UMA ABORDAGEM BASEADA EM APRENDIZADO DE MÁQUINA PARA SELEÇÃO DE ESTRUTURAS DE DADOS

TARSILA BESSA NOGUEIRA ASSUNÇÃO

UMA ABORDAGEM BASEADA EM APRENDIZADO DE MÁQUINA PARA SELEÇÃO DE ESTRUTURAS DE DADOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Gisele Lobo Pappa
Coorientador: Fernando Magno Quintão Pereira

Belo Horizonte

Dezembro de 2020

TARSILA BESSA NOGUEIRA ASSUNÇÃO

A MACHINE LEARNING APPROACH FOR DATA STRUCTURE SELECTION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Gisele Lobo Pappa
Co-Advisor: Fernando Magno Quintão Pereira

Belo Horizonte

December 2020

Assunção, Tarsila Bessa Nogueira.

A851m A machine learning approach for data structure selection [manuscrito] / Tarsila Bessa Nogueira Assunção. – 2020.
xix, 67 f. il.

Orientadora: Gisele Lobo Pappa.
Orientador: Fernando Magno Quintão Pereira.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.61-67
.
1. Computação – Teses. 2. Aprendizado de máquina – Teses. 3. Estrutura de dados – Seleção – Teses. 4. Redes neurais (Computação) – Teses. I. Pappa, Gisele Lobo. II. Pereira, Fernando Magno Quintão. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*82.6(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

A Machine Learning Approach for Data Structure Selection

TARSILA BESSA NOGUEIRA ASSUNÇÃO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. GISELE LOBO PAPPA - Orientadora
Departamento de Ciência da Computação - UFMG

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Coorientador
Departamento de Ciência da Computação - UFMG

DR. DIEGO ELIAS DAMASCENO COSTA
Department of Computer Science and Software Engineering - Concordia University

PROF. ADRIANO CÉSAR MACHADO PEREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 14 de Dezembro de 2020.

Aos meus pais, José Tarcísio e Teresa Cristina.

Agradecimentos

Primeiramente, eu gostaria de agradecer aos meus pais José Tarcísio e Teresa Cristina, por sempre me amarem e me apoiarem incondicionalmente. Eu não estaria aqui hoje sem vocês. Ao meu irmão Eduardo, por ter decidido estudar Engenharia da Computação primeiro, mostrando que esse era um caminho que eu também poderia seguir. À minha tia Simoni, por todo o apoio, e visitas à Belo Horizonte nestes últimos sete anos. E a todos os meus familiares, que estão sempre torcendo por mim.

Gostaria também de agradecer à minha orientadora Gisele e ao meu coorientador Fernando por acreditarem neste trabalho e pelo suporte e ensinamentos durante este mestrado. Agradeço aos meus amigos da graduação e da pós: André, Ana Luiza, Bernardo, Cristiano, Gabriel, Hugo, Lívia, Matheus, Nathália, Rodrigo, Ronaldo, e Vinicius, por toda ajuda com trabalhos práticos, sessões de estudo e monitoria, e pela companhia por todos estes anos na UFMG. Agradeço também ao Artur, por revisar esta dissertação, e por estar sempre disposto a me ajudar a *debuggar* meu código ou discutir ideias.

Aos meus amigos do Laboratório de Compiladores: Angélica, Breno, Caio, Carina, Guilherme, Junio, Marcelo, Marcus, Rubens, Roberto, Pedro Caldeira, Pedro Ramos, e Yukio, por toda a ajuda e companhia desde 2015, pelas idas ao Aeroburguer, pelos Compilassandos, e pelas maratonas de *Switch* no laboratório. Aos meus amigos do Laboratório de Inteligência Computacional: Alex, Karen, Iago, Felipe, Renato, Pedro, por me receberem tão bem no grupo, por toda ajuda com aprendizado de máquina, pelas ótimas sugestões durante a prévia de defesa, e pelos *happy hours*.

Também gostaria de agradecer ao meu time durante o estágio e trabalho no Google por sempre me ensinar tanto. Agradeço, em especial, aos meus maiores mentores na empresa, que também se tornaram meus grandes amigos: Luciana, por estar sempre disposta a ouvir sobre minhas alegrias e frustrações com este projeto, e por ter discutido ideias comigo, sugerindo possíveis soluções para os problemas que encontrei, e ao Vitor, por sempre me apoiar e me lembrar que devo confiar em mim.

Finalmente, gostaria de agradecer à CAPES por ter financiado este trabalho, e

ao DCC pelos ótimos programas de graduação e mestrado. Agradeço aos funcionários do departamento por todo o suporte, e em especial à Sônia pela atenção e cuidado na preparação para a defesa. Finalmente, agradeço também à UFMG, por ter sido a minha segunda casa pelos últimos sete anos.

"Trouble, trouble, trouble."
(Taylor Swift)

Resumo

Uma seleção de estrutura de dados adequada pode melhorar bastante o desempenho e uso de memória de um programa. Porém, é difícil entender o comportamento de uma estrutura de dados em uma aplicação antes de executá-la, o que torna complexa a tarefa de escolher a melhor estrutura de dados para o uso em um programa de computador. Esta dissertação apresenta um *framework* capaz de selecionar, de maneira automática, estruturas de dados para um programa e sua entrada. O *framework* é formado por quatro componentes: um componente para extração de dados, que coleta informação sobre o comportamento das estruturas de dados usadas no programa dado como entrada; um construtor de *features*, que transforma as informações extraídas em features para modelos de aprendizado de máquina; dois modelos de aprendizado de máquina que selecionam automaticamente estruturas de dados para serem usadas com o conjunto de programa e entrada; e um reconstrutor de programas que cria uma nova versão da aplicação dada como entrada usando as estruturas de dados selecionadas pelos modelos. O *framework* é avaliado em aplicações sintéticas e reais e é capaz de atingir a mesma acurácia que os desenvolvedores originais ao escolher estruturas de dados para as aplicações.

Palavras-chave: Seleção de estrutura de dados, aprendizado de máquina, classificação de sequências, LSTMs.

Abstract

Accurate data structure selection can vastly improve a program's performance and memory usage. However, it is hard to understand the behavior of a data structure in an application before running it, making choosing the best data structure to use in a computer program a complex task. This dissertation presents a framework capable of automatically selecting data structures for a given program and input. The framework consists of four components: a program data extractor, which collects information on the behavior of the data structures used in the input program; a feature constructor that transforms the extracted data into machine learning features; two different offline machine learning models which automatically select data structures to be used with the given application/input combination; and a program reconstructor that creates a new version of the input program using the data structures selected by the models. The framework is evaluated with synthetic and real world applications and is capable of reaching the same level of accuracy as the original developers when selecting the data structures for the applications.

Keywords: Data Structure selection, machine learning, sequence classification, LSTMs.

Resumo Estendido

Uma estrutura de dados é um formato especializado eficiente para organizar e armazenar dados. Linguagens de programação disponibilizam *frameworks* de coleções que oferecem ao desenvolvedor tipos abstratos de dados para gerenciar coleções de dados, enquanto escondem a implementação dessas estruturas. Este é o caso do *framework* de coleções de Java e da *Standard Template Library* de C++. Algoritmos e estruturas de dados são fortemente conectados. Por exemplo, ao utilizar uma lista ordenada em uma aplicação, o desenvolvedor usaria um algoritmo de busca otimizado para essa estrutura. Portanto, a escolha da melhor estrutura de dados para uma aplicação é uma parte importante de programação. A escolha de uma estrutura inadequada para um programa pode trazer problemas de desempenho e memória. Muitos estudos identificaram as estruturas de dados como responsáveis pela ineficiência do desempenho de programas.

A escolha da estrutura de dados para um programa de computador é uma tarefa complexa. Geralmente, os desenvolvedores utilizam bibliotecas de estrutura de dados e confiam na complexidade assintótica para decidir qual implementação utilizar. Essa abordagem funciona bem para algoritmos, mas complexidade assintótica não é tão útil para estruturas de dados, pois não considera o comportamento das coleções durante a execução de um programa. Por exemplo, quando consideramos a linguagem Java, buscar por um elemento em um `TreeSet` ou `HashSet` tem tempo logarítimico e constante amortizado, respectivamente. Porém, se estamos lidando com um conjunto de poucos elementos, uma busca linear será mais rápida na prática. Além disso, a eficiência de uma estrutura de dados também depende de outros fatores além do comportamento assintótico, como a arquitetura que executa o programa e o que é passado como entrada.

Nesta dissertação, nós investigamos o problema de selecionar a melhor estrutura de dados para uma determinada aplicação e entrada. Nós propomos um *framework* baseado em aprendizado de máquina para escolher, de maneira automática, a estrutura de dados que melhora o desempenho de um programa. No nosso caso, desempenho se refere ao tempo de execução da aplicação. O *framework* disponibiliza uma solução

completa, que recebe um programa como entrada, o modela como entrada para um modelo de aprendizado de máquina, executa um modelo preditivo, e retorna a aplicação modificada com as melhores estruturas de dados. O *framework* é instanciado usando duas abordagens para classificação de sequências, um baseado em *features*, e o outro em modelos. Classificação de sequências é uma modelagem preditiva para um problema, no qual, dado um conjunto de classes, o objetivo é associar cada classe à uma sequência. Neste trabalho, nós focamos na substituição de variações da estrutura de dados `Set` de Java. Nós decidimos trabalhar com essa interface porque ela é uma implementação estabelecida, recomendada para armazenar elementos únicos.

O nosso modelo baseado em *features* extrai informações para aprendizado da representação sequencial de um programa, como o número de operações `add`, `remove`, `contains`, e `iterator` invocadas, e o número máximo de elementos armazenados na estrutura de dados. Esse conjunto de *features* é dado como entrada para diferentes classificadores, como *Random Forest*, e *Support Vector Machine*. A abordagem baseada em modelos utiliza uma *Long Short-Term Memory Network* (LSTM), que recebe dados sequenciais como entrada. A LSTM é muito utilizada para problemas como reconhecimento de fala e tradução graças à sua capacidade de aprender dependências de longo prazo em sequências.

O *framework* proposto foi avaliado em conjuntos de dados sintéticos e reais, e um de seus modelos de aprendizado é capaz de atingir o mesmo nível de acurácia que o desenvolvedor original ao escolher estruturas de dados para os programas. Além disso, o outro modelo conseguiu encontrar a estrutura correta para metade dos casos em que o programador fez uma escolha equivocada, comprovando que o *framework* pode auxiliar o desenvolvedor na tarefa de seleção de estrutura de dados para aplicações.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

A data structure is a specialized format for organizing and storing data efficiently. Programming languages offer collection frameworks that provide the developer with abstract data types for managing collections of data, while hiding the underlying data structure implementation [Shacham et al., 2009]. That is the case of the Java framework collection[1] and the Standard Template Library (STL) in C++ [Stepanov and Lee, 1995]. Algorithms and data structures are strongly connected. For example, when using a sorted list in an application, one would use a search algorithm optimal for this data structure. Therefore, selecting the best data structure for an application is an important part of programming. Choosing the wrong data structure for a program may result in problems with performance and memory bloat [Costa and Andrzejak, 2018]. Multiple studies have identified data structures as responsible for applications performance inefficiency [Gil and Shimron, 2012; Georges et al., 2007; Liu and Rus, 2009].

Choosing which data structure to use in a computer program is a complex task. Usually, developers use data structure libraries and mainly rely on asymptotic complexity to choose which implementation to use. This approach works well when it comes to algorithms, but asymptotic complexity is not as useful for data structures, as it does not consider the actual behavior of the collection during the execution of a program. For example, when we consider the Java language, a search for an element in a `TreeSet` or `HashSet` has logarithmic and constant amortized time, respectively. However, if we are dealing with a small set of elements, a linear search will be faster in practice. Besides, the efficiency of a data structure also depends on different aspects other than asymptotic behavior, such as the underlying architecture and program inputs.

Most data structure libraries provide different implementations for an abstract

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

data type, with each one focusing on offering at least one distinct feature. As an example, in Java, there are five concrete implementations of the `Set` collection. The `HashSet` implementation offers constant time performance for the basic operations (`add`, `remove`, `contains`, and `size`), while the `TreeSet` maintains a sorted set of elements. All of these concrete implementations are interchangeable because they share the same abstract data type. For this reason, choosing which implementation to use for an abstract data type is a more specific instance of the data structure selection problem, and also a complex task.

In this work, we investigate the problem of selecting the best data structure for a given application and input, while focusing on different concrete implementations of one abstract data type. We propose a framework based on machine learning to automatically choose the data structure implementation that improves the performance of a program. In our case, performance refers to the execution time of a given application. However, our framework can be easily extended to focus on other performance aspects, such as energy and memory consumption. Following a machine learning approach is promising here, as these approaches have shown to be more effective than human designed models in a number of different situations related to compiler optimization [Leather et al., 2009; Monsifrot et al., 2002; Stephenson et al., 2003].

The framework provides a fully-fledged solution, which receives as input a program, creates an appropriate machine learning representation to it, runs a prediction model and returns the program with the modified data structure. The framework was instantiated using two different types of sequence classification models: a feature-based and a model-based. Sequence classification is a predictive modeling problem in which, given a set of class labels, the goal is to associate one class label to a sequence [Xing et al., 2010]. In this first version, we test the replacement of variations of the `Set` data strucute in Java. We decided to work with the `Set` interface as it is a long-established collection, highly recommended for storing unique elements.

The feature-based model extracts features from the sequence representation of the program, such as number of `add`, `remove`, `contains`, and `iterator` instructions invoked, and the maximum number of elements in the data structure, and feed them to conventional classifiers such as Random Forest [Breiman, 2001] and Support Vector Machine [Cortes and Vapnik, 1995]. The model-based sequence classification approach is based on a Long Short-Term Memory Network (LSTM) [Hochreiter and Schmidhuber, 1997], which receives sequence data directly. An LSTM is a type of Recurrent Neural Network [Rumelhart et al., 1985], which has been notably used for speech recognition and machine translation due to its capability to learn long term dependencies in sequences.

The framework offers both a static and a dynamic solution to the data structure selection problem. While the feature-based model is static, as it outputs one alternative data structure to be used during the whole execution of the program, the model-based with LSTM approach finds different optimal data structures for different parts of a program, identifying points in the execution when one implementation ought to be replaced by another.

Constructing a diverse set of data for training machine learning models is a hard task. For this work, there was not a well-known large dataset focused on Java collections available. For this reason, we have implemented an application generator that can simulate synthetic applications exploring the different behaviors of multiple data structures. We have trained our models on this data and applied the trained models in real-world algorithms written by us, and from the DaCapo benchmarks suite [Blackburn et al., 2006].

The contributions of this work include:

- A definition of the data structure selection problem modeled as a sequence classification problem.
- The implementation of a framework that automatically selects data structure implementations for a program.
- An analysis of which characteristics of an application's behavior are important to take into account when choosing a data structure implementation.
- An empirical evaluation of two different learning models on six real-life applications from the DaCapo benchmarks suite.

This dissertation is organized as follows. Chapter 2 addresses related work. Chapter 3 presents a general view of the framework proposed here and details the process of creating a machine learning representation from input programs. Chapter 4 describes the feature-based and model-based approaches proposed for data structure selection. In Chapter 5, we evaluate both learning models on synthetic data and real-world applications. Chapter 6 concludes the dissertation and addresses future work.

Chapter 2

Related work

Many works have previously investigated the problem of data structure selection [Shacham et al., 2009; Liu and Rus, 2009; Jung et al., 2011; Xu, 2013; Manotas et al., 2014; Basios et al., 2018; Costa and Andrzejak, 2018]. These works proposed different approaches for data structure selection, such as automatic tools and hand-made models. Some solutions are dynamic, as they find different optimal data structures for different points of the execution of a program. Others are static: one unique solution is given for the whole application. In a different perspective, numerous researchers have focused on how the use of inappropriate data structures causes performance inefficiencies regarding energy consumption, execution time and memory use [Mitchell and Sevitsky, 2007; Xu and Rountev, 2010; Yang et al., 2012; Hasan et al., 2016; Pinto et al., 2016; de Oliveira Júnior et al., 2019; Costa et al., 2017]. Their works do not automatically provide alternative data structures to be used in a program. In this chapter, we review these two categories of related works, and emphasize the unique characteristics of the framework presented in this dissertation. Furthermore, we explore a few works regarding sequence classification, as the models we propose for data structure selection are based on this approach.

2.1 Data structure performance impact

Wirth [1976] stated that Algorithms + Data Structures = Programs, making the selection of efficient data structures for an application an important part of programming. Numerous researchers have studied the impact of the inappropriate use of data structures as a cause of performance inefficiencies regarding energy consumption [Hasan et al., 2016; Pinto et al., 2016; de Oliveira Júnior et al., 2019], execution time [Costa et al., 2017], and memory consumption [Mitchell and Sevitsky, 2007; Yang

et al., 2012; Xu and Rountev, 2010]. In this section, we focus on studies related to the performance of Java collections. Costa et al. [2017] present an analysis of memory consumption and execution performance of collection classes offered by six libraries. They used a dataset containing 10,896 Java projects from GitHub to analyze the popularity and usage pattern of each library. Then, they collected the execution time and memory allocation of the benchmarks using the four most used collection types with different implementations. Their work indicates that alternative libraries can provide a significant reduction of execution time and memory consumption.

Various studies have focused on memory bloat as a performance bottleneck. The work by Mitchell and Sevitsky [2007] proposes a framework that offers the ability to compare different implementations and understand when it makes sense to use a specific implementation. They introduce the concept of health of a data structure, which depends on the relationship between the actual data and structural overhead. Yang et al. [2012] propose a dynamic analysis that tracks and records the flow of elements to and from containers to analyze behavior and performance. Their results indicate that optimization focused on containers is a target for performance analysis and tuning. The work by Xu and Rountev [2010] presents static and dynamic tools to find inappropriate use of Java containers. They identify interface functions such as `add` and `get` and analyze their execution frequencies. They conclude that it is possible to find optimization opportunities when focusing on containers with high allocation frequency during runtime.

In a different perspective, some researchers present studies on the energy consumption of Java collections. The work by Hasan et al. [2016] presented detailed profiles of the energy consumed by common operations done on Java `List`, `Map`, and `Set` abstractions. They conclude that choosing the wrong data structure implementation can increase the energy consumption of the application in up to 300%. Pinto et al. [2016] performed an empirical investigation on the energy consumption of Java's thread-safe collections. They observed that different implementations of the same collection can have widely different energy consumption behaviors. This also applies to individual operations. An implementation that works well with insertions can be way less effective than another implementation when it comes to iterating through data. The work by de Oliveira Júnior et al. [2019] proposes a framework that combines static analyses and application-independent energy profiles to produce energy-saving recommendations for alternative collection implementations. Their results indicate that widely used collections such as `ArrayList` and `HashMap` should be avoided when energy consumption is a major concern.

While the aforementioned works identify performance inefficiencies related to data

structure selection, they do not provide alternatives that could improve application performance. The framework proposed in this dissertation comes to fill this gap: it finds the best implementation for each object instance of a data structure in the code to improve performance.

2.2 Data structure selection methods

The most direct inspiration of this work has been Brainy [Jung et al., 2011]. Like our framework, Brainy is a program analysis tool that automatically selects the best data structure for a given program. It generates traces of various runtime characteristics of a combination of application, input and architecture, and feeds them into an offline model constructed using machine learning; this model is then used to select the best data structure. Jung et al. have chosen to work with C++ programs and use a conventional artificial neural network modeling the data structure selection problem as a classification task. Brainy collects information from the program, such as the number of interface operations called, combined with a cost model for each invocation based on the number of elements accessed. Their approach selects only one data structure to be used in the whole program, regardless of the number of objects created. In contrast, our work selects different data structures for each allocation site in the application. Furthermore, we use sequence data to represent the interface functions called in the program.

The work by Basios et al. [2018] presents the definition of Darwinian Data Structures: distinct data structures that are interchangeable because they share an abstract data type. Their goal is to solve the Darwinian data structure optimization problem, which tries to find the optimal implementation for a Darwinian data structure used in an input program. Their tool, ARTEMIS, finds tunable Darwinian data structures in the code, and uses a multi-objective genetic search algorithm to provide optimized solutions. Unlike ours, their framework is source-to-source, and they provide as outpust a transformed version of the original code with the optimized data structures. Additionally, they rely on testing to define a performance search space, so their tool can only be applied to programs with a test suite.

The Chameleon [Shacham et al., 2009] and Perflint [Liu and Rus, 2009] projects instrument applications to collect runtime statistics on behaviors such as interface function calls. Like us, Chameleon works with Java programs, while Perflint works with C++. Chameleon also collects heap-related information from the garbage collector. Both projects feed the collected features into hand-constructed diagnostics to determine

if the data structures should be replaced. Our work expands on this by using machine learning models to automatically select the best data structure instead of relying on hand-constructed models. The SEEDS [Manotas et al., 2014] project uses exhaustive search to tune the collections implementation used in a program. However, while our work focuses on improving runtime performance, SEEDS focuses on energy efficiency.

Some researches have focused on dynamic solutions to the data structure selection problem, which transforms one data structure implementation into another in specific points of the program. For example, the CollectionSwitch [Costa and Andrzejak, 2018] project presents a framework that selects, at runtime, collection implementations to optimize the execution and memory performance of an application. Their tool identifies allocation sites which instantiate suboptimal collection variants, and selects optimized implementations for future instantiations. They present a dynamic approach and deal with overhead as the collection variants presented are capable of changing their internal data structures depending on the collection size. The work by Xu [2013] presents CoCo, a framework that identifies at runtime, for each container instance used in a program, whether or not there exists another container implementation that is more suitable for the execution. If so, the tool automatically performs the switch to the other implementation for increased efficiency. While our work does not swap data structure implementations at runtime, our LSTM-based solution is capable of identifying points in the execution when one implementation should be replaced by another.

Table 2.1 presents a summary comparing our proposed approaches to the works referenced in this section. For each work, it is presented which programming language the work focus on, whether it's a static or dynamic solution, what methods are used for solving the problem, and which performance aspect they focus on.

2.2.1 Language support

Using a different perspective, some researchers suggest language support for data structure selection. For example, the work by Dewar et al. [1979] presents, for the high-level programming language SETL, new declarations that allow the developer to control the data structures that will be used to implement an algorithm that has already been written in pure SETL. They describe the features of this new sublanguage and the data structures that it can generate at runtime. Besides, they also propose an heuristic, based on global program analysis, which is capable of automatically selecting an efficient data structure representation for the SETL program. Additional work on this subject by Schonberg et al. [1986] focuses on eliminating the need for manually specifying the structures in the sublanguage. They present an optimizer that

Work	Language	Static or dynamic	Method	Optimization
ARTEMIS	Java	Static	Multi-objective genetic search algorithm	Execution time, memory and CPU usage
Brainy	C++	Static	Modeling the problem as a classification task with ANNs	Execution time
Chameleon	Java	Static	Feeding runtime statistics to hand-constructed models	Execution time and memory consumption
Coco	Java	Dynamic	Monitoring application usage at the instance level	Memory bloat
CollectionSwitch	Java	Dynamic	Cost model and selection rules based on the behavior of collection variants	Execution time and memory consumption
Perflint	C++	Static	Feeding runtime statistics to hand-constructed models	Execution time
SEEDS	Java	Static	Exhaustive search	Energy consumption
Our feature-based approach	Java	Static	Modeling the problem as a classification task with feature extraction and classifiers	Execution time
Our model-based approach	Java	Dynamic	Modeling the problem as a classification task with LSTMs	Execution time

Table 2.1: A summary of the work related to our framework.

uses techniques that allow relations of inclusion and membership to be established, the domains and ranges of mappings to be estimated, and the single-valuedness of mappings to be proved. With these definitions, automatic selection of data structures is possible. This line of work focused more on using only static analysis for data structure selection and the performance of those tools was generally worse than hand-selected implementations.

2.3 Sequence classification

A sequence is an ordered list of elements that can be represented as symbolic values, real numbers, a vector of real values or a complex data type. In this work, we propose to represent the behavior of a data structure in a given input program as a sequence of instructions. Therefore, we can also model the data structure selection problem as a sequence classification problem. The task of sequence classification is to associate one class label to a sequence, when given a set of class labels [Xing et al., 2010]. In this section, we go over a few methods for solving the sequence classification problem.

Sequence classification methods can be divided into three groups: feature-based, distance-based, and model-based [Xing et al., 2010]. In feature-based sequence classification, we transform a sequence into a vector of features and then run conventional classifiers, such as Random Forest [Breiman, 2001], Support Vector Machine (SVM) [Cortes and Vapnik, 1995], and k-Nearest Neighbor [Tan et al., 2016]. The simplest method for feature extraction is to use each element and its position as a feature. Among other alternatives of features extracted from sequences are k-grams: consecutive k symbols of the original sequence. They have been heavily used in biological sequence classification [Ounit et al., 2015; Kawulok and Deorowicz, 2015; Yousef et al., 2017; Breitwieser et al., 2018]. For example, Wang et al. [2000] propose an approach for protein classification consisting of transforming a sequence into a feature vector of k-grams and feeding them as input to an ANN. Different methods can be used to reduce the dimensionality of k-grams vectors when the sequence vocabulary is large. For instance, some approaches allow inexact matchings for k-grams, such as the gapped k-grams. Additionally, extracted features can also be pattern-based [Dong and Pei, 2007]. Arbitrary sequence patterns can be used as features or their running counts, which is the number of matches of a pattern P in each position of a sequence S.

Distance-based methods define a distance function to measure the similarity between two sequences [Xing et al., 2010]. The Levenshtein [Miller et al., 2009] and Hamming [He et al., 2004] distances are two of the most well-known distance functions used in this problem. The former considers the number of editions (inserting or deleting an element) needed to transform one sequence into another. The latter is used for sequences with the same length and counts the number of elements that are different. Both distance functions consider one mismatch as one unit of dissimilarity. Works by Wilbur [1985] and Henikoff and Henikoff [1993] propose different cost measures for mismatches. Furthermore, in biological sequence classification, alignment-based dis-

tances functions are the most applied, with tools such as the ones by Altschul et al. [1997] and Pearson [1990].

Model-based classification is based on generative models, which assume that sequences in a class are generated by an underlying model [Xing et al., 2010]. The Naive Bayes sequence classifier [Rish et al., 2001] is one of the most used model-based classifiers due to its simplicity and low-computation cost. As an example, Cheng et al. [2005] propose protein classification based on text documentation classification using Naive Bayes. The work by Yan et al. [2004] presents a framework that combines an SVM and a Bayesian classifier to identify protein interface residues. Blasiak and Rangwala [2011] propose a Hidden Markov Model [Baum et al., 1970] variant to produce a set of fixed-length description vectors from a set of sequences. They use three different algorithms to infer model parameters and conclude that their approach is useful for classifying sequences of amino acids into structural classes.

Another common model for sequence classification is Artificial Neural Networks (ANN) [Goodfellow et al., 2016], which is one of the models we explore in this work. ANNs are classification methods inspired by biological neural networks. They are used to solve numerous problems in pattern recognition, prediction, and optimization [Yegnanarayana, 2009]. In sequence classification, ANNs are heavily used in the biological field to classify sequences of proteins and DNA [Wang et al., 2000; Blekas et al., 2005; Ma et al., 2001; Wu, 1997].

A Long Short-Term Memory Network (LSTM) [Hochreiter and Schmidhuber, 1997] is a special type of a Recurrent Neural Network (RNN) [Rumelhart et al., 1985]. RNNs are artificial neural networks that allow information to persist. They are composed of hidden states, which act as the memory of the network, holding information on previous data. This chain-like nature makes RNNs suitable for learning sequences and lists. However, this network becomes unable to connect information when the length of the sequence grows. LSTMs have been designed to fill this gap, being capable of learning long-term dependencies, as remembering information for long periods of time is their default behavior. The key idea behind the architecture of LSTMs is a memory cell that can maintain its state throughout the whole processing of the sequence, and gating units that control the information flow into and out of the cell [Greff et al., 2016]. Due to its capability to learn long-term dependencies, the use of LSTMs has been notable in language modeling [Zaremba et al., 2014], speech-to-text transcription [Graves et al., 2013], and handwriting recognition [Breuel et al., 2013]. For sequence classification, LSTMs have been heavily used in the biological field. As an example, Hu et al. [2019] propose a combination of Convolutional Neural Networks (CNN) [LeCun et al., 1999] and bidirectional LSTMs to identify DNA binding proteins. Furthermore,

Muller et al. [2018] present an LSTM approach for classification of amino acid sequences and generation of new data instances.

Making a parallel between sequences of amino acids and sequences of program operations, here we propose a feature-based and a model-based approaches for data structure selection in Java programs.

Chapter 3

A Framework for Data Structure Selection

As previously mentioned, the data structure selection problem is defined as the task of selecting data structures that optimize a performance-related aspect of a given input program, such as memory consumption, execution time, and energy consumption. In this work, we propose a framework that focuses on the runtime performance under a machine learning perspective.

Figure 3.1 presents the framework proposed for data structure selection. It is composed of four components. The program data extractor (1) receives the input code in binary format and runs it to extract information on the behavior of the data structures used in the application, such as the number of interface functions invoked, the number of elements in the collection, and execution time of each function called. This component will be explained in more details in Section 3.1. The feature constructor (2) uses the information extracted from the program to create features for different machine learning models (3). The models are responsible for finding the best data structures to improve the runtime performance of the input program. In this work, we model the data structure selection problem as a classification task. The feature construction and the learning models components are explained in more details in Chapter 4. The last component is the program reconstructor (4), which creates a new version of the input code, in binary format, using the data structures recommended by the models. Due to its similarity to the program data extractor, this component will be described in Section 3.2.

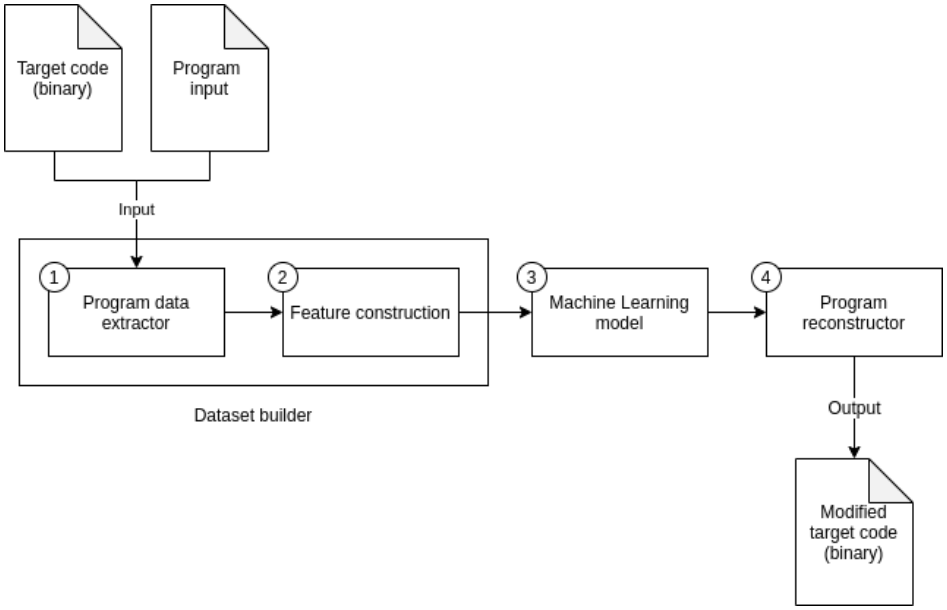


Figure 3.1: Fluxogram of the proposed framework.

3.1 Program data extractor

The program data extractor is built on top of Soot [Vallée-Rai et al., 2010], a framework for analyzing and transforming Java and Android applications. Our approach is to iterate through an intermediate representation of an input program, which is generated by Soot, searching for each use of the target data structure, e.g. `HashSet`. Then, we replace it with our implementation of a subclass for that structure, e.g. `InstrHashSet`. The `InstrHashSet` class creates a log of each `add` (A), `remove` (R), `contains` (C), and `iterator` (I) operation in the code. Its source code can be found in Appendix A.

Figure 3.2 presents an example of input and output for the program data extractor. In Figure 3.2a, there's an application that creates a new instance of `HashSet` and invokes different interface functions. Figure 3.2b, presents the information that the program data extractor generates by using Algorithm 3.1 as input. In the *id* column of the comma-separated values (CSV), there's the object ID; the type of operation invoked in *op*; in the *st* column, we present the instant when the instruction was called; the execution time, in nanoseconds, of the operation, in the *et* column; the number of elements in the data structure in *size*; and in the *impl* column, what implementation of set is being used. This information is collected for each interface operation mentioned above and refers to the state of the data structure in that point of the program execution.

We can safely replace objects of type `HashSet` with objects of type `InstrHashSet`

```java
public class Example {
  public static void main() {
    Set<Integer> s =
        new HashSet<>();

    for (int i = 0; i < 3; i++) {
      s.add(i);
    }

    s.remove(0);
    if (s.contains(1)) {
      for (int element : s) {}
    }
  }
}
```

Algorithm (3.1) Example of a program creating a new instance of `HashSet` and invoking interface functions.

(a) Input program for the data extractor generator.

id,	op,	st,	et,	size,	impl,
0,	A,	6709502610239,	5130,	1,	HS,
0,	A,	6709502694527,	1604,	2,	HS,
0,	A,	6709502703089,	1087,	3,	HS,
0,	R,	6709502711180,	7889,	2,	HS,
0,	C,	6709502726049,	3473,	2,	HS,
0,	I,	6709502736179,	142464,	4,	HS,

(b) Information generated by our program data extractor for Algorithm 3.1.

Figure 3.2: Example of input and output of the program data extractor.

due to the Liskov substitution principle. It states that properties that are provable about objects of type T should be true for objects of type S, with S being a subtype of T [Liskov and Wing, 1994]. However, some code practices prevent us from simply replacing the instructions that instantiate a new object. For instance, in line 2 of Algorithm 3.2, the variable `ts` is declared with the concrete type `TreeSet`, instead of the abstract data type (ADT) `Set`. Replacing the variable type in this case triggers a compilation error. To handle these cases, our framework first transforms the code by declaring each variable with its ADT. We apply this transformation to function parameters and return type, local variables, and class fields.

```java
void foo() {
  TreeSet<Integer> ts = new TreeSet<>();
  ts.add(1);
}
```

Algorithm 3.2: An application that declares a variable using its concrete type `TreeSet`, instead of its abstract type `Set`.

The code instrumentation is divided into two phases: *abstract* and *concrete*. In the abstract phase, our tool converts the type of the target implementation to that of their ADT, for example, `TreeSet<T>` → `Set<T>`. In the concrete phase, we transform

the type of the original data structure to their subtype, one of our instrumentation classes. As an example, the code in line 2 of Algorithm 3.2 is replaced with `new InstrTreeSet<>()`. In the concrete phase, we only need to change the type used in instructions of object creation (method `new`) and casting. We also add a function call at each exit point of the program to print the collected information. All data collected from a program is added to a list to avoid calling Java's `print` instruction multiple times in the execution. After both phases end, the Soot framework transforms the intermediate representation back to Java bytecode.

During the concrete phase, we also store the source code line of each object creation and casting instruction. We do this in order to know which instructions to replace with a different data structure later. For each object creation, we store the object ID, the code line of the instruction, and in which class the object was created. For the casting instructions, we run an analysis that returns the definition sites for the variable being cast. Then, we store the line of the definition site of the variable, the class in which it is used, and the source code line of the cast operation. When the instrumentation ends, we create two output files with the stored values. Figure 3.3 presents the output files of the instrumentation of Algorithm 3.3. In line 3, there is an object creation. So in Figure 3.3a the *id* column represents the object ID, the *class* column is the name of the class `ConstructorCastingExample`, and the *line* column is the source code line of the instruction. Figure 3.3b presents information on the casting operation in line 5. It shows that if we change the `Set` implementation defined in line 3 of the `ConstructorCastingExample` class, then we also need to change the type of the casting instruction in line 5.

```
1  public class ConstructorCastingExample {
2      public static void main() {
3          Set<String> sample = new HashSet<String>();
4          check_1(sample, 1);
5          check_2((HashSet)sample, 2);
6      }
7
8      private static void check_1(Set<String> sample, int x) {
9          sample.add("check-1");
10     }
11
12     private static void check_2(HashSet<String> sample, int x) {
13         sample.add("check-2");
14     }
```

```
15 }
```

Algorithm 3.3: An application that creates a new instance of `HashSet` and uses a casting operation.

id,	class,	line,
0,	ConstructorCastingExample,	3,

(a) File with information about instructions of object creation.

def,	class,	cast,
3,	ConstructorCastingExample,	5,

(b) File with information about casting instructions.

Figure 3.3: Output files generated by our program data extractor for Algorithm 3.3.

The `TreeSet` class implements a sorted version of `Set` by using the natural ordering of its elements, or a provided comparator. If the elements are not comparable and a comparator is not provided, the program will throw an exception at runtime. We need to mitigate this issue in order to replace any `Set` implementation with our `InstrTreeSet` class. For this reason, we have implemented a customized comparable that works with any type of element. If the set is composed of primitive types, our comparator simply uses its natural ordering. However, if the type is unknown, the comparator uses the element's hash code for the comparison. All of the `IntrTreeSet` constructors call `TreeSet`'s comparator constructor, passing our customized comparator as the argument. The comparator source code is provided in Appendix A.

In this work, we use the program data extractor to generate appropriate program representations for machine learning. As here we deal with a classification task, our classes are the possible data structures that can be selected, which in our case are three: `HashSet`, `TreeSet`, and `LinkedHashSet`. Classification algorithms usually work with two sets of data: a training set and a testing set. Each example in these datasets correspond to a program. In the training set, used to learn the model, information about the actual best data structure to be used needs to be available. In our case, we also need this information for the testing set, in order to evaluate the generalization of the produced models in new programs.

Having said that, we use the extractor to create three versions of the input program: one only using `InstrHashSet`, one using `InstrLinkedHashSet`, and the other using `InstrTreeSet`. This way, we can have information on the program while using different implementations of `Set`. Then, we run analyses to find which of the three implementations is the fastest for each object instance in the input code. We extract the program runtime by adding the execution time of each interface function invoked. With this approach, we can find different optimal solutions for each object in the code,

but we only take into account the time needed to perform interface operations. To find the fastest data structure, we run the three generated versions of the code ten times and calculate the average time of each operation. Then, we sum the average time and find which data structure took the least time to run. We consider one data structure faster only if its runtime is at least 5% smaller than the second alternative.

3.2 Program reconstructor

Our framework also provides a program reconstructor, which replaces a target data structure with another implementation with the same ADT. For example, is allows us to replace an instance of `LinkedHashSet` with `TreeSet`. This tool is used to transform the input program by using the best data structure implementations chosen by the machine learning models that we will describe in Chapter 4. Similarly to the program data extractor, this replacement process is also divided into two phases. The first phase (abstract) converts the type of the target implementation to that of their ADT. In the second phase (concrete), we use two files received as input. The input files describe which object creation and casting instructions should be replaced. As illustrated in Figure 3.4, in each file, for every line, in the *line* column is the source code line of the instruction, the *class* column is the class in which the operation is called, and in *impl* is the implementation type that should be used. When we analyze casting and object creation instructions, we check if the source code line is in one of the input files, and if its class matches the one in the file. If both criteria is reached, we perform the transformation.

line,	class,	impl,
13,	Example,	java.util.HashSet,
6,	Example,	java.util.TreeSet,
3,	Test,	java.util.LinkedHashSet,
13,	Benchmark,	java.util.TreeSet,

Figure 3.4: Example of input file given to the program reconstructor.

Chapter 4

Learning to select data structures

In Chapter 3, we presented a general view of our framework and how we extract information from programs to generate appropriate data classification representations. This chapter presents two more components of our framework: the feature constructor and the machine learning models, as detailed in Figure 3.1. We propose two learning models to solve the data structure selection problem modeled as a sequence classification problem.

A sequence is an ordered list of events, in which an event can be represented as a symbolic value, a numerical real value, a vector of real values, or a complex piece of data. The interaction between data structures and an application can be represented as a sequence of operations that are performed in the collection. Therefore, we can represent a data structure behavior in Java as a sequence of interface operations. Figure 4.1 presents an example of this representation. On the left, there is a snippet of code creating an instance of `HashSet`, and on the right, a sequence representation of the same code.

Given L as a set of class labels, the task of sequence classification is to associate one class label to a sequence [Xing et al., 2010]. In this work, we represent the data structure selection problem as a sequence classification problem. In this representation, the class labels are the set of possible data structures to be selected, and the sequences are the sequence representation of that data structure in the program.

4.1 Feature-based sequence classification

In sequence classification, the feature-based approach is a method that transforms a sequence into a feature vector and then apply conventional classification methods, such as Random Forest, and SVM. Sequence features can be divided into pattern-

```
1  public static void main() {
2    Set<Integer> hs = new HashSet<>();
3
4    for (int i = 0; i < 5; i++) {
5      hs.add(i);
6    }
7
8    for (int i = 0; i < 3; i++) {
9      if (hs.contains(i)) {
10       hs.remove(i);
11     }
12   }
13
14   for (int element : hs) {}
15 }
```

(a) Application creating an instance of `HashSet` and invoking different interface functions.

AAAAACRCRCRI

(b) Sequence representation of the instance of `HashSet` in Algorithm 4.1a.

Figure 4.1: On the left: an application that creates a `HashSet` object and invokes the `add`, `remove`, `contains`, and `iterator` interface operations. On the right: a sequence representation of the behavior of the `HashSet` object in the application on the left.

based, or property-based [Dong and Pei, 2007]. The former is composed of patterns the appear in the original sequence, while the latter refers to properties of the sequence or the objects in it. For this first model, the next sections describe both their feature extraction and machine learning model components of the framework presented in Figure 3.1.

4.1.1 Feature extraction

When dealing with sequence data, it is desirable to keep the sequential nature of the elements. K-grams are pattern-based features that keep the order of the items in a sequence. They are defined as short segments of k symbols of a sequence, and can be represented with vectors of the absence or presence of a k-gram, or vectors of their frequencies. As an example, considering that our problem has four possible symbols, when $x = 2$, we can have as many as $C_{4,2}$ 2-grams. For the sequence in Figure 4.2, the 2-grams present are AC, CA, AR, RA, AA, and AI. The 2-grams AC and CA appear two times in the sequence, while the other 2-grams only appear once.

ACARAACAI

Figure 4.2: A sequence of add, remove, and contains operations.

In this work, we have used our feature constructor to generate the input for our

feature-based model. This component is used to extract k-grams from the sequences extracted from the programs by the program data extractor component from our framework. We chose $k = 1, 2, 3$ to be used as features for the learning model. We have used the frequencies of each k-gram in the sequence, totaling 84 features. Since all sequences must have the same number of features, when a k-gram is not present in the sequence, we use 0 for their frequency. We also extract other properties of the sequences, such as the number of operations in the sequence (length); the maximum, the minimum, and the median of the number of elements in the data structure during the whole program execution. In total, we extracted 88 features from each sequence of our dataset.

4.1.2 Machine learning model

After feature extraction, we can apply any classification method to learn a model that associates these features with the most appropriate data structure. We propose using four classifiers: Decision Tree (DT) [Rokach and Maimon, 2008], k-Nearest Neighbors (KNN) [Tan et al., 2016], Random Forest (RF) [Breiman, 2001], and Support Vector Machine (SVM) [Cortes and Vapnik, 1995]. The Decision Tree classifier splits the dataset into smaller subsets based on splitting rules created according to the information gain of the features. These rules are applied recursively to each derived subset until splitting no longer adds value to the predictions. The k-Nearest Neighbors algorithm assumes that similar examples (programs) are close to each other in the feature space. The algorithm classifies an unlabeled vector by assigning the label which is the most frequent between the k training samples closest to the unlabeled one. Random Forest consists of a number of decision trees that work as an ensemble. Each individual tree in the forest predicts one class, and the class with the most votes becomes the final prediction. This is based on the idea that a group of uncorrelated outcomes outperforms individual predictions. The Support Vector Machine classifier represents the dataset as points in a n-dimensional space and finds hyperplanes that divides the points into different classes.

4.2 Model-based sequence classification with LSTMs

A long short-term memory network (LSTM) [Hochreiter and Schmidhuber, 1997] is a special type of a recurrent neural network (RNN). The use of LSTMs has been notable in language modeling, speech-to-text transcription, and machine translation

due to its capability to learn long term dependencies in sequences. In this work, the interaction between data structures and an application can be represented in terms of sequential data. So, provided that a sufficiently diverse and representative training set is available, the model can directly learn from the operations sequence. In this section, we present an LSTM-based approach for solving the data structure selection problem.

4.2.1 Machine learning model

We have used the recurrent network structure presented in Figure 4.3 for model training. We trained a unidirectional LSTM with one hidden layer with 100 neurons. The output of the layer was fed into a densely connected feed-forward layer with 3 output neurons, combining the output signals with a softmax function. We have used the Adam [Kingma and Ba, 2014] adaptive algorithm for learning rate optimization and categorical cross-entropy as loss function. To mitigate overfitting, we also added dropout to our hidden layer. The network receives 70 inputs and the three outputs represent, respectively, `HashSet`, `TreeSet`, and `LinkedHashSet`. The number of inputs is based on the length of the input sequences and the number of features, which are described next. We have implemented our LSTM model in Python using the Keras library (version 2.1.5) [Chollet et al., 2015] with the Tensorflow backend (version 1.4.0) [Abadi et al., 2015].

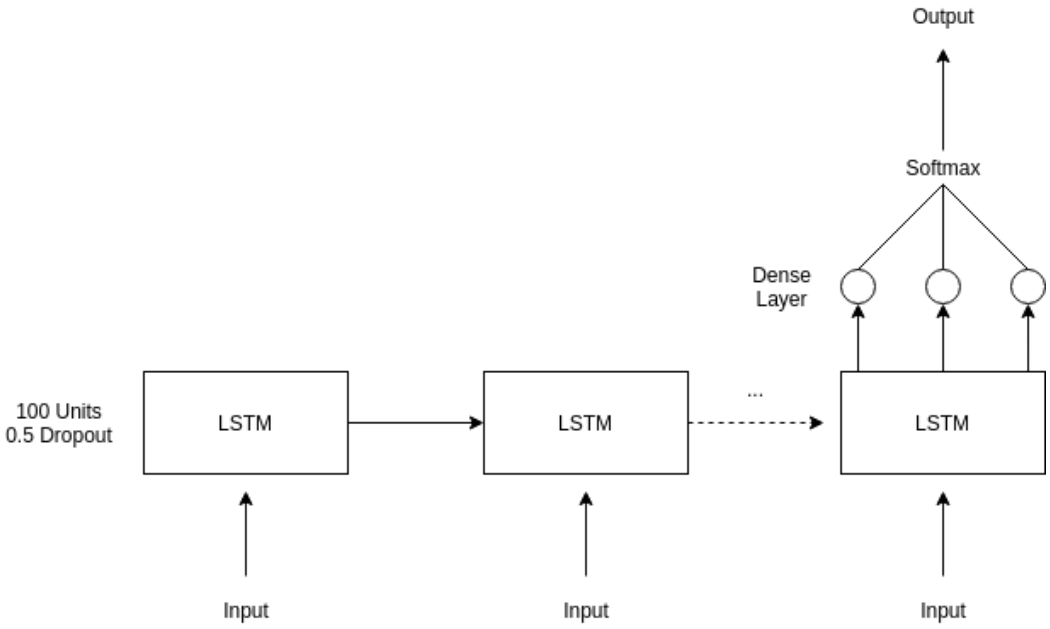


Figure 4.3: Schematic of the LSTM model used in this work.

4.2.2 Feature construction

As in the previous model, the feature constructor is used to generate the inputs for the network. We generate three different types of input features: (i) the sequence of interface operations called in the program, (ii) the sequence of the number of elements in the data structure after each interface instruction is called, and (iii) the length of the original sequence of operations. The first feature is transformed into one-hot vectors, binary vectors with length equal to the number of the possible values, which in our case are the four interface operations and a space character that is used to pad the sequences. The second and third features are rescaled so that all values are withing the range of zero and one. We use the min-max normalization, where the scaled value y is given by the formula $y = (x - min)/(max - min)$, where the minimum and maximum values comes from the set which contains the x being normalized. We use the `MinMaxScaler` class from the Sci-kit Learn library to perform the normalization. We use only the training data to estimate the minimum and maximum observable values to avoid leaking validation data to our model.

As later showed in Section 5.1, the sequences extracted by the program data extractor component of the framework have lengths ranging from 25 to 12000 operations. Given that we want to feed these sequences to a classification model, we need to find a way for all sequences to have the same size. One solution is to pre-process the data by splitting it into smaller subsequences of operations. Our feature constructor performs this pre-processing using a sliding window approach, which is presented in Figure 4.4. Firstly, if needed, we pad the sequence with space characters up to the point it reaches the size of the window we are using. Then, we extract a subsequence of size N starting from the first character; then from the second character; then from the third, until we get to the end of the original sequence. For instance, for P1 in Figure 4.4, the original sequence is composed of nine operations. When using a sliding window of length four, we first generated a subsequence with the first four operations of the original sequence (ARAI), then with the second to fifth operations (RAIC), third to sixth (AICI), and so on. In this example, the sliding window always moves one character to the right, but we can choose how many characters to slide.

We also need to pre-process the other two features: the size of the data structure and the sequence length. Each operation in a sequence is linked to a value of data structure size, so we apply the sliding window approach to the sequences of sizes as well. For the length of the original sequence, we repeat its value for each subsequence generated. Figure 4.5 illustrates this approach for P1 of Figure 4.4. For the sequence of data structure size, we extract the subsequence of length four starting from the first

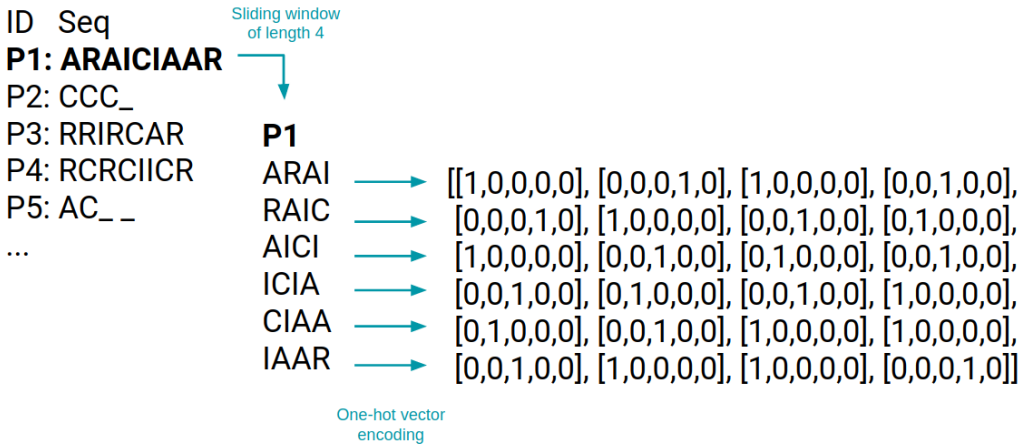


Figure 4.4: Pre-processing of sequences of operations using the sliding window approach.

value, then from the second, and so on. For the sequence length, we repeat the original sequence length, which is nine in the example, for all subsequences. As previously mentioned, our LSTM model receives 70 inputs. That is because we have used a sliding window of length ten and seven features. The seven features are a set of the encoded sequences of operations (five features), the size of the data structure, and the length of the sequence.

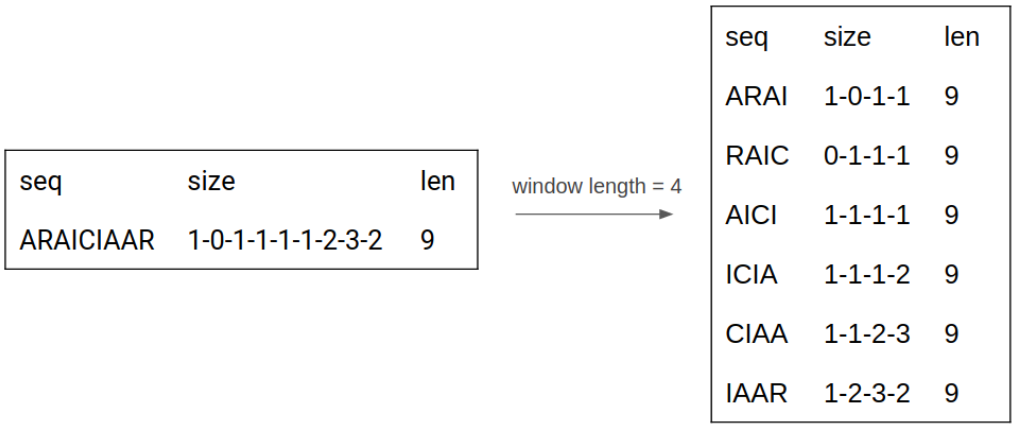


Figure 4.5: Pre-processing of program data using the sliding window approach.

The biggest advantage of the sliding window approach is the generation of more subsequences. This is interesting because it helps mitigating a great problem in data structure selection: datasets with not enough data. Additionally, this method allows us to find points in the execution of the input program when a data structure should be replaced by another. For example, the model could output that `HashSet` should be

used for the first 100 interface operations of a sequence, and `LinkedHashSet` should replace it afterwards. However, since our framework does not focus on dynamically replacing data structures at runtime, we need to select only one data structure as the final output. We propose two methods for this: selecting the most recommended implementation and the last recommended.

Chapter 5

Experimental evaluation

In this chapter, we will present the datasets generated to test the proposed framework in Section 5.1, and evaluate the effectiveness of the proposed feature-based and model-based approaches, in Sections 5.2 and 5.3. Next, in Section 5.4 we will present results on the application of both models on real-world benchmarks, comparing our models to the best data structures found empirically and the ones chosen by the developer. Finally, in Section 5.5, we will present different alternatives that were considered, but that are not present in our final models. All the experiments described in this chapter were performed on a system with Intel Xeon microarchitecture, and a GeForce GTX GPU. The detailed system configuration is described in Table 5.1.

Table 5.1: System configuration

CPU Model	Intel (R) Xeon (R) CPU E5520 @ 2.27 GHz
CPU core count	16
Memory	50 GiB
Operating system	Debian GNU/Linux 9.12 (stretch)
GPU Model	GeForce GTX 1080 Ti
Cuda Version	9.0
OpenJDK Version	1.8.0_272
OpenJDK Runtime Environment	build 1.8.0_272-8u272-b10-0+deb9u1-b10
OpenJDK 64-bit Server VM	build 25.272-b10, mixed mode
Javac version	javac 1.8.0_272

5.1 Programs data

In this work, we focus on the `Set` abstract data type. It is a well-known Java collection interface used for algorithms that require storing a collection of unique values. Its implementations can be grouped into general-purpose and special-purpose. We focus on the special-purpose implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. To generate accurate machine learning models, it is important to have datasets that are representative of the many different behaviors of these data structures in real-world applications. Representative training sets help avoid overfitting the model. Overfitting happens when a model fits the training data too well by memorizing various peculiarities such as noise, rather than finding a general predictive rule [Hawkins, 2004]. However, the large design space makes the creation of representative training sets complex. It is hard to understand the behavior of real-world benchmarks before instrumentation, making them ineffective for training the learning models.

As an example, we have instrumented the benchmarks from the DaCapo suite [Blackburn et al., 2006]. The suite is composed of a collection of open-source, client-side Java benchmarks, which comes with built-in evaluation of execution time and memory consumption. Our instrumentation tool could find instances of `Set` in six benchmarks from DaCapo: `antlr`, `avrora`, `bloat`, `luindex`, `lusearch`, and `pmd`. From these benchmarks, we extracted only 135 sequences of operations that could be used for training. This number is not enough to train an accurate learning model. Therefore, we propose a program generator, which creates several synthetic applications that explore different behaviors of multiple data structures.

5.1.1 Program generator

Algorithm 1 presents the pseudocode of the proposed generator. We create an empty set and a queue of L numbers, in lines 2 and 3. The queue is initialized by taking the modulo with a constant, which is an input parameter of the algorithm. The generator receives a configuration file with the following constants as input: L, which limits the number of operations invoked in the code; M, N, O, P, and Q, which define if we add e to our set (line 12), remove e from it (line 14), search for e (line 16), get the maximum element from the set (line 18), or simply iterate through the set once (lines 20 and 21). By changing the values of these constants, we can simulate the behavior of different data structures.

We have decided not to use random functions in our code, in order to better control the generated applications. For example, we can easily limit the number of

```
1  ApplicationGenerator (config)
2      Q ← empty queue ;
3      S ← empty set ;
4      for i ← 0; i < config.L; i ← i + 1 do
5          Q.push(i%config.M) ;
6      end
7      i ← 0 ;
8      while Q is not empty do
9          i ← i + 1 ;
10         e ← Q.pop() ;
11         if i % config.N == 0 then
12             S.add(e) ;
13         else if i % config.O == 0 then
14             S.remove(e) ;
15         else if i % config.P then
16             S.contains(e) ;
17         else if i == config.Q then
18             max ← max(S) ;
19         else if i == config.R then
20             foreach elem ∈ S do
21             end
22     end
```

Algorithm 1: Pseudocode of the proposed application generator.

operations called in the code with the constant L, since this value controls the number of iterations of the loop in line 8 of Algorithm 1. It is also easy to estimate the number of instructions called in the program, grouped by operation type, before running the generator. This makes it easier to simulate situations where we perform one specific operation more, for example, performing more additions. By controlling the number of operations called, we can simulate different behaviors.

It is important that our training sets are also representative of the multiple implementations of `Set` available. To create training examples for the different implementations, we simply run the program generator with our program data extractor, which was described in Chapter 3. We use our instrumentation to create three versions of our generator: one using `InstrHashSet`, one with `InstrLinkedHashSet`, and the last one with `InstrTreeSet`. Then, we run each version with the same configuration files, check which version is faster, and add this sample to our training set. However, since we can not know for certain when one implementation is better than the others, creating a balanced training set regarding the three data structures is still a hard task. We chose not to use the `Set` implementations `ConcurrentSkipListSet` and `CopyOnWriteArraySet`

as they are recommended for concurrent execution with multiple threads. Yet, we can easily add these two implementations as alternative data structures for our framework as well.

5.1.2 Datasets

We used the proposed generator to create four datasets: small, medium, large, and mixed sequences. The small dataset contains sequences with lengths between 25 and 500. For medium sequences, the length is between 500 and 1,000; and between 1,000 and 12,000 for large sequences. The mixed dataset contains all of the sequences of the other three datasets. Table 5.2 presents the number of samples of each class for each dataset. Even though our generator can easily create different examples, it is hard to generate samples of an specific class. Given a configuration file for the generator, we can calculate how many operations of each type will be called. However, it is a complex task to understand which data structure will run faster with that configuration. We can only know which data structure is the fastest after executing the generated application and measuring its runtime. For this reason, the classes are not always equally represented in our datasets. Besides, equally represented classes is not our focus here, instead, we aim on creating a dataset that represents real application behavior.

Dataset	**Sequence length**	`HashSet`	`LinkedHashSet`	`TreeSet`	**Total**
small	$25 \leq x \leq 500$	1786	1511	640	3937
medium	$501 \leq x \leq 1000$	481	330	76	887
large	$1001 \leq x \leq 10000$	1720	717	133	2570
mixed	$25 \leq x \leq 10000$	3987	2558	849	7394

Table 5.2: Number of samples of each class in each dataset.

5.2 Feature-based sequence classification model

In this section, we present analyses regarding the feature-based model described in Section 4.1. We will present an analysis of the distribution of the features extracted from the datasets described in the previous section and a study on the effectiveness of each feature for the learning model. Table 5.4 presents a profile of the mixed dataset

that was generated with the Pandas Profiling[1] tool. We show the distribution of seven features out of the 88 features described in Section 4.1.1: the number of `add` (A), `remove` (R), `contains` (C), and `iterator` (I) operations; the sequence length; the median and the maximum number of elements in the dataset throughout the execution of the program. Table 5.3 presents the definition of these seven features. We chose to display only these features as the others are mainly a combination of the first four.

Table 5.3: Definition of the seven features analyzed in this section.

A	Number of `add` operations in the sequence
C	Number of `contains` operations in the sequence
I	Number of `iterator` operations in the sequence
R	Number of `remove` operations in the sequence
len	Total number of operations in the sequence
max	Maximum number of elements in the data structure
median	Median number of elements in the data structure

This analysis presents some interesting aspects of our mixed dataset. Table 5.4 presents, for each feature, the percentage of values equal to zero, the minimum, maximum, mean, and median of values. All of the sequences are composed of at least one and at most 5,550 `add` operations. It is the only instruction that is present in all of the sequences. In the other extreme, the `iterator` operation is absent in 32% of the sequences. The length feature presents information on the number of operations in the sequences. While the shortest sequence contains only 26 instructions, the longest sequence is composed of 8,368 operations. The average sequence length is 1,390.6 instructions. The median and max features refer to the number of elements in the data structure during the whole execution of the program. At any execution point, there are at most 5550 elements in the set, so the data structures used here are of moderate size.

5.2.1 Feature selection

It is important to understand which subset of features is relevant for our feature-based model or if all of them contribute to selecting different data structures. Using more features than necessary may add noise to the model and slow down convergence [Guyon and Elisseeff, 2003]. We use the information gain metric to

[1] https://pandas-profiling.github.io/pandas-profiling/docs/master/rtd/

Feature	Zeros (%)	Min	Mean	Median	Max
A	0.0	1	519.90	157.0	5550.0
C	24.1	0	279.57	65.0	4967.0
I	32.0	0	236.82	47.0	4950.0
R	13.7	0	354.36	93.0	5124.0
len	0.0	26	1390.66	519.0	8368.0
median	2.0	0	219.10	49.5	2775.5
max	0.0	1	423.65	78.0	5550.0

Table 5.4: Distribution of a few features for the mixed dataset.

decide which features are relevant for our model. The information gain measures the reduction in entropy or surprise when making changes to a dataset. In feature selection, information gain is also called mutual information, and is defined as the amount of information one can obtain from one random variable given another [Witten and Frank, 2002]. If the mutual information between one feature and the target classes is high, then this feature brings a lot of information to the model. We have used the Sci-kit Learn library (version 0.22.1) [Pedregosa et al., 2011] to calculate the mutual information between the 88 features described in Chapter 4 and the target classes. Table 5.5 presents the information gain for the five features with higher gain. Note that the top five features are the same for the small, medium, and mixed datasets. On the other hand, for the large dataset, the number of `add` operations brings more information than the length of the sequences, and the number of `reduce` instructions is more important than the number of `iterator` operations. The full table with the information gain values for all features is in Appendix B.

Small		Medium		Large		Mixed	
Feature	**Score**	**Feature**	**Score**	**Feature**	**Score**	**Feature**	**Score**
max	0.4189	max	0.4605	max	0.4267	max	0.4267
median	0.3945	median	0.4220	median	0.3902	median	0.3947
len	0.3908	len	0.3287	A	0.3702	len	0.3496
A	0.3097	A	0.2918	len	0.2580	A	0.3315
I	0.2520	I	0.2593	R	0.2023	I	0.2135

Table 5.5: Information gain for the top five features for the four datasets.

5.2.2 Model evaluation

Having a ranking of features generated according to the information gain, we used the features incrementally to train a classifier. We started with one and increased the size of the set up to 88. This procedure was performed using five-fold cross-validation using from one to 88 features. For example, when using two features, we used only the top two features with higher information gain.

We ran our experiments with four classifiers: Random Forest (RF), k-Nearest Neighbors (KNN), Support Vector Machine (SVM), and Decision Tree (DT). The classifiers were implemented with the Sci-kit Learn library. During each iteration of cross-validation, we performed a grid search to tune the values of each parameter of the method in the training set, and used the model with the best parameters to predict the testing samples. The grid search method also uses the cross-validation approach to find the best parameters, but in this case, we used only three folds. Table 5.6 presents the search space for grid search with RF, SVM, and KNN. Usually, the only hyperparameter that is tuned for decision tress is the depth of the tree, so we did not perform grid search for this model. Instead, we use entropy as a criterion for split and we do not limit the depth of the tree.

Model	**Hyperparameter**	**Values**	**Default**
RF	Number of trees	10, 100, 1000	10
	Max features	$\sqrt{features}$, 0.25, 0.5, 0.75	$\sqrt{features}$
	Max depth	None, 2, 4, 6, 8, 10, 20	None
SVM	γ	0 or 2^x for x in -15, -13, ..., 1, 3	0
	C	1 or 2^x for x in -5, -3, ..., 13, 15	1
KNN	Number of neighbors	3, 5, 11, 19	5
	Weights	uniform, distance	uniform
	Metric	euclidean, manhattan	minkowski

Table 5.6: Search space for grid search with RF, SVM, and KNN.

Results were evaluated using the F1-score, which is the harmonic mean of the precision and recall measures. In binary classification, precision is the fraction of true positive examples among all of the samples classified as positive by the model. Recall is the fraction of examples classified as positive among the total number of positive examples. Our work deals with three classes, so the F1-score is calculated as the average of F1-score for each class, weighted by the number of instances of each label. This metric is commonly used for imbalanced datasets like ours. We have used Sci-Kit

Learn's `metrics` module to calculate the weighted F1-score for our learning models both during training and testing.

Figure 5.1 summarizes the effectiveness improvement of each classifier according to the number of features used. While the Decision Tree classifier and the Random Forest had similar values of F1-score during training, RF reached the highest values of F1-score during testing on all datasets. The SVM was the worst classifier during training. However, it did better than the k-Nearest Neighbors classifier during testing depending on the dataset and the number of features used. The model produced by all classifiers converges fast as we increase the number of features considered, so it is not necessary to use all of the 88 features during training. We selected the best number of features for each dataset by finding the point in the training curve where the F1-score became the highest. This point was different for the datasets, but all of them reached a F1-score value of 0.99 during training and 0.88 during testing.

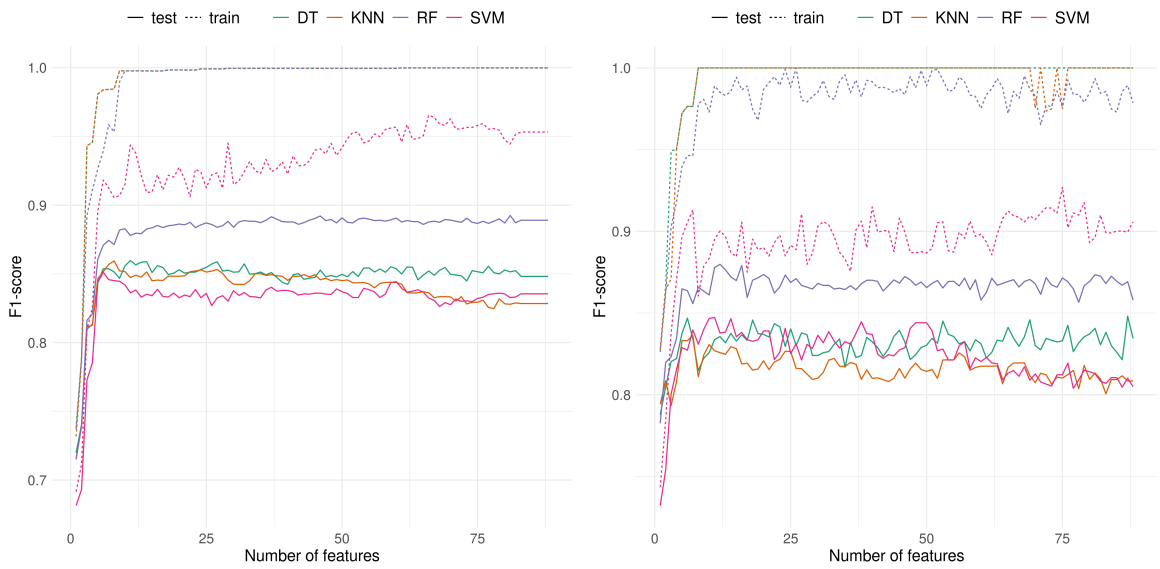
Table 5.7 presents the features that were selected for each dataset. They are sorted from the higher to lower information gain value. The medium dataset needs the most number of features and it is also the dataset with the least number of samples. The large and mixed datasets needed the least number of features to be effective. Table 5.8 presents the best parameters for each dataset when using the features in Table 5.7. Experiments described next use these features and parameters for the Random Forest classifier.

Dataset	**Number of features**	**Features**
small	10	max, median, len, A, I, AA, AI, IA, R, II
medium	11	max, median, len, A, I, AA, AAA, R, IA, RI, C
large	7	max, median, A, len, R, AA, I
mixed	7	max, median, len, A, I, AA, R

Table 5.7: Final features selected for each dataset using the Random Forest classifier.

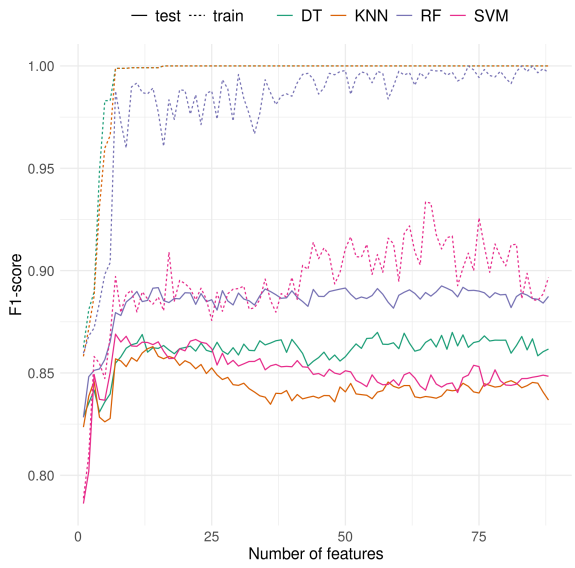
Dataset	**Number of trees**	**Max features**	**Max depth**
small	100	0.75	10
medium	100	0.75	10
large	100	0.25	None
mixed	1000	0.25	20

Table 5.8: Final values for the hyperparameters of the Random Forest classifier for all datasets.

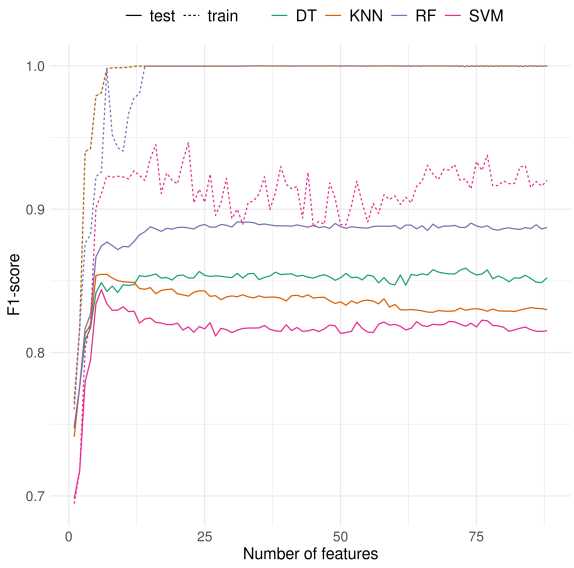


(a) Small dataset.

(b) Medium dataset.



(c) Large dataset.



(d) Mixed dataset.

Figure 5.1: Effectiveness of the proposed feature-based model with four different classifiers and four datasets with different number of features.

Another point to account for is that our datasets are imbalanced, as the classes are not equally represented. The machine learning community has addressed the issue of class imbalance in two ways: assigning distinct costs to training examples or re-sampling the original dataset. In this work, to deal with data imbalance, we used two re-sampling methods: Random Oversampling (ROS) and Synthetic Minority Oversampling (SMOTE) [Chawla et al., 2002]. In the former, instances of the minority class

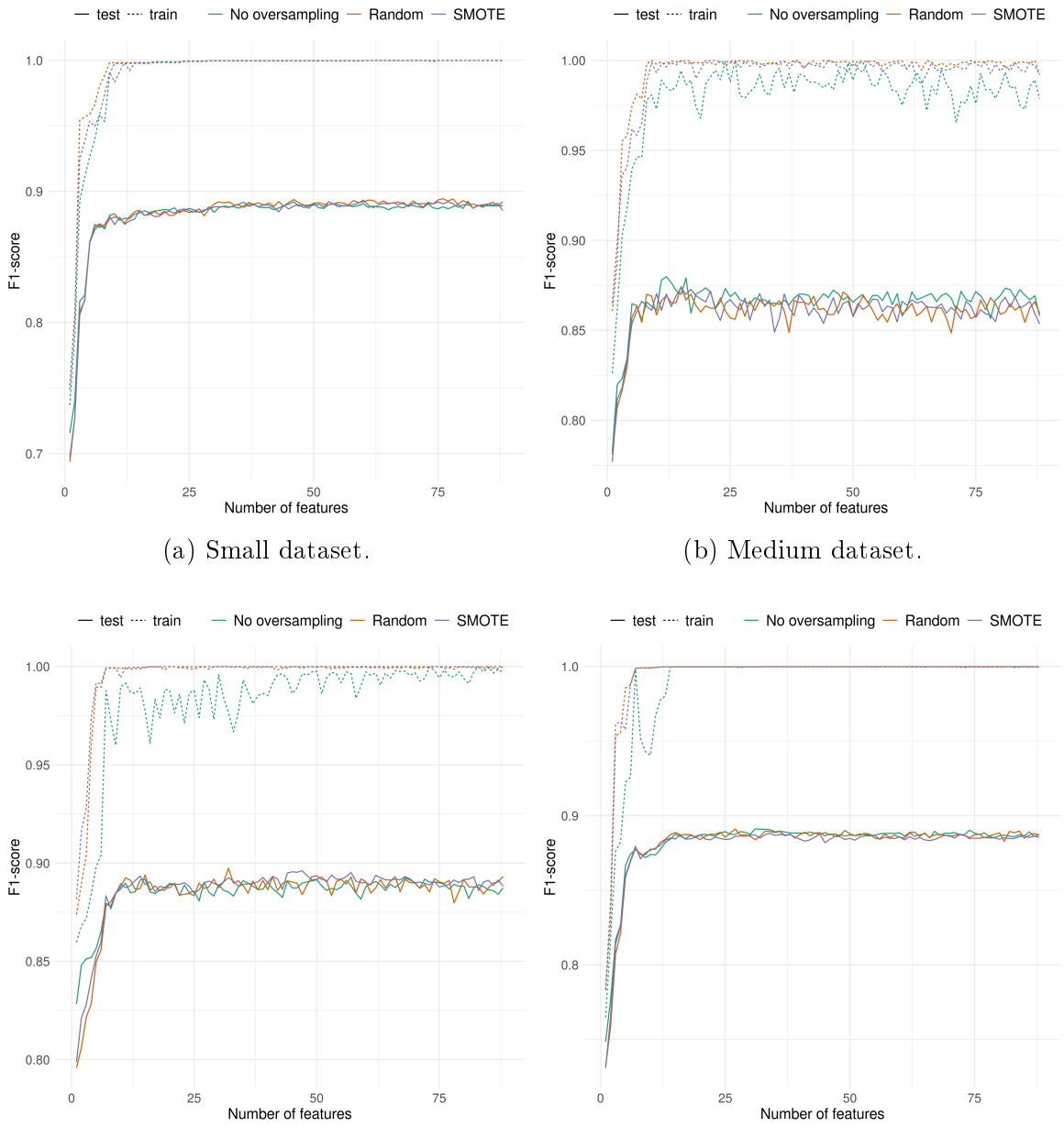
are randomly duplicated. The latter approach re-samples the minority class by adding new synthetic samples instead of simply duplicating original examples. In our implementation, we used both methods from the imbalanced-learning library [Lemaître et al., 2017]. Figure 5.2 presents the effectiveness of the RF model with the original dataset and using the oversampling methods described above. While there are some differences between the three methods, the training and testing curves are really similar for all datasets, and specially for the mixed one. We believe that performing the oversampling adds complexity to the models and the results show it does not improve the values of F1-score, so we still use the RF model with the original dataset for following experiments described in Section 5.4.

We performed one last experiment with the features in Table 5.7 to measure the execution time of the feature-based model. We extracted the average execution time for the feature construction for each dataset. The time is really small as at most only 11 features are used. Table 5.9 presents the execution time of feature construction and model training for each dataset. It takes less than one second to perform feature extraction on the small, medium, and large datasets, while it takes a little bit more than one second for the mixed dataset. Furthermore, it does not take long to train the datasets, with medium and large needing less than one second to train. The highest execution time is for the small dataset, which takes almost 7 seconds to train. Therefore, our feature-based approach is a really fast, and does not a lot of time overhead to the program.

Dataset	**Feature extraction (sec)**	**Model training (sec)**
small	0.2137	6.8778
medium	0.1112	0.2717
large	0.7478	0.2471
mixed	1.0401	5.0886

Table 5.9: Execution time, in seconds, of feature construction and model training for all datasets.

Our feature-based model for sequence classification achieved an F1-score value of 0.88 when trained with the RF classifier and the mixed dataset. This model is the most relevant for this work as it might better represent a real-world scenario, due to its variation on the sequence length. Additionally, the model is really fast, taking less than ten seconds both to extract the relevant features and to train the model.



(a) Small dataset.

(b) Medium dataset.

(c) Large dataset.

(d) Mixed dataset.

Figure 5.2: Effectiveness of the proposed feature-based model using the Random Forest classifier and different methods of oversampling.

5.3 Model-based sequence classification with LSTM

In this section, we present the evaluation results for our second model described in Section 4.2. As previously mentioned, we need to pre-process our sequences before feeding them to the LSTM network. In these experiments, we have used the sliding

window approach to split our data into smaller subsequences of ten operations. At each step of this method, the sliding window also skips ten characters. Both values were chosen empirically. At first, we proposed starting with two values for the window length: the minimum and median sequence length of the sequence. However, that did not generate as many subsequences as we needed for training, so we decreased the window length to ten.

Due to the splitting method previously described, this model is capable of recommending alternative data structures for different points in the execution of the input program. The number of data structure swaps in the application is directly linked to the number of characters that are skipped during window sliding. To decide on this number, we performed an analysis that found the number of data structure swaps that did not slowdown the original programs in our datasets. From this analysis, we decided on always skipping ten characters when generating new subsequences with the sliding window. The full analysis will be described in details in Section 5.5.

Even though the sliding window method is practical for augmenting the number of samples, it can generate identical subsequences with different labels. For this reason, we removed most sequences that generated duplicate subsequences from different classes. Table 5.10 presents the final number of subsequences used for training our LSTM. The ratio of duplicate sequences from different classes is below 1% for each dataset, so this should not affect our results.

Dataset	**Total of subsequences**	**Duplicate subsequences**	**Ratio (%)**
small	122,965	242	0.1968039686
medium	81,963	234	0.2854946744
large	826,428	23	0.0027830616
mixed	1,031,356	553	0.0536187311

Table 5.10: Number of subsequences generated from each dataset with the sliding method, and the ratio of duplicated subsequences from different classes.

We have used the Keras library to implement the LSTM model used in this work. It is composed of only one hidden layer, 70 inputs, and three output neurons. We performed five-fold cross-validation in order to train our model, tuning the following parameters: number of neurons in the hidden layer, dropout and learning rates, and number of training epochs. Table 5.11 presents the values tested during parameter tuning. We used SciKit Learn's `StratifiedKFold` class to create the training and testing sets for cross validation. It splits the data into folds that preserve

the percentage of samples for each class. To avoid leaking information to the testing phase, we performed the split in the original sequences and only then generated the subsequences from it. This way, we ensure that the testing set does not contain pieces of sequences from the training set.

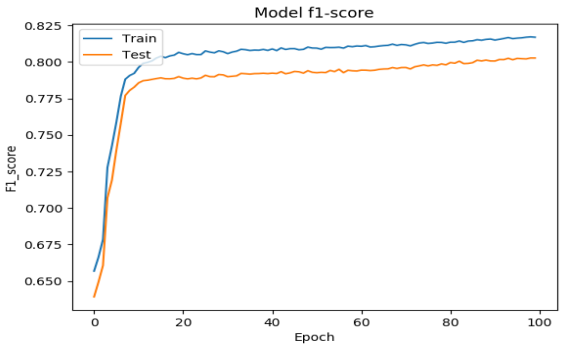
Parameter	**Values**
Neurons in the hidden layer	50, 100, 150, 200, 250, 300
Dropout rate	0.5, 0.6, 0.7, 0.8, 0.9
Learning rate	0.1, 0.01, 0.001, 0.0001, 0.00001
Training epochs	50, 100, 200, 300, 400, 500

Table 5.11: Hyperparameters tested during cross validation.

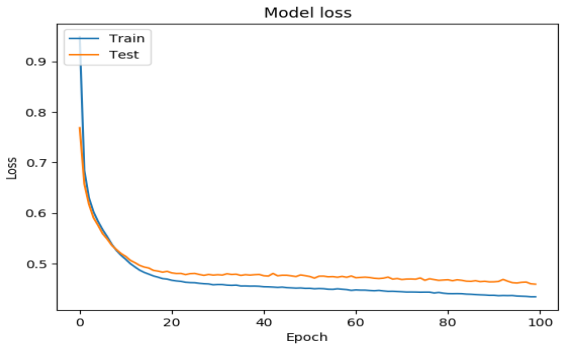
Figure 5.3 presents the effectiveness of our final LSTM model. It uses 100 neurons in the hidden layer, a dropout rate of 0.5, and a learning rate of 0.00001. We have trained it for 100 epochs, and it reached an F1-score of 0.87 for the mixed dataset. Similar to the feature-based model, we used the weighted F1-score metric from the Sci-Kit Learn library for training and testing. For all datasets, our network reached an F1-score value of at least 0.82 during training and 0.80 during testing. The loss function was smaller than 0.5 for all training and testing sets. These results show that our network is capable of generalizing and having good values of F1-score in both training and testing phases.

We also analyzed the execution time of the LSTM model. Table 5.12 presents how long it takes to train all datasets with the LSTM network running on a GeForce GTX 1080 Ti GPU, with Cuda 9.0. Training this model takes longer than the feature-based one mainly due to the huge difference on the data size, as our sliding window approach generates a lot more samples for the LSTM training. For this reason, training takes from 30 minutes to 14 hours depending on the dataset size. Even though training is not fast, this phase needs to be performed only once and then the trained model can be used to predict other unseen data.

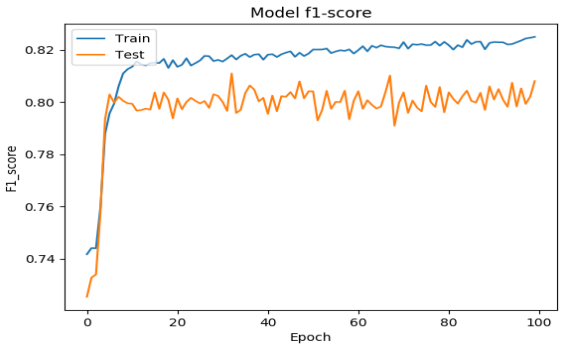
The previous sections presented an evaluation of the feature-based and the model-based with LSTMs sequence classification models from our framework. Both models reached an F1-score of 0.87 when trained with the mixed dataset, which, as previously mentioned, is the closest representation of a real-world scenario. The feature-based model should be used for small programs and when time is a priority, as feature extraction and training time is really low. On the other hand, the model-based with



(a) F1-score for the small dataset.



(b) Loss for the small dataset.



(c) F1-score for the medium dataset.

(d) Loss for the medium dataset.

(e) F1-score for the large dataset.

(f) Loss for the large dataset.

(g) F1-score for the mixed dataset.

(h) Loss for the mixed dataset.

Figure 5.3: Effectiveness of the proposed LSTM model for the four datasets.

Dataset	Execution time (min)
small	51.1053
medium	34.9117
large	332.0877
mixed	882.5953

Table 5.12: Execution time, in minutes, of LSTM training for all datasets.

LSTMs approach is recommended for large applications, as it outputs different data structures for different points in the execution, taking into account the diverse behaviors of the application throughout its execution.

5.4 Application on real programs

We have previously demonstrated the effectiveness of our framework with a synthetic dataset created using our program generator. While the generator effectively simulates different behaviors of data structures, it is not a real algorithm. It only calls different interface functions according to a configuration file. In this section, we apply our framework to real algorithms: two created by us, and five benchmarks from the DaCapo suite. In this case, we used the models already trained with the RF classifier and LSTM and gave this new data as input to test the model.

5.4.1 Synthetic programs

We have implemented two well-known algorithms to help illustrate how to apply the framework proposed in this work. Algorithm 2 receives two strings as input and checks if they are anagrams. Algorithm 3 uses a brute force approach to find the number of pairs in a set whose sum is equal to a given input. The latter receives a list of possible sums and outputs the pair of values for each sum.

We implemented Algorithms 2 and 3 using an instance of `HashSet`. We used our framework to verify if this implementation was indeed the fastest one for both programs. We used the program data extractor to collect information on the `Set` implementation used. We also created other two versions of each algorithm to empirically find which implementation was the fastest. We used a random generator to create inputs for both programs. The two input strings were composed of 10,000 characters. Similarly, the list of target sums in Algorithm 3 was composed of 10,000 values varying from 1 to 15,000, while the list of numbers added to the `Set` contained 100 numbers varying from 1 to 20,000.

```
1  AreAnagrams (string s1, string s2)
2  |  S ← empty set ;
3  |  for elem in s1 do
4  |  |  S ← S + elem ;
5  |  end
6  |  for elem in s2 do
7  |  |  if elem not in S then
8  |  |  |  print "s1 and s2 are not anagrams." ;
9  |  end
10 |  print "s1 and s2 are anagrams" ;
```

Algorithm 2: Pseudocode of an algorithm to find if two strings are anagrams.

```
1  FindSums (list of numbers, list of target sums)
2  |  setS ← list of numbers ;
3  |  for sum in targetsums do
4  |  |  for elem1 in S do
5  |  |  |  for elem2 in S do
6  |  |  |  |  if elem1 ≠ elem2 then
7  |  |  |  |  |  if elem1 + elem2 = sum then
8  |  |  |  |  |  |  print elem1 plus elem2 are equal to sum. ;
9  |  |  |  end
10 |  |  end
11 |  end
```

Algorithm 3: Pseudocode of an algorithm to find two values whose sum is equal to a given input.

We have used the feature constructor component of our framework to extract features from the sequences for our feature-based model, and to pre-process the sequences for the model-based with LSTMs approach. For the former, we used the features for the two sequences of length 10,000. For the latter we generated 2,000 subsequences for Algorithm 2 and 50,510 subsequences for Algorithm 3. Both models were previously trained using the mixed dataset.

For both algorithms, the two learning models correctly suggested to keep using `HashSet` in Algorithm 2 and to replace it with `LinkedHashSet` in Algorithm 3. Table 5.13 presents the execution time, in milliseconds, of both Algorithms using the three different implementations of `Set`.

	AreAnagrams	FindSums
HS	64.74	26664.67
LH	71.59	18220.44
TS	101.85	21806.93

Table 5.13: Execution time, in milliseconds, of the `AreAnagrams` and `FindSums` algorithms with different implementations of `Set`.

5.4.2 DaCapo

The DaCapo benchmarks suite is a set of open-source, Java real world applications with non-trivial memory loads. It comes with built-in evaluation of execution time and memory consumption and is used for Java benchmarking by the programming language, memory management and computer architecture communities. In this section, we present the results of applying our trained learning models on some benchmarks from DaCapo.

We have used the program data extractor described in Section 3.1 on the `antlr` and `bloat` benchmarks from the 2006 DaCapo suite and the `avrora`, `luindex`, `lusearch`, and `pmd` benchmarks from the 2009 version. We only used the benchmarks from the older version that are not available in the newer one. We used a bash script[2] written by Eric Bodden from McGill University to process the 2006 benchmarks with our Soot-based program data extractor. For the 2009 suite, we used TamiFlex [Bodden et al., 2011], a tool for handling reflection in the static analysis of Java programs. In total, we extracted the interface operations of 155 instances of `Set` in the code. We created three versions of each benchmark using one of `InstrHashSet`, `InstrLinkedHashSet`, and `InstrTreeSet`. We ran each version ten times and calculated the total runtime by addition of the execution time of each interface operation extracted. With this approach, we found the best implementation for each `Set` instance, which are used as class labels for testing our learning models. The class distribution found is the following: 91 samples of `HashSet`, 31 of `LinkedHashSet`, and 33 of `TreeSet`.

We used the feature constructor described in Chapter 4 to transform the extracted information from DaCapo into inputs for our already trained models. Table 5.14 presents the weighted F1-score values of the models when compared with the best data structures found empirically as described above. The **dataset** column identifies which dataset was used to train our models. The **original** column presents the accuracy of the original benchmark, identifying if the developer made a good choice for the data

[2] http://dacapobench.sourceforge.net/batchsoot

structures in the code. We used different approaches to apply our learning models to the DaCapo suite. Since it does not take long to train the feature-based model, we retrained it with the complete datasets using the best parameters found by the grid search with cross-validation. Then, we applied the trained model to the data extracted from DaCapo. On the other hand, training the LSTM network takes longer, so we used the trained model that had the highest F1-score for validation during cross-validation. We did not retrain it with the complete dataset before applying it to DaCapo.

Our feature-based model reached an F1-score of at most 0.34, which shows that while this model had great results with our synthetic data, it had a hard time generalizing for real-world applications. This could be explained due to the differences between the data extracted from DaCapo and our datasets. Tables 5.15 and 5.16 present the comparison of the mean, median, and maximum values of the features from DaCapo and the small and medium datasets. The model trained with the small dataset had the lowest F1-score on DaCapo and the model trained with the medium dataset had the highest. The biggest difference between our synthetic datasets and DaCapo is the length of the sequences. The ones from the benchmark suite have a large range, having up to 1,384,527 operations. However, the median length on the medium dataset is 748, similar to DaCapo's 768. This could contribute to the better results achieved by our model trained with the medium dataset. Another huge difference between our sets and DaCapo is the number of addition operations. The latter is composed of sequences that have up to 978,874 operations, while the maximum is 1,000 for the medium dataset and 500 for the small one. However, the median of `add` operations on DaCapo is in the middle of both sets.

Dataset	**Programmer**	**Feature-based**	**LSTM - most**	**LSTM - last**
small	0.43	0.13	0.07	0.09
medium	0.43	0.34	0.09	0.10
large	0.43	0.21	0.12	0.28
mixed	0.43	0.22	0.45	0.44

Table 5.14: F1-score of original benchmarks and proposed learning models when compared with the best versions found empirically.

Our LSTM network is capable of finding after how many operations in the program execution the current data structure implementation should be replaced. However, as this work does not focus on dynamically changing the data structure, we use two different methods to decide on only one implementation to be used throughout

	Mean		Median		Max	
	small	**DaCapo**	**small**	**DaCapo**	**small**	**DaCapo**
a	92.69	15520.46	70	168	500	978874
aa	27.10	6618.6	0	52	355	454031
ai	19.22	1058.33	5	0	437	104709
i	88.99	8341.53	27	2	495	405653
ia	19.13	1057.33	5	0	438	104710
ii	45.18	3779.83	0	0	490	165566
len	308.13	30246.13	254	768	500	1384527
max	68.94	178.58	36	15	500	6366
median	36.57	50.99	19	2	250.5	2764
r	70.78	84.70	50	0	497	2817

Table 5.15: Distribution of features for the small dataset and DaCapo benchmarks.

	Mean		Median		Max	
	medium	**DaCapo**	**medium**	**DaCapo**	**medium**	**DaCapo**
a	324.38	15520.46	280	168	1000	978874
aa	27.10	123.82	0	52	999	454031
aaa	87.38	4136.36	0	22	998	288340
len	919.94	30246.13	748	768	5000	1384527
max	252.06	178.58	200	15	1000	6366
median	132.59	50.99	110	2	500.5	2764
r	202.20	84.70	167	0	997	2817
ri	47.41	5.16	0	0	975	320

Table 5.16: Distribution of features for the medium dataset and DaCapo benchmarks.

the whole execution. The first method finds the implementation that was the most recommended for a certain instance. The second chooses the last recommended implementation. The F1-score values for both methods on the DaCapo benchmarks is presented in Table 5.14. Our LSTM, trained with the mixed dataset, reached an F1-score value of 0.44. Even though this is very low, it is a bit higher than F1-scores reached by the programmer in the current implementations.

Since our LSTM network can make suggestions of when to replace one data structure with another, it is important to consider that swapping a data structure implementation at runtime has a cost. So it is desirable do minimize the number of swaps throughout the execution of the application. Table 5.17 presents the median and maximum of number of swaps suggested for DaCapo. The median suggested varies between 2 and 4 swaps, which is really low and would not cause too much overhead on the program execution. The maximum values recommended go up to 8,486, which

is a high number, but it is a direct reflection of the length of this sequence, which is composed of 1,384,527 operations.

Dataset	Median	Max
small	2	3264
medium	2	5103
large	4	8486
mixed	2	6088

Table 5.17: The median and maximum of number of swaps recommended by our LSTM network for the DaCabo benchmark suite.

Even though our learning models had difficulties to generalize for real-world applications, both models correctly predicted the data structure for a few samples when the original developer did not. Tables 5.18, 5.19, and 5.20 presents the confusion matrices when comparing the best data structures for DaCapo found empirically to the ones recommended by our learning models, and chosen by the original developer. From Table 5.18, we find that the feature-based model predicted a lot of instances of `HashSet` as `LinkedHashSet`. This confusion is expected as the two implementations are the most similar to each other. On the other hand, this model correctly predicted 14 instances of `LinkedHashSet`, and eight instances of `TreSet` that were not chosen by the developer. This shows that this model can still help the developer with the task of selecting which data structures to use on a given application, even though it also mispredicted samples that the developer chose correctly.

The model-based with LSTMs approach had a performance similar to the developer, when trained with the mixed dataset. Tables 5.19 and 5.20 present that both our model and the developer chose to mainly use `HashSet` in the benchmarks, with 100% and 98% of recall, respectively. For our model, this could be explained by the large number of `HashSet` samples in our unbalanced training sets. On the other hand, the original developer may have primarily chosen this implementation as it is highly recommended by Java's documentation. It is worth highlighting that our model-based approach correctly predicted the best implementation for two instances when the developer did not: one for `HashSet`, and another for `TreeSet`. Besides, the developer also wrongly used two instances of `TreeSet` when `HashSet` should have been used.

Finally, we used the program reconstructor described in Section 3.2 to create versions of the DaCapo suite using the data structures recommended by our models.

		Predicted class			
		HS	LH	TS	Recall (%)
True class	HS	12	77	2	13
	LH	11	14	8	42
	TS	7	16	8	26
Precision (%)		40	13	44	

Table 5.18: Confusion matrix of the RF classifier, trained with mixed dataset, applied to the DaCapo suite.

		Predicted class			
		HS	LH	TS	Recall (%)
True class	HS	91	0	0	100
	LH	33	0	0	0
	TS	30	0	1	3
Precision (%)		59	0	100	

Table 5.19: Confusion matrix of the model-based with LSTMs approach, trained with mixed dataset, applied to the DaCapo suite.

		Predicted class			
		HS	LH	TS	Recall (%)
True class	HS	89	0	2	98
	LH	33	0	0	0
	TS	31	0	0	0
Precision (%)		58	0	0	

Table 5.20: Confusion matrix of the choices of the original developer.

Figure 5.4 presents the speedup of the versions generated with our framework, compared to the original benchmarks. In the chart, the benchmarks are represented with their first two letters and a letter representing its input size. For example, **an-l** represents the `antlr` benchmark with large input. For most of the benchmarks, there is not a significant difference in execution time between each version. This can be explained by the fact that DaCapo is an older benchmark suite, which has been improved over the years. Besides, the majority of the benchmarks are already pretty fast and take less than five seconds to run. The exceptions are `avrora` with default and large input sizes and `bloat` with default input size. Furthermore, the DaCapo benchmarks do not focus on the performance of data structures, so improving this aspect would not necessarily result in an improvement for the program. The execution time of input and output operations, for example, has way more impact on the benchmarks' performance than

the time needed to perform operations on the data structures used. While there is not a significant difference in runtime between our models and the original code, as the highest speedup was 1.31x, replacing the data structures with our suggestions did not add much slowdown to the benchmarks either, the lowest value being 0.78x.

It is important to mention that we can not replace all `Set` implementation with another in all applications. For example, if one program depends on printing the elements of a set in the order that they were added, we can not use `HashSet`. We can not replace `TreeSet` with another implementation when we expect the set to be sorted either. For the same reason, if the elements in a set are not comparable, we can not use `TreeSet`. We have mitigated this with our customized classes for instrumentation, but that is not true for the original `TreeSet` implementation. Our framework does not take these cases into account. This was not a problem for our synthetic applications, as we only implemented algorithms that could use any `Set` implementation. However, that is not the case for DaCapo. As there are not that many `Set` instances on the benchmark suite, we decided not to take a conservative approach, like filtering the instances to avoid replacing `TreeSet`. We replaced all data structures with the ones recommended by our learning models and relied on DaCapo's verification to check if the new version did not change the original code. This was true for all benchmarks, except for `bloat`. The implementations recommended by our feature-based model did not work on the original code and, for this reason, we did not add its execution time to the chart in Figure 5.4.

5.5 Alternatives considered

We studied different approaches before deciding on the final learning models previously described in this chapter. In this section, we present the alternative approaches that we tried, explaining their positive points, and why we decided to not follow with these ideas.

5.5.1 Dataset with no iterations

At first, we thought that the `add`, `contains`, and `remove` interface functions contained sufficient information of a data structure behavior during the execution of a program. We extracted data for only these operations in our dataset and ran both the feature-based and neural network approaches with the resulting features and sequences. Figure 5.5 presents the histogram of the number of sequences that contain the `add`, `contains`, and `remove` operations in the ith position, in which $0 <= i <= 10$. This

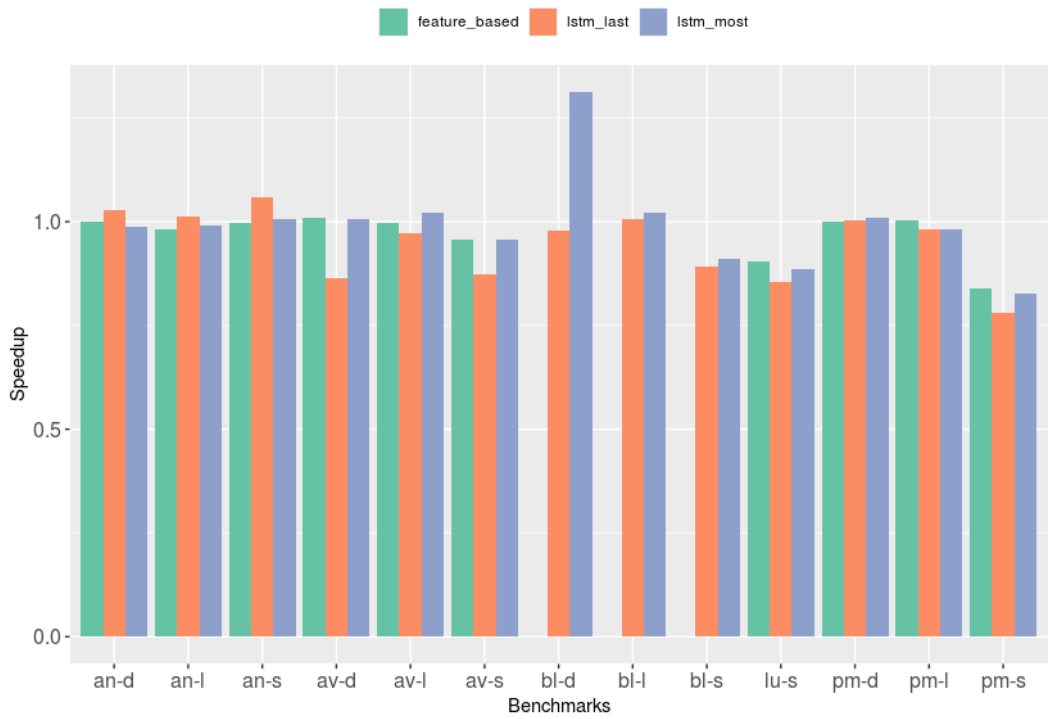
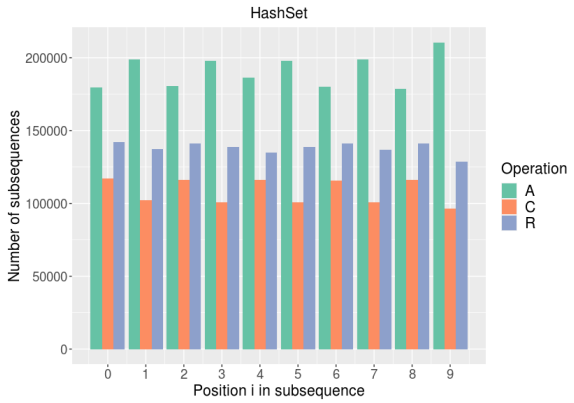


Figure 5.4: Speedup of the versions using the data structures recommended by our learning models when compared to the original benchmark.

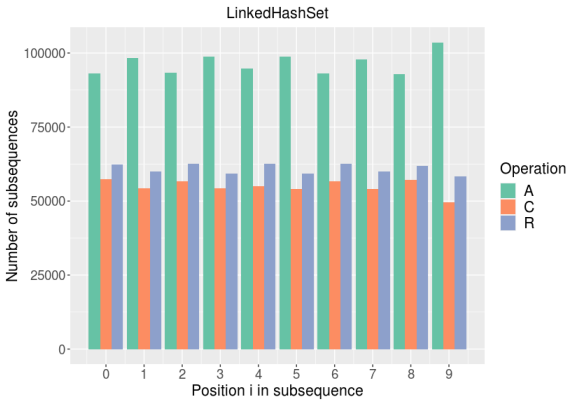
is computed for each of the target data structures, and we can see that there is not a significant difference between them. For all of the classes, the number of `add` operations is higher, followed by `remove`, and then `contains`. With the distribution of operations being similar, it is hard for our LSTM model to differentiate between the classes. Adding the number of `iterator` operations mitigated this problem, as illustrated by Figure 5.6. In this last histogram, we can see how iterations are heavily used in programs that are faster using `LinkedHashSet`. On the other hand, when we use the `TreeSet`, the number of iterations is really low.

5.5.2 No sliding window

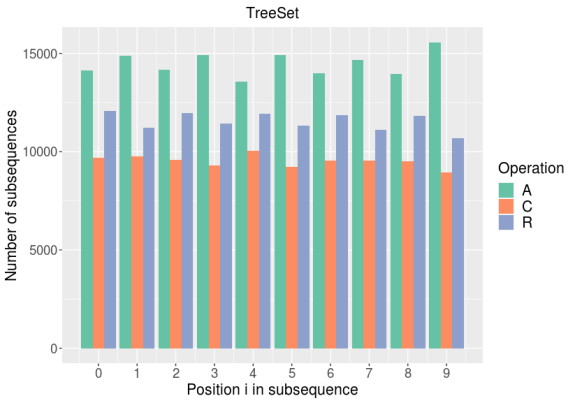
We proposed an alternative approach for splitting a sequence into smaller subsequences. Figure 5.7 presents this approach: after padding the sequence with space characters, we simply split it into smaller chunks of the same length. Each subsequence is then transformed into a one-hot vector as well. This splitting method is simpler as it is not necessary to consider other parameters, such as the window length, and how many steps it takes when sliding. On the other hand, it does not contribute to generating more subsequences like the sliding window method. Our



(a) Sequences that have `HashSet` as the suitable data structure.



(b) Sequences that have `LinkedHashSet` as the suitable data structure.



(c) Sequences that have `TreeSet` as the suitable data structure.

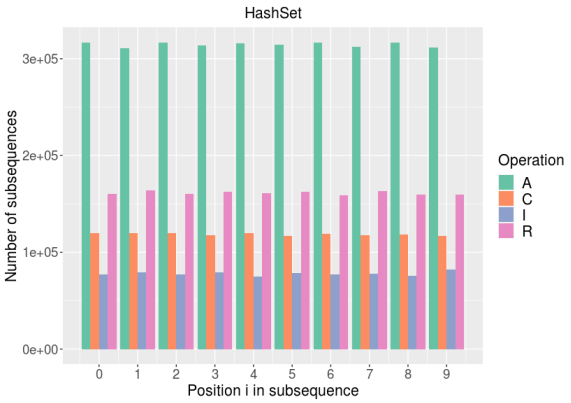
Figure 5.5: Number of sequences, of each class, containing `add`, `contains`, and `remove` operations in the ith position, in which $0 <= i <= 10$.

datasets do not contain enough sequences for training an LSTM network, so generating more subsequences with the sliding window contributed to a larger training set.

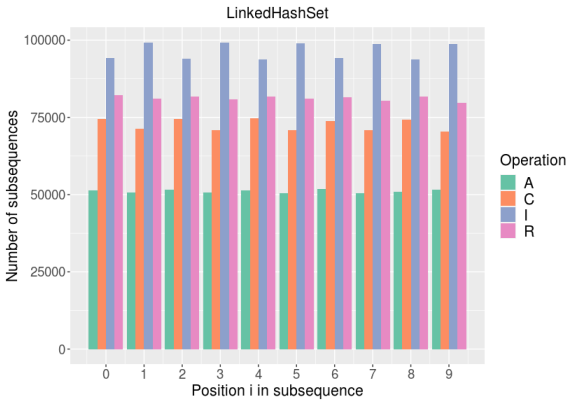
5.5.3 Sliding window parameters

The sliding window approach for splitting a sequence depends on two parameters: the window length, which is the length of the generated subsequences, and the step size, which is the number of characters that we skip when sliding the window. As an example, if we set the window len to 3 and the step size to 2, we generate the following subsequences from the sequence **"ARIACC"**: ARI, IAC, and CC_. These parameters have direct effect on the number of generated subsequences.

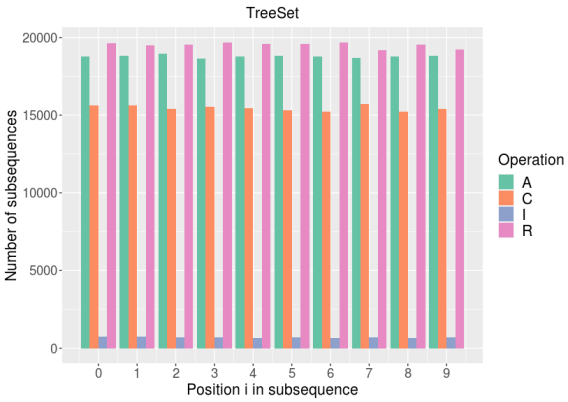
We proposed starting with two different values for the window length: the min-



(a) Sequences that have `HashSet` as the suitable data structure.



(b) Sequences that have `LinkedHashSet` as the suitable data structure.



(c) Sequences that have `TreeSet` as the suitable data structure.

Figure 5.6: Number of sequences, of each class, containing `add`, `contains`, `remove` and `iterator` operations in the ith position, in which $0 <= i <= 10$.

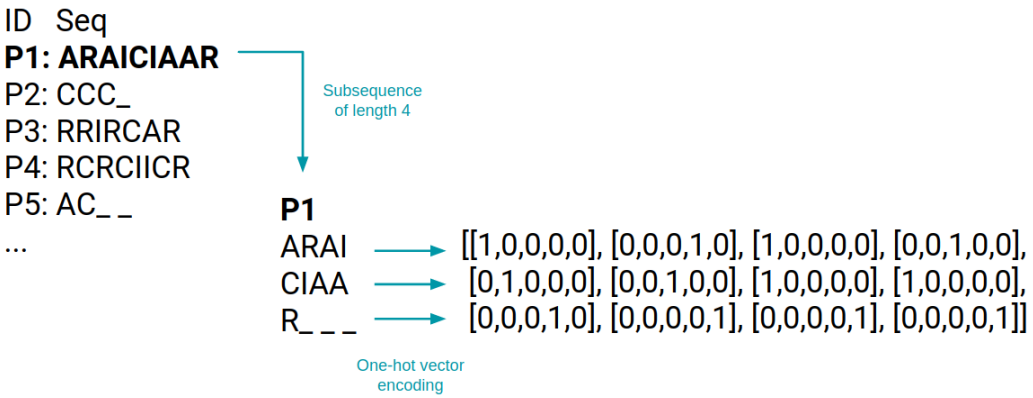


Figure 5.7: Preprocessing of sequences using the split approach.

imum and median sequence length in the dataset. We also set the step size as one. However, after running our LSTM with the small dataset, we thought we still did not

have enough samples for training. For this reason, we decided to choose an arbitrary smaller value that would result in a larger number of sequences. The window length set to ten worked well with our experiments and is the value used in this work.

Our LSTM is capable of suggesting different data structure implementations for different points in the execution of a program due to its sliding window approach. For example, our model could recommend one implementation for the ARI subsequence mentioned above, and another one for the IAC subsequence. However, replacing a data structure causes an overhead, so it is desirable to minimize the number of data structure swaps throughout the execution of an application. The step size directly affects this.

We wrote Algorithm 5.1 to measure the overhead of replacing one `Set` implementation with another. The code creates a new instance of `TreeSet` and add 100 elements to it in lines 11 to 14. The `swap` function in lines 3 to 8, creates a new `HashSet` with the elements in the `Set` passed as parameter, and returns its execution time, in seconds. We call this function 10000 times for warm up (lines 16 to 19), and finally we call it one more time in line 22 to measure the time needed to swap the data structure. We changed the implementation of the data structure in line 11 and the one the replaces it in line 5, and we also varied the number of elements in the set.

```
1  public class Swap {
2
3    private static long swap(Set<Integer> s) {
4      long start = System.nanoTime();
5      s = new HashSet(s);
6      long end = System.nanoTime();
7      return end - start;
8    }
9
10   public static void main() {
11     Set<Integer> s = new TreeSet<>();
12     for (int i = 0; i < 100; i++) {
13       s.add(i);
14     }
15
16     System.out.println("Warming up.");
17     for (int i = 0; i < 10000; i++) {
18       swap(s);
19     }
20
21     System.out.println("Starting actual benchmark.");
```

```
22      long sw = swap(s);
23      System.out.println("Swapping data structures: " + sw);
24    }
25  }
```

Algorithm 5.1: Application creating an instance of `TreeSet` and replacing it for a `HashSet`.

We used the algorithm described above to get the average execution time of performing a data structure replacement in a program. Then, we used our synthetic datasets to calculate the difference of runtime between the code with the fastest and slowest `Set` implementation. By dividing this value by the average swap time, we could find how many data structure swaps could be done during the execution of our input applications. Table 5.21 presents the median and maximum number of swaps for each dataset that does not affect a program's performance. We used this values to find how many steps our sliding window should take at each iteration without decreasing the program's efficiency. With this experiment, the step size was set to ten. While this analysis was only done on our old datasets, which did not contain information on the `iterator` operation, we continued using this value with our new datasets. As previously presented, this value worked well for our experiments, keeping the median of swaps very low for the DaCapo benchmarks.

Dataset	Median	Max
small	3.88	24.27
medium	11.97	51.71
large	24.67	108.55

Table 5.21: Median and maximum number of data structure swaps that can be done in a program without affecting performance.

Chapter 6

Conclusion

This dissertation has presented a framework to automatically select the best data structure for a given application and input. This framework is composed of four components: a program data extractor, a feature constructor, machine learning models, and a program reconstructor. The program data extractor is built on top of the Soot framework and extracts information on the behavior of the data structures used in a given input program. The feature constructor transforms this information into features that are fed to machine learning models capable of automatically selecting data structures for the input program. Finally, the program reconstructor creates a new version of the input program, in binary format, which uses the data structure implementations selected by our models. Even though this work focus on improving the execution time of applications, this framework can be used to improve other performance aspects such as memory and energy consumption.

We have evaluated our framework on synthetic applications and real world programs from the DaCapo benchmarks suite. One of our models was capable of reaching slightly better values of F1-score as the original developer when selecting data structures to be used in the benchmarks. Besides, our models could also correctly select data structure implementations that were not chosen by the developer, but that improved the programs's execution time. When comparing the original benchmarks with the versions generated by our framework, we did not observe any slowdown.

Future work should primarily invest on improving the proposed machine learning models by augmenting the training sets with data from real-world applications. The differences between our synthetic datasets and real benchmarks make it hard to train machine learning models that can perform well on real data. Furthermore, we should also regenerate our synthetic benchmark using specific frameworks for reliable performance measurement in Java. These frameworks would allow us to generate syn-

thetic data with a warmup phase, and multiple iterations. Besides, we could also try to use cost-sensitive classification, in which we aim to minimize the model costs without assuming that all misclassification errors are equal. As an example, developers would normally already choose to use `HashSet` in their code, as it is recommended on Java's documentation, so misclassifying one of these samples would not be as bad as misclassifying `LinkedHashSet` and `TreeSet` samples. So we could add a higher cost to the incorrect prediction of the latter implementations. More work can be done on parameter tuning as well.

For the LSTM model, we should also work on tuning the window length, and the number of characters that it skips with the window slide. After improving the learnings models, we could also work on expanding our framework by collecting examples of parallel programs that use the `ConcurrentSkipListSet` and `CopyOnWriteArraySet` data structures. These two implementations could easily be added as possible classes to be selected by the models. Besides, the framework could be expanded to work with different abstract data types such as `List` and `Map`. Extending its implementations to collect information on interface functions would be enough to make our program data extractor work with these classes as well.

Appendix A

Classes for Instrumentation

In order to extract data structure information from computer programs, we use the Soot instrumentation tool to replace all instances of `Set` in the code with a class used for instrumentation. As an example, all instances of `HashSet` are replaced with `InstrHashSet`. Algorithm A.1 presents its implementation.

```
1  import java.util.HashSet;
2
3  public class InstrHashSet<E> extends HashSet<E> {
4
5    private static List<String> log = Collections.synchronizedList(new
       ArrayList<String>());
6    private long id;
7
8    public InstrHashSet(long id) {
9      super();
10     this.id = id;
11   }
12
13   @Override
14   public boolean add(E e) {
15     long start = System.nanoTime();
16     boolean a = super.add(e);
17     long end = System.nanoTime();
18     InstrHashSet.log.add(this.id + ",A," + start + "," + (end - start) +
       "," + this.size() + ",HS");
19     return a;
20   }
21
22   @Override
```

```
23   public boolean contains(Object o) {
24     long start = System.nanoTime();
25     boolean c = super.contains(o);
26     long end = System.nanoTime();
27     InstrHashSet.log.add(this.id + ",C," + start + "," + (end - start) +
     "," + this.size() + ",HS");
28     return c;
29   }
30
31   @Override
32   public boolean remove(Object o) {
33     long start = System.nanoTime();
34     boolean r = super.remove(o);
35     long end = System.nanoTime();
36     InstrHashSet.log.add(this.id + ",R," + start + "," + (end - start) +
     "," + this.size() + ",HS");
37     return r;
38   }
39
40   public static void printLog() {
41     for (String s : InstrHashSet.log) System.out.println(s);
42   }
43 }
```

Algorithm A.1: Implementation of the `InstrHashSet` class which logs information regarding its interface functions.

`TreeSet` is an implementation of `Set` in which the added elements are sorted. Elements can be ordered using their natural ordering or by a comparator provided at set creation time. If the elements are not comparable, and a comparator is not provided, the code will throw an exception at runtime. To avoid this, we have implemented the `HashComparator` class. It uses natural ordering for primitive types, and sorts the elements using their hash code for other types. Its source code is presented in Algorithm A.2. When replacing any implementation of `Set` with `InstrTreeSet`, our instrumentation class for `TreeSet`, we use `TreeSet`'s constructor with comparator, passing `HashComparator` as argument.

```
1 import java.util.Comparator;
2
3 public class HashComparator<E> implements Comparator<E> {
4
5   @Override
6   public int compare(E arg0, E arg1) {
```

```
7       if (arg0 instanceof Integer) {
8         return ((Integer) arg0).compareTo((Integer) arg1);
9       } else if (arg0 instanceof Double) {
10        return ((Double) arg0).compareTo((Double) arg1);
11      } else if (arg0 instanceof String) {
12        return ((String) arg0).compareTo((String) arg1);
13      } else if (arg0 instanceof Long) {
14        return ((Long) arg0).compareTo((Long) arg1);
15      } else if (arg0 instanceof Boolean) {
16        return ((Boolean) arg0).compareTo((Boolean) arg0);
17      } else if (arg0 instanceof Short) {
18        return ((Short) arg0).compareTo((Short) arg1);
19      } else if (arg0 instanceof Character) {
20        return ((Character) arg0).compareTo((Character) arg1);
21      } else if (arg0 instanceof Float) {
22        return ((Float) arg0).compareTo((Float) arg1);
23      } else if (arg0 instanceof Byte) {
24        return ((Byte) arg0).compareTo((Byte) arg1);
25      } else {
26        if (arg0.hashCode() < arg1.hashCode()) {
27          return -1;
28        } else if (arg0.hashCode() > arg1.hashCode()) {
29          return 1;
30        } else {
31          return 0;
32        }
33      }
34    }
35  }
```

Algorithm A.2: Implementation of the `InstrHashSet` class which logs information regarding its interface functions.

Appendix B

Feature selection

We used our feature constructor to extract 88 different features from the datasets created with our program generator. However, using too many features can add noise to the model, making it important to verify if all features are relevant to the learning model. To do this, we calculated the information gain for each extracted feature. In feature selection, information gain measures the amount of information one feature brings to the model. Table B.1 presents the results.

Table B.1: Information gain for all features using the four datasets.

Small		**Medium**		**Large**		**Mixed**	
Feature	**Score**	**Feature**	**Score**	**Feature**	**Score**	**Feature**	**Score**
max	0.4189	max	0.4605	max	0.4267	max	0.4267
median	0.3945	median	0.4220	median	0.3902	median	0.3947
len	0.3908	len	0.3287	A	0.3702	len	0.3496
A	0.3097	A	0.2918	len	0.2580	A	0.3315
I	0.2520	I	0.2593	R	0.2023	I	0.2135
AA	0.2068	AA	0.2427	AA	0.1973	AA	0.1845
AI	0.1874	AAA	0.1971	I	0.1946	R	0.1648
IA	0.1871	R	0.1859	AAA	0.1874	AI	0.1614
R	0.1411	IA	0.1749	RRA	0.1775	IA	0.1600
II	0.1402	RI	0.1641	RAR	0.1617	AAA	0.1291
RI	0.1373	C	0.1473	CCC	0.1587	ARA	0.1087
IR	0.1330	IR	0.1420	RA	0.1546	RA	0.1035
AAA	0.1217	AI	0.1395	AR	0.1534	RR	0.1004
IC	0.1213	AR	0.1250	RRR	0.1473	AR	0.1003

CI	0.1168	IAI	0.1237	CCA	0.1472	IRI	0.0968
IRI	0.1162	RR	0.1211	IA	0.1471	C	0.0961
AAI	0.1115	RA	0.1141	ARR	0.1464	II	0.0940
ARA	0.1107	IRI	0.1132	AI	0.1342	IR	0.0922
IAI	0.1089	RRA	0.1110	RAA	0.1324	CCC	0.0921
IAA	0.1045	IC	0.1086	ARA	0.1322	RI	0.0919
RAA	0.1039	AAI	0.1077	AAR	0.1301	RRR	0.0892
AIA	0.0995	CRI	0.1020	ACC	0.1277	RAI	0.0887
RII	0.0939	ARA	0.1011	CAC	0.1261	IC	0.0878
RA	0.0933	AIR	0.0998	RR	0.1246	CI	0.0865
IAR	0.0922	II	0.0993	CC	0.1209	IAA	0.0863
ICI	0.0881	RIA	0.0960	IC	0.1204	RRA	0.0858
IIR	0.0873	CA	0.0957	C	0.1193	RAA	0.0806
RAI	0.0866	RAA	0.0945	RI	0.1182	CC	0.0796
RR	0.0856	IIA	0.0925	ACA	0.1141	IAR	0.0779
C	0.0827	RAR	0.0924	CA	0.1119	AIA	0.0776
IIC	0.0820	IAA	0.0918	IR	0.1106	AAI	0.0769
CII	0.0805	CI	0.0905	CCR	0.1104	ARR	0.0766
RRA	0.0801	AIA	0.0868	AC	0.1087	RAR	0.0762
ARI	0.0800	CCC	0.0852	CI	0.1067	IIR	0.0759
IRA	0.0789	IRC	0.0841	CAA	0.1062	ICI	0.0754
AAR	0.0775	IAC	0.0833	RCC	0.1006	RII	0.0732
AIC	0.0760	CAI	0.0831	RAC	0.0996	IAI	0.0716
IIA	0.0743	RRR	0.0824	AAC	0.0994	AIR	0.0707
AII	0.0741	RAI	0.0820	CRC	0.0966	AII	0.0701
AR	0.0734	CR	0.0803	RCR	0.0864	RIA	0.0678
AIR	0.0726	ARR	0.0803	RIC	0.0826	ARI	0.0669
CIR	0.0726	CC	0.0797	RCI	0.0816	CII	0.0669
RRR	0.0671	AII	0.0797	CIR	0.0814	AAR	0.0664
ICA	0.0671	CIR	0.0795	IRI	0.0778	AC	0.0654
ICR	0.0669	AIC	0.0787	CR	0.0767	CRI	0.0638
CC	0.0664	CAC	0.0783	RCA	0.0760	CAI	0.0635
CCC	0.0656	ARI	0.0778	CRA	0.0754	CA	0.0635
ACA	0.0649	RII	0.0750	CAR	0.0746	CRC	0.0633
CAI	0.0631	AAR	0.0713	CRR	0.0713	CIR	0.0623
IAC	0.0599	IIR	0.0700	RRC	0.0707	IIC	0.0618

RIC	0.0595	RRI	0.0695	AIR	0.0680	ACA	0.0607
RCI	0.0591	RCI	0.0680	IAI	0.0673	RIC	0.0595
III	0.0586	IAR	0.0673	IAR	0.0637	AIC	0.0592
ACI	0.0585	IIC	0.0669	ICR	0.0636	CIA	0.0584
IRC	0.0579	III	0.0643	ARC	0.0634	IRA	0.0579
RIA	0.0578	ACC	0.0642	RC	0.0622	IIA	0.0575
CAA	0.0575	CIA	0.0641	IRC	0.0615	mIn	0.0572
CRI	0.0574	ACA	0.0639	RIR	0.0609	IAC	0.0572
mIn	0.0567	RAC	0.0636	IRA	0.0599	CCR	0.0567
CIA	0.0556	RIC	0.0628	II	0.0592	CCA	0.0549
RAR	0.0540	RIR	0.0626	ACR	0.0584	RCC	0.0543
CR	0.0533	CII	0.0609	AAI	0.0572	CAC	0.0523
AAC	0.0523	mIn	0.0599	RIA	0.0571	RCI	0.0515
CIC	0.0512	IRA	0.0596	CRI	0.0569	ICA	0.0514
CCR	0.0479	ACI	0.0577	RAI	0.0552	ICR	0.0514
ARR	0.0476	CAA	0.0575	IAA	0.0532	CAA	0.0498
CRC	0.0476	ICI	0.0554	AIA	0.0506	III	0.0495
RIR	0.0444	IRR	0.0547	ARI	0.0503	ACI	0.0490
CRR	0.0401	ICR	0.0545	ACI	0.0499	IRC	0.0485
RC	0.0401	CRC	0.0494	ICI	0.0470	RRC	0.0482
RCR	0.0388	AC	0.0484	IRR	0.0459	ACC	0.0471
RCC	0.0388	RCR	0.0475	CIC	0.0450	RIR	0.0457
AC	0.0366	AAC	0.0449	IIR	0.0441	CRR	0.0417
CCI	0.0347	RC	0.0409	ICA	0.0441	CRA	0.0381
CA	0.0317	CRR	0.0404	AIC	0.0434	AAC	0.0376
RRC	0.0275	CAR	0.0396	CAI	0.0425	RCR	0.0374
CCA	0.0265	RCC	0.0354	CIA	0.0411	CR	0.0368
ICC	0.0249	CCR	0.0353	ICC	0.0392	RAC	0.0366
CAR	0.0203	ACR	0.0328	IIC	0.0380	RC	0.0346
ACC	0.0167	ARC	0.0311	mIn	0.0374	RCA	0.0321
ACR	0.0149	ICA	0.0308	CII	0.0338	CAR	0.0320
RAC	0.0143	CCA	0.0299	CCI	0.0338	CCI	0.0304
RRI	0.0126	RRC	0.0271	AII	0.0313	ICC	0.0276
CRA	0.0094	CRA	0.0269	RII	0.0309	ARC	0.0271
CAC	0.0092	ICC	0.0232	IAC	0.0297	CIC	0.0265
RCA	0.0042	RCA	0.0205	III	0.0275	RRI	0.0262

ARC	0.0035	CCI	0.0179	IIA	0.0274	IRR	0.0220
IRR	0.0030	CIC	0.0150	RRI	0.0206	ACR	0.0118

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389--3402.

Basios, M., Li, L., Wu, F., Kanthan, L., and Barr, E. T. (2018). Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 118--128.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, 41(1):164--171.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical report TR-CS-06-01, ANU. http://www.dacapobench.org.

Blasiak, S. and Rangwala, H. (2011). A hidden markov model variant for sequence classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Blekas, K., Fotiadis, D. I., and Likas, A. (2005). Motif-based protein sequence classification using neural networks. *Journal of Computational Biology*, 12(1):64--82.

Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241--250. IEEE.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5--32.

Breitwieser, F., Baker, D., and Salzberg, S. L. (2018). Krakenuniq: confident and fast metagenomics classification using unique k-mer counts. *Genome biology*, 19(1):1--10.

Breuel, T. M., Ul-Hasan, A., Al-Azawi, M. A., and Shafait, F. (2013). High-performance ocr for printed english and fraktur using lstm networks. In *2013 12th International Conference on Document Analysis and Recognition*, pages 683--687. IEEE.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321--357.

Cheng, B. Y. M., Carbonell, J. G., and Klein-Seetharaman, J. (2005). Protein classification based on text document classification techniques. *Proteins: Structure, Function, and Bioinformatics*, 58(4):955--970.

Chollet, F. et al. (2015). Keras. `https://keras.io`.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273--297.

Costa, D. and Andrzejak, A. (2018). Collectionswitch: a framework for efficient and dynamic collection selection. In *Proceedings of*, pages 16--26.

Costa, D., Andrzejak, A., Seboek, J., and Lo, D. (2017). Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 389--400.

de Oliveira Júnior, W., dos Santos, R. O., de Lima Filho, F. J. C., de Araújo Neto, B. F., and Pinto, G. H. L. (2019). Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 160--170. IEEE.

Dewar, R. B., Grand, A., Liu, S.-C., Schwartz, J. T., and Schonberg, E. (1979). Programming by refinement, as exemplified by the setl representation sublanguage. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):27--49.

Dong, G. and Pei, J. (2007). *Sequence data mining*, volume 33. Springer Science & Business Media.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57--76.

Gil, J. and Shimron, Y. (2012). Smaller footprint for java collections. In *European Conference on Object-Oriented Programming*, pages 356--382. Springer.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645--6649. IEEE.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222--2232.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157--1182.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225--236.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1--12.

He, M. X., Petoukhov, S. V., and Ricci, P. E. (2004). Genetic code, hamming distance and stochastic matrices. *Bulletin of mathematical biology*, 66(5):1405--1421.

Henikoff, S. and Henikoff, J. G. (1993). Performance evaluation of amino acid substitution matrices. *Proteins: Structure, Function, and Bioinformatics*, 17(1):49--61.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735--1780.

Hu, S., Ma, R., and Wang, H. (2019). An improved deep learning method for predicting dna-binding proteins based on contextual features in amino acid sequences. *PloS one*, 14(11).

Jung, C., Rus, S., Railing, B. P., Clark, N., and Pande, S. (2011). Brainy: effective selection of data structures. In *ACM SIGPLAN Notices*, volume 46, pages 86--97. ACM.

Kawulok, J. and Deorowicz, S. (2015). Cometa: classification of metagenomes using k-mers. *PloS one*, 10(4).

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Leather, H., Bonilla, E., and O'Boyle, M. (2009). Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81--91. IEEE Computer Society.

LeCun, Y., Haffner, P., Bottou, L., and Bengio, Y. (1999). Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319--345. Springer.

Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811--1841.

Liu, L. and Rus, S. (2009). Perflint: A context sensitive performance advisor for c++ programs. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 265--274. IEEE.

Ma, Q., Wang, J. T., Shasha, D., and Wu, C. H. (2001). Dna sequence classification via an expectation maximization algorithm and neural networks: a case study. *IEEE*

Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 31(4):468--475.

Manotas, I., Pollock, L., and Clause, J. (2014). Seeds: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, pages 503--514.

Miller, F. P., Vandome, A. F., and McBrewster, J. (2009). Levenshtein distance: Information theory, computer science, string (computer science), string metric, damerau? levenshtein distance, spell checker, hamming distance.

Mitchell, N. and Sevitsky, G. (2007). The causes of bloat, the limits of health. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 245--260.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *International conference on artificial intelligence: methodology, systems, and applications*, pages 41--50. Springer.

Muller, A. T., Hiss, J. A., and Schneider, G. (2018). Recurrent neural network model for constructive peptide design. *Journal of chemical information and modeling*, 58(2):472--479.

Ounit, R., Wanamaker, S., Close, T. J., and Lonardi, S. (2015). Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC genomics*, 16(1):236.

Pearson, W. R. (1990). [5] rapid and sensitive sequence comparison with fastp and fasta.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825--2830.

Pinto, G., Liu, K., Castor, F., and Liu, Y. D. (2016). A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 20--31. IEEE.

Rish, I. et al. (2001). An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41--46.

Rokach, L. and Maimon, O. Z. (2008). *Data mining with decision trees: theory and applications*, volume 69. World scientific.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Schonberg, E., Schwartz, J. T., and Sharir, M. (1986). An automatic technique for selection of data representations in setl programs. In *Readings in artificial intelligence and software engineering*, pages 235--243. Elsevier.

Shacham, O., Vechev, M., and Yahav, E. (2009). Chameleon: adaptive selection of collections. In *ACM Sigplan Notices*, volume 44, pages 408--418. ACM.

Stepanov, A. and Lee, M. (1995). *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77--90. ACM.

Tan, P.-N., Steinbach, M., and Kumar, V. (2016). *Introduction to data mining*. Pearson Education India.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214--224. IBM Corp.

Wang, J. T., Ma, Q., Shasha, D., and Wu, C. H. (2000). Application of neural networks to biological data mining: a case study in protein sequence classification. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 305--309.

Wilbur, W. J. (1985). On the pam matrix model of protein evolution. *Molecular biology and evolution*, 2(5):434--447.

Wirth, N. (1976). *Algorithms+ Data Structures= Programs Prentice-Hall Series in Automatic Computation*. Prentice Hall.

Witten, I. H. and Frank, E. (2002). Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, 31(1):76--77.

Wu, C. H. (1997). Artificial neural networks for molecular sequence analysis. *Computers & chemistry*, 21(4):237--256.

Xing, Z., Pei, J., and Keogh, E. (2010). A brief survey on sequence classification. *ACM Sigkdd Explorations Newsletter*, 12(1):40--48.

Xu, G. (2013). Coco: sound and adaptive replacement of java collections. In *European Conference on Object-Oriented Programming*, pages 1--26. Springer.

Xu, G. and Rountev, A. (2010). Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 160--173.

Yan, C., Dobbs, D., and Honavar, V. (2004). A two-stage classifier for identification of protein–protein interface residues. *Bioinformatics*, 20(suppl_1):i371--i378.

Yang, S., Yan, D., Xu, G., and Rountev, A. (2012). Dynamic analysis of inefficiently-used containers. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, pages 30--35.

Yegnanarayana, B. (2009). *Artificial neural networks*. PHI Learning Pvt. Ltd.

Yousef, M., Khalifa, W., Acar, İ. E., and Allmer, J. (2017). Microrna categorization using sequence motifs and k-mers. *BMC bioinformatics*, 18(1):170.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.