using the expressions found in the AST input specification.

- *Variable* represents the abstract node in the tree. It matches the types of the node instead of the content of the node.

- *Context* represents the presence of expressions surrounding the node modified, which constitutes a location. This location contains a pattern that holds some expressions to be matched to a particular sub-tree in the AST.

- *Size* represents the size of the tree, in terms of the total of the nodes that constitutes it.
- *Nodes* represents all the nodes in the AST that have children, which means, all the nodes that are not terminal ones.
- *ParentOne* represents the nodes being connected to one parent node. In this case, the surrounding context of this node has only one node or expression that can be matched.
- *ParentTwo* represents the nodes being connected to a grandparent node. In this case, the surrounding context of this node has two nodes or expressions that can be matched.
- *ParentThree* represents the nodes being connected to great-grandparent node. In this case, the surrounding context of this node has three node or expressions that can be matched.
- *NodeItself* represents the node that stands by itself, which means that the node has no surrounding context to be matched.
- *Patterns* represents a single or multiple expressions that compose the surrounding context of a node. These surrounding expressions compose a sub-tree used to be matched using filtered nodes from the input.
- *Operations* represents tree operations such as delete, insert and update applied to the nodes of the input AST that generate a new AST corresponding to the desired modifications in the specification.

We automatically assign values to these features based on the number of times they occur in the transformations, building feature vectors. For instance, consider Figure 4.4. It represents a program in the DSL generated from input-output examples and is consistent with the description of program transformation p_2. We present it with the purpose of exemplify how the feature vectors are computed. We can build a feature vector according to number of occurrences of the features in the program. For instance, for the transformation in Figure 4.4, the following features vector can be built, which is presented in Figutre 4.3.

4.4 Model Training and Testing

In Step 3, we train and test the ML model given our feature vectors. Since we have a supervised learning problem, which learns from labeled training data, we focused on commonly used and studied supervised learning models [5]. One of the major challenges in learning supervised ML models is

```
<transformation>: Transformation(<rule>)
<rule>: EditMap(\target => (operation, location)
<operation>: Update(x, <ast>)
x: target
<ast>: <const>| <ref>
<const>: Node(Object CreationExpression, NList(ConstNode(new),
NList(Node(IdentifierName, SN(ConstNode(VersionedModel))),
SN(Node(ArgumentList,SN(Node(Argument)))))))
<ref>: Reference(target, <match1>, 1)
<match1>: Context(<pattern1>,<path1>)
<pattern1>: Context (Abstract("InvocationExpression"))
<path1>: "."
<location>: EditFilter(\ x => Match(x, <match2>), AllNodes(node,
"PostOrder"))
<match2>: Context(<pattern2>,<path2>)
<pattern2>: Concrete (= context.GetModel()),
<path2>: "/[1]"
```

Figure 4.3: A program in the DSL generated consistent with input-output examples

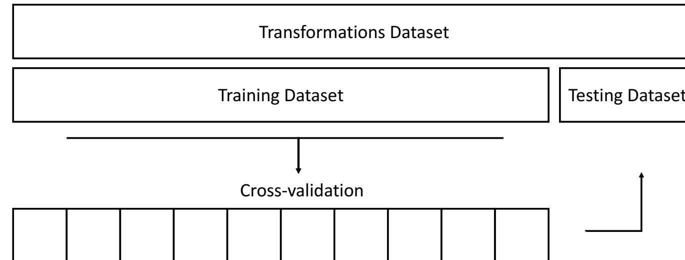
Features	Value	Features^2	Value	Feature^3	Value
ConstNode	2	ConstNode^2	4	ConstNode^3	8
Reference	1	Reference^2	1	Reference^3	1
ConcretePattern	1	ConcretePattern^2	1	ConcretePattern^3	1
AbstractPattern	1	AbstractPattern^2	1	AbstractPattern^3	1
Node	1	Node^2	1	Node^3	1
Pattern	0	Pattern^2	0	Pattern^3	0
ParentOne	1	ParentOne^2	1	ParentOne^3	1
ParentTwo	0	ParentTwo^2	0	ParentTwo^3	0
ParentThree	0	ParentThree^2	0	ParentThree^3	0
NodeItself	1	NodeItself^2	1	NodeItself^3	1
SizeProg	25	SizeProg^2	625	SizeProg^3	15625
NumberOp	1	NumberOp^2	1	NumberOp^3	1

Figure 4.4: Example of feature vector for a program in the DSL

dealing with over-fitting. To avoid over-fitting, we used cross-validation, a technique that successively divides the data in training and testing datasets, as seen in Figure 4.5. We selected the k-fold cross-validation [24], which is commonly used in practice with k set to 10. It splits the training dataset in 10 equal sized subsets. The first subset is treated as a validation set, in which is evaluated the model fit on the other 9 subsets. Each subset is given the opportunity to be treated as a validation set for the model trained on remaining 9 subsets, thus this process is repeated 10 times. At the end, the model

with the best performance is used. This strategy allows us to access how the model generalizes to unseen data.

Figure 4.5: Cross-validation strategy for estimating the performance of the model



Once we have trained our model using a cross-validations strategy, we apply it to a testing dataset, which is not included in the training dataset, as see in Figure 4.5. We want a model that is able to fit the training data presenting a high precision and recall while being able to generalize to unseen data, thus to access how accurately the models perform on the unseen data, we have focused on f-measure. The test set works as a model's validation approach and can be used to analyze how well the model can generalize to data that are not present in the training dataset. In practice, the testing dataset consist of around 25% of the original dataset [54].

When we separate the testing dataset from the training data, we have to be sure not to include only one type of transformation, such as only correct or incorrect transformations. In order to avoid that our testing dataset consists only of one type of transformation, we have used a stratified approach to divide our data. A stratified approach allows to divide the data such that classes are equally balanced in both training and testing datasets. The mean of correct and incorrect transformations of the original dataset remains in the training and testing subsets to avoid situations in which we could only have correct or incorrect in one of the classes.

4.5 Ranking Instantiation

The output of the Model Training and Testing step consists of a function. The function is given by a set of coefficients, which are the features, along with corresponding weights, which are real number that represent the relevance of each feature in the classification. Each feature receives a weight. Thus, in Step 5, we substitute the original ranking approach of a PbE tool by the function obtained in the ML-based approach. The output of our proposed approach, the function, works as the input for the tool, which used the ranking function to order the transformations. This process consists of evaluating

each of the learned transformations assigning them ranking scores by employing the ranking function. The function receives a transformation as an input, evaluates it regarding its properties, and assigns it a real number, the score, which corresponds to how correct is that transformation. Each of the transformations receives a score, which is used to differentiate between them. The output of the tool consists of an ordered set of transformations.

Since REFAZER works in the program transformation domain and allows us to modify its ranking scheme testing other approaches, we select it and evaluate the ML-based ranking approaches. The ranking scheme allows us to test different ranking approaches by substituting its original weights to new ones, a process the we call instantiation. The ranking scheme is linear, follows a tree structure and the function is computed in a bottom-up fashion, where the score of a node is computed in terms of the score of its children.

For instance, consider the `Context` node in Figure 4.6. This node has two expressions `pattern` and `path` as its children. Let us suppose that the `pattern` expression has a weight 0.2 and the `path` expression has a weight 0.1. The score of the `Context` is computed in terms of the values of its children' expressions, applying mathematical operations (e.g., summing-up these values $0.2+0.1 = 0.3$ and multiplying these values $(0.2+0.1)*0.1 = 0.03$. The function to compute the `Context` node is characterized by: `Context` = (`pattern` + `path` * `Context`). The score for the `Context` and other bottom-up nodes flow-up in the tree, helping to compute the score for the node immediately above it until reaching the tree root node. Thus, the function to compute the immediately node above depends of the results for the computed function. The ranking function gives the final aggregated score by computing the involved mathematical operations over the weights from the external nodes back through their ancestors to the topmost node.

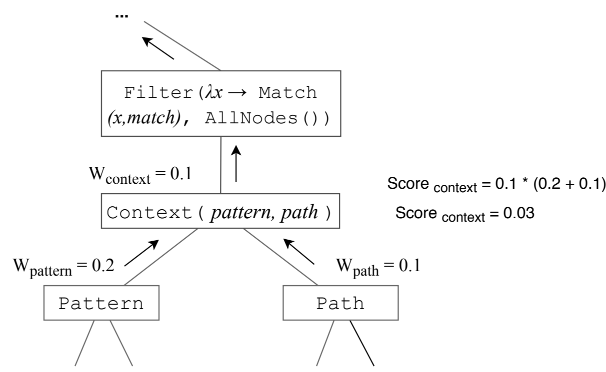


Figure 4.6: Ranking score computation process

LR and SVM models are easier to interpret than NN, and they provide a more explicit function,

thus being easier to instantiate. For instance, let us consider the SVM. In the SVM model, each feature receives a weight, which is a real number that represents the influence of that feature in relation to the others to generate a hyper-plane, a function, that best separates two classes. Thus, the instantiation consists of assigning the weights to the features in REFAZER ranking scheme, which are used to compute the ranking score. The ranking score is characterized by a probability of the transformation of belonging to one class or another, which is, of being correct or not.

LR aims to find a function that estimates the probability of an unseen observation belonging to one class or another. The weight of a feature represents its influence or relevance in relation to the others in the classification task. A higher weight means that the feature has a higher contribution in the classification. These weights are assigned to the features in the instantiation in REFAZER ranking scheme and the ranking score represents the probability of the transformation being correct.

However, NN provides a more difficult solution to interpret and more difficult to instantiate than the previous models SVM and LR. NN models are composed of layers and nodes. Each connection between nodes is associated with a weight, which in NN represents the strength of a connection. The idea is to updates these weights in order to decrease the error, or missclassified transformations. Each node represents a function, such as ReLU, that for a real value as an input, the corresponding output of the function will be 0 if the value is less than 0, or the same value given, is it greater than 0. The output of this function in a node works as an input for all the nodes in the next layer, that by its hand, works as an input for the next layer, and successively until the last layer, which is the output of the neural network. In the instantiation, the NN model receives the number of each feature, propagates its values through the nodes in the hidden layers, and outputs a class, which corresponds to the probability of the program being correct.

Chapter 5

Evaluation

In this section, we describe the evaluation of our approach. We evaluate it in real edit scenarios of C# projects of GitHub and provide a comparison with other approaches to rank program transformations. In Section 5.1, we present the goal of our evaluation along with our research questions and metrics. In Section 5.2, we show the planning of the experiments. In Section 5.3, we present the main results obtained and, in Section 5.4, we discuss our results. In Section 5.5, we describe the threats to validity and finally, in Section 5.6, we answer our research questions.

5.1 Definition

We define our experiment according to the Goal Question Metric (GQM) approach [3]. Thus, the goal of our study consists of analyzing ML-based ranking approaches *for the purpose of* characterizing them *with respect to* their efficiency and efficacy to rank correct program transformations *from the point of view of* Programming-by-Example tools' designers *in the context of* software evolution.

For this purpose, we address the following research questions:

- RQ_1: How efficiently does an ML-based ranking approach rank program transformations when compared to a manual or random ranking approach?

  Considering that an efficient ranking function can rank correct transformations in the first position with the minimum number of examples (preferably 1), we measure the number of examples required to rank the correct transformation in the first position by the ML-based ranking approaches, SVM, NN, and LR, and compare them to a HE and RW approaches.

- RQ_2: How effectively does an ML-based ranking approach rank program transformations

when compared to a manual or random ranking approach?

Two metrics for evaluating ranking quality commonly used in information retrieval are Precision@k and Normalized Discounted Cumulative Gain (NDCG@k) [31]. Here, Precision is not based on the prediction but on the data obtained after instantiated the model. Typically, users do not feel encouraged to analyze many instances of a top-k ranking positions. Thus, we focused on the 10 first positions as the more relevant ones. We measure the Precision@10 (i.e., rate of correct ranked transformations by the classifier models in top-10 positions) and NDCG@10 (i.e., instead of rewarding only the first correct transformation, all of the top-10 correct transformations are rewarded at a decaying discount factor).

5.2 Planning of the experiment

In this section, we describe our benchmark, instrumentation, training databse generation, how we trained the models, how we used negative examples, and how the ranking approaches were evaluated.

5.2.1 Benchmark

We start by building a benchmark of repetitive code edits. We have reused edit scenarios manually analyzed and obtained from three open-source C# projects from GitHub, namely Roslyn [35], Entity Framework [33], NuGet available for public usage [7]. We have selected these edit scenarios because of their availability for public usage and also because they were used in REFAZER evaluation using a ranking approach based on domain knowledge of en expert. This allowed us to compare the performance of distinct approaches with REFAZER original ranking in the same edit scenarios.

In addition, similarly to the previous reused scenarios, we have analyzed commits from other two C# projects, namely ShareX [37] and [34] Newtonsoft [36]. We compare the before and after version of the modified codes looking for syntactic similar changes. As repetitive edits, we considered those that were performed systematically by the developers and occurred in three or more locations in the codebase. We focused on similar edits, which are edits that share the same structure involving different or same expressions and present distinct granularity, involving changes in a single line and in multiple lines. Once we have selected the edit scenarios, we build a before and after file of the developer based on the diff. This file is be used as an oracle so that we can judge whether a transformation edits exactly as specified by the developer.

We used 43 distinct scenarios of repetitive edits: Roslyn (15), Entity Framework (9),

NuGet (9), Newtonsoft (6), and ShareX (4). The number of edited locations in each scenario ranges from 3 to 24, with a median of 5 locations. Each project contains at least one scenario with 8 edited locations. These scenarios along with their edited locations are characterized in Table 5.1. More details about the edits used are available in the approach's website.[1] In 31 (72%) out of the 43 scenarios, the edits are complex and context-dependent. The notion of complex edits relies on that fact that a search/replace strategy available in IDEs is not capable of correctly applying the edits to all the necessary locations automatically. For instance, one of these scenarios are characterized by the edits `(Owner.GetModel())` and `context.GetModel()` to `(new VersionedModel(Owner.GetModel()))` and `new VersionedModel(context.GetModel())`. Moreover, 35 (81.6%) of these scenarios include changes in a single line. The most common type of edit consists of changes in methods. The changes are characterized as adding, deleting or updating the methods' properties, such as names, their parameters, return statements, and others. In addition, the edit scenarios are divided in two groups according to the role they play in the study. We have the scenarios used for train/test and for evaluation. We randomly select 15 of the 43 edit scenarios, which corresponds to approximately 1/3 of it, for generating training and testing data and evaluate the models on the other 28 edit scenarios. For evaluation, we mean the step, in which, having trained and tested the model, we deploy it in a tool. We had to provide a balance between number of scenarios used for training the models and number of scenarios used for evaluation. We wanted to generate as much data as we could from the edit scenarios while preserving as many scenarios as we could to evaluate the approaches, and provide a comparison with other ranking approaches on the same scenarios.

5.2.2 Instrumentation

Given that we evaluate the ML-based approach in program domain, we have used REFAZER [7] to generate the transformations for our training dataset, given examples of edits performed in the edit scenarios. From an example-based specification, REFAZER generates multiple transformations. Originally, REFAZER employs a HE based ranking approach and is available in both C# [48] and Python [60] programming languages. In our experiment, we have used the C# version, which received more recent updates at the time of writing. The experiments were performed on a Windows, Intel Core i5 3.20 GHz CPU with 16 GB RAM.

For the purpose of learning ML models, we have used scikit-learn (version 0.19.0), an open

[1] https://sites.google.com/view/ml-to-rank-transformations/

Table 5.1: Edit scenarios characterization. Project = name of the project; Scenario = edit scenario that represents a specific task; Locations = number of edit locations; Identical = whether the edit is the same for all the location; Granularity = whether one or more lines were changed; Scope = scope of the edit; Role = whether the scenario was used for training/testing the model or for evaluating the distinct ranking approaches

Project	Scenario	Locations	Identical	Granularity	Scope	Role
NewtonSoft	NJ025	3	No	Multiple lines	Method	Train/Test
	NJ059	9	No	Single line	Method	Evaluation
	NJ224	4	No	Single line	Method	Evaluation
	NJ236	3	No	Single line	Method	Evaluation
	NJ844	11	Yes	Single line	Method	Evaluation
	NJ1491	24	No	Single line	Method	Train/Test
ShareX	S564	4	Yes	Single line	Method	Train/Test
	S583	3	Yes	Single line	Method	Evaluation
	S863	5	No	Single line	Method	Evaluation
	S1088	8	Yes	Single line	Conditional	Evaluation
Entity Framework	E1	13	No	Multiple lines	Method	Train/Test
	E3	18	No	Single line	Method	Evaluation
	E5	3	Yes	Multiple lines	Method	Train/Test
	E6	3	No	Single line	Method	Evaluation
	E7	10	Yes	Multiple lines	Method	Evaluation
	E9	3	No	Multiple lines	Method	Train/Test
	E10	12	Yes	Single line	Method	Evaluation
	E11	7	No	Single line	Method	Evaluation
	E12	3	No	Single line	Method	Evaluation
Nuget	N13	4	No	Single line	Property	Evaluation
	N14	13	No	Single line	Method	Evaluation
	N15	8	Yes	Single line	Method	Train/Test
	N18	8	No	Single line	Method	Train/Test
	N20	4	No	Single line	Method	Evaluation
	N21	11	No	Multiple lines	Method	Train/Test
	N23	3	No	Multiple lines	Method	Train/Test
	N26	20	No	Single line	Method	Evaluation
	N27	4	No	Single line	Method	Evaluation
	N28	4	No	Single line	Method	Train/Test
Roslyn	R30	21	Yes	Single line	Method	Train/Test
	R36	4	No	Single line	Method	Evaluation
	R41	15	No	Single line	Class	Evaluation
	R42	14	Yes	Single line	Method	Evaluation
	R44	12	Yes	Single line	Method	Evaluation
	R45	4	No	Single line	Method	Evaluation
	R46	5	No	Multiple lines	Method	Evaluation
	R48	11	No	Single line	Method	Train/Test
	R49	5	No	Single line	Method	Evaluation
	R50	5	No	Single line	Constructor	Train/Test
	R51	5	No	Single line	Method	Evaluation
	R53	6	No	Single line	Method	Evaluation
	R54	16	Yes	Single line	Using Directive	Evaluation
	R56	4	No	Single line	Method	Train/Test

source ML library for Python (version 3), which contains a number of state-of-the-art algorithms implemented in Python and usable for a variety of supervised and unsupervised learning problems. For finding the ideal hyper-parameters, which is known as *tuning* process, we have used `GridSearchCV` strategy, available in scikit-learn [53]. It searches exhaustively over a set of parameters values for the best estimators using cross-validation and can be used to tune models such as SVM and LR where we have to find the ideal hyper-parameters, such as C value. A very high or low value for this hyper-parameter can cause the model to be over-fitted or under-fitted. In addition, in these models, we tested both L1 and L2 regularization term, and L1 presented higher performance. Thus, we have used L1 regularization or Lasso Regression (Least Absolute Shrinkage and Selection Operator), as a means of penalty for the complexity of the function, thus avoiding over-fitting. It shrinks the values of some features to zero, which works as a feature selection.

To prevent unbalanced training database where we have more data belonging to one class than to another, which is common in classification problems, we have used a scikit-learn helper function that divides the training data in train and test subsets. This function returns stratified folds and preserve the percentage of samples for each class.

5.2.3 Training Database Generation

Now we describe how the training/testing dataset was generated. Out of 43 scenarios, we randomly selected 15 of them for generating transformations to be used as training. For each scenario, we provide REFAZER with positive examples corresponding to the edited locations by the developers. The examples are given incrementally until REFAZER is able to learn a correct transformation in the first position. It is possible, given constraints in the generalization process, that REFAZER may not be able to learn a correct transformation in the first position using all the examples. In N21, for instance, where that happens, all the learned transformations are incorrect. In Section 5.2.5 we discuss more about this scenario. Thus, the dataset of transformations inferred for each scenario are used as an input to learn ML models. REFAZER builds on PROSE, a framework that allows synthesizing program transformations. Since REFAZER can learn a huge number of transformations, which may not to fit the computer memory, we used a feature of PROSE that allows to restrict the number of the learned programs to the top-200 programs generated, except for the scenarios where the number of transformations does not reach 200. This value influences the number of inferred transformations, restricting them. The number of transformations could be higher. However, to cope with that, we need more memory power. Each transformation automatically receives a label (correct or incorrect) based on the edits in our benchmark. Thus, for the 15 pre-selected scenarios to generate

the training/testing dataset, we have 1,677 transformations, from which 1,026 were labeled as *Correct* and 651 as *Incorrect*. The number of correct and incorrect labeled transformations according to each scenario is depicted in Table 5.2.

Table 5.2: Dataset of labeled transformations. Project = name of the project; Scenario ID = ID of the edit scenario; Learned transformations = number of learned transformations; Correct = number of transformations labeled as correct; Incorrect = number of transformations labeled as incorrect

Project	Scenario ID	Learned Transformations	Correct	Incorrect
NewtonSoft	NJ025	200	136	64
	NJ1491	58	54	4
ShareX	S564	6	6	-
Entity Framework	E1	5	5	-
	E5	135	50	85
	E9	135	135	-
NuGet	N15	138	60	70
	N18	200	200	-
	N21	200	-	200
	N23	200	200	-
	N28	6	6	-
	N30	45	43	2
Roslyn	R48	106	28	78
	R50	200	110	90
	R56	43	43	-
	Total	**1,677**	**1,026**	**651**

To perform the feature extraction, we count the occurrences of the 12 transformation features, build feature vectors with the computed values and train three models SVM, LR, and NN. We added pseudo-features to obtain non-linear models, by adding new features corresponding to the second and third power of each of the 12 features, resulting in 36 features.

5.2.4 Training Models

We have tuned the models by observing their performance in the predictions using the testing dataset. This dataset consists of the 20% of the transformation dataset, randomly selected and set apart in the training step for the purpose of only testing the learned models, as seen in Figure 4.5. For this purpose, we have used a scikit-learn helper function, which divides the transformation dataset in training and

testing subsets. We set the train size in 80%. For SVM and LR, we have tuned the C hyper-parameter. For SVM, the *C* represents how much we want to avoid misclassifications, in which high values for *C* mean small margin in the hyper-plane, and low value for *C* mean large margin. Since increasing or decreasing the margin leads to the inclusion of more or less training examples, we want a value that does not misclassify and is useful. For both LR and SVM models, using the `GridSearchCV` strategy, we have tested the following values for *C*: 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 1.5, 2, 2.5, 3, 4, 5, 6, 7, 8, 9, 10. For the SVM approach, the best estimator *C* was 0.1. For LR, the best estimator *C* was 2.5. We also tested different NN architectures with different number of nodes and hidden layers, in order to find the best parameters, and the one with best performance was 3 hidden layers of 30 nodes each. The NN model receives the features as inputs, process them propagating the weights feed-forward and outputs a label.

After training and testing the model using the testing data, we achieved the f-measures depicted in Table 5.3. After this process, we instantiated the ranking approaches and evaluate them on the other 28 edit scenarios of the initial 43 C# projects. Along with the ML models, we evaluate the HE and RW approaches on the same scenarios. We added RW as a control group to evaluate whether its outcome compared to the ML-based models' ones were statistically different, meaning that the ML models' weights have an effect.

Table 5.3: Performance of models predictions

Ranking Approach	Precision	Recall	F-measure
SVM	0.88	0.88	0.88
LR	0.90	0.90	0.90
NN	0.91	0.90	0.90

5.2.5 Using Negative Examples

Until now, we have discussed how we collected the edit scenarios, the instrumentation process, how we generated the training database, and how we trained the models. Another important aspect regarding the planning of the experiment, which can be considered a contribution, consists of the use of negative examples. The literature has pointed out two ways of evaluating learning techniques from examples, which is from only positive, and positive and negative examples [4]. In addition, in PbE, the use of negative examples is an open topic in program transformations [7]. Originally, REFAZER and other PbE tools employ a ranking approach based on only-positive based specification [59;

44]. In the dataset of labeled transformations, we realize that for one specific scenario, N21, the inferred transformations are labeled as incorrect. N21 is characterized by having 11 non-identical edited locations involving more than one line. In this scenario, all the positive examples, corresponding to the number of edited locations, were given but they are not enough for the employed HE ranking approach to rank the correct transformation in the first position. One of the possibilities can be due to the HE, which may benefit more specific transformations and the scenario may require a more general transformation, or the HE ranks an overgeneralized transformation in the first position. Considering the latter, the best-ranked transformation for the giving ranking approach incorrectly applies edits to locations that should not be applied (false positive locations). We refer to them as False Positive Generators (FPG), since false positives were generated by these overgeneralized transformations. Since giving only positive example leads to overgeneralized transformations, we need an approach to prevent them, filtering them out. Thus, we implemented a solution in REFAZER so that negative examples could be used to prevent overgeneralized transformations.

To understand how efficient this solution can be when applied to other FPG scenarios, we measured the number of positive and negative examples required by the ranking approaches to rank transformations compared to an only positive example-based specification. In Table 5.4, we learn that the ranking approaches could rank the correct transformation in all the FPG scenarios by adding negative examples. Thus, we have used them in our evaluation.

5.2.6 Ranking Evaluation

For the RQ_1, we measure the ranking efficiency in terms of the number of positive and negative examples required to rank the correct transformation in the first position. The same metric has been used to evaluate ranking functions in previous work for measuring ranking efficiency [59]. Each edit scenario has a predefined number of positive examples available, which corresponds to the number of edited locations by the developer. The number of negative examples depends on the number of unneeded edited locations by an incorrect transformation ranked in the first position. To simulate what would happen in a real situation with a developer making use of a PbE tool with the ML-based ranking approach, for each scenario, we start by providing one random positive example. The transformation in the first position is evaluated whether it edits all the needed locations correctly.

For instance, consider the random positive location `if (trivia.CSharpKind() == SyntaxKind.None)` edited to `if (trivia.IsKind(SyntaxKind.None))`. With this example, a set of transformation can be learned, ranging from the most specific to the most general. For instance, consider p_1, p_2 and p_3 as transformations learned from this given example. These trans-

Table 5.4: The "FPG" represents treatments where the highest number of examples available were given but were not enough to rank the correct transformation on the first position. The symbol "-" represents the scenarios where approaches with only positive examples were able to put the correct transformation in the first position

Project	Scenario ID	Only Positive					Positive and Negative				
		RW	HE	SVM	NN	LR	RW	HE	SVM	NN	LR
NewtonSoft	NJ224	FPG	-	-	-	-	3	-	-	-	-
	NJ236	-	-	FPG	-	-	-	-	3	-	-
ShareX	S583	FPG	-	-	-	-	2	-	-	-	-
Entity Framework	E3	FPG	-	-	-	-	2	-	-	-	-
	E6	-	-	FPG	-	-	-	-	3	-	-
	E7	FPG	-	FPG	FPG	FPG	3	-	2	2	2
NuGet	N14	FPG	-	FPG	-	FPG	2	-	2	-	2
	N20	FPG	-	-	-	-	2	-	-	-	-
	N26	FPG	-	-	FPG	FPG	2	-	-	2	2
Roslyn	R36	FPG	-	-	-	-	4	-	-	-	-
	R41	-	-	FPG	FPG	FPG	-	-	2	2	2
	R42	FPG	-	-	FPG	-	3	-	-	3	-
	R44	FPG	-	FPG	-	FPG	2	-	2	-	2
	R46	FPG	-	-	-	-	2	-	-	-	-
	R49	FPG	-	-	-	-	2	-	-	-	-
	R51	-	-	-	FPG	-	-	-	-	3	-
	R54	FPG	-	-	-	-	2	-	-	-	-

formations are ranked using a ranking approach. Suppose that the ranking approach ordered 1st: p_2, 2nd: p_3, and 3rd: p_1. We than evaluate the p_2, the first position, regarding whether or not it edits all the other needed locations similar to the example given. If it does not edit all the needed locations, we add another positive example for refinement purpose and reevaluate the transformations. However, if it edits all needed locations, we also evaluate whether, in addition of the needed locations, p_2 also edits unneeded locations. If it does, we start providing a random negative example corresponding to the unneeded edited locations by that transformation. But if it edits only the needed locations correctly, we stop the process and consider p_2 a correct transformation. For the RQ_2, we measure the accuracy and relevance of the ranked transformations. Thus, we provide only one positive example to all approaches. Our purpose is to evaluate how accurate and relevant are the top-10 transformations ranked with the minimum number of examples available, which is one. Providing more than one would benefit some approaches over the others, thus the specification has to be the same for each one. For instance, given the aforementioned examples, we evaluate p_1, p_2, .., p_{10}. However, whenever any of the transformations edits more or less locations than needed, instead of giving another example, we consider it an incorrect transformation.

5.3 Results

Table 5.5 summarizes our results for our RQ_1. Compared to the HE, which is our baseline, the LR ranking approach is the most efficient one, requiring the mean of 1.67 and median of 2 examples to rank the desired transformation in the highest position of the rank. NN provides a more efficient solution than a SVM with mean and median of 1.76 and 2 examples, respectively, in despite of SVM with mean of 1.78 and median of 2 examples. When compared to a RW ranking approach, LR provided a superior performance in terms of examples. The RW approach required the highest number of examples on average overall, with mean of 2.32 and median of 2.16 examples.

Table 5.5: Comparison of the number of examples required by ranking approaches. N/A represents the absence of the value due to memory errors and other types of issues

Project	Scenario ID	Positive and Negative				
		RW	HE	SVM	NN	LR
NewtonSoft	NJ059	2.3	3	2	2	2
	NJ224	1.6	1	1	1	1
	NJ236	3	2	3	N/A	2
	NJ844	1.6	1	1	1	1
ShareX	S863	2.3	3	2	3	1
	S583	2.6	2	2	2	2
	S1088	1.3	1	1	1	1
Entity Framework	E3	2.6	1	1	1	1
	E6	3	2	3	N/A	2
	E7	2.3	1	2	2	2
	E10	1.6	1	1	1	1
	E11	4	3	3	3	3
	E12	1.3	1	1	1	1
NuGet	N13	1	2	2	2	2
	N14	2	1	2	1	2
	N20	2.6	2	2	2	1
	N26	1.6	1	1	2	2
	N27	2.5	2	2	N/A	2
Roslyn	R36	3.3	2	2	2	2
	R41	1	1	2	2	2
	R42	3	2	2	3	2
	R44	1.6	1	2	1	2
	R45	3	2	2	2	2
	R46	2	1	1	1	1
	R49	2	1	1	1	1
	R51	2	2	2	3	2
	R53	5	3	3	3	3
	R54	2	1	1	1	1
	Mean	**2.31**	**1.64**	**1.78**	**1.76**	**1.67**
	Median	**2.16**	**1.5**	**2**	**2**	**2**
	St. Deviation	**0.88**	**0.73**	**0.68**	**0.77**	**0.61**

In order to have a more consistent idea on how an approach based on randomly generated weights rank correct transformations, we have generated three distinct random ranking approaches. For each of these three random approaches, different weights were computed. This number allows us to understand how the number of examples vary from one random approach to another. In cases where a random approach was not able to rank the correct transformation given memory constraints, we have used a strategy pointed out in the literature. We have computed the mean of the values of those that were able to rank the correct transformation in the first position and imputed in the missing values, which is a method commonly applied to handle missing data [49]. We collected the number of examples required by each approach and performed an aggregated of the means, which is depicted in Table 5.6. The mean of all the means for the evaluated scenarios was 2.31 and median 2.16.

Table 5.6: Comparison of the number of examples required by distinct random approaches. N/A represents the absence of the value for memory errors and types of similar issues

Project	Scenario ID	Comparing Random Approaches			
		R1	R2	R3	Mean
NewtonSoft	NJ059	1	3	3	2.3
	NJ224	2	1	2	1.6
	NJ236	N/A	3	N/A	3
	NJ844	2	1	2	1.6
ShareX	S863	2	3	3	2.6
	S583	3	2	2	2.3
	S1088	1	2	1	1.3
Entity Framework	E3	3	2	3	2.6
	E6	N/A	3	N/A	1.6
	E7	3	1	3	3
	E10	2	1	2	1.6
	E11	N/A	4	N/A	4
	E12	2	1	1	1.3
NuGet	N13	1	1	1	1
	N14	2	2	2	2
	N20	3	2	3	2.6
	N26	2	1	2	1.6
	N27	3	2	N/A	2,5
Roslyn	R36	4	2	4	3.3
	R41	1	1	1	1
	R42	4	2	3	3
	R44	2	1	2	1.6
	R45	3	3	3	3
	R46	2	2	2	2
	R49	2	2	2	2
	R51	N/A	2	N/A	2
	R53	N/A	5	N/A	5
	R54	2	2	2	2
				Mean	**2.31**
				Median	**2.16**
				St. Deviation	**0.88**

We have performed a statistical test t-test for evaluating whether there is a difference among

ranking approaches. A t-test is one of the most common tests, which analyzes whether the means of two groups are equal to each other [64]. We used a paired test with confidence interval of 95% for all the comparisons and our results can be seen in Table 5.7. Statistically, the LR and HE ranking approaches are not significantly different from each other, with p-value of 0.76, which indicates that both approaches have similar performance. As observed, HE, SVM, NN, and LR approaches are significantly different from RW, with p-values of 0.00001, 0.0002, 0.004, and 0.00008 respectively.

Table 5.7: Comparison of the p-values for the distinct approaches

p-value	RW	HE	SVM	NN	LR
RW	-	0.00001	0.0002	0.004	0.00008
HE	0.00001	-	0.16	0.10	0.76
SVM	0.0002	0.16	-	0.42	0.18
NN	0.004	0.10	0.42	-	0.32
LR	0.00008	0.76	0.18	0.32	-

Table 5.8 summarizes our results for our RQ_2. For NDCG@10 metric, the HE approach presented the closest results to the ideal ranking, which is 1.0, followed by LR and the other approaches. However, for the Precision, HE and LR were able to put the highest number of correct transformations on the 10 highest positions of the ranking, also followed by the other approaches. The RW approach was the one with the most inferior effectiveness overall. We have performed statistical tests for evaluating whether there is a difference among ranking approaches, which can be seen in Table 5.9. The results confirm how similar are the ML-based approaches, specially LR, and HE approach. In terms of NDCG, the difference between both approaches present p-value > 0.05. The same result is observed for the precision metric. In addition, SVM and NN are not statistically different from HE, thus presenting a similar performance.

5.4 Discussion

5.4.1 Characteristics of the domain

In the string transformation domain, the ML-based, in average, require less examples than HE approach. Evaluated in terms of average of the number of examples required to learn the desired task,

Table 5.8: Comparison of the NDCG and Precision metrics for the top-10 positions in ranking approaches

Project	Scenario ID	NDCG					Precision				
		RW	HE	SVM	NN	LR	RW	HE	SVM	NN	LR
NewtonSoft	NJ224	0.41	1	1	1	1	0.40	1	1	1	1
	NJ236	0	0	0	0	0	0	0	0	0	0
	NJ844	0.41	1	1	1	1	0.40	1	1	1	1
ShareX	S863	0.42	0	0	0	0.87	0.29	0	0	0	0.29
	S1088	1	1	1	1	1	1	1	1	1	1
Entity Framework	E3	0.44	1	1	1	1	0.40	1	1	1	1
	E7	0	1	0.39	0.31	0.60	0	1	0.29	0.1	0.69
	E10	0.55	1	1	1	1	0.50	1	1	1	1
	E11	N/A	0	0	0	0	N/A	0	0	0	0
	E12	0.90	1	1	1	1	0.69	1	1	1	1
NuGet	N13	1	0	0	0	0.30	0.57	0	0	0	0.10
	N14	0.50	1	0.60	0.77	0.51	0.40	1	0.69	0.59	0.50
	N20	0.42	0	0	0	0.92	0.29	0	0	0	0.29
	N26	0.67	1	1	0.72	0.50	0.5	1	0.29	0.50	0.10
	N27	0	0	0	0	0	0	0	0	0	0
Roslyn	R36	0	0	0	0	0	0	0	0	0	0
	R41	0.77	1	0.50	0	0.43	0.40	1	0.50	0	0.29
	R44	0.61	0.96	0.85	0.78	0.82	0.59	0.80	0.69	0.69	0.69
	R45	0	0	0	0	0	0	0	0	0	0
	R46	0.80	1	1	1	1	1	0.66	1	1	1
	R49	0.71	1	0.95	1	0.95	0.50	0.50	0.50	0.50	0.50
	R51	0	0	0	0	0	0	0	0	0	0
	R53	0	0	0	0	0	0	0	0	0	0
	R54	0.71	1	0.80	1	0.95	0.50	0.50	0.50	0.50	0.50
	Mean	**0.22**	**0.56**	**0.44**	**0.45**	**0.50**	**0.22**	**0.50**	**0.41**	**0.38**	**0.50**
	Median	**0**	**1**	**0**	**0**	**0,5**	**0**	**0,5**	**0**	**0**	**0,5**
	St. Deviation	**0.44**	**0.50**	**0.51**	**0.51**	**0.51**	**0.44**	**0.51**	**0.50**	**0.50**	**0.51**

Table 5.9: Comparison of the p-values for the distinct approaches

p-value	NDCG					Precision				
	RW	HE	SVM	NN	LR	RW	HE	SVM	NN	LR
RW	-	0.07	0.32	0.51	0.03	-	0.04	0.17	0.33	0.09
HE	0.07	-	0.03	0.06	0.94	0.04	-	0.10	0.09	0.29
SVM	0.32	0.03	-	0.43	0.22	0.17	0.10	-	0.33	0.47
NN	0.51	0.06	0.43	-	0.11	0.33	0.09	0.33	-	0.23
LR	0.03	0.94	0.22	0.11	-	0.09	0.29	0.47	0.23	-

an ML-based solution for string transformations required 1.49 examples compared to a manual ranking as baseline, which required 4.17 examples [59]. However, in program transformation domain, we do not observe this effect. Indeed, the average of the HE approach for program transformation, which

is our baseline, already ranks the correct transformation with 1.64 examples. It implies that much effort has been given by an expert to find the ideal weights for the features so that the HE approach could achieve a high efficiency in program transformation, compared to how a HE approach performs in string domain. But we have to consider the characteristics of the domain.

In string transformation, for example, the edits are characterized by manipulating spreadsheets, such as formatting and performing name abbreviations or extracting last names, formatting phone numbers, and typically these edits must be generalized to a list of locations [11]. For instance, "`Maria Silva`" and "`Pedro Costa`" formatted to "`Silva, M.`" and "`Costa, P.`" The transformation has to generalize to match locations "`name1 name2`", updating them to "`name2`, first letter of `name1`." On the other hand, in program domain, typically we tend to have a more specific instance involved in different locations. For instance, "`a.CSharpKind() == b`" and "`c.CSharpKind() == d`" edited to "`a.isKind(b)`" and "`c.isKind(d)`". In order to produce the desired output, the transformation has to match the locations with the specific instance "`exp1.CSharpKind() == exp2`" updating to "`exp1.isKind(exp2)`", generalizing only the surrounding context expressions. Thus, the characteristics of the application domain may favor more or less generalized transformations. In program transformation, HE approach assigns higher weights than ML-based ones to specialized transformations features and provides high efficiency in this domain. On the other hand, we observed that the ML-based ranking approaches assigns lower weights to features that favors specialized transformations, favoring more generalized transformations. One of the reasons for that may rely on the fact that, in the training data, we might have more generalized transformations associated with the label *correct* than *incorrect*, making this characteristic more relevant.

5.4.2 Characteristics of the edits

Regarding the characteristics of the repetitive edits applied by the developers in our benchmark, we learn that the edits vary from simply adding, removing, updating constants in identical locations to more complex and context-dependent ones, such as invoking a new method to perform comparisons. As expected, for complex and context-dependent edit locations, ranking approaches require more examples to rank the correct transformation. For instance, consider the scenario NJ844, in which all the approaches require only one example, except for RW. This scenario is characterized by an edit `reader.Read()` to `CheckedRead(reader)` and all the other locations follow the same pattern with same expressions, thus being identical and not requiring any generalization. The same occurs in S1088 and E10. In these cases, specialized

transformations perform the user intent and, thus, need to be ranked higher. On the other hand, N20, R36, and R51, in which all the approaches require 2 or more examples, are characterized by requiring generalization. For instance, the scenario R51 is characterized by the edits `useHexadecimalNumbers:false)+"U"` and `useHexadecimalNumbers:false)+"L"` being updated to `ObjectDisplayOptions.IncludeTypeSuffix)` in both locations. The locations to be edited are not identical, involving different expressions. In these cases, generalized transformations are the ones to perform the user intent and, thus, need to be ranked higher.

5.4.3 Efficiency of ranking approaches

In scenarios E7, N14, N26, R41, and R44, HE performs more efficiently than LR. These five scenarios are the only ones in which LR is characterized by being FPG, generating false positives from an only positive example-based specification. For instance, in E7, the edits are characterized by the edits `c()` to `ExtendedSqlAzureExecutionStrategy.ExecuteNew(c)`. For these specific scenarios, LR can rank the correct transformation only by adding negative examples, which prevents overgeneralized transformations that cause false positives. Two of the scenarios (E7 and R44) are characterized by involving identical changes and, in those cases, overspecialized transformations can rank correctly. In the other three, the changes made are identical for all the locations, but they are context-dependent, which means that the context around the instance changed varies from location to location. In these cases, the generalization has to occur in the context around the instance modified. However, for these cases, overgeneralized transformations tend to generate false positives and need to be prevented by adding negative examples in ML-based approaches.

By refining the specification with negative examples, ML-based approaches are more efficient in FPG scenarios. For the scenarios where HE requires less examples than LR, LR could not rank the correct transformation in the first position, even using all the available positive examples. It reinforces the idea that ML favors more generalized transformations and, in some cases, overgeneralized transformations, which causes an increase in the number of examples for specification refinement purpose. Overgeneralized transformations generate false positives, which causes FPG scenarios. However, by adding only one negative example, the LR was able to rank the correct transformation. It was also observed in the RW approach, where the majority of the scenarios were FPG, and by adding negative examples, FPGs did not occur. Thus, by providing negative examples, overgeneralized transformations are avoided, which enables ranking approaches to rank the correct transformation in FPG scenarios. The results presented by the HE, which did not differ from only positive to positive and negative examples, can be due to the fact that the HE approach tends to benefit more specialized transformations,

naturally preventing more generalized transformations, disregarding negative examples.

Regarding the ML-based approaches when compared to HE and RW approaches in terms of efficiency, it is important to consider the edit scenarios S863 and N20. The LR approach was the only one to be able to rank the desired transformation in the highest position with one positive example. The performance of LR in these scenarios can be due to the fact that they are all characterized by requiring generalization and LR tends to benefit more generalized transformations, assigning higher weights to their characteristics, and consequently, assigning higher ranking scores to them. For instance, in N20, the two locations `CaptureWindow(wi.Handle)` and `CaptureScreenshot(CaptureType.Region)` are edited by the developer to `CaptureTaskHelpers.CaptureWindow(wi.Handle)` and `CaptureTaskHelpers.CaptureScreenshot(CaptureType.Region)`, respectively. A specific transformations in these cases would not produce the desired edits. The HE approach required one more example in these cases for refinement purpose, and probably, because the transformation in the first position was too specific, since it was not characterized by being FPG. On the other hand, RW generated FPG transformation in these scenarios and required two more negative examples to rank the correct transformation. In addition, it was also able to decrease the number of examples in NJ059 from 3 to 2 examples, compared to HE.

5.4.4 Generalization versus specialization

When we compare the ranking approaches in terms of weights used to build the ranking functions, focusing on the essential features related to generalization and specialization capacity, we learn that they differ from each other. For example, the HE approach employs a heuristic to give a higher weight to the feature *Reference* than *ConstNode*, to favor transformations that reuses nodes from the input AST rather that creating a node from the scratch. It also favors patterns that consider the surrounding context of a location. Most importantly, *Concrete* patterns receive higher weights than *Abstract* patterns, which favors transformations that match more specific edits, thus reducing the capacity of generalization of the transformations, restricting the matches to specific locations. For instance, the human expert assigns a value close to 1000 to the feature *Concrete* in HE ranking function. In the LR approach, the feature's weight indicate the size of the effect of that feature in predicting correct and incorrect transformations. L1 has shrunk the value of *Constant* and *Concrete* features to zero and with that, they are no longer relevant in predicting the transformations. While HE assigns a high value to *Concrete*, LR considers this feature irrelevant. For *Abstract* and *Reference* features, negative weights were assigned. Therefore, in LR, the capacity of generalization of the

transformations are not restricted as HE.

Regarding the distinct ML-based approaches used, NN, SVM, and LR, all of them presented a statistical difference from a RW approach but not from the baseline, HE. This implies that any of the ML-based approaches can be used to build ranking functions for program transformations, presenting a similar efficiency to the ranking approach based on knowledge domain. However, among the ML-bases approaches, the LR presented the closest mean of examples to HE. In addition, LR and SVM have the advantage of being more simple models and easier to interpret than NN, for example. NN model does not provide an intuitive representation of the relationship between the input and the outputs. Moreover, LR and SVM have presented high accuracy from a fairly small training dataset, which is a disadvantage for NN models.

Another aspect relates to the transformation themselves. For example, it is expected that all the locations in an edit scenario could be correctly edited by two or more transformations with different characteristics, ranging from more specific to more general ones. However, the transformations are evaluated only in terms of performing exactly what the developer intends, given the before and after version of the edits applied by them in our benchmark. If this goal is achieved by the transformation, editing correctly only the needed locations, it is a correct transformation. Since different ranking functions are compared, the transformations in the first position in each scenario are not expected to be the same for all the ranking approaches, but certainly is the one with the highest probability of producing the desired edits.

5.4.5 Effectiveness of ranking approaches

Regarding the effectiveness of the ranking approaches, the comparison analysis in terms of the Precision and relevance of ranked transformations allowed us to make important observations. In seven scenarios, the computed NDCG and Precision exhibit the ideal ranking for most of the approaches, which is represented by the value 1. This means that the top-10 are all correct transformations and consists of a valuable resource for interacting with the user, allowing her to choose between any of the top-10 ranked transformations. Considering all the scenarios, overall, we have the HE and LR approach presenting higher Precision and more correct transformations closer to the first position.

The reason why RW presented an inferior performance using these metrics compared to the other approaches can be explained by the fact that RW was the one to have the highest number of FPGs, followed by the ML-based approaches. FPGs can be solved by adding a negative example, thus, for some scenarios, RW and LR approaches can rank the correct transformation with one positive and one negative example. For instance, the scenario E7, where `c()` is edited to

`ExtendedSqlAzureExecutionStrategy.ExecuteNew(c).` In this scenario, HE puts all correct transformations in top-10, with Precision and NDCG 1. Since we have only one positive examples available for all the approaches, RW and ML-based ones were expected to present an inferior performance on the FPG scenarios, such as in E7, where Precision and NDCG of RW was zero, while LR results were 0.69 and 0.39, respectively. However, HE approach does not present FPGs since it favors more specialized transformations, disregarding the need of a negative examples.

5.4.6 Characteristics of ML-based ranking approaches

Observing the results obtained regarding efficiency and effectiveness, we learn that ML-based approaches are promising in building ranking functions. The results of ML-based approaches are similar to an approach based on domain knowledge of an expert, which requires effort and time to find the ideal weights and build the function manually. ML-based approaches perform similar while reducing the manual effort in building the functions. The reason why LR performed best compared to the other ML-based approaches may be due to the characteristics of the problem.

It is worth mentioning that ML-approaches also involve effort. For instance, obtaining examples of edits and selecting an appropriate tool to learn transformations from the given examples are activities that take effort. Labeling training data is also costly, even though is not done manually. The effort in selecting features is reduced since ML strategies can perform features selection. Feature extraction can be done automatically. When we know which configurations to use, training and testing models do not take so much time. In addition, the instantiation process may require effort to substitute the original weights of the PbE tool used. However, even though it takes effort to use an ML-based approach to build the ranking function, when compared to finding the weights manually by using a trial and error approach, it is advantageous. Since the number of combinations of weights and features, and distinct functions that can be obtained is so enormous, this process can take years. Even if an ML-based approach takes one month, it is a reduction of time and effort.

5.5 Threats to Validity

The number of examples had a significant role in determining the efficiency of the ranking functions in our work. The examples selected may influence the total number of examples necessary to rank the transformations. However, instead of choosing examples that cover more similar or more different patterns in the experiment, we minimized this threat by randomly selecting the examples. We also used the same seed for randomly selecting the examples so that each ranking approach could

be given the same examples, thus being treated equally. Concerning our benchmark, the collected edit scenarios contain edits with distinct granularity, ranging from perfective evolution to bug fixing and adding features. As pointed out in the discussion, the ranking approaches tend to require more examples to rank correct transformations that perform edits that require generalization. This threat was reduced since we did not prioritize edit scenarios with more or less generalized or specialized edits. Our priority in the code edit mining process was whether they shared a similar structure and occurred in three or more locations.

We have evaluated our approach in C# programming language. Even though the approach can also be used in other languages, using features based on their own DSLs, additional experiments are required to give us a more clear idea on it could work or ways to be improved. Another limitation concerns to the projects and edit scenarios. The number of scenarios used gave us evidence that ML-based approaches can be as efficient and effective as HE, but we would need more scenarios to provide a broader evaluation. With respect to the ML models, we focused on three common classifiers used in practice, but Decision Trees, K-Nearest Neighbors and Random Forests could also be viable approaches, which also need to be explored. Another direction could possibly be learning-to-rank techniques, such as RankSVM. We tried RankSVM, but it requires some effort to find an appropriate available algorithm that works with our data. Regarding REFAZER technique to evaluate the ranking approaches, it is restricted to PROSE framework an its limitations to learn program transformations. However, this threat concerns more the PbE tool than the ranking approaches employed by them.

Regarding the selected features and the number of them, more features could give a broader range of characteristics to be analyzed, and more options be analyzed as relevant or not to the ML-based models. However, this threat is reduced since we have used features pointed out in the literature as relevant across domains. In addition, each domain has their own particularities, which makes harder to find a set that can be used across any domain.

Another aspect to consider relates to the size of the training/testing dataset for the ML approaches. Given memory issues, we had to focus on 200 programs to generate our dataset, which restricts the number of transformations. Classifier models, specially the NN, typically have higher performances with a large number of training examples. The size of the training dataset in program transformations depends on the amount of learned and labeled transformations, which depends on the number of number of edit scenarios mined from code repositories. It may have had an effect on the performance of the NN in this study. However, with fairly small training database, it provided a more effective approach than a SVM in terms of number of examples required to rank the correct transformation.

5.6 Answers to the Research Questions

Next, we summarize the answers of our research questions.

- RQ_1: How efficiently does an ML-based ranking approach rank program transformations when compared to a manual or random ranking approach?

  An ML-based ranking approach provides a similar efficiency to HE approach, with LR being the most efficient among ML-based ones, requiring the mean of 1.67 example. However, ML performs similar to HE while reducing the manual effort involved by automatically finding the weights to build the ranking functions. Compared to RW, LR is superior providing a statistical difference with p-value of 0.00008.

- RQ_2: How effectively does an ML-based ranking approach rank program transformations when compared to a manual or random ranking approach?

  An ML-based approach, LR, presented a similar effectiveness in terms of Precision and NDCG compared to HE, with the average of 0.50 for both metrics. On the other hand, LR presented a higher Precision and NDCG compared to the RW, with 0.22 for both metrics. Statistical difference was observed for NDCG but not for Precision.

Chapter 6

Related Work

This chapter describes the main related works. In Section 6.1 we describe the main studies on PbE techniques based on human-expert knowledge domain to build ranking approaches. In Section 6.2 we describe the main studies based on ML techniques to build ranking approaches.

6.1 Human-expert based ranking approaches

Aiming to support developers by automating repetitive code edits, Rolim et al. [7] propose REFAZER, a PbE technique that learns program transformations from input-output examples. The technique uses examples of code edits applied by developers to synthesize a program transformation that applies the same edits to other similar locations in the source code. Moreover, in the education scenario, it learns program transformations to fix new students' submissions with similar faults in the programs. To sort the set of program transformations learned and pick the correct one, REFAZER employs a ranking function. The function uses particular weights, given by an expert in the domain, to favor specific features such as: constant nodes over references in the AST, presence of context surrounding nodes and shorter transformations. The weights are used to compute the transformation ranking scores. Even though we use the idea of assigning weights to transformation features, our weights are automatically computed using an ML-based approach, which helps to reduce the manual effort in finding them testing distinct configurations to build ranking functions.

There is a set of tools that mine software repositories and APIs searching for code fragments to support programmers. Often, the code fragments found or synthesized from an example based specification do not equally satisfy the user's desired intent. To provide the best-fit code, these tools usually build ranking approaches that employ similar sorting strategies. PROSPECTOR [30] tool uses

a technique for automatically synthesizing code fragments given a query that expresses the input and output examples of the desired code. The ranking approach employed builds on a heuristic that smaller and simpler solutions, the *jungloids*, are usually the correct ones. The tool sorts the *jungloids* by their length, assigning the top ranks to the shortest ones. The idea of designing ranking functions considering specific features of the code and giving preference to shorter solutions is also shared by other tools. Perelman et al. [43] propose a technique for generating and ranking completions for partial expressions and XSnippet [52], a framework that finds all the code snippets that satisfy a given query in a code repository. To rank the multiple XSnippet uses the notion of heuristics to favor certain solutions, such as the shortest one, most frequent snippets in the source code, and also context. We do not give preference to any of the features used, since that requires specific knowledge domain. Instead, we use an ML algorithm that computes the relevance of each feature based on their simultaneous influence on the final label of the training data.

In the domain of string transformation, PbE tools support end users performing tasks in spreadsheets, such as data wrangling. FlashFill [11] has been an expressive tool in this context. This tool is capable of synthesizing a wide range of programs in spreadsheets from input-output examples. The ranking strategy adopted to rank the programs uses features based on their syntactic structure and observes the Occam's razor principle, that states that smaller and simpler explanations are usually the correct ones. Wrangler [21], a system for interactive data transformation, follows the same heuristic preferentially ranking simpler solutions. In addition, the system uses other criteria such as explicit observed types from the interaction with the user, length of selected text, specification difficulty and frequency of equivalent transformations in the corpus. Diversely, BlinkFill [56], a system that learns string transformation from spreadsheets, uses a ranking approach that gives preference to token sequences that have larger contexts around them as they are more likely to correspond to the desired transformation. FIDEX (FIltering Data using EXamples) [68] uses positive and negative examples of string edits to learn a set of filtering expressions in spreadsheets. To sort the learned expressions, the system uses an algorithm to identify the highest ranked filter expression and scores are given empirically according to their generality, thus a general token receive higher score than a constant token. Another program synthesis approach for text and web extraction infers programs given input examples and the ranking relies on the correspondence between the extractions, that is how they align with one another [47]. Similar to FIDEX, we include negative examples in our example-based specification. However, differently from these approaches, we use an ML-based approach for building ranking functions and we evaluate it in the program transformation domain.

6.2 Machine Learning based ranking approaches

Singh and Gulwani [57] propose a solution for efficiently predicting a correct program from a large number of programs induced from examples, in the string transformation domain. The solution relies on building the ranking function using automatically generated weights based on supervised machine learning, a concept of ML. The key idea is to rank any correct program higher than all the incorrect ones for each example of the task. The features used in the ranking approach includes frequency-based features, as well as boolean features and their corresponding weights. Menon et al. [32] introduce a framework for learning general programs that speed up the search and is also used for ranking. The framework is based on ML techniques and the weights are given based on textual features. The underlying idea considers that textual features may help bias the search over possible solutions providing clues about the most relevant features to the problem. The idea of assigning weights to guide the search and rank solutions is also present in InSynth [16], a tool that synthesizes code snippets. The tool generates and suggests a list of expressions of a given type. The ranking approach to sort the expressions is based on weights derived from a corpus of code. Such as, more frequently occurring declarations receive smaller weight, and therefore are favored in the ranked suggested expressions. Likewise, we use the idea of assigning weights to features, such as frequency of patterns, size, and others, to rank the transformations. However, we do not favor any of them based on heuristics. Moreover, we evaluate in the program transformation domain.

Lau [25] presents SMARTedit, a string transformation tool that allows a user to demonstrate a desired task and the tool infers and generalizes the demonstration, generating a set of hypothesis consistent with the given demonstrated actions. Through the proposition of a probabilistic framework, probabilities are assigned to each hypothesis that represents how probable the hypothesis will produce the right output. The tool presents a list of output states, ranked by probability, and allows the user to interact and choose the most consistent one. Similarly, our ML-based approach uses the idea of probabilities to disambiguate between the transformations and the higher the probability assigned by the ML model, the higher the position in the rank.

Raychev et al. [46] propose RESYNTH, an approach to refactoring based on program synthesis from examples of code edits. The approach builds on a search strategy to synthesize a sequence of invocations of refactorings functions based on examples of edits performed by users. The search engine employs an A* algorithm to synthesize the refactoring sequences, guided by a heuristic function that minimizes the edit distance and expression distance from the user edits. The edit distance is given by the number of node renames, leaf node inserts and deletes required to get from the input tree to the

target tree. The expression distance is given in terms of the number of expressions the are present in one tree that are not present in the other one. Likewise, the ranking scheme used in the evaluation of our approach follows a tree structure and the score is computed in a bottom-up fashion, where the weight of a node is computed in terms of the weights of its children. However, the weights given by the approach are not restrict to a tree structure.

Chapter 7

Conclusions

In this work, we propose an ML-based ranking approach for automatically building ranking functions. From a dataset of automatically labeled transformations, we train and test ML models that automatically compute the ideal weights to build ranking functions to rank program transformations.

We have trained and evaluated three ML-based ranking approaches from 15 edit scenarios of five C# projects and compared them with HE and RW approaches, evaluating them in other 28 edit scenarios. We evaluate their efficiency and effectiveness in ranking program transformations by measuring the minimum number of examples required to rank the correct transformation in the first position, as well as the Precision and relevance of the ranked transformations in the top-10 position with only one example. The results obtained indicated that ML-based approaches can be as efficient and effective as HE approach in ranking program transformations while reducing the manual effort in finding the weights based on the expert domain knowledge. In addition, including negative examples allowed ML-based approaches to prevent over-generalized transformations and rank the correct transformation in the first position.

Since each domain has their own particularities and properties, which reflect in the DSLs, the feature selection based on these properties for assigning automatically obtained weights is not automated. However, compared to a manual approach, the effort for selecting relevant features from the DSL to be used for training is reduced, since ML techniques, such as L1 regularization, performs a feature selection. Regarding the training data, for program transformations in C#, edit scenarios are available for generating transformations for training the algorithms [7], which obviates the need to obtain edit scenarios. However, to operate in another domain, there is a need for obtaining edit scenarios for training, selecting features to be used, and instantiation in a PbE tool, the same process used by another ML-based approach for string transformation domain [59].

For PbE tools' designers, building ranking functions based on human-expert domain knowledge involves manual effort to find the ideal weights of transformations features, which follows a trial and error approach. This task is cumbersome because the number of different combinations grows exponentially according to the number of features. By employing an ML-based ranking approach, compared to a trial and error approach, the effort is reduced since the weights are obtained automatically. Other domains, such as string transformation, also provide a promising problem domain in which our approach can be used, but additional experiments are necessary. We have to build a benchmark mining data from Excel product team and help forums, select features based on the DSL from the domain, and find a tool to instantiate the model trained.

For end-users, providing a solution that requires a minimum number of examples to rank correctly the transformations helps to obviate their manual effort in providing examples to PbE tools. Another important aspect that concerns end-users consists of providing options to them, such as showing the first 10 positions of the rank. In this aspect, an effective ranking approach that presents high accuracy in the first 10 positions becomes useful. Comparing the ML-based approaches, LR presented a superior effectiveness, which is similar to HE approach. Thus, we conclude that, in this domain, the LR presents the best ML-based approach to be used for transformations. To the best of our knowledge, there is no study using LR approach for ranking transformations, specially in transformation domain.

For future work, we intend to investigate how to automate the process of obtaining repetitive edit scenarios in the code mining process. A solution that automatically finds repetitive patterns obviates the need of manually analyzing code edits looking for similar patterns. This would allow us to reduce the effort in obtaining edit scenarios to generate a training dataset. We also intend to investigate other supervised ML-based models that were not included in this study, such as Decision Trees, Random Forests, and K-Nearest Neighbors, aiming to improve performance such as obtaining a higher accuracy and higher efficiency. Learning to rank approaches could also be used to build ranking functions for transformations, which can use semi-supervised or reinforcement learning. In addition, an unsupervised learning approach could be used to group transformations regarding their characteristics. Other categories of transformations with higher granularity also need to be explored. We also intend to gather more training data to test a Deep Learning approach. It has been used in learning problems, such as providing recommendations, which can be similar to our problem in the sense of recommending transformations based on their probability of being correct in general, but it requires thousands of observations [65].

Human-computer interaction approaches also need to be explored, which can have an influence on the ranking approach performance. We intend to analyze how the user can interact with the top-

10 ranked solutions in order to express her preference or even filter out incorrect transformations, providing negative examples visually. Instead of randomly selecting a positive or negative example for the PbE tools, such as in our evaluation, an approach to help users selecting examples also could possibly help ranking approaches to be more efficient.

Bibliography

[1] Apache Software Foundation. Netbeans IDE, 2018. `https://netbeans.org/`, Last accessed on 2018-09-11.

[2] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Pearson Education India, 1995.

[3] Victor R Basili-Gianluigi Caldiera and H Dieter Rombach. Goal question metric paradigm. *Encyclopedia of Software Engineering*, 1:528–532, 1994.

[4] Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. An overview of machine learning. In *Machine Learning, Volume I*, pages 3–23. Elsevier, 1983.

[5] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 161–168. ACM, 2006.

[6] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *ACM SIGMOD Record*, volume 25, pages 493–504. ACM, 1996.

[7] Reudismam Rolim de Sousa, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE'17, Piscataway, NJ, USA, 2017. IEEE Press.

[8] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[9] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International*

Conference on Software Engineering, ICSE'14, pages 653–663, New York, NY, USA, 2014. ACM.

[10] Google. ErrorProne, 2018. `https://errorprone.info/`, Last accessed on 2018-09-11.

[11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[12] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning*, pages 9–14. Springer, 2016.

[13] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[14] Sumit Gulwani and Prateek Jain. Programming by examples: PL meets ML. In *Asian Symposium on Programming Languages and Systems*, pages 3–20. Springer, 2017.

[15] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2014.

[16] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. *SIGPLAN Not.*, 48(6):27–38, June 2013.

[17] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std*, 1990.

[18] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[19] JetBrains. ReSharper, 2018. `https://www.jetbrains.com/resharper/`, Last accessed on 2018-09-11.

[20] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[21] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3363–3372, New York, NY, USA, 2011. ACM.

[22] Miryung Kim and Na Meng. Recommending program transformations. In *Recommendation Systems in Software Engineering*, pages 421–453. Springer, 2014.

[23] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and software technology*, 55(12):2049–2075, 2013.

[24] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[25] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, 2003.

[26] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[27] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[28] Meir Lehman and Juan C Fernández-Ramil. Software evolution. *Software evolution and feedback: Theory and practice*, 1:7–40, 2006.

[29] Meir M Lehman and Juan F Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002.

[30] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005.

[31] Brian McFee and Gert R Lanckriet. Metric learning to rank. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 775–782, 2010.

[32] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–187–I–195. JMLR.org, 2013.

[33] Microsoft. Entity framework 6, 2010. `http://www.asp.net/entity-framework`, Last accessed on 2019-06-02.

[34] Microsoft. Nuget 2, 2011. `https://github.com/nuget/nuget2`, Last accessed on 2019-06-02.

[35] Microsoft. Project roslyn, 2011. `https://github.com/dotnet/roslyn`, Last accessed on 2019-06-02.

[36] Microsoft. Newtonsoft, 2018. `https://github.com/JamesNK/Newtonsoft.Json`, Last accessed on 2019-06-02.

[37] Microsoft. Sharex, 2018. `https://github.com/ShareX/ShareX`, Last accessed on 2019-06-02.

[38] Microsoft. Visualstudio, 2018. `https://visualstudio.microsoft.com/`, Last accessed on 2018-09-11.

[39] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[40] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[41] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[42] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324. ACM, 2010.

[43] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. *SIGPLAN Not.*, 47(6):275–286, June 2012.

[44] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'15, pages 107–126, New York, NY, USA, 2015. ACM.

[45] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[46] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. Refactoring with synthesis. *SIGPLAN Not.*, 48(10):339–354, October 2013.

[47] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. January 2017.

[48] Reudismam Rolim. Refazer. `https://github.com/reudismam/Refazer/`, 2018.

[49] Donald B Rubin. Multiple imputation after 18+ years. *Journal of the American statistical Association*, 91(434):473–489, 1996.

[50] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 1995.

[51] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach, 3rd Edition*. Pearson Education, 2009.

[52] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. *SIGPLAN Not.*, 41(10):413–430, 2006.

[53] scikit-learn developers. sklearn.model_selection.gridsearchcv, 2007. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`, Last accessed on 2019-1-28.

[54] scikit-learn developers. sklearn.model_selection.train_test_split, 2007. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html#sklearn.model_selection.train_test_split`, Last accessed on 2019-5-2.

[55] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016.

[56] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, June 2016.

[57] Rishabh Singh and Sumit Gulwani. *Synthesizing Number Transformations from Input-Output Examples*, pages 634–651. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[58] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651. Springer, 2012.

[59] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer, 2015.

[60] Gustavo Soares. Refazer. `https://github.com/gustavoasoares/refazer`, 2017.

[61] Ian Sommerville et al. *Software engineering*. Addison-wesley, 2007.

[62] Source Forge. FindBugs, 2018. `http://findbugs.sourceforge.net/`, Last accessed on 2018-09-11.

[63] The Eclipse Foundation. Eclipse, 2018. `https://www.eclipse.org/`, Last accessed on 2018-09-11.

[64] University of california, Berkeley. Using t-tests in R, 2014. `https://statistics.berkeley.edu/computing/r-t-tests`, Last accessed on 2019-29-18.

[65] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Advances in Neural Information Processing Systems*, pages 2643–2651, 2013.

[66] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *ACM SIGPLAN Notices*, volume 52, pages 452–466. ACM, 2017.

[67] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):63, 2017.

[68] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. Fidex: Filtering spreadsheet data using examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'16, pages 195–213, New York, NY, USA, 2016. ACM.

[69] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[70] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 495–504. ACM, 2013.

Appendix A

Training Models Parameters

The parameters used for training SVM were:

```
LinearSVC(C=0.1, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, loss='squared_hinge',
max_iter=1000, multi_class='ovr', penalty='l1', random_state=None,
tol=0.0001, verbose=0)
```

The parameters used for training LR were:

```
LogisticRegression(C=2.5, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, max_iter=100,
multi_class='ovr', n_jobs=1, penalty='l1', random_state=None,
solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

The parameters used for training NN were:

```
MLPClassifier(activation='relu', alpha=0.0001,
batch_size='auto', beta_1=0.9, beta_2=0.999, early_stopping=False,
epsilon=1e-08, hidden_layer_sizes=(30, 30, 30),
learning_rate='constant', learning_rate_init=0.001,
max_iter=200, momentum=0.9, nesterovs_momentum=True, power_t=0.5,
random_state=None, shuffle=True, solver='adam', tol=0.0001,
validation_fraction=0.1, verbose=False, warm_start=False)
```