Decision Algebra: A General Approach to Learning and Using Classifiers
DECISION ALGEBRA: A GENERAL APPROACH TO LEARNING AND USING CLASSIFIERS

ANTONINA DANYLENKO

LINNAEUS UNIVERSITY PRESS
Decision Algebra: A General Approach to Learning and Using Classifiers
Doctoral dissertation, Department of Computer Science, Linnaeus University, Växjö, Sweden, 2015

ISBN: 978-91-87925-47-4
Published by: Linnaeus University Press, 351 95 Växjö
Printed by: Elanders Sverige AB, 2015
Abstract


Processing decision information is a vital part of Computer Science fields in which pattern recognition problems arise. Decision information can be generalized as alternative decisions (or classes), attributes and attribute values, which are the basis for classification. Different classification approaches exist, such as decision trees, decision tables and Naive Bayesian classifiers, which capture and manipulate decision information in order to construct a specific decision model (or classifier). These approaches are often tightly coupled to learning strategies, special data structures and the special characteristics of the decision information captured, etc. The approaches are also connected to the way of how certain problems are addressed, e.g., memory consumption, low accuracy, etc.

This situation causes problems for a simple choice, comparison, combination and manipulation of different decision models learned over the same or different samples of decision information. The choice and comparison of decision models are not merely the choice of a model with a higher prediction accuracy and a comparison of prediction accuracies, respectively. We also need to take into account that a decision model, when used in a certain application, often has an impact on the application's performance. Often, the combination and manipulation of different decision models are implementation- or application-specific, thus, lacking the generality that leads to the construction of decision models with combined or modified decision information. They also become difficult to transfer from one application domain to another.

In order to unify different approaches, we define Decision Algebra, a theoretical framework that presents decision models as higher order decision functions that abstract from their implementation details. Decision Algebra defines the operations necessary to decide, combine, approximate, and manipulate decision functions along with operation signatures and general algebraic laws. Due to its algebraic completeness (i.e., a complete algebraic semantics of operations and its implementation efficiency), defining and developing decision models is simple as such instances require implementing just one core operation based on which other operations can be derived.

Another advantage of Decision Algebra is composability: it allows for combination of decision models constructed using different approaches. The accuracy and learning convergence properties of the combined model can be proven regardless of the actual approach. In addition, the applications that process decision information can be defined using Decision Algebra regardless of the different classification approaches. For example, we use Decision Algebra in a context-aware composition domain, where we showed that context-aware applications improve performance when using Decision Algebra. In addition, we suggest an approach to integrate this context-aware component into legacy applications.

Keywords: classification, decision model, classifier, Decision Algebra, decision function
To my family

Присвячується моїй родині
This thesis consists of two major parts: Theory of Decision Algebra and Decision Algebra in Context-Aware Composition. The thesis is based on the following refereed publications:

Theory of Decision Algebra:


Antonina Khairova, Jonas Lundberg, Welf Löwe (2010). Decision Algebras for Capturing and Manipulating Decision Information (Best Doctoral Forum Poster Award). Tenth SIAM International Conference on Data Mining;


Antonina Danylenko, Welf Löwe (2014). Combining Classifiers of Different Classification Approaches. Incremental Classification. Concept Drift and Novelty Detection Workshop (IcaNove’14) at International Conference on Data Mining;

Decision Algebra in Context-Aware Composition:


This thesis is also a direct extension of:

Acknowledgments

Now, at this last stage of finalizing the thesis, I would like to thank everyone who was around me during the journey of my Ph.D studies. First of all I would like to express my gratitude to my supervisors who made an enormous contribution to my academic, research and even personal development and without whose tremendous support this thesis would not have been possible. I am sincerely thankful to Professor Welf Lowe for being my supervisor, sparking my interest to this research problem and inspiring the course of this thesis. I am grateful for our challenging and productive discussions and for your ability with one support word change my mood for the whole working week. Special gratitude goes to Associate Professor Jonas Lundberg for his support, suggestions and comments that helped to gain a different prospective to the research problem. Thank you for your good sense of humor and positive attitude. I would have never started my journey without endless enthusiasm and hard work of Associate Professor Tetyana Shatovska and Associate Professor Victoria Repka who first started the university collaboration program between Kharkiv National University of Radioelectronics and Linnaeus University through which I learned about the opportunity to apply for Ph.D studies.

During this journey I was surrounded by great colleagues and friends at the Linnaeus University, whom I want to acknowledge for being supportive and for interesting discussions we usually have outside the university. I appreciate your friendship and time that we spent together regardless all the hard work we usually had to do.

My parents Nina Khairova and Alik Khairov and my grandparents Antonina and Feliks Gutsalenko are people who always stand by me and who always loved me regardless of my achievements, work, behavior or knowledge. Without your commitment and careful education that you gave me I would not write these acknowledgement words today. You all are those people whose support and love accompany me during all my life, and I cannot express here how much I love you and how much I have missed you. Thank you for everything.

Guys from the "Ukrainian community", thank you for our meetings, for the support and friendship you provide during such difficult times for our country, for the moments when we laugh and forget about the current problems.

There are two persons in my life who are extremely important to me who always stand by me in both joyful and disappointing moments. These are my beloved husband Oleg and my son Taras. Thank you for your love, support and patience, and for being sometimes the voice of wisdom in my head. You’re my daytime and my nighttime. My world. You are my life.
Зразка, на останній стадії завершення написання дисертації, я хочу подякувати всім, хто оточував і підтримував мене на шляху до Ph.D. Перш за все, я хочу висловити подяку моїм керівникам за вагомий внесок у мій академічний, науково-дослідницький і навіть особистий розвиток, без їх величезної підтримки ця дисертація була б неможлива. Я щиро вдячна професорові Велфу Льове за його керівництво, за те, що викликав в мене інтерес до цієї дослідницької проблеми надихаючи на цю роботу. Я вдячна за наші складні й продуктивні дискусії і Вашу здатність одним словом підтримки змінити мій настрій на весь робочий тиждень. Особлива подяка доценту Йонасу Лундбергу за його підтримку, пропозиції та зауваження, які допомогли побачити нові перспективні напрямки в моїй роботі. Дякую Вам за гарне почаття гумору і позитивне ставлення до життя.

Я можливо ніколи б не розпочала цей шлях без нескінченного ентузіазму і наполегливої праці доцентів Тетяні Шатовської і Вікторії Рінки, які започаткували програму співпраці між Харківським національним університетом радіоелектроніки та Ліннеус Університетом, завдяки якій я дізналася про можливість здобути Ph.D. у Швеції.

На цьому шляху мене оточували чудові колеги і друзі, яких я хотіла б відзначити за підтримку та цікаві дискусії поза стінами університету. Я вдячна за дружбу й час який ми провели разом незважаючи на високу зайнятість.

Мої батьки Ніна та Алік Хайрови і мої бабуся з дідусем Антоніна та Фелікс Гуцаленко це ті люди, які люблять мене незалежно від моїх досягнень, роботи, поведінки або знань. Без вашої прихильності і ретельної освіти, якими ви мене оточили, я би не писала ці слова подяки сьогодні. Ви є ті люди, чиї підтримка і любов супроводжують мене впродовж моєї життя, і я не можу висловити як сильно я вас люблю і як сильно я сумую за вами. Дякую Вам за все.

Шановне панство з "Української громади", дякую за наші зустрічі, за підтримку та дружбу, які Ви надаєте в такі важкі для нашої країни часи, за моменти сміху, що дозволяють забути про поточні проблеми.

Є в моєму житті дві дуже важливі для мене людини, які завжди підтримують мене як в хвилині щастя так і в хвилині розчарувань. Це мій коханні чоловік Олег і мій син Тарас. Дякую вам за вашу любов, підтримку і терпіння, а також за те, що іноді ви є голосом мудrostі в моїй голові. Ви мій день та моя ніч. Мій світ. Ви моє життя.
# Contents

1 Introduction ........................................... 1
   1.1 Subject of Study .................................. 3
   1.2 Goals of the Thesis ................................. 3
   1.3 Goals Criteria ..................................... 4
   1.4 Methodology ...................................... 5
   1.5 Background and Motivation ....................... 8
   1.6 Contribution of the Thesis ....................... 10
   1.7 Thesis Outline .................................. 10

2 Decision Information: Background and Motivation ........ 13
   2.1 Decision Information in Computer Science: Literature Study . 13
   2.2 Notations of Decision Information ................ 21
   2.3 Summary .......................................... 24

3 Decision Algebra ..................................... 25
   3.1 Decision Functions ................................. 25
   3.2 Core Operations of Decision Functions ............ 30
   3.3 Learning and Deciding ............................. 32
   3.4 Auxiliary Operations ............................... 34
   3.5 Summary .......................................... 36

4 Accuracy of Decision Functions .......................... 39
   4.1 Conservative and Optimistic Decision Functions .... 39
   4.2 Comparing Decision Functions ..................... 42
   4.3 Combining General Decision Functions ............. 46
   4.4 Approximating Decision Functions ................ 57
   4.5 Parametrized Specification Decision Algebra ....... 59
   4.6 Summary .......................................... 61

5 Instantiations of Decision Algebra ......................... 63
   5.1 Decision Graphs ................................. 63
   5.2 Decision Tables ................................... 72
   5.3 Naïve Bayesian Classifier ......................... 76
   5.4 Merging Different Decision Functions ............... 83
   5.5 Summary .......................................... 86
## Contents

6 **Experiments over Decision Algebra** 89  
6.1 Decision Graphs versus Decision Trees 90  
6.2 Accuracy of Merged Decision Functions 97  
6.3 Summary 102  

7 **Decision Algebra in Context-Aware Composition** 105  
7.1 Context-Aware Composition 106  
7.2 Decision Algebra in Context-Aware Composition 107  
7.3 Object-Oriented Design and Concerns 109  
7.4 Context-Aware Recommender Systems 111  
7.5 Aspect-Oriented Context-Aware Composition 114  
7.6 Summary 119  

8 **Experiments over Context-Aware Composition** 121  
8.1 Evaluation of Decision Models in Contact-Aware Composition 121  
8.2 Context-Aware Composition with AOP 130  
8.3 Summary 138  

9 **Related Work** 141  
9.1 Decision Algebra 141  
9.2 Decision Algebra in Context-Aware Composition 146  

10 **Conclusions and Future Work** 151  
10.1 Review of the Goals and Goal Criteria 153  
10.2 Future Work 155  

A **Generalized Weak Law of Large Numbers** 157
# List of Figures

1.1 Diagram of the iterative hypothetico-deductive method. 7

2.1 Decision models distribution 17
2.2 Decision models in the problem domains 18
2.3 Distribution of the rationales for choosing the decision models in the problem domains 19
2.4 The application of the decision information in the application domain 23

3.1 A tree (left) and graph (right) representation of $df^2$. 29
3.2 A redundant (left) and a non-redundant (right) tree representation of $df^2$. Nodes are labelled as in the previous example. 30
3.3 Equivalent decision functions: $df^2$ (left) $\equiv df^2$ (right) 31

4.1 Merging of the Decision Functions (Scenario 1) 53
4.2 Merger of Decision Functions (Scenario 2) 55
4.3 Merger of Decision Functions (Scenario 3) 56
4.4 An tree representation of $df^2$ (left), and a tree representation of an approximated decision function $df^1^2$ (right). 58

5.1 Diagram of DA instantiations 64
5.2 Approximation and k-approximation of $df^2$ 69
5.3 Three scenarios of merging DG and NB 84

6.1 The percentage of reduced internal nodes and leaves compared to the total tree size (100%) 93
6.2 Times of learning and deciding based on a DG as % of DT (100%) 93
6.3 The accuracy gained by pruning DTs and using k-approximated DGs 96
6.4 Learning and approximation times of DGs as % of DTs (100%) 97
6.5 Average accuracy (%) of the merged DG, DG learned over 1/8 of a dataset (Regular dec. graph) and a line of 100% accuracy 98
6.6 Average accuracy (%) of the merged DG, DG learned over 1/8 of a dataset (Regular dec. graph) and a line of 100% accuracy 100
List of Figures

6.7 Positive permutations and the number of positive results per permutation. ........................................... 101

7.1 Object-oriented design for adaptation to CAC. ............. 110
7.2 The recommender system in a software development process. 113
7.3 Design of the profiling/learning phase in CAC ............... 116
7.4 Design of a Composition Block - AOP-based composition phase in CAC ............................................ 117

8.1 Homogeneous Quicksort and Context-Aware Sorting using DGs ("Opt Graph") and DTables ("Opt Table"). The x-axis displays the array size, the y-axis the time in msec. .................. 129
8.2 Homogeneous algorithms and CAC using AOP-base and manual approaches. The x-axis displays the problem size, the y-axis the time in msec. .............................................. 133
8.3 Sequential ProductInlined and CAC approaches in Matrix-Multiplication. The x-axis displays the problem size, the y-axis the time in msec. .................. 138
List of Tables

2.1 Example Dataset Characteristics ........................................... 22
3.1 Parameterized Algebraic Specification of $D(C)$ ........................................... 27
4.1 Comparing conservative, optimistic and general decision functions (where the co-domain is a power lattice $\mathcal{P}^C$). $df_1, df_2$ denote the respective decision functions and the $df_{oracle}$ denotes the oracle-accurate decision function. $P, R$ and $F$ denote precision, recall and harmonic F-score. ........................................... 43
4.2 Combining decision functions $df_1, df_2$ (where the co-domain is a power lattice $\mathcal{P}^C$). ........................................... 43
4.3 Parameterized Algebraic Specification of Decision Algebra ........................................... 60
5.1 Parameterized Algebraic Specification of Decision Graph Algebra ........................................... 71
5.2 Example of (a) a decision table of $df^2$, (b) a decision table of $bind_{A_2}(df^2, v_{high})$ and (c) a decision table of $approx_{A_2}(df^2, v_{high})$ ........................................... 73
5.3 Parameterized Algebraic Specification of Decision Table ........................................... 75
5.4 Naïve Bayesian classifier for "Car Evaluation" data set ........................................... 78
5.5 Parameterized Algebraic Specification of the Naïve Bayesian classifier ........................................... 82
6.1 Dataset Characteristics for Comparison Decision Graphs and Decision Trees ........................................... 91
6.2 Dataset Characteristics for Merging Decision Functions ........................................... 99
7.1 Application-specific and general CAC concerns. ........................................... 110
7.2 Usage scenarios. ........................................... 114
8.1 Memory overhead of different classification models. ........................................... 124
8.2 Decision overhead of different classification models. ........................................... 124
8.3 Errors of different decision approaches. ........................................... 126
8.4 Time overhead (in %) of different decision approaches. ........................................... 128
8.5 Speed-up of different CAC approaches for Sorting ........................................... 132
8.6 Speed-up of different CAC approaches for Matrix-Multiplication. 132
List of Tables

8.7 Speed-up of CAC approach with Exchangeable Data Representation for Matrix-Multiplication. . . . . . . . . . . . . . 137
Chapter 1

Introduction

Classification is a vital part of the different application domains within Computer Science, such as information storage, retrieval and manipulation, knowledge management, artificial intelligence, image processing, data processing and visualization in social and behavioural science, and software and hardware engineering. In general, classification is used to make certain decisions in a certain context (e.g., to diagnose a patient based on his/her health symptoms). A context can be represented by a set of attribute values (e.g., a set of symptoms) that can be retrieved from a particular situation or state (e.g., a patient’s health state). A decision is, basically, an inference reached on the basis of a context, often given as a class (e.g., the diagnosis of the patient). We refer to the information that is necessary for classification as decision information.

Many classification approaches exist that capture and manipulate decision information in order to construct a specific decision model (e.g., decision trees [86], Naïve Bayesian classifiers [54], support vector machines (SVMs) [19]). Basically, a decision model is a black-box of a set of rules for classifications. Classification approaches are often tightly coupled to certain learning strategies, special data structures and how common problems of classification are addressed. Therese problems may include fragmentation (i.e., reducing the statistical support of data that decreases predictive performance), replication (i.e., increasing memory consumption due to the duplication of rules) and model overfitting (i.e., increasing the complexity of a model that decreases predictive performance) in decision trees [100]. Decision models represent decision information in different ways. For instance, decision trees capture trees with class distributions in the leaves, Naïve Bayesian classifiers capture tables of class probabilities and SVMs capture coefficient vectors.

Selecting an appropriate classification approach to fit a particular classification problem is a difficult task since no single approach has been found superior to all others [40]. The wrong choice of a decision model may have a negative impact on application performance. For instance, high memory consumption could occur due to a redundancy in a decision model that makes the decision model grow considerably and high execution time is
Chapter 1. Introduction

often a consequence of low accuracy (i.e., non-optimal decisions lead to non-optimal actions) or low robustness of the model (i.e., non-resistance to a change in decision information or application behaviour that leads to low model performance). However, accuracy, robustness and scalability are actually contradictory goals, which can lead to trade-offs. Therefore, different classification approaches may be appropriate in different applications [75].

Often, when adapting classification approaches and decision models to the needs (in regard to accuracy, robustness and scalability) of specific application domains, one tries to overcome common problems by introducing new data structures or algorithms. Since these solutions are usually domain specific, the generality becomes lost as decision models become incomparable and, hence, make benchmarking difficult. Moreover, domain specific solutions also prevent a simple combination of different decision models.

In order to unify the classification approaches, this thesis proposes a theoretical generalization over decision models, referred to as Decision Algebra, which defines models as higher order decision functions. Decision Algebra separates the interfaces and implementations of decision models, making them (re-)usable as interchangeable black-box components. In fact, several existing classification approaches, including decision trees, decision graphs, decision tables, Naïve Bayesian classifier have come out as the default implementations of Decision Algebra. Furthermore, Decision Algebra abstraction enables a general approach for used to combine decision models, which allows for symbolic computations with the decision information captured, regardless of the classification approach.

This thesis consists of two major parts. The first part (Chapters 2–6) focuses on theory and experiments of Decision Algebra, specifically generalizing decision models (including decision trees, decision graphs, and decision tables, Naïve Bayesian classifiers) and defining a Decision Algebra common abstractions of these models. The second part of the thesis ( Chapters 7 and 8) discusses how different applications can benefit from Decision Algebra and suggests an approach to be used to integrate a Decision Algebra component into legacy applications.

In this chapter, we formulate a problem statement that this thesis addresses in Section 1.1. In Sections 1.2 and 1.3 the research goals and goal criteria, respectively, are presented. Furthermore, in Section 1.4, we outline our intended approach and, in Section 1.5, we motivate our research goals. Section 1.6 defines the main contributions of this thesis. Finally, Section 1.7 provides the outline of the thesis. All of the concepts and ideas presented in this Introduction chapter will be revised and further explored in later chapters.
1.1 **Subject of Study**

The subject of the study of this thesis can be characterized using a problem definition that the thesis will solve and a set of "unknowns" that needs to be researched in order to solve the problem.

1.1.1 **Problem Statement**

The problem of the study is the absence of a unified theoretical abstraction for existing and future decision models that allows for:

1. Choosing and exchanging decision models (based on the application’s functional and non-functional requirements) regardless of the classification approach used.
2. Theoretically and practically combining, approximating and manipulating decision models by generalizing over specific implementation details.

1.1.2 **Unknowns**

The set of "unknowns" to be answered consists of:

1. What are the common functional requirements (i.e., operations) of the decision models?
2. What are the common non-functional requirements (i.e., axioms, properties) of decision models?
3. Does a single general approach exist that can be used to combine the decision models (i.e., an operation that will be applicable for different decision models)?
4. How can exchangeable decision models be integrated into applications in order to improve application performance (i.e., the design of a component that enables one to adapt application behaviour to the changes in the application environment)? These applications are called context-aware applications.

This thesis proposes a solution to the given problem based on the possible answers provided throughout this thesis to the above questions.

1.2 **Goals of the Thesis**

Based on the problem statement given give above, the **goal of this thesis is to define a unified abstraction for decision models**. This goal can further
be divided in two sub-goals based on the answers to the "unknowns". As such the goals of this thesis are as follows:

1 Provide a **unified theoretical** formalization of classification approaches. We refer to this formalization as *Decision Algebra* (This goal shall provide answers for unknowns 1, 2 an 3):

1.a Provide an algebraic specification of Decision Algebra;
1.b Provide instances for several decision models based on Decision Algebra formalization in [1.a];
1.c Provide an interface and its implementations in Java based on the specifications defined in [1.a].

2 Create a context-aware component to be used to apply Decision Algebra in different applications in the field of Software Engineering (This goal shall provide answers for unknown 4.):

2.a Define and implement a design for a context-aware component with built-in Decision Algebra for exchangeable decision models;
2.b Define and test an integration of a component to legacy applications.

### 1.3 Goals Criteria

The criteria for fulfilling our first goal are:

1.1 Completeness: Decision Algebra shall be algebraically complete (i.e., it shall provide a complete algebraic semantics of base sets, constants and operations that describe decision models based on the fundamentals of algebraic specification).

1.2 Soundness: Decision Algebra shall be sound (i.e., any property that is provable for Decision Algebra shall be also true on all instantiations (decision models) upon which formalization of Decision Algebra is based.

1.3 Composability: Decision Algebra shall be composable (i.e., it shall allow recombinant instantiations that can be combined and constructed in various combinations: (1) any instantiation of Decision Algebra shall combine with any other instantiation based on a general combining operation and the result of the combining instantiations of Decision Algebra is also an instantiation of Decision Algebra and (2) any instantiation of Decision Algebra can be constructed (approximated) by different algorithms.
1.4 Methodology

1.4 Implementation efficiency: Decision Algebra shall be efficient in implementation (i.e., it shall have a minimum number of core operations, which will make Decision Algebra reusable as it reduces the work required to implement or adapt a decision model).

The criteria for fulfilling our second goal (and provide an answer for unknown 4) are:

2.1 Integrability: Decision Algebra shall be integrated into existing context-aware legacy applications based on a minimum well-defined integration steps;

2.2 Performance efficiency: The context-awareness component with integrated Decision Algebra shall improve application performance by allowing the application to switch between different decision models.

1.4 Methodology

In order to reach the research goals presented above, a systematic approach has been chosen. It is structured as follows.

The first research goal shall be reached by:

1 Defining a problem and conducting background research based on existing theories and observations via a literature study and systematic literature review.

2 Proposing a theoretical framework using Decision Algebra as a solution:

- Justifying that a wide variety of applications can be effectively dealt with using Decision Algebra;
- Justifying that Decision Algebra can improve rationales for applying decision models in particular problem domain;
- Providing uniform definitions of decision information, context, decisions, and decision models;

3 Specifying the requirements of Decision Algebra as the list of operations, parameters and properties:

- Defining a formal representation of a decision model;
- Defining general (core and derived) operations that can be used to manipulate decision information captured in decision models;
- Defining the pre- and post-conditions of the operations; At this point, it should be possible to reach goal criterion 1.1.
Chapter 1. Introduction

4 Checking the soundness of Decision Algebra over existing theories:
- Describing how Decision Algebra can be instantiated towards decision models; **At this point, it should be possible to reach goal criterion 1.2.**
- Describing how decision models can be fairly compared and combined under a common Decision Algebra interface; **At this point, it should be possible to reach goal criterion 1.3.**

5 Building a prototype of Decision Algebra;

6 Evaluating the Decision Algebra using a set of experiments. Then analysing and interpreting the results:
- Comparing and combining the decision models designed based on the common Decision Algebra interface;
- Improving the non-functional requirements of the decision models represented as instances of Decision Algebra **At this point, it should be possible to reach goal criterion 1.4.**

The second goal shall be reached by:

1 Defining a hypothesis that Decision Algebra improves the efficiency of context-awareness applications and conducting background research based on existing theories and observations via a literature study:
- Showing that context-aware application processes decisions information and benefits from Decision Algebra in non-functional requirements;

2 Providing prerequisites for context-awareness using Decision Algebra:
- Describing a base-line approach that shows what and how semi-manual efforts must aid the integration of Decision Algebra into a context-aware application. **At this point, it should be possible to reach goal criterion 2.1.**

3 Building a prototype of a context-aware application using the built-in Decision Algebra;

4 Evaluating the hypothesis in a prototype using a set of experiments and then analysing and interpreting the results:
- Improving context-aware application performance in terms of non-functional requirements by applying Decision Algebra. **At this point, it should be possible to reach goal criterion 2.2.**
1.4. Methodology

Figure 1.1: Diagram of the iterative hypothetico-deductive method.

The comparisons between decision models integrated in context-aware applications (or independently) should be fair, showing the advantages and disadvantages of the decision models instead of the advantages and disadvantages of the different implementations. Thus, in order to enable a fair comparison, we need to bias the:

- Accuracy of the decision (i.e., does a given decision model always decide the optimal variant and what is the impact of a suboptimal decision on the overall performance of the application);
- Decision time and its impact on the overall performance;
- Memory consumption required for capturing the decision information.

In general, our approach can be described using a combination of scientific methods and engineering design processes as shown in Figure 1.1. Based on this method, we iteratively performed Steps 1–5, which update and improve Decision Algebra. Step 6 symbolizes a new theoretical framework for Decision Algebra and step 7 shows a comparison of the competitive theories, which are referred to as related work (Chapter 9).
Chapter 1. Introduction

1.5 Background and Motivation

Decision information, in general, is a set of contexts mapped to a decision, where each context corresponds to a tuple of the values of certain attributes on which certain decision can be reached. Decision models represent the information necessary for classification (e.g., distributions, probabilities and coefficients). Decision models, such as decision trees, support vectors, Bayesian classifiers and neural networks, are often constructed automatically using machine learning. Machine learning processes a set of contexts and corresponding classes that sample a certain classification problem. This sample is usually called a *training data set*.

Learning is not an easy task and appropriate learning algorithms and decision models need to address several issues [60, 100]:

- Accuracy (i.e., the ratio of correct decisions in all decisions) is an issue, especially, with missing or contradicting decision information;
- Robustness (i.e., the accuracy of decision models) learned with only a limited amount of decision information is a related issue. Learning needs to avoid decision model overfitting (i.e., to avoid decisions based on statistically insignificant data);
- The scalability of learning and classifications (i.e., the time required for constructing and applying a decision model, respectively) is another issue, since decision model size grows, in the worst case, exponentially with the number of context attributes. Data replication (i.e., redundancy in decision models) adds to this problem.

Learning algorithms and corresponding data structures often address these problems. For instance, memory consumption is usually reduced by redundancy elimination in the decision information captured, while model overfitting can be solved by approximating the decision information captured [31, 97]. Learning algorithms can be presented in a general algorithmic framework [90]. Data structures used to implement decision models are generally well-known, along with efficient implementations thereof. Thus, adapting learning algorithms and decision models to the needs (in accuracy, robustness and scalability) of specific application domains, in general, may have a negative impact on memory consumption and application performance. Moreover, advances made in one domain are not trivially propagated to others.

For instance, static program analysis uses decision graphs, a type of decision model, to capture context-sensitive analysis information (constructed by program analysis, not learning) [107]. Precise program analysis is quite expensive in terms of time and memory consumption. Therefore, decision
graphs optimize memory consumption by removing any redundancies and trade accuracies against scalability. Decision graphs might even be beneficial in classification problems of other application domains with similar requirements, but it is hard to compare them with other, also highly specific, decision models. Moreover, the approach of trading accuracy for the scalability used in decision graphs might be applicable even to other decision models, but, again, it is hard to transfer this approach before the commonalities of the different models are understood.

In addition, decision models may be constructed from different data sets sampling the same classification problem. If the context for these datasets differs, then the trade-offs specified above may lead to different classification approaches. This prevents a simple combining of decision models learned over different data sets for the same (or even different) problem domain.

As the variety of application domains with classification problems each comes with different learning algorithms, combining algorithms, decision models, variants thereof and tailored implementations - sometimes even with different notations - we consider it worthwhile to introduce Decision Algebra. Several interface operations can be implemented at the abstract level using primitive operations, which are specific to individual decision models. This does not exclude more efficient algorithms and data structures that override abstract implementations. Due to this generalization, insights can be gained at an abstract level and reused between different domains, paving the way for a deeper problem understanding. Some properties, for instance, for combined decision models can be proven based on Decision Algebra level and still be valid for all its implementations. The objective that Decision Algebra allows for reusing operations and implementations of decision models between different application and problem domains can lead to more efficient implementations of decision models and to more efficient solutions of existing issues in the problem domains.

Furthermore, as an example of application of Decision Algebra in application domain, we chose context-aware composition that enables solutions for improving adaptation of software to dynamic changes of the environment. This domain was chosen due to its clear dependency on the decision model to be applied for selecting best adaptation. Context-aware composition allows for automatically selecting optimal variants of algorithms, data-structures and schedules at runtime often using dynamic dispatch tables. However, these tables grow exponentially with the number of significant context attributes. Therefore, to make a context-aware composition scale, alternative Decision Algebra instantiations can be used and non-functional requirements (i.e., memory consumption and execution time) can be automatically optimized statically or dynamically by providing possible component variants based on the scalable context-aware Decision Algebra component. Introducing context-aware composition in existing applications usually re-
Chapter 1. Introduction

requires a high re-engineering and implementation effort and, therefore, can be
time-consuming and error-prone. Our proposed approach provides a simple way to adapt the existing applications to context-awareness. Assuming a good object-oriented design, adaptation does not require changes within the legacy applications. This enables the (re-)engineering of self-adaptive and performance-portable (legacy) applications, which makes them run efficiently on modern hardware.

1.6 Contribution of the Thesis

The contribution of this thesis can be summarized as follows:

1 An overview on recent existing research in different application domains of Computer Science where decision models are used for classification purposes (Chapter 2);

2 Identification of potential issues due to the absence of a unifying decision model, and overview of the benefits of processing decision information using one general approach (Chapters 2 and 6);

3 Providing a theoretical foundation for Decision Algebra, which generalizes the classification approach and common aspects of decision information (Chapters 3 and 5);

4 Providing a new general approach to combining different decision models regardless of the actual implementation (Chapter 4); and

5 Developing a context-aware component with build-in Decision Algebra for integration into legacy codes, that allows developers to easily exchange decision models (Chapter 7 and 8).

1.7 Thesis Outline

The reminder of this thesis is structured as follows. In Chapter 2, we give a general introduction to the problem and present a notion of decision information. Moreover, we give an overview of existing decision models and applications that process decision information. Chapter 3 presents Decision Algebra along with its algebraic specifications. Decision Algebra defines a general representation of decision models, referred to as decision functions, and the operations over these functions. Chapter 4 shows how Decision Algebra can be instantiated with existing decision models: decision graphs, decision trees, decision tables, and Naïve Bayesian classifiers. In Chapter 5, we show how different decision models can be compared and combined with
respect to their accuracy. In Chapter 6, we evaluate our Decision Algebra using two experiments: (1) a comparison of decision trees and decision graphs using a common Decision Algebra interface and (2) an evaluation of the accuracy of combined decision functions. Chapter 7 gives a general overview of the context-aware composition and defines a general approach for integrating context-aware components into applications based on Aspect-Oriented programming. Moreover, this chapter gives an example of how to apply this component to recommender systems for non-functional requirements. In Chapter 8, we evaluate the context-aware composition component using the built-in Decision Algebra. Finally, Chapter 9 presents related work and Chapter 10 concludes this thesis and discusses the future work.
Chapter 2

Decision Information: Background and Motivation

The purpose of this chapter is to justify the choice of research goals presented in Chapter 1 and to motivate Decision Algebra concept given in the next Chapter 3.

We introduce the general idea of decision information, which can be considered to be an essential component in different application domains. The chapter is structured as follows. Section 2.1 discusses the results of a literature study that backs up the observations discussed in the previous chapter (see Section 1.5). Section 2.2 introduces a common vocabulary and a set of basic notations that will be used throughout this thesis. These notions characterize decision information used in different Computer Science domains by different types of applications. Finally, we conclude the chapter in Section 2.3, where we outline the problems of processing decision information, motivate the chosen research topic and guide the reader towards the next chapter.

2.1 Decision Information in Computer Science: Literature Study

Despite the vast body of literature on applications that use decision models in different application domains of Computer Science, no systematic study has been performed on the usage of decision models in different domains and the rationales behind their selection. In this section we describe such a study of the research papers published in the Journal of Universal Computer Science (J.UCS) from January 2010 till August 2014. The choice in a favour for this journal was made due to the variety of research studies over different application domains in Computer Science.

2.1.1 Objective

The objective is to study and summarize recent existing research in different application domains of Computer Science where decision models are used
Chapter 2. Decision Information: Background and Motivation

for classification and to:

A identify what decision models are typically used,

B assess the connection between the problem domains and the decision models used,

C retrieve the rationales for applying specific decision models in particular problem domains.

2.1.2 Method and Conduction of the Study

Our study is comprised of the primary steps of a systematic literature review as suggested by [61]. It is a well-defined approach to identifying, evaluating and interpreting all relevant studies regarding a particular research question, topic area or phenomenon of interest.

We searched for papers to be studied further using five steps:

1. We automatically searched—the actual search string is given below—for papers that used well-known or developed special decision models as tools for solving other Computer Science research problems.

2. We manually inspected the papers found in Step 1 and selected those papers that we considered relevant. As our primary objective was to understand the reasoning behind and consequences of the choices of decision models applied to Computer Science research problems, we excluded papers about theoretical aspects, surveys and roadmaps as well as papers that addressed non-Computer Science problems (e.g., e-learning, decision-making in society and classifications of general methods). We also excluded short papers of one or two pages as well as papers mentioning decision models only briefly in related or future work. Finally, we excluded special issues.

3. We calculated matching frequencies of the search string in the papers found in Step 1.\(^1\)

4. We assessed the accuracy of the automated search by calculating the F-score\(^2\) based on precision \(P\) and recall \(R\) of the retrieved papers of Step 1 and the relevant papers analyzed in Step 2.

5. We adjusted the search string to increase the accuracy of the automated search.

\( \text{\textsuperscript{1}using PDF-XChange Viewer http://pdf-xchange-viewer.en.softonic.com/} \)

\( \text{\textsuperscript{2}F = (2PR)/(P + R)} \)

\( \text{\textsuperscript{3}P = |relevant papers| ∩ |retrieved papers|}/|retrieved papers| \)

\( \text{\textsuperscript{4}R = |relevant papers| ∩ |retrieved papers|}/|relevant papers| \)
These steps were repeated iteratively until the F-score did not further increase. The search string used to produce the final set of papers is: ("genetic algorithm", "bayesian", "bayes", "neural network", "neural networks", "clustering", "support vector", "support vectors", "reinforcement learning", "incremental learning", "collaborative filtering", "continuous learning", "learn continuously", "decision tree", "decision graph", "decision table", "dispatch table", "opinion mining", "hidden-markov-model", "hidden markov model", "utility function", "utility-based technique", "logistic regression", "linear regression", "BDD", "nearest neighbors") AND ("unsupervised learning", "supervised learning", "classifier", "decision model", "machine learning", "data mining", "pattern recognition", "artificial intelligence", "image processing", "decision tree", "genetic algorithm", "incremental learning", "classification", "linear regression", "BDD") with 145 (86) retrieved (relevant) papers, a precision (recall) of 0.59 (1) and an F-score of 0.74. We studied the 86 relevant out of a total of 474 papers.

For each paper, the following data items were collected:

F1 the title of the paper; and

F2 the year of the paper;

F3 the category of the paper as selected by the author(s) based on the list of topics pre-defined by J.UCS – a paper can have more than one category;

F4 a short description of the problem addressed in the paper;

F5 a decision model that was used or implemented in the paper. It is the model that captures the decision information required for learning, deciding or continuous learning. This model could be decision trees, Naïve Bayesian classifier (probabilistic model), support vector machines, or neural networks (maximum-margin model) or others found in the paper;

F6 a short description of the rational for using this decision model. Such rationales were given within the discussion in the paper, by formal proofs or by some references justifying the choice;

F7 any relevant additional information, such as the purpose for using the decision model or a tool that was used as an implementation of the decision model. Every paper was read carefully and the data was extracted in the form as described above.

2.1.3 Results

We will now discuss the study results based on Objectives A, B and C.

Objective A: Identify what decision models are typically used.
Altogether around 30 types of decision models were used in the 86 papers. We further classified them based on the type of data the decision model captured for the actual decision making:

DM1 The tree-based models captured a search tree for the decision making. For each attribute value, the search space was restricted, which lead to a class, such as decision tree, decision table, decision rule, multi-variant binary decision diagram and decision graph;

DM2 The probability-based models capture the probabilities of the attribute values belonging to the different classes: Naïve Bayes classifiers, Bayesian networks, conditional-probability models, and hidden Markov models.

DM3 The maximum-margin models captured the hyperplanes separating vectors of the attribute values belonging to the different classes: support vector machines (SVMs), artificial neural networks, and similar.

DM4 The vector-based models define vectors of attribute values as centroids of different classes. They are the results of instance-based learning, such as k-nearest neighbours, and clustering algorithms, such as k-means, hierarchical clustering and distribution- and density-based clustering.

DM5 The regression models capture the coefficients of certain function families that map attribute values to classes. They capture the coefficients of the linear and logistic functions as derived from the linear and logistic regression, respectively.

DM6 Ad-hoc solutions are self-developed decision models that do not fall into any of the above categories.

DM7 Related papers that discuss the learning method, not the decision model. The decision model itself is unclear as the learning methods do not imply a particular model of any of the above categories. These generic learning methods include genetic algorithms, collaborative filtering, population-based incremental learning and reinforcement learning.

Figure 2.1 shows the categories of the decision models introduced in the 86 relevant papers of the study. We are particularly interested in the first category (DM1) as it contains decision models that serve as a natural representation of our theoretical framework of Decision Algebra. Therefore, these models are commonly used in examples in Chapters 3 and 4. Also, we will look at the first six categories of decision models (DM1–DM6). In addition, we should look at the first six categories of decision models (DM1–DM6).
2.1. Decision Information in Computer Science: Literature Study

These are decision models that can be generalized using Decision Algebras. We present this generalization in Chapter 3. The first (six) category (-ies) covers around 12% (almost 83%) of the decision models used in the papers. In total, we found 14 (95) decision models in DM1 (DM1–DM6). Note that some of the papers introduce more than one model. Most of the popular models are vector-based models (DM4) (36% of the papers).

Around 17% of the decision models fall into the "others" category (DM7). It cannot be excluded that there are decision models of one of the categories DM1 – DM6 even among those.

**Objective B:** Assess the connection between the problem domains and decision models used in these domains.

The problem domains were derived from the data items F3 and F4. Below, we define the five problem domains addressed in the 86 relevant papers:

- **P1** Storage, retrieval and manipulation of information,
- **P2** Knowledge management,
- **P3** Applied mathematics including artificial intelligence, image processing, logics, and formal languages,
- **P4** Data processing and visualization in social and behavioral sciences, and
- **P5** Software and hardware engineering, including software technology, programming, operating and control systems, and logic circuit design.

Figure 2.2 shows how the decision models are distributed over the problem domains and decision model categories: the bars are the number of all of the
decision models used in a problem domain, the number of decision models in categories DM1 – DM6, the number of decision models in DM1 and the number of decision models in the most popular category for each problem domain, respectively.

In all of the problem domains, the decision models in DM1 are used, and the decision models in DM1 – DM6 are dominating. The vector-based models (DM4) were the most popular models in most of the domains. The tree-based models (DM1) were the second most popular models for DM4 and DM7 in the domains P1 and P5, respectively. However, no single decision model category dominated all of the problem domains or any particular domain.

**Objective C:** Retrieve the rationales for applying specific decision models to particular problem domains.

In order to consider this objective, we drew on the data extracted from a short description of the rationale for using the decision model (F6) and any, relevant to this rationale, additional information (F7). The set of rationales derived from the papers can be generalized into four groups:

R1 References to previous studies in this problem domain (i.e., choosing a well-known decision model for this particular problem),

R2 References to the requirements of a specific type of input or output data that suggests the use of a particular decision model,

R3 References to the requirements of a specific performance or representation property that suggests the use of a particular decision model,

R4 None of the above; the choice was made based on a popular, commonly used, random decision model.
2.1. Decision Information in Computer Science: Literature Study

Figure 2.3 shows how the rationales are distributed among the papers and problem domains. Notice, that several papers use more than one rational to motivate the choice of a specific decision model. Around 30 papers (35% of all papers) do not specify any particular rationales for using one or the other decision model. Otherwise, 34% of the studies (or 29 studies) use rationale R1, 30% (or 30 studies) use rational R2 and 15% (or 13 studies) use rational R3.

In two out of the five problem domains (P1 and P2), the major group of papers did not motivate the choice of the specific decision model. For P1 around 33% (9 studies) used rational R4, for P2 around 46% (6 studies), for P3 around 25% (7 studies), for P4 around 25% (5 studies) and for P5 around 21% (3 studies). A reference to the non-functional properties of a decision model (R3) was the least frequently used (13 papers) and did not at all occur in one domain (P2): P1 - 11% (3 studies), P3 - 14% (4 studies), P4 - 15% (3 studies) and P5 - 21% (3 studies).

In this study, we observed that:

A the decision models [DM1 - DM6] that we attempted to abstract using our theoretical framework, Decision Algebra, are popular (83%). The tree-based models [DM1] take 12% among them,

B the decision models that we abstracted using Decision Algebra included tree-based models, which were popular in all (considered) of the Computer Science problem domains, but, there is no single decision model dominated any problem domain, and

C the selection of a decision model is mostly ad-hoc.

As we have only looked at a limited set of papers within the J.UCS (2010
There is a threat to the external validity of the generalization of these observations. However, the results indicate that (A) different decision models co-exist, they are (B) applicable across problem domains and (C) the culture of comparing the pros and cons of the decision models to select one could be further developed in general and in any individual problem domain (assessed). The reason is due to the difficulties in adapting, configuring or even re-implementing the decision models for a specific problem domain, which leads to problems in regard to benchmarking their accuracy, robustness and scalability. This motivates the present work as Decision Algebra allows for us to use decision models as black-box components hiding the different types (categories) of decision models and their implementation details behind a common interface.

2.1.4 Non-functional Requirements for the Choosing Decision Model

As was discussed above, a rational in favor of the non-functional properties of decision models (performance or representation properties) is the least frequently used. In this section, we present a short overview on selected papers that use this particular rational to choose a decision model to solve Computer Science research problems. This presentation was created in order to give a general overview of the types of requirements usually used to select decision models.

Bonnel et al. [6] proposed an Information Retrieval Interface (IRI) evaluation framework aimed at evaluating the suitability of any IRI to different IR scenarios. In this work, the authors used decision trees as the decision model in order to identify the scenarios in which the particular IRI was effective. The decision trees model was chosen based on its simplicity in representation, interpretation and rules extraction.

Zulkernain et al. [119] proposed a system architecture that automatically administrated personal unavailability based on cell phones in order to manage cell phone disruptions. The decision making process was based on a decision tree model in order to process the data from the phone sensors and activate a corresponding correct action. The rational for choosing the decision tree included low computational complexity at runtime and its suitability to a discrete set of a small number of outcomes.

Chamlertwat et al. [14] proposed a system that automatically analyzed customer opinions from a Twitter micro-blog service based on sentiment analysis. Needing a decision model that classified each tweet into "opinion" or "non-opinion," the authors used SVMs as the results gained in their previous study showed that SVMs gave the best performance in terms of accuracy for filtering opinion tweets.

Lee et al. [66] proposed a spam detection model that enabled a parameter
optimization and optimal feature selection in order to improve an accuracy of detection. In order to maximize the detection rates, the authors used a Random Forests decision model. This algorithm was chosen based on its high execution speed for high-dimension data.

Rosa et al. [92] proposed a symbolic-connectionist hybrid system that predicted the thematic roles assigned to the word in a sentence context. The authors used a symbolic connection hybrid decision model that was constructed based on Neural Networks. The main reasons for this decision were the short training time and a possibility of a simple extraction of symbolic knowledge.

Finally, Dvorak et al. [28] presented a computer-aided technique for the design of digital systems that could produce representations of arbiters and allocators in the form of a Multi-Terminal Binary Decision Diagram (MTBDD). This representation was chosen based on its compact and non-redundant representation characteristics for Boolean functions.

It’s interesting that only 15% of the studies referred to the non-functional requirements of the decision models. As we discussed before, this may be caused by the fact that benchmarking and adopting decision models for a specific problem domain are non-trivial tasks. Therefore, a generalized DA can benefit from the way in which a specific decision model is chosen.

### 2.2 Notations of Decision Information

In this section, we will first introduce a data set that will be used in all of the examples in this thesis. Then, we will discuss the general characteristics of decision information and decision models. We will also present a set of formal definitions that will be used throughout this thesis.

We introduce the “Car Evaluation” data set from the UCI Machine Learning Repository [34], which is used as an example throughout the thesis. The data set contains six categorical attributes that are presented in Table 2.1. The total number of training examples is 1728 with no missing values. Each classifier that we present is constructed using the FC4.5 algorithm [42] over a random sample of the data set equal to 1/4 (432 instances). Sometimes, we just used a subset of the attributes to construct a classifier.

**Definition 1.** A decision tuple \((a, c)\) is a tuple that relates to an actual context \(a \in \tilde{A}\) with an actual decision \(c \in C\), where \(\tilde{A}\) is a formal context and \(C\) is a formal decision.

The decision tuple can also be referred to as a decision fact or a training instance. Notice that we distinguish between an actual context and a formal context.
Table 2.1: Example Dataset Characteristics

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Buying price</td>
<td>very high, high, medium, low</td>
</tr>
<tr>
<td>2</td>
<td>Maintenance price</td>
<td>very high, high, medium, low</td>
</tr>
<tr>
<td>3</td>
<td>Doors number</td>
<td>2, 3, 4, 5, more</td>
</tr>
<tr>
<td>4</td>
<td>Persons capacity</td>
<td>2, 4, more</td>
</tr>
<tr>
<td>5</td>
<td>Size of language boot</td>
<td>small, medium, big</td>
</tr>
<tr>
<td>6</td>
<td>Safety</td>
<td>low, med, high</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>Car acceptability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not acceptable, acceptable, good, very good</td>
</tr>
</tbody>
</table>

**Definition 2.** An actual context $\vec{a} = (a_1, \ldots, a_n)$ is a tuple of attribute values $a_i \in A_i$, where $A_i$ is an attribute that corresponds to a property in a certain problem domain. A formal context $\vec{A}$ is the set of all actual contexts $\vec{a} = (a_1, \ldots, a_n)$ for all possible $a_i \in A_i$. Hence, it is the Cartesian product $\vec{A} = A_1 \times A_2 \ldots \times A_n$ over sets of possible values of attributes $A_1, \ldots, A_n$. Finally, an actual decision $c \in C$ is one out of a set of alternative decisions. A formal decision $C$ is the set of all alternative decisions.

As an example of a formal context, consider car attributes, such as “buying price” and “Maintenance price”. An actual context could be a pair of values, such as “low” and “high”. A corresponding formal decision could be “Car acceptability” with the actual decisions “not acceptable”, “acceptable”, “good” or “very good”.

**Definition 3.** The problem domain of decision information $\text{DI} \subseteq \vec{A} \times C$ determines the actual classification problem. It is defined by a pair $(\vec{A}, C)$, where the formal context $\vec{A}$ is a subspace of $\vec{A}$ ($\vec{A} \subseteq \vec{A}$) (i.e., $\vec{A}$ has at least the same attributes as $\vec{A}$ but possibly more).

**Definition 4.** Decision Information is a multi set of decision tuples: $\text{DI} = \{(\vec{a}_1, c_1), \ldots, (\vec{a}_n, c_n)\}$. Decision information is:

- **complete** if and only if: $\forall \vec{a} \in \vec{A}: (\vec{a}, c) \in \text{DI}$ and

- **non-contradictive** if and only if: $\forall (\vec{a}_i, c_i), (\vec{a}_j, c_j) \in \text{DI}: \vec{a}_i = \vec{a}_j \Rightarrow c_i = c_j$

Decision information can also be referred to as dataset, training set or training sample. The complete decision information contains decisions for all of the possible actual contexts within a given problem domain. In non-contradictive decision information, no two tuples have the same actual context $\vec{a}$ that leads to different decisions.

Complete and non-contradictive decision information can be represented as a decision function.
2.2. Notations of Decision Information

![Diagram of decision information and its components]

**Figure 2.4:** The application of the decision information in the application domain

**Definition 5.** A decision function \( df \) maps each actual context \( \vec{a} \in \vec{A} \) to an actual decision \( c \in C \).

\[
df : A_1 \times \cdots \times A_n \rightarrow C
\]

In order to use the given decision information in a particular application domain, it has to be processed such that a decision function \( df \) is constructed (i.e., we construct a certain representation of the decision function that is based on a given actual context \( \vec{a} \in \vec{A} \), which determines a decision \( c \in C \)).

We define a specific representation of the decision function as the decision model.

**Definition 6.** A decision model (or a classifier) is an implementation of a decision function.

For instance, decision trees capture formal context represented as internal nodes and decisions as class distributions in the leaves, Naïve Bayesian classifiers capture tables of class probabilities and SVMs capture coefficient vectors. Furthermore, in order to construct a decision model from the given decision information, we have to select a classification approach, which might depend on the: (1) problem domain of the decision information, (2) application domain where the classifier is to be used, (3) learning algorithm and (4) decision algorithm.

**Definition 7.** A classification approach is a combination of a decision model and a learning algorithm.

**Definition 8.** Learning maps decision information into decision function. Learning algorithm constructs a decision function stored in a decision model from given decision information.
Chapter 2. Decision Information: Background and Motivation

Definition 9. **Deciding** applies a decision function \( df \) to a given actual context \( \vec{a} \in \vec{A} \) in order to determine a concrete decision \( c \in C \).

For instance, deciding based on the decision trees corresponds to a search of a path from a root node to a leaf where a decision is stored.

2.3 **Summary**

In this chapter, we introduced the notion of decision information composed by actual contexts and decisions. We also introduced the notions of the decision tuple, which relates a context to its decision, and the decision model, which represents a decision function. In addition, we performed a literature study of J.UCS published papers. The objective of this literature study was to summarize recent research in different Computer Science domains in terms of the use of decision models as black-box components. Moreover, we aimed to assess the rationales between the problem domains and decision models used and provide evidence for the necessity to develop a generalized theoretical concept by which to uniformly describe decision models. The observations made in this chapter motivate the present work: Decision Algebra allows us to use decision models as black-box components hiding the different types (categories) of decision models and their implementation details behind a common interface. This concept should allow one to uniformly describe and evaluate existing decision models; choose the right decision model for a particular problem domain, avoiding adding new functionality, which can have a negative impact on non-functional requirements; and combine different decision models. The next chapter formally introduces Decision Algebra using decision functions and operations over these functions.
Chapter 3

Decision Algebra

In Chapter 2, we presented the different types of decision models that are usually used in Computer Science as well as the rationales that are applied when choosing a specific decision model. Moreover, we defined a notion of decision information and discussed the benefits of Decision Algebra to the existing theories of classification approaches.

The aim of this chapter is to show that Decision Algebra provides a unified theoretical formalization for classification approaches (see Section 2.1) as higher-order decision functions. Decision Algebra tends to describe both decision information and the main operations required in a classification process.

The notion of a decision function, its term representation and specific requirements are discussed in Section 3.1. The basic core operations of Decision Algebra are given in Section 3.2. In Section 3.3, we specify the two most-commonly used operations for decision functions (i.e., learning and deciding) that fulfil the basic requirements of most decision models. In this section, we also formally describe an example of an algorithm used when learning a decision function and explain how to decide which given decision function to use in certain actual contexts.

In Section 3.4, we define the auxiliary operations evert, apply and equals for defining combination and approximation of decision functions. Each operation of a decision function is defined using an algebraic specification based on notations given in [29]. Finally, in Section 3.5, we summarize this chapter.

3.1 Decision Functions

In Section 2.2, we gave a general definition of a decision function df that represented complete and non-contradictive decision information as a mapping between an actual context $\vec{a} \in \vec{A}$ and an actual decision $c \in C$. However, in most of the cases decision information that is captured by decision models is contradictory and has missing data. In this situation, decision models represent possible alternative decisions as distributions over classes $C$. For
instance, decision trees capture distributions in the leaves, Naïve Bayesian classifiers capture conditional probabilities and support vector machines capture a distance between the support vectors and maximum-margins that separate the classes [19].

For one and the same actual context \( \vec{a} \) different decision \( c \in C \) is possible following a distribution \( D(C) \) that defines the set of all possible distributions over \( C \). For this thesis, we define the distribution \( D(C) \) as a function that generalizes over natural numbers or probabilities (probability distribution). Throughout the thesis, we assume that \( D(C) \) is a distribution of natural numbers unless its precisely stated as a ’probability distribution’.

Therefore, a formal definition of a decision function is:

**Definition 10.** A decision function \( df \) is a mapping of an actual context \( \vec{a} \in \vec{A} (\vec{A} = A_1 \times A_2 \times \cdots \times A_n) \) (domain of \( df \)) to a (probability) distribution \( d(C) \in D(C) \) (co-domain of \( df \)) over classes \( C \):

\[
df : \vec{A} \rightarrow D(C)
\]

In what follows we define a constructor and several operations over \( D(C) \). These definitions are important since most of the operations over decision functions in a base-case scenario process decisions captured in these distributions. The constructor takes \( k = |C| \) pairs \( (p, c) \) of frequencies of \( p \in P \) and the corresponding classes \( c \in C \) and returns a distribution \( d(C) \in D(C) \):

\[
\text{cons}_D : (C \times P)^k \rightarrow D(C)
\]

\((c, p) \in d(C)\) denotes that decision \( c \in C \) is supported by a frequency (probability) \( p \in P \). The specification of \( D(C) \) can be given using a parameterized algebraic specification (PAS) with \( C \) as the parameter. This specification consists of a pair of specifications [29]: a formal parameter specification for \( C \) and target specification of a distribution \( D(C) \) itself (\( D_i \hat{=} D(C_i) \)). We use a well-defined formalized specification since it allows for the definition of large software systems regardless the implementation details. The PAS of \( D(C) \) is defined in Table 3.1, where the notation “+” (e.g., \( C+, P+ \) and \( BOOL+ \)) stands for a disjoined union of sorts, operation symbols, equations and axioms, which are syntactical notions of a signature.

Now, we can also define a decision function as a parameterized algebraic specification \( DF[\vec{A}, D] \) with \( \vec{A} \) and \( D(C) \) as parameters. By \( DF \), we denote a set of all decision functions with the same signature \( A_1 \times \cdots \times A_n \rightarrow D(C) \). Parameterized specifications shall provide a general representation of the decision information as an abstract decision model along with a set of operations. The complete specification will be given at the end of Chapter 4, when we explain and define all of the operations of Decision Algebra.
3.1. Decision Functions

\[ D(C) = C+, P+, BOOL^+ \]

**sort** \( D \)

**ops**

- \( \text{cons}_D: (C \times P)^k \rightarrow D; \)
- \( g_D: D_1 \times \cdots \times D_m \rightarrow D; \) Any function to apply;
- \( \sqcup_D\): \( D \times D \rightarrow BOOL; \) combining function;
- \( \equiv_D: \prod D \rightarrow BOOL; \) equivalence function

**mode**: \( D \rightarrow C; \) returns a most-probable class

**prob**: \( D \times C \rightarrow P; \) return frequency for a class

**eqns**

\[
\forall c, c_1, \ldots, c_k: C, \forall p_1, \ldots, p_k \in P
\]

\[
\text{mode}(\text{cons}_D(c_1, p_1, \ldots, c_k, p_k)) :=
\]

\[
\text{arg} \max_{c : C} \text{prob}(\text{cons}_D((c_1, p_1), \ldots, (c_k, p_k)), c)
\]

\[
\text{prob}(\text{cons}_D((c_1, p_1), \ldots, (c_k, p_k)), c) := p_i \text{ when } c_i = c
\]

**axioms**

\[
\forall d: D
\]

\[
d \sqcup_D d = d;
\]

Table 3.1: Parameterized Algebraic Specification of \( D(C) \)

For simplicity of explanation in this chapter, we assume that \( A_1, \ldots, A_n \) and \( C \) are discrete (or categorical domains), but discretization of continuous domains is captured using our implementation of decision functions.

The *arity* of a decision function \( df : A_1 \times \cdots \times A_n \rightarrow D(C) \), denoted by \( \text{arity}(df) \), is the number \( n \) of attributes. If important, we annotate the decision function with its arity \( n \) as a superfix, \( df^n \).

A decision function can be implemented in different ways by different decision models. For example, a decision function \( df : A_1 \times \cdots \times A_n \rightarrow D(C) \) over finite domains \( A_i \) can be defined explicitly by all its decision tuples \( (\vec{a}, (c, p)) \) with \( \vec{a} \in A_1 \times \cdots \times A_n \) and \( (c, p) \in D(C) \). The tuples can be captured in a decision table with \( n + 1 \) columns, one for each attribute \( A_i \in \vec{A} \) and a final column for the actual decision distribution of decision \( c_j \in C \) with \( \text{prob}(d(C), c_j) = 1 \). Such a decision table has \( |A_1| \cdot \cdots \cdot |A_n| \) rows. Decision tables are one type of decision models, but, there many other decision models exist that often differ from one domain to another as discussed in Section 2.1.

Decision functions can be described as higher order (or curried) functions where 0-ary (constant) decision functions \( df^0 : \rightarrow D(C) \) are the result of an 1-ary decision function \( df^1 : A_1 \rightarrow (\rightarrow D(C)) \). A decision function is then represented by a curried function:

\[
df^n : A_1 \rightarrow (A_2 \rightarrow (\ldots \rightarrow (A_n \rightarrow D(C)) \ldots)
\]

1. The particular implementation of \( \sqcup_D \) may vary, especially if it merges probability distributions and need normalization. We refer to Chapter 4 for a discussion of variants.
Chapter 3. Decision Algebra

A *curried* decision function \( df^n \) takes one argument (attribute) of type \( A_1 \) and generates a new decision function of a type \( (A_2 \to (\ldots \to (A_n \to D(C))\ldots) \), which, in turn, takes the next argument of type \( A_2 \) and yields a new decision function \( (A_3 \to \ldots) \). The final decision function will be captured as: \( \to D(C) \).

These curried functions can easily be represented as a *decision tree* or *decision graph*. Therefore, all of the examples in this and next chapters are given based on this natural term representation of decision functions.

\[
df^n = x^1(df_1^{n-1}, \ldots, df_{\mid A_1\mid}^{n-1})
\]

where the \( \mid A_1\mid \)-ary selection operator \( x^1 \) is applied to the arguments of \( A_1 \). There are \( \mid A_1\mid \) result functions, one for each attribute value \( a \in A_1 \). These are \( (n - 1) \)-ary decision functions:

\[
df_{\mid A_1\mid}^{n-1} : A_2 \to \ldots \to A_n \to D(C)
\]

with \( \text{id}_{\mid x\mid}(a) \) being a bijective mapping of each attribute value \( a \in A_i \) to a unique Natural index number. If necessary for distinction, we index a selection operator \( x^i \) with the index of the attribute \( A_i \), which it switches on.

The constructor \( df \) of a decision function \( df \in DF[\vec{A}, D] \) reflects its higher-order representation. Thus, it takes attribute values of the first attribute \( a_i \in A_1 \) and \( \mid A_1\mid \) corresponding decision functions \( DF[\vec{A}', D] \), where \( \vec{A}' = A_2 \times \ldots \times A_n \):

\[
\text{cons}_{DF} : A_1 \times DF[\vec{A}', D] \times \ldots \times A_1 \times DF[\vec{A}', D] \to DF[\vec{A}, D]
\]

If all decision functions \( df_{\mid A_1\mid}^{n-1} \in DF[\vec{A}', D] \) are equivalent, then the result of this construction is equivalent to \( df^n \):

\[
\text{cons}_{DF}(a_1, df_{\mid A_1\mid}^{n-1}, \ldots, a_{\mid A_1\mid}, df_{\mid A_1\mid}^{n-1}) \equiv df^n
\]

The equivalence property of decision functions will be discussed in Section 3.1.1. The constructor of the *constant decision function* discussed above \( df^0 \in DF_\emptyset[[0], D] \) (with zero-dimensional attribute vector space \( \{0\} \)) is:

\[
\text{cons}_0^{DF} : D \to DF_\emptyset[[0], D]
\]

As a simple example, Figure 3.1 (left) shows a tree-representation of a 2-ary decision function \( df^2 \) with two attributes “Buying price” \( (A_1) \) and “Maintenance price” \( (A_2) \) of the “Car evaluation” data set presented in the previous chapter in Section 2.2:

\[
df^2 = x^1(\text{na}, x^2(\text{na}, \text{na}, \text{a}, \text{a}), x^2(\text{na}, \text{na}, \text{a}, \text{a}), \text{a}).
\]
3.1. Decision Functions

Each intermediate node represents a decision term with a selection operator $x^n$ corresponding to the attribute $A_n$. Each leaf node corresponds to the most-probable decision $c \in C$ of a distribution $d(C) \in D(C)$. The outermost selection operator $x^1$ gives the result “not acceptable” (na) if $A_1$ is “very high” (vhigh). It returns “acceptable” (a) if $A_1$ is “low”. In case $A_1$ is “high” or “medium” (med) the decision in favor of a certain class is based on the actual attribute value of $A_2$ according to $x^2(na, na, a, a)$, which is the 4-ary selection operator mapping each attribute value of $A_2$ to a decision. By removing and reusing redundant subtrees, this (and every other) decision tree can easily be represented as a directed acyclic graph (Figure 3.1 (right)).

Figure 3.1: A tree (left) and graph (right) representation of $df^2$.

In contrast to OBDDs, decision graphs contain multiple decisions instead of binary ones due to multiplicity of context attributes. Also, decision values $c \in C$ are not binary either. Thus, more general decision graphs inherit the multi-terminal multi-valued decision diagrams [76] with all their properties.

3.1.1 Redundancy and Equivalence

A graph representation of a decision function requires redundancy elimination in the decision information captured. Thus, when the decision function is implemented, it may lead to decreased memory consumption. Therefore, in this section, we need to define what is a redundant and non-redundant decision function and when the two decision functions are equivalent.

Definition 11. An n-ary decision function $df^n = x^n(df^{n-1}, \ldots, df^1)$ is redundant if all of its sub-functions $df^{n-1}$ are equivalent (i.e., represent the same decision, and can, therefore, be replaced with this decision).

That is, a decision function $df$ containing a redundant sub-function $df_r = x'(df_j, \ldots, df_j)$, where each branch leads to the same decision $df_j$ can, without
any loss of information, be rewritten as:
\[ df = x(\ldots, x^{df_j}, \ldots, \ldots) \equiv df' = x(\ldots, df_j, \ldots) \]

In a decision tree representation, this definition corresponds to replacing a term with root \( df \), by any of its (equivalent) sub-terms. The process of removing redundancy is called **redundancy elimination**.

As an example, Figure 3.2 shows redundant (left) and non-redundant (right) tree representations of \( df^2 \). Nodes are labelled as in the previous example.

![Figure 3.2: A redundant (left) and a non-redundant (right) tree representation of \( df^2 \). Nodes are labelled as in the previous example.](image)

**Figure 3.2**

The right-hand side of Figure 3.2 shows a non-redundant representation of \( df^2 \).

**Definition 12.** Two decision functions \( df \) and \( df' \) are **equivalent**, denoted by \( df \equiv df' \), if they give the same distribution \( d(C) \) for the same actual context \( \vec{a} \) disregarding permutations of the attribute vector positions.

As an example, Figure 3.3 depicts two equivalent decision functions since, for each set of actual attributes \((a_1, a_2) \in A_1 \times A_2\), they give the same decision \( c_j \in C \).

### 3.2 Core Operations of Decision Functions

To instantiate a decision function with different decision models (the discussion about the instantiation of Decision Algebra is given in Chapter 4),
3.2. Core Operations of Decision Functions

we need to define the core operations that must be implemented in each instance. Defining the core operations allows to define all other non-core operations of decision functions using the core operations, thus, we make them reusable between different implementations.

One of the goal criteria for Decision Algebra is “Implementation Efficiency” (see Section 1.3), which requires a minimum number of core-operations. Therefore, due to the higher-order representation of decision functions, we defined only one core operation, called bind, which binds attribute $A_i$ to an attribute value $a \in A_i$.

An $n$-ary decision function $df^n \in DF[\vec{A}, D]$ bound this way returns a new $(n - 1)$-ary decision function $df^{n-1} \in DF[\vec{A}', D]$, with $\vec{A}' = A_1 \times \cdots \times A_{i-1} \times A_i$:

\[
bind_{A_i}: DF[\vec{A}, D] \times A_i \rightarrow DF[\vec{A}', D],
\]

where

\[
bind_{A_i}(cons_{DF}(a_1, df_1, \cdots, a_{|A_i|}, df_{|A_i|}, a)) := df_i, \text{ if } a = a_i.
\]

If we bind, for example, $df^2$ of Figure 3.1 to $a = \text{high} \in A_1$ we get:

\[
bind_{A_1}(df^2, \text{high}) = x^2(na, na, a, a)
\]

Binding an attribute value $a \in \hat{A}$ of an “unknown” attribute $\hat{A} \notin \vec{A}$ for a decision function $df^n \in DF[\vec{A}, D]$ is still well-defined, but leaves the decision function unchanged:

\[
bind_{\hat{A}}(df^n, a) := df^n.
\]

Furthermore, the interpretation of this operation in the base case scenario for $df^0 \in DF_0([0], D)$ is straightforward:

\[
bind_{A_i}(df^0, a) := df^0,
\]
Chapter 3. Decision Algebra

where \( a \) is a value of any attribute \( A_j \).

### 3.2.1 Decision Algebra: Formal Definition

As we have now formally defined a decision function, we can formally define **Decision Algebra** as a theoretical framework that provides a general representation of decision information as an abstract decision function with core and non-core operations. Classification approaches that intend to instantiate Decision Algebra have to implement at least the core operations. More formally:

**Definition 13.** A **Decision Algebra** is a triple \( DA = < DF, \Omega, R > \), where \( DF \) is a set of decision functions, \( \Omega \) is a set of operations defined over \( DF \), and \( R \) is an equivalence relation.

The equivalence relation \( R \) partitions the terms of the algebra \( DA \) into a number of equivalent classes. This relation enables us to determine that two decision functions, which are syntactically distinct do, nevertheless, represent the same decisions (see equivalence property).

In what follows, we will define a set of non-core operations \( (\Omega) \), which allows for general learning, deciding, combining and manipulating of the decision functions.

### 3.3 Learning and Deciding

Learning and deciding are the main operations used in all decision models. Therefore, they also need to be defined over decision functions. The goal criterion (1.4) of “Composability” (see Section 1.3) requires that a decision function is self-contained and stateless, allowing the application of any learning algorithm for the implementation of this function to any decision model. Therefore, in this section we aim to define these operations in such a way that they become reusable regardless of the implementation details.

#### 3.3.1 Learning

Decision information may be incomplete (i.e., it does not contain a decision for all possible combinations of attribute values) or contradictive (i.e., it contains different decisions for the same combination of attribute values) (see Section 2.2). Both require learning to transform the decision information into a decision function.

Learning is the process of constructing, from decision information, a decision function that heuristically solves a decision or a prediction or classification problem for which only expensive or no algorithmic solutions are known [22]. Usually, learning depends on a chosen classification approach,
3.3. Learning and Deciding

thus a chosen decision model [77, 54, 19]. What we want to reach is a separation of a learning algorithm from a chosen decision model (e.g., we want to capture the conditional probabilities required for the construction of a Naïve Bayesian classifier using a learning algorithm that is commonly used to learn decision trees).

As an example, we will sketch the algorithm for learning decision trees. Algorithm 1 presents the commonly used C4.5 algorithm [89]. (We also use this algorithm in our experiments presented in Chapter 5.)

**Algorithm 1 Learning**

Dataset \( DS \rightarrow \text{decision function} \ df \)

1: compute distribution of classes \( D(C) \) in \( DS \);
2: compute error of just selecting the mode of \( D(C) \) as the decision;
3: if error acceptable then
   4: return \( df^0 \): \( \rightarrow D(C) \)
5: end if
6: let \( A \) - set of the dataset attributes;
7: for each \( A_i \in A \) do
   8: compute \( \text{gain}(A_i) \)
9: end for
10: choose the attribute \( A_i \) with \( \text{max} \, \text{gain}(A_i) \)
11: if gain not acceptable then
   12: return \( df^0 \): \( \rightarrow D(C) \)
13: end if
14: for each \( v \in A_i \) do
   15: let \( DS_v \leftarrow \{(\bar{a}, c) \in DS|A_i = v\} \)
   16: recursively construct \( df_v \) for a data subset \( DS_v \)
17: end for
18: return \( df \), where \( \forall v \in A_i, \, \text{bind}_{A_i}(df, v) = df_v \)

The algorithm works by recursively selecting the "best attribute" to split the training set (lines 6–10) and expanding the terms of the decision function (lines 14–17) until the stopping criteria are met (lines 3 and 11). For this algorithm, the "best attribute" means the attribute that most effectively partitions the data set into subsets enriched in one class or the other. Every decision function is created only when each corresponding training set \( DS_v \) is processed (line 16).

This algorithm only uses a constructor of a decision function, which is a general Decision Algebra operation. All other computations are done on the training data set and are not dependent upon a decision model (e.g., a tree structure). Therefore, the algorithm can be reused by any instantiation of Decision Algebra (decision model) that implements a constructor of a
decision function. For instance, it can be used to learn a Naïve Bayesian classifier or decision tables.

### 3.3.2 Deciding

Deciding is applying a decision function \(df\) to a given actual context \(\vec{a} = (A_1 \times \ldots \times A_n)\) in order to determine a concrete decision \(c \in C\). Therefore, in general, we evaluate \(df\) for \(\vec{a}\) and then select the most-probable class of the resulting distribution \((d(c))\) using a mode operation discussed before (see Section 3.1):

\[
df : A_1 \rightarrow \ldots \rightarrow A_n \rightarrow D(C)
\]

\[
declare : DF[\vec{A}, D] \times \vec{A} \rightarrow C
\]

\[
declare(df, \vec{a}) := \text{declare(bind}_{A_1}(df, a_1), \vec{a}_1) \quad a = (a_1, \vec{a})
\]

\[
declare(df^0, \vec{0}) := \text{mode} \circ df^0,
\]

where “\(\text{mode} \circ df^0\)” means that we apply the mode operation over \(D(C)\) on a constant decision function \(df^0\).

Obviously, we can pre-compute \(\text{declare}\) if the learning phase precedes and is not interleaved with the decision phase. That is, we apply the mode-function on each of the distributions learned for a \(df\)-function. This action saves space and decision time, but loses the information captured in the decision distributions.

The \text{declare} operation is based on a core operation bind of Decision Algebra. Therefore, any decision model implementing this operation can use the general \text{declare} operation. In what follows, we will define two auxiliary operations of Decision Algebra that will help to define the equivalence, combining and approximation of decision functions.

### 3.4 Auxiliary Operations

We now motivate and define two auxiliary operations of decision functions: \text{evert} and \text{apply}. In order to reduce the memory consumption for a decision graph we can reorder the nodes of the graph in an optimal way such that the amount of nodes will be reduced. Furthermore, the nodes of the graph can be reordered using the learning algorithm C4.5, presented in Section 3.3, in favor of the “best” splitting attribute. This reordering operation was inspired by a Shannon expansion of OBDDs \cite{11} and is called \text{evert}.

In general, such operations used as \text{declare} or \text{equivalence} apply functions, \text{mode} and \(\equiv_D\), respectively, are defined on the co-domain \(D(C)\) of a decision function. We define an \text{apply} operation over decision functions, that can
push a certain function $g$ defined on the co-domain $D(C)$ toward the constant decision functions $df^0$.

### 3.4.1 Everting Decision Functions

Everting changes the order in which attributes occur in the decision functions. This is used, for example, in heuristics for saving space during learning. Its implementation on a specification level generalizes the Shannon expansion of OBDDs [11] over a decision function $df$ with the $i$th attribute:

$$
evert_{A_i} : DF[\vec{A}, D] \rightarrow DF[\vec{A}', D] \quad \vec{A}' = A_i \times A_1 \times \cdots \times A_{i-1} \times A_{i+1} \times \cdots \times A_n$$

$$i \in [1 \ldots n]$$

$$evert_{A_i}(df) := \text{cons}_{DF}(a_1, \text{bind}_{A_i}(df, a_1), \ldots, a_{|A_i|}, \text{bind}_{A_i}(df, a_{|A_i|}))$$

$$a_1, \ldots, a_{|A_i|} \in A_i$$

$$evert_{A_i}(df) \equiv df$$

$$evert_{A_i}(df^0) = df^0$$

Generally, the Shannon expansion creates a new decision function that corresponds to a new decision model, but does not change the decisions. It is just a rewrite rule that can be used to reorder the attributes of a decision function, sometimes making the representation more compact.

Hence, a decision function $df$ is equivalent to $evert_{A_i}(df)$ (equivalence was discussed in Section 3.1.1). For example, Figure 3.3 in Section 3.1.1 shows a tree-representation of $df^2 = x^1(na, x^2(na, na, a, a), x^2(na, na, a, a), x^2(na, na, a, a), a)$ (on the left) and its everted version $df^1 = evert_{A_i}(df^2)$ (on the right). Notice, that index $i$ of decision function $df^2$ just defines another 2-ary decision function. For $df^2$ it holds:

$$df^2_1 = evert_{A_i}(df^2) = x^1(na, na, na, a), x^1(na, na, na, a), x^1(na, a, a, a), x^1(na, a, a, a))$$

$$df^2_2 = evert(1, df^2) = df^2$$

### 3.4.2 Applying Functions to Decision Functions

We can apply a general function $g$ defined on the co-domain of a decision function $D(C)$ (e.g., $\text{mode}$ or $\equiv_D$) to the constant elements $df^0 \in DF[\{0\}, D]$ of the set of decision functions $df_1, \ldots, df_k \in DF$.

In Section 3.3, in the definition of the $\text{decide}$ operation, we applied $\text{mode}$ to the bases of a decision function represented by distribution $d(C) \in D(C)$ in order to get an actual class $c \in C$. Moreover, based on the $\text{apply}$ operation, we can now define an $\text{equals}$ operation that recognizes two equivalent decision functions by recognizing the equivalence ($\equiv_D$) of the decisions for all of the attribute vectors $\vec{a} \in \vec{A}$.
Chapter 3. Decision Algebra

These examples are just useful special cases of applying general functions to decision functions. We define such a general application of the apply of the arbitrary $k$-ary function $g$ to the $k$-tuples of decision functions as:

$$g : D_1 \times \ldots \times D_k \rightarrow D$$

$$apply : (D_1 \times D_2 \times \ldots \times D_k \rightarrow D) \times DF[\vec{A}_1, D_1] \times DF[\vec{A}_2, D_2] \times \ldots \times DF[\vec{A}_k, D_k] \rightarrow DF[\vec{A}, D]$$

$$apply(g, df_1, \ldots, df_k) := cons_{DF}(a_1, apply(g, bind_{A_1}(df_1, a_1), \ldots, bind_{A_1}(df_k, a_1))), \ldots$$

$$a_{|A_1|}, apply(g, bind_{A_1}(df_1, a_{|A_1|}), \ldots, bind_{A_1}(df_k, a_{|A_1|})))$$

where $a_1, \ldots, a_{|A_1|} \in A_1$

In a case when a decision function is represented as a tree $apply(g, df_1 \ldots df_k)$, we recursively apply $g$ to the respective sub-trees of the arguments and eventually evaluate it on the leaves. The result is a decision function over $D(C)$.

For a base case scenario, these operations are defined as follows. Notice that we need a certain function to get a distribution $d(C)$ from a constant decision function $df^0$. For getting the distribution we use an eval operation. The definitions above will be still valid.

$$eval(cons^{\emptyset}_{DF}(d(C)), \_ ) = d(C)$$

$$apply(g, df_1, \ldots, df_k) := cons^{\emptyset}_{DF}(g(eval(df^0_1), \ldots, eval(df^0_k)))$$

$$\equiv (df^0_1, df^0_2) := eval(df^0_1, \_ ) \equiv_D eval(df^0_2, \_ ),$$

where $\_ \text{ means that any parameter that fits the signature can be accepted.}$

The special cases decide and equals discussed earlier in Sections 3.3 and 3.1.1, respectively, could be redefined as:

$$decide(df, \_ ) := eval(apply(mode, df), \_ )$$

$$\equiv (df^0_1, df^0_2) := apply(\equiv_D, df^0_1, df^0_2) \equiv df^0_1$$

In general, all functions defined on the co-domain of the decision functions could be lifted to the decision functions over this co-domain.

3.5 Summary

In this chapter, we formally defined a notion of a decision function. We presented its default higher-order representation, which can be straightforwardly captured using decision trees or decision graphs. Furthermore, we defined its general constructor, a non-redundancy property and a minimum number of core operations, namely one: bind. Operations of learning
and *deciding*, which are always implemented in existing decision models, were specified over a general decision function. We showed that a decision function is composable (i.e., these operations can be used regardless of the specific decision function implementation). In what follows (in Chapter 4 and in Chapter 5) we will confirm this assumption by defining a general operation `merge` used to *combine* decision functions.

Auxiliary operations of decision functions, such as `evert` and `apply`, were also defined. These operations are non-core operations and, thus, use a core operation `bind` for their implementation. In the next chapter, we consider the accuracy of comparing, combining and approximating decision functions. The operations `merge` and `approx` used for the combining and approximating of decision functions, respectively, are based on `apply`. 
Chapter 4

Accuracy of Decision Functions

In the previous chapter, we formally defined decision functions using a core operation \textit{bind} and several auxiliary operations: \textit{evert}, \textit{apply} and \textit{equals}. What we want to define now is how we can generally compare, combine and approximate decision functions. These operations require a notion of accuracy of the decision functions. Thus, in order to define and use these operations later on, we first need to focus on: (1) how to compare the accuracy of the decision functions, (2) what happens to the accuracy of a combined decision function and (3) what happens to the accuracy of an approximated decision function.

This chapter is structured as follows. Section 4.1 presents two types of decision functions depending upon their accuracy, while Section 4.2 discusses how decision functions can be compared in terms of their accuracy. In Section 4.3, we first discuss the accuracy of a combined decision function and then define a \textit{merge} operation, which allows, generally, for the combining of different decision functions. Furthermore, in Section 4.4, we discuss the consequences of the approximation of a decision function on its accuracy and define a general \textit{approx} operation. As before, for each operation, a corresponding algebraic specification is given. Finally, in Section 4.5 we define a complete parametrized algebraic specification of Decision Algebra.

4.1 Conservative and Optimistic Decision Functions

Intuitively, a decision function \(df_1\) is \textit{more accurate} than a decision function \(df_2\) iff as it, more often, gives the "right" classification (i.e., we define it based on some ground truth). However, this ground truth is, in general, not known to us usually due to the fact that the formal context \(\vec{A}\) does not model all properties that have an influence on a decision. For one and the same actual context \(\vec{a} \in A\), different decisions \(c \in C\) are possible following a distribution \(D(C)\). This distribution is usually not known to us (just samples thereof with the training data sets). Still, we can define a "more accurate" relation based on the idea of a "right" classification from a theoretical "oracle".
Chapter 4. Accuracy of Decision Functions

Definition 14. Let \( \text{oracle}_{\vec{a}} : C \rightarrow \mathbb{R} \) be the accurate classification distribution of a decision given a concrete context \( \vec{a} \in \vec{A} \), and \( df_{\text{oracle}} : \vec{A} \rightarrow D(C) \) as the corresponding accurate decision function with \( \forall \vec{a} \in \vec{A} : \text{eval}(df_{\text{oracle}}, \vec{a}) = \text{oracle}_{\vec{a}} \).

Given the \( df_{\text{oracle}} \) decision function we can express the classification error in a concrete context \( \vec{a} \) and a general error of \( df \).

Definition 15. Let \( df : \vec{A} \rightarrow D(C) \) be any decision function, and \( \text{error}_{df}(\vec{a}) \) of a distribution of decisions \( d_{\vec{a}} = \text{eval}(df, \vec{a}) \) in a concrete context \( \vec{a} \in \vec{A} \) be defined as

\[
\text{error}_{df}(\vec{a}) = \frac{1}{|C|} \sum_{c \in C} (\text{prob}(\text{oracle}_{\vec{a}}, c) - \text{prob}(d_{\vec{a}}, c))^2
\]

and a general \( \text{error}_{df} \) of \( df \) is defined as

\[
\text{error}_{df} = \frac{1}{|\vec{A}|} \sum_{\vec{a} \in \vec{A}} \text{error}_{df}(\vec{a}).
\]

Finally,

Definition 16. The decision function \( df_1 : \vec{A} \rightarrow D(C) \) is more accurate than another decision function \( df_2 : \vec{A} \rightarrow D(C) \) iff as \( df_1 \) is closer to the \( df_{\text{oracle}} : \vec{A} \rightarrow D(C) \) accurate decision function than \( df_2 \):

\[
\text{error}_{df_1} \leq \text{error}_{df_2}.
\]

The theoretical background for comparing and combining decision functions reflects our previous work on points-to analysis [37]. In this chapter, we aim to apply our results from the points-to analysis domain to decision functions. The main challenge is that a points-to analysis co-domain can always be represented by a lattice \( L(C, \cup, \cap) \) (usually a power set lattice) with a defined supremum \( \cup \) and infimum \( \cap \), and, thus, the theory behind comparing and combining points-to analysis information is based on the lattice theory.

The results of the points-to analysis can be captured by a decision function with a lattice co-domain. However, as we discussed before, most of decision models capture class distributions \( D(C) \). Therefore, we chose the following approach. First, we show the theory of comparing and combining decision functions, where we restrict the co-domain of decision function to a power lattice \( L(\mathcal{P}^C, \cup, \cap) \), and then we show our attempt to adapt it to decision functions with class distribution co-domains \( D(C) \). Notice that we will use \( \mathcal{P}^C \) for a co-domain notion and \( d(C) \in \mathcal{P}^C \) for particular decision, until the further notice when we will switch to the decision distribution \( D(C) \) again.
4.1. Conservative and Optimistic Decision Functions

In general, a problem domain consists of infinitely many decision tuples and the ground truth of a decision in an actual context is not known. Therefore, a decision function $df$ can contain decisions that are not in the $df_{oracle}$ (garbage) (i.e., decision information captured is an over-approximation of a ground truth) defined by an expert in a problem domain. We refer to these decision functions as to conservative decision functions.

**Definition 17.** A decision function $df : \vec{A} \rightarrow \mathcal{P}^C$ is conservative with respect to the oracle decision function $df_{oracle} : \vec{A} \rightarrow \mathcal{P}^C$ iff:

$$ df_{oracle} \subseteq df \iff df_{oracle} \cup df = df $$

$$ \Rightarrow d_{oracle}(C) \subseteq d(C) \iff d_{oracle}(C) \cup d(C) = d(C), \quad \forall d(C), d_{oracle}(C) \in \mathcal{P}^C $$

Where the supremum $\cup$ and infimum $\cap$ of decision functions $df_1$ and $df_2$ are defined as follows:

$$ df_1 \cup df_2 = apply(\cup, df_1, df_2) $$

$$ df_1 \cap df_2 = apply(\cap, df_1, df_2) $$

where

$$ \subseteq : \mathcal{P}^C \times \mathcal{P}^C \rightarrow \text{BOOL} $$

$$ d_1(C) \subseteq d_2(C) \iff d_1(C) \cup d_2(C) = d_2(C) $$

The supremum $\cup$ and infimum $\cap$ over the co-domain of decision functions (as discussed before the co-domain for this example is a power lattice) are defined as union and intersection operations over sets, respectively. Thus, the operation $\subseteq$ over decision functions can be defined as follows:

$$ df_1 \subseteq df_2 = apply(\subseteq, df_1, df_2) \equiv df_{true} $$

$$ df_1 \subseteq df_2 \iff df_1 \cup df_2 = df_2 $$

where $df_{true}$ is the decision function over boolean true.

Decision functions can also miss some of the decisions contained in $df_{oracle}$ (misses) (i.e., decision information captured is a sample (under-approximation) of a ground truth, which is observed in a restricted environment). We refer to these decision functions as to optimistic decision functions.

**Definition 18.** A decision function $df : \vec{A} \rightarrow \mathcal{P}^C$ is optimistic with respect to the oracle decision function $df_{oracle} : \vec{A} \rightarrow \mathcal{P}^C$ iff:

$$ df \subseteq df_{oracle} \iff df \cup df_{oracle} = df_{oracle} $$

$$ \Rightarrow d(C) \subseteq d_{oracle}(C) \iff d(C) \cup d_{oracle}(C) = d_{oracle}(C), \quad \forall d(C), d_{oracle}(C) \in \mathcal{P}^C $$
Chapter 4. Accuracy of Decision Functions

For a better understanding of conservative and optimistic decision functions, let us take an example of our “Car evaluation” data set that was first presented in Section 2.2. A short reminder: the decision of this data set determines whether a car is in acceptable condition based on several attributes, such as “buying price” and “maintenance price”, etc. An optimistic decision function of this problem domain would be a decision function that captures the decision information observed in let us say some auto service center. All of the information captured is true, but we also miss the information that could be taken from other auto service centers (thus, under-approximating the ground truth). A conservative decision function, in this scenario, is a decision function that captures the information taken from an auto-expert, who over-approximates the ground truth by excluding the cases that are not possible in the reality. For instance, “Lada” marque can never be in an “acceptable” condition regardless its attribute values.

4.2 Comparing Decision Functions

In order to compare decision functions, we have to evaluate the garbage (false positives) and misses (false negatives) that they contain and to trade the garbage off against the misses. Let \( df_{\text{oracle}} : \mathcal{A} \rightarrow \mathcal{P} \mathcal{C} \) be the oracle decision function of a certain problem domain and \( df : \mathcal{A} \rightarrow \mathcal{P} \mathcal{C} \) be some decision function that captures the decision information from the same problem domain. Then, the amount of misses of a decision function \( df \) can be assessed using a metric \( R \) called recall. High recall means that a decision function captures most of the relevant decisions.

\[
R = \frac{|df \cap df_{\text{oracle}}|}{|df_{\text{oracle}}|}
\]

where to compute the cardinality of \( df \) (\(|df|\)), we need to sum up the cardinalities of the decision sets for all actual contexts \( \vec{a} \in \mathcal{A} \) of \( df \):

\[
|df| = \sum_{\vec{a} \in \mathcal{A}} \text{eval}(df_{\text{cardinality}}, \vec{a})
\]

\[
df_{\text{cardinality}} = \text{apply}(\text{cardinality}, df)
\]

A cardinality operation over sets returns a natural number \( \mathbb{N} \).

We can assess the amount of “garbage” using a metric \( P \) that defines precision. High precision means that a decision function captures substantially more relevant decision information than irrelevant decision information.

\[
P = \frac{|df \cap df_{\text{oracle}}|}{|df|}
\]
4.2. Comparing Decision Functions

Table 4.1: Comparing conservative, optimistic and general decision functions (where the co-domain is a power lattice \( \mathcal{P}^C \)). \( df_1, df_2 \) denote the respective decision functions and the \( df_{\text{oracle}} \) denotes the oracle-accurate decision function. \( P, R \) and \( F \) denote precision, recall and harmonic F-score.

| Special case of decision functions \( df_1, df_2 \) | Comparison of \( df_1, df_2 \) with respect to \( df_{\text{oracle}} \) | \( P \) \( R \) \( F \)-score |
|--------------------------------------------------|--------------------------------------------------|--|--|
| \( df_1, df_2 \) conservative                   | \( P_1 \geq P_2 \iff |df_1| \leq |df_2| \) | \( R_1 = R_2 = 1 \) | \( F_1 \geq F_2 \iff |df_1| \leq |df_2| \) |
| \( df_1, df_2 \) optimistic                      | \( P_1 = P_2 = 1 \) | \( R_1 \geq R_2 \iff |df_1| \geq |df_2| \) | \( F_1 \geq F_2 \iff |df_1| \geq |df_2| \) |
| \( df_1 \) conservative, \( df_2 \) optimistic  | \( P_1 \leq P_2 = 1 \) | \( R_1 = 1 \geq R_2 \) | \( F_1 \geq F_2 \iff \text{model} \leq |df_1| \) |
| \( df_1 \) conservative, \( df_2 \) general     | \( P_1 \geq P_2 \iff |df_1| \leq |df_2| \) | \( R_1 = 1 \geq R_2 \) | \( F_1 \geq F_2 \iff |df_1| \leq |df_2| \) |
| \( df_1 \) optimistic, \( df_2 \) general       | \( P_1 = 1 \geq P_2 \) | \( R_1 \geq R_2 \iff |df_1| \geq |df_2| \) | \( F_1 \geq F_2 \iff |df_1| \geq |df_2| \) |

Table 4.2: Combining decision functions \( df_1, df_2 \) (where the co-domain is a power lattice \( \mathcal{P}^C \)).
Recall and Precision range between 0 (worst) and 1 (best). The recall can often be increased at the cost of reducing the precision and vice versa. Therefore, these two measures are combined into a single measure called $F$-score, which balances $R$ and $P$:

$$F = \frac{2 \times P \times R}{P + R}$$

Consequently, $R_{\text{cons}} = 1$ for a conservative and $P_{\text{opt}} = 1$ for an optimistic decision function. The precision of a conservative $P_{\text{cons}}$ (recall of an optimistic $R_{\text{opt}}$) decision function is:

$$P_{\text{cons}} = \frac{|d_{\text{oracle}}|}{|d_f|}$$

$$R_{\text{opt}} = \frac{|d_f|}{|d_{\text{oracle}}|}$$

Learning algorithms construct decision functions that may be conservative, $d_{\text{oracle}} \subseteq d_f$, optimistic, $d_f \subseteq d_{\text{oracle}}$, or general (neither conservative nor optimistic). Usually, they are optimistic as training data is observed in reality.

To compare the accuracy of two decision functions, one could measure the precision $P$ and recall $R$ of these functions with the known $d_{\text{oracle}}$ and compare the resulting $F$-scores. This approach is impracticable if no such $d_{\text{oracle}}$ can be provided.

### 4.2.1 Direct Comparison

A comparison of decision functions $d_{f_1}$ and $d_{f_2}$ without an $d_{\text{oracle}}$ decision function at hand exploits the fact that one of the two decision functions or both might be conservative or optimistic. Five situations may occur when comparing different decision functions with special cases involved. They are shown in Table 4.1 as well as the implications of the result of comparing their accuracy.

The proofs for all claims in Table 4.1 follow directly from the definitions of precision, recall and $F$-score in the case of the conservative, optimistic and general decision functions, respectively.

As we can see, comparisons without knowing the $d_{\text{oracle}}$ decision function are (trivially) possible for the first two cases when both decision functions are conservative or optimistic, respectively. When comparing a conservative with an optimistic, we need the $d_{\text{oracle}}$ to get an exact comparison. If either decision functions is general only semi-decisions can be made. Therefore, we need to find another way to compare the accuracy of these decision functions.
4.2. Comparing Decision Functions

4.2.2 Approximating Accuracy

If we assume that we can over- or under-approximate a decision function \(df_{\text{oracle}}\), then such approximations can be calculated either from a conservative decision function leading to \(df_{\text{oracle}}\) over-approximations or from an optimistic one leading to under-approximations. Let us define \(df_{\text{oracle}} \sqsubseteq df_{++}\) (\(df_{\text{oracle}} \sqsubseteq df_{++}\)) as an over- (under-) approximation of the decision function \(df_{\text{oracle}}\). Let \(F^+(F^-)\) be an upper (lower) bound of the actual \(F\)-score. The idea is as follows. If we could get the upper and lower bounds of the exact \(F\)-scores \(F_1\) and \(F_2\) of two decision functions \(df_1\) and \(df_2\), then we could (semi-) compare the decision functions using \(F_1 \geq F_2 \Rightarrow F_1 \geq F_2\).

Assume a conservative decision function \(df\). In this case, the exact \(F\)-score reduces to:

\[
F = \frac{2 \times |df_{\text{oracle}}|}{|df_{\text{oracle}}| + |df|}
\]

Operations plus + and multiplication \(\times\) are defined over Natural numbers \(\mathbb{N}\). Applying the \(F\)-score definition to over- and under-approximations of the \(df_{\text{oracle}}\) yields:

\[
\hat{F} = \frac{2 \times |df_{++} \cap df|}{|df_{++} \cap df| + |df|}
\]

\[
\tilde{F} = \frac{2 \times |df_{--} \cap df|}{|df_{--} \cap df| + |df|}
\]

From \(df_{++} \sqsubseteq df\) (resp. \(df_{--} \sqsubseteq df\)) it follows \(df_{++} \cap df = df_{++}\) (\(df_{--} \cap df = df_{--}\)) and together with \(|df_{++}| \geq |df_{\text{oracle}}| (|df_{--}| \leq |df_{\text{oracle}}|)\) it follows that \(\hat{F} \geq F \geq \tilde{F}\).

Note that \(df_{++} \sqsubseteq df\) does not necessarily hold for any over-approximation of \(df_{\text{oracle}}\). Assume an over-approximation \(df_{++}\) with \(df_{++} \setminus df \neq \text{cons}_D(\emptyset)\), where

\[
df_{++} \setminus df = \text{apply}(\setminus, df_{++}, df)
\]

Then we can construct an even better over-approximation \(df_{\text{oracle}} = df_{++} \cap df\) for which \(df_{++} \sqsubseteq df\) holds.

Hence, we can always derive lower and upper bounds for the \(F\)-scores of conservative decision functions: \(F^- = \tilde{F}, F^+ = \hat{F}\). This result has no effect on the comparability of the conservative decision functions as we can compare them directly. Dually, we can also derive lower and upper bounds for the \(F\)-scores of the optimistic decision function. In this case \(F^- = \tilde{F}, F^+ = \hat{F}\). Again, this result has no effect on the comparability of the optimistic decision functions as we can compare them directly, as well.
Chapter 4. Accuracy of Decision Functions

Unfortunately, for general decision functions, we can neither establish upper nor lower bounds of the $F$-scores based on an over- or under-approximated decision function $df_{oracle}$. To see it, we construct a counter example from two extreme cases of an under-approximated oracle $df^-_{oracle}$: (i) $df^-_{oracle} \subseteq df_{oracle} \subseteq df \Rightarrow F > \tilde{F}$ and (ii) $df^-_{oracle} = df \cap df_{oracle} \Rightarrow F < \tilde{F}$. Symmetrically, upper and lower bounds cannot be constructed with $df^+_{oracle}$ either, which leads to the conclusion that, even with over- or under-approximations of the $df_{oracle}$ decision function, we still cannot get any closer to comparing the general decision functions.

4.2.3 Accuracy of Learning by Combining Decision Functions

Combining optimistic decision functions by computing the union $\cup$ over co-domain $\mathcal{P}^+$ and conservative decision functions by computing the intersection $\cap$ over co-domain $\mathcal{P}^C$ are obvious improvements. In Table 4.2, we summarize the results of the direct combining of the two decision functions. The first two cases, combining two conservative and two optimistic decision functions, respectively, are quite obvious based on the conclusions from the previous subsection. The combined decision function is always more accurate than either of the two input decision functions and it is conservative or optimistic, respectively.

When combining a conservative (optimistic) decision function with a general one, the accuracy of the general decision function improves as we take away decisions that are not (add decisions that are) contained in the decision function $df_{oracle}$. However, we cannot say if the conservative (optimistic) input decision function $df$ is improved. The fact that we compare a conservative (optimistic) decision function $df$ with a general combined decision function $df_{comb}$ allows us to apply the semi-decider $F_1 \geq F \iff |df| \leq |df_{comb}|$ (resp. $F_1 \geq F \iff |df| \geq |df_{comb}|$). However, as $df_{comb}$ is the intersection $\cap$ (union $\cup$) of $A_1$ with something, $df_{comb} \subseteq df$ ($df \subseteq df_{comb}$) and, hence, $|df_{comb}| \leq |df|$ ($|df_{comb}| \geq |df|$). Therefore, we can only guarantee $F_1 \geq F$ in the trivial case of $|df_{comb}| = |df|$ (i.e., when we haven’t cut off (added) something to the conservative (optimistic) decision function). This property continues to hold in the special case where the general decision function is an optimistic (conservative) one. This case is, therefore, omitted in Table 4.2.

4.3 Combining General Decision Functions

Combining decision functions is interesting if the combined decision function becomes more accurate than the original input decision functions. Optimistic decision functions (the usual case) become more accurate. As we
4.3. Combining General Decision Functions

have bot previously discussed combining (or comparing) general decision functions, we will discuss the above theory in terms of class distribution co-domain $D(C)$ in this section. Then, we will present an operation merge over decision functions and discuss how the accuracy of the combined decision function changes the general decision functions. Finally, three possible scenarios for merging decision functions are given.

Previously, we discussed decision functions over lattice-based co-domains. However, in general, learning algorithms represent decision information with distributions over classes $D(C)$ (or sometimes with probability distributions). Therefore, what we want to discuss now is how the above theory for comparing and combining decision functions can be applied to decision functions with such co-domains.

First, we will try to represented co-domain as a lattice of fuzzy sets $F(C, \mu(C))$ [118], where each class $c \in C$ has a grade of membership $\mu(c)$. Based on the fuzzy logic, we define an intersection $\cap$ over decision functions $d_1, d_2$ as a decision function over the intersection $\cap$ of their respective fuzzy sets $d_1(C) \in F(C, \mu(C))$ and $d_2(C) \in F(C, \mu(C))$ with membership functions $\mu_1$ and $\mu_2$, respectively:

$$d_1 \cap d_2 = apply(\cap, d_1, d_2)$$

$$d_1(C) \cap d_2(C) = \min(\mu_1(C), \mu_2(C)), \forall c \in C$$

and we define a union $\cup$ over decision functions $d_1, d_2$ as a decision function over the union $\cup$ of their respective fuzzy sets $d_1(C) \in F$ and $d_2(C) \in F$:

$$d_1 \cup d_2 = apply(\cup, d_1, d_2)$$

$$d_1(C) \cup d_2(C) = \max(\mu_1(C), \mu_2(C)), \forall c \in C$$

However, even after defining precision and recall we cannot really apply the results of these implications. The reason is that we cannot determine whether the decision function constructed by a certain learning algorithm is conservative or optimistic due to the fact that the learning algorithm approximates the decision information and the number of training instances does not really indicate whether the constructed decision function is conservative, optimistic or general one. Additionally, the source of the data sometimes is not known and contains misses and junk information. Therefore, in this scenario, we should always consider a decision function as a general one (i.e., we cannot apply the theoretical implications presented before in Section 4.2).

Secondly, we tried to define the same operations over decision functions with probability distributions co-domains $P(C)$. In order to do this action we defined union $\cup$ and intersection $\cap$ operations over probability distributions
Chapter 4. Accuracy of Decision Functions

\( X, Y \in P(C) \) as:

\[
\begin{align*}
p_i &= P(X = c) \\
p'_i &= P(Y = c) \\
p_i \cap p'_i &= \frac{P(X = c, Y = c)}{P(X = Y)} = \frac{p_i p'_i}{\sum_{k \in |C|} (p_k p'_k)} \\
p_i \cup p'_i &= \frac{p_i p'_i}{\sum_{k \in |C|} (p_k p'_k)} \\
p_i \cup p'_i &= 1 - \frac{p_i p'_i}{\sum_{k \in |C|} (p_k p'_k)}
\end{align*}
\]

However, in this scenario, the basic axiom of union and intersection do not hold (i.e. \( X \cup X \neq X \) and \( X \cap X \neq X \)). For instance, \( X = (0.7, 0.3) \), then:

\[
X \cap X = \left\{ \frac{0.7 \times 0.7}{0.7 \times 0.7 + 0.3 \times 0.3}, \frac{0.3 \times 0.3}{0.7 \times 0.7 + 0.3 \times 0.3} \right\}
\]

\( X \cap X = \{0.84, 0.16\} \)

However, we still want to combine two general decision functions, and, obviously, we want to have an accuracy improvement for such a combination. In what follows, we define the combination operation called merge over two general decision functions with a class distribution co-domain \( D(C) \).

### 4.3.1 Merge Operation

In order to combine two decision functions, a merge operator \( \sqcup_D \) has to be defined over the co-domain of these functions \( D(C) \) (see Section 3.1):

\[
\sqcup_D : D(C) \times D(C) \rightarrow D(C)
\]

When it comes to combining two general decision functions, the merge operator \( \sqcup_D \) can be specified in different ways. We even can combine two decision functions to get a new decision function over a different co-domain \( D(C) \). For instance, one decision function specifies the height and the other the weight of the person. We can combine these decision functions to get a body mass index (bmi) by applying a certain merge operator defined as a formula for calculating bmi.

We define the \( \sqcup_D \) operation over class distributions \( D(C) \) as:

\[
d(C) \sqcup_D d'(C) = \{(c, p + p')|(c, p) \in d(C), (c, p') \in d'(C)\}
\]

The + operator on frequencies \( p \in P \) is the plus of the Natural numbers. We consider the \( \sqcup_D \) of the two distributions to be the base case of our merge operation of the decision functions.
4.3. Combining General Decision Functions

A general merge operation of the two decision functions \( df_1 \in DF_1[\vec{A}, D] \) and \( df_2 \in DF_2[\vec{A}, D] \) has the signature of:

\[
\sqcup : DF_1 \times DF_2 \rightarrow DF
\]

A merge operation is based on \( \text{apply} \). It applies a merge function \( \sqcup_D \) on two decision functions:

\[
\sqcup(df_1, df_2) := \text{apply}(\sqcup_D, df_1, df_2)
\]

with the following base case implementation over constant decision functions \( df_1^0, df_2^0 \in DF_0[[0], D] \):

\[
\sqcup(df_1^0, df_2^0) := \text{cons}^0_{DF}(\sqcup_D(\text{eval}(df_1^0, \_), \text{eval}(df_2^0, \_)))
\]

### 4.3.2 Accuracy of the Combined Decision Function

What we wish to establish is that combining decision functions gives us a new decision function that is tententially more accurate (i.e., reduces the error). As a consequence, we could define a simple and general learning approach based on combining decision functions. This combining learning approach would work as an online algorithm, as a new decision function can be learned howsoever (any learning algorithm would do) from new decision information and be combined with an existing decision function, which thereby gets more accurate over time. Moreover, as combining is linear in regard to the size of the decision functions involved, partitioning (i.e., dividing given decision information and learning the individual decision functions that are combined) is faster than any hyper-linear learning (i.e., learning on the whole based on the given decision information). This way, we can save learning time and guarantee scalability even for large data sets.

Unfortunately, we cannot guarantee that the combination of two general decision functions will always create a more accurate decision function. This conclusion is even intuitively clear if we think of merging a very accurate decision function with another one that makes random decisions, which clearly cannot improve the already accurate decision function. However, we can assume somewhat accurate input decision functions in order to expect an even more accurate output. The following holds for any distribution of co-domain \( D(C) \). We formalize the notion of "somewhat accurate" as probably accurate:

**Definition 19.** A decision functions \( df : \vec{A} \rightarrow D(C) \) is **probably accurate** with respect to an accurate decision function \( df_{\text{oracle}} \) iff \( \forall \vec{a} \in \vec{A} : \text{eval}(df, \vec{a}) \) is a random sample of \( \text{oracle}_d \).
Chapter 4. Accuracy of Decision Functions

The combination of probably accurate decision functions leads to a more accurate one.

**Theorem 4.3.1.** Let $df_1, \ldots, df_n$ be a series of independently learned decision functions $df : \vec{A} \rightarrow D(C)$ that are probably accurate with respect to an accurate decision function $df_{\text{oracle}} : \vec{A} \rightarrow D(C)$. For large $n$, the merged decision function $df_1 \sqcup \ldots \sqcup df_n$ converges in probability to the $df_{\text{oracle}}$.

**Proof.** Let $d_{a,1} = \text{eval}(df_1, a), \ldots, d_{a,n} = \text{eval}(df_n, a)$ be a series of distributions of decision functions $df_1, \ldots, df_n$ in a concrete context $\vec{a}$, each a random sample of $oracle_{\vec{a}}$. As we prove the theorem independently for each concrete context $\vec{a} \in \vec{A}$, we drop the index $\vec{a}$ from now on and assume an arbitrary but fixed actual context. For the merger $\sqcup$ of the series of distributions the following holds

$$\forall c \in C : \text{prob}((d_1 \sqcup \ldots \sqcup d_n), c) = \text{prob}(d_1, c) + \ldots + \text{prob}(d_n, c)$$

and

$$|d_1 \sqcup \ldots \sqcup d_n| = |d_1| + \ldots + |d_n|$$

with $|d_i| = \sum_{c \in C} \text{prob}(d_i, c)$ is the size of a distribution (i.e., the sum of all frequencies $\text{prob}(d_i, c)$ of all of the possible classes $C$). As each $d_i$ is a random sample of the $oracle$, it can be understood as a result of a series of $|d_i|$ random decisions:

$$\text{prob}(d_i, c) = [D_1 = c] + \ldots + [D_{|d_i|} = c]$$

or

$$d_i(c) = \frac{[D_1 = c] + \ldots + [D_{|d_i|} = c]}{|d_i|}$$

which is the unweighed average of a series of random decisions for which Corollary A.1 applies (see Appendix A). If to consider $X_i = \frac{d_i(c)}{|d_i|}$ as random variables, then $E(X_i) = p_c$ and $Var(X_i) \leq p_c(1 - p_c)$.

Moreover, for the merged distribution it holds:

$$\text{prob}((d_1 \sqcup \ldots \sqcup d_n), c) = \text{prob}(d_3, c) + \ldots + \text{prob}(d_n, c)$$

$$= \frac{d_3(c)}{|d_1|} |d_1| + \ldots + \frac{\text{prob}(d_n, c)}{|d_n|} |d_n|$$

$$= X_1|d_1| + \ldots + X_n|d_n|.$$ 

Instead of the merged distribution $(d_1 \sqcup \ldots \sqcup d_n)$, we consider its corresponding distribution $(d_1 \sqcup \ldots \sqcup d_n)_N$, which is normalized with $N = \sum_{i=1}^n |d_i|$.\(^1\)

$$\text{prob}((d_1 \sqcup \ldots \sqcup d_n)_N, c) = \frac{\sum_{i=1}^n X_i \times |d_i|}{\sum_{i=1}^n |d_i|}$$

\(^1\)Normalization: $d(c) = N \times p_c \Leftrightarrow d_N(c) = p_c$.
4.3. Combining General Decision Functions

The above results in the weighted average of a series of random variables $X_i$ with weights $N_i = |d_i|$ for which Lemma A.2 in Appendix A applies. Hence,

$$\forall c \in C : \lim_{n \to \infty} P(\| \text{prob}(d_1 \cup \ldots \cup d_n)_{N_i} - p_c \| \geq \epsilon) = 0 \text{ for any } \epsilon > 0.$$ 

This property holds for any actual context $\vec{a} \in \vec{A}$ which concludes the proof.

Recall that $\text{decide}(df, \vec{a}) = \text{eval}(\text{apply}(\text{mode}, df), \vec{a})$ (i.e., applying the $\text{mode}$ to the underlying distributions). The result of the $\text{mode}$ is unaffected by a normalization with any positive constant $N$, especially $N = |d|$, which is the size of the distribution:

$$\text{mode}(d) = c \Leftrightarrow \text{prob}(d, c) = \max_{c' \in C} \text{prob}(d, c') \Leftrightarrow$$

$$\Leftrightarrow \text{prob}(d_{N}, c) = \frac{\text{prob}(d, c)}{N} = \max_{c' \in C} \frac{\text{prob}(d, c')}{N}, N > 0.$$ 

In what follows, we define a general operator $\text{merge}$ for combining decision functions regardless of their implementation details. The only prerequisite for applying this operation is an implemented co-operation $\text{bind}$.

Previously, we discussed the accuracy of the combined decision functions and we showed that we needed to define a "probably accurate" decision function in order to guarantee that a combined decision function becomes tendentiously more accurate.

Having these prerequisites, we can now establish three scenarios of combining decision functions. These scenarios depend upon the formal context $\vec{A}$ of the combined decision functions, where the contexts can be same, intersecting or disjoint. The scenarios depend on the sample of a decision information from which a decision function was learned. Each scenario includes the prerequisites for applying the scenario, the definition of a merge operation, the algorithm for combining the two decision functions and an example.

4.3.3 Scenarios for Combining Decision Functions

The general $\text{merge}$ operation applies in three different scenarios as discussed below.

4.3.3.1 Scenario One: Same Formal Context.

Prerequisite: The decision functions $df_1 \in DF_1[\vec{A}, D]$ and $df_2 \in DF_2[\vec{A}', D]$ are constructed over different samples of the same problem domain and $\vec{A} = \vec{A}' = A_1 \times \ldots \times A_n$. 

51
Chapter 4. Accuracy of Decision Functions

Algorithm 2  \( \sqcup(d_1, d_2), d_1, d_2 \in DF[A_1 \times \ldots \times A_n, D] \)

1: if \( d_1 \in DF_0[[\overline{0}], D] \) \& \( d_2 \in DF_0[[\overline{0}], D] \) then 
2: \( \text{return } x(\sqcup_D(d_1, d_2)) \)
3: end if 
4: for each \( a \in A_1 \) do 
5: \( d_a = \sqcup(\text{bind}_1(d_1, a), \text{bind}_1(d_2, a)) \)
6: end for 
7: \( \text{return } \text{cons}_{DF}(a_1, d_{a_1}, \ldots, a_{|A_1|}, d_{a_{|A_1|}}) \)

Merge: Let \( A_1 = \{a_1, \ldots, a_k\} \). The merge \( \sqcup \) operation over the decision functions has the following recursive abstract implementation:

\[
\sqcup(d_1, d_2) := x^{a_1}(\sqcup(\text{bind}_{A_1}(d_1, a_1), \text{bind}_{A_1}(d_2, a_1)));
\]

\[
\ldots,
\]

\[
a_{k}(\sqcup(\text{bind}_{A_1}(d_1, a_k), \text{bind}_{A_1}(d_2, a_k)))
\]

\[
\sqcup(d_1^0, d_2^0) := x^0(\sqcup(\text{eval}(d_1^0, \_), \text{eval}(d_2^0, \_)))
\]

We recursively applied \( \sqcup \) to the respective sub-terms of the arguments and, eventually, evaluated it on the constant decision functions \( d_1^0 \) and \( d_2^0 \). The result is a new decision function \( d \in DF'[A, D] \) recursively constructed from the results.

Algorithm: Algorithm 2 outlines an implementation of a merge operation over two decision functions. Lines 1–3 handle the base case where both of the decision functions are constant. Lines 4–7 handle the recursive case where both of the decision functions are partially evaluated with regard to the first attribute \( A_1 \in \overline{A}_1, \overline{A}_2 \). We will extend this algorithm in the next scenarios.

Example: Figure 4.1 shows the merging of two decision functions \( d_1, d_2 \in DF[A_1, D] \) as depicted in (a) and (b), respectively. Note that in \( d_2 \) (b), the second attribute “Maintenance price” does not contribute to the decision. The first call applies the statements in Lines 4-6. The roots of both decision functions correspond to the same attribute “Buying price.” In each iteration, a recursive call merges the sub-terms of \( d_1 \) and \( d_2 \) with “Buying price” bound to each of the possible values (very high, high, medium, low). Algorithm 2 is called recursively four times on decision functions with “Maintenance price” the root attribute. Figure 4.1 (c) shows the intermediate result before finally merging the resulting functions in Line 7. In each recursive call, Lines 4-6 are executed again and the algorithm is again recursively called four times with “Maintenance price” bound to each of its possible values (very high, high, medium, low). These second recursions lead to the base case in Lines 1-3.
4.3. Combining General Decision Functions

\[ \text{Figure 4.1: Merging of the Decision Functions (Scenario 1)} \]

where the actual distributions are merged.

4.3.3.2 Scenario Two: Disjoint Formal Contexts.

**Prerequisite:** The decision functions \( df_1 \in DF_1[\vec{A}, D] \) and \( df_2 \in DF_2[\vec{A}', D] \) are constructed over samples with disjoint formal contexts of the same problem domain: \( \vec{A} = A_1 \times \ldots \times A_n \) and \( \vec{A}' = A'_1 \times \ldots \times A'_m \) and \( \{A_1, \ldots, A_n\} \cap \{A'_1, \ldots, A'_m\} = \emptyset \).

**Merge:** The definition of merge needs to be extended accordingly with an additional base case:

\[ \sqcup(df_1^0, df_2) := \sqcup(df_2^0, df_1^0) \]

We recursively applied \( \sqcup \) to the sub-terms of \( df_1 \) and the whole of \( df_2 \) since all attributes of \( \vec{A} \) are unknown in the context \( \vec{A}' \) of \( df_2 \), until the sub-terms are constant decision functions. In this base case, we flipped the arguments and recursively applied \( \sqcup \) to the sub-terms of \( df_2 \) and the constant decision functions of \( df_1 \) until we merged the constant decision functions and recursively constructed the results.

**Algorithm:** Algorithm 3 extends Algorithm 2 and implements the new base case in Lines 4–6.
Chapter 4. Accuracy of Decision Functions

Algorithm 3 \( \sqcup (df_1, df_2), df_1 \in DF[\bar{A}, D], df_2 \in DF[\bar{A}^\prime, D] \)

1: if \( df_1 \in DF[\emptyset, D] \) \& \( df_2 \in DF[\emptyset, D] \) then
2: \hspace{1em} return \( x(\sqcup_D(df_1, df_2)) \)
3: end if
4: if \( df_1 \in DF[\emptyset, D] \) then
5: \hspace{1em} return \( \sqcup (df_2, df_1) \)
6: end if
7: for each \( a \in A_1 \) do
8: \hspace{1em} \( df_a = \sqcup (\text{bind}_1(df_1, a), \text{bind}_1(df_2, a)) \)
9: end for
10: return \( \text{cons}_{DF}(a_1, df_a_1, \ldots, a_{|A_1|}, df_{a_{|A_1|}}) \)

Algorithm 4 \( \sqcup (df_1, df_2), df_1 \in DF[\bar{A}, D], df_2 \in DF[\bar{A}^\prime, D], \bar{A} = A_1 \times \ldots \times A_n, \bar{A}^\prime = A_1^\prime \times \ldots \times A_m \)

1: if \( df_1 \in DF[\emptyset, D] \) \& \( df_2 \in DF[\emptyset, D] \) then
2: \hspace{1em} return \( x(\sqcup_D(df_1, df_2)) \)
3: end if
4: if \( df_1 \in DF[\emptyset, D] \) then
5: \hspace{1em} return \( \sqcup (df_2, df_1) \)
6: end if
7: if \( A_1 \neq A_1^\prime \) \& \( A_1 \in \{A_2^\prime, \ldots, A_m^\prime\} \) then
8: \hspace{1em} return \( \sqcup (df_1, \text{evert}_{A_1}(df_2)) \)
9: end if
10: for each \( a \in A_1 \) do
11: \hspace{1em} \( df_a = \sqcup (\text{bind}_1(df_1, a), \text{bind}_1(df_2, a)) \)
12: end for
13: return \( \text{cons}_{DF}(a_1, df_a_1, \ldots, a_{|A_1|}, df_{a_{|A_1|}}) \)
4.3. Combining General Decision Functions

Example: Figure 4.2 shows the merging of the two decision functions $d_{f1} \in DF_{1}[\vec{A}, D]$ (a) with the formal context $\vec{A} = (5, 1, 3)$ and $d_{f2} \in DF_{2}[\vec{A}', D]$ (b) with the formal context $\vec{A}' = (6, 4)$. Algorithm 3 evaluates $d_{f1}$ with regard to the formal context of attributes 5, 1, 3 (Lines 7–9). Then, for each constant decision function $d_{f1}^0$, the new base case applies (Lines 4–6) and the algorithm evaluates the decision function $d_{f2}$ for its formal context (attributes 6, 4) (Lines 7–9). Finally, the constant decision functions $d_{f1}^0, d_{f2}^0$ are merged using $\sqcup_D$ and a merged decision function is constructed (Lines 1–3, and in the recursive ascent in Line 10). The graphs in Figure 4.2 (c) and (d) show the sub-graphs before the final decision graph construction and the resulting decision graph, respectively.

4.3.3 Scenario Three: General Case.

Prerequisite: For this general case, scenarios one and two were just special cases. The decision functions $d_{f1} \in DF_{1}[\vec{A}, D]$ and $d_{f2} \in DF_{2}[\vec{A}', D]$ are constructed over samples with arbitrary formal contexts of the same problem domain: $\vec{A} = A_1 \times \ldots \times A_n$ and $\vec{A}' = A'_1 \times \ldots \times A'_m$.

Merge: The definition of merge needs to be extended once more such
Chapter 4. Accuracy of Decision Functions

Figure 4.3: Merger of Decision Functions (Scenario 3)

that it aligns the common attributes of the two formal contexts $\vec{A}$ and $\vec{A}'$.

$$\sqcup(d_f_1, d_f_2) := \sqcup(d_f_1, \text{evert}_{A_1}(d_f_2)) \text{ iff } A_1 \in \{A'_2, \ldots, A'_m\}$$

**Algorithm:** Algorithm 4 extends Algorithm 3 and implements the reordering of the second formal context in Lines 7-9.

**Example:** Figure 4.3 shows the merging of the two decision functions $d_f_1 \in DF_1[\vec{A}, D]$ (a) with the formal context $\vec{A} = (6, 5, 1, 3)$ and $d_f_2 \in DF_2[\vec{A}', D]$ with the formal context $\vec{A}' = (6, 4)$ (b) with one overlapping attribute 6. The roots of both decision functions are different (attributes 6 and 4 respectively). Therefore, on the first iteration, $d_f_2$ needs to be aligned to the common attribute 6 (lines 7-9) as shown in Figure 4.3 (b). Then, the decision functions are recursively merged (lines 10-12). On the second iteration, the common attributes are no longer available, therefore we merged the rest according to the scenario two: Algorithm 4 evaluates $d_f_1$ with regard to the formal context attributes (5, 1, 3) (Lines 7-9). Then, each $d_f_1^0$ algorithm evaluates $d_f_2$ with regard to its formal contexts (4). Finally, the constant decision functions are merged. The intermediate result before the final merge is presented on Figure 4.3 (c). The final merged decision function is shown on Figure 4.3 (d).
4.4 Approximating Decision Functions

4.3.4 Scalability of Combining the Decision Functions.

Decision functions \(\text{df} : \vec{A} \rightarrow D(C)\) require memory \(\Omega(|\vec{A}| \times |C|)\) (i.e., they need to capture, for each actual context \(\vec{a} \in \vec{A}\) and each class \(c \in C\), the corresponding frequency \(d_a(c)\)). Hence, this memory is obviously a lower bound for the time complexity of any learning algorithm creating such decision functions.

Assuming that accessing the distribution of a decision function in an actual context \(d_a = \text{eval}(\text{df}, \vec{a})\) takes constant time, then the time complexity of the merged based learning is exactly this lower bound. Hence, other learning algorithms are at least as complex as the merged based learning. Therefore, we can save the learning time of the more complex learning algorithms and guarantee scalability even for large training data sets by learning on smaller data sets and merging the resulting decision functions.

Moreover, learning on the smaller data sets can trivially be parallelized as using \(p\) processors, it takes time \(O(l(n/p))\) to learn \(p\) decision functions given a data set of size \(n\) and a complexity of \(O(l(n))\) of the learning algorithm. Using a parallel sum technique, the merge operation of the resulting decision functions can be parallelized as well, requiring \(O(\log(p))\) sequential merge operations of decision functions on \(p\) processors. Merging distributions takes time \(O(|C|)\) on one processor or \(O(1)\) on \(|C|\) processors. Merging decision functions take \(O(|\vec{A}| \times |C|)\) on one processor or \(O(1)\) on \(|\vec{A}| \times |C|\) processors. Altogether, simple learning can be done in time \(O(l(n/p) + \log(p \times |\vec{A}| \times |C|))\) on \(p\) processors or \(O(l(n/p) + \log(p))\) on \(p \times |\vec{A}| \times |C|\) processors.

In general, we can use \(\sqcup\) to formalize a simple learning algorithm: a neutral element \(\bot \in D(C)\) is a default class representing “don’t know” and learning starts without any knowledge (i.e., with the initial decision function \(\text{df} \equiv \text{df}_\bot\)). Each tuple \((\vec{a}, c)\) in the training set corresponds to a decision function:

\[
df' (\vec{b}) = \begin{cases} c & \text{if } \vec{b} = \vec{a} \\ \bot & \text{otherwise} \end{cases}
\]

For each \(\text{df}'\) corresponding to a tuple of the dataset, the learning incrementally sets \(\text{df} := \sqcup(\text{df}, \text{df}')\).

4.4 Approximating Decision Functions

In general, pre-computing decide after learning may save representation space without sacrificing decision accuracy. Alternatively, space can be traded off against information accuracy if we approximate a decision by ignoring one attribute and merging different sub-functions of alternative values of that attribute. Notice, that this merge over the sub-function of the decision functions does not differ for conservative and optimistic decision functions.
Chapter 4. Accuracy of Decision Functions

![Decision Function Diagram]

Figure 4.4: An tree representation of $df^2$ (left), and a tree representation of an approximated decision function $df_1^2$ (right).

To fairly approximate these decision functions we always have to use union operation over sub-functions, so that no decision information captured is lost.

### 4.4.1 Accuracy of the Approximation from Merging Decision Functions.

It is intuitively clear that, although based on merge, the approximation of a decision function does not converge (in probability) to an accurate decision function. This statement is not a contradiction to Theorem 4.3.1 as, in general, the theorem’s prerequisites are not fulfilled by the merge operations used in approximations (we merge not independently learned decision functions, but sub-functions of the same decision function).

Basically, we merge distributions $d_{\vec{a}} = eval(df, \vec{a})$ and $d_{\vec{a}'} = eval(df, \vec{a}')$ of different actual contexts $\vec{a}, \vec{a}' \in \vec{A}$. That is, if we approximate a decision function $df$ with an attribute $A_i \in \vec{A}$, then we merge all of the sub-functions $df_1 \ldots df_{|\vec{A}|}$ of the decision function $df$ that correspond to a selection operator $x^i$. Thus, the actual context vectors $\vec{a}, \vec{a}'$ differ in position $i$ and take different values $a, a' \in A_i$.

The formal context attribute $A_i$ has an impact on the decision. Then, however, $oracle_\vec{a}$ and $oracle_{\vec{a}'}$ are different classification distributions. Still $d_{\vec{a}}$ and $d_{\vec{a}'}$ could be random samples of, and probably accurate with respect to, the same accurate distribution, either $oracle_\vec{a}$ or $oracle_{\vec{a}'}$. However, it could only happen by chance and must not be assumed in general. Therefore, we expect to lose accuracy by using merge-based approximation operations.
4.5. Parametrized Specification Decision Algebra

4.4.2 Approximation Operation

In this section, we present approximation operation \( \text{approx} \). It approximates the decision function by ignoring one (or more) attribute(s) \( A_i \). The \( \text{approx} \) operation is based on \( \text{merge} \) and \( \text{bind} \) operations:

\[
\text{approx}_{A_i} : DF[\vec{A}, D] \rightarrow DF[\vec{A}', D] \quad \vec{A}' = A_1 \times \cdots \times A_{i-1} \times A_{i+1} \times \cdots \times A_n
\]

\[
i \in [1 \ldots n]
\]

\[
\text{approx}_{A_i}(df) := \sqcup (\text{bind}_{A_i}(df, a_1), \ldots, \text{bind}_{A_i}(df, a_{|A_i|}))
\]

\[
\text{where} \quad a_1, \ldots, a_{A_i} \in A_i
\]

Notice that the \( \text{merge} \) operation \( \sqcup \) for approximation operation always fulfills the criteria defined for scenario one. Thus, it always merges the decision functions with the same formal contexts \( \vec{A}' \), because they are, basically, bounden versions of the same attribute \( A_i \) of the same decision function \( df \).

In fact, the decision function \( df \) can be approximated by ignoring any number of attributes \( A_i \in \vec{A} \). For our approximation, we use the operation \( \sqcup_D \) over \( D(C) \) as defined in the previous Section 4.3. For instance, Figure 4.4 shows an example of the approximation of a decision function given before \( df^2 = x^2(\text{na}, x^2(\text{na}, \text{na}, a, a), x^2(\text{na}, \text{na}, a, a), a) \):

\[
df_1^2 = \text{approx}_{A_1}(df^2)
\]

\[
df_1^2 = x^1(\text{na}, \text{na}, \text{na}, a)
\]

The right-hand side depicts the original decision function \( df^2 \) and the left-hand side depicts an approximated decision function \( df_1^2 \). The sub-terms of the selection operator \( x^2 \) that refers to the attribute \( A_2 \), which are merged by a base-case scenario (since the sub-terms are represented by constant decision functions \( df^0 \)), where the actual distributions are merged (the most probable class is ‘\( \text{na} \)’).

4.5 Parametrized Specification Decision Algebra

Finally, after defining all of the operations over decision functions, we can define a parametrized algebraic specification of Decision Algebra in Table 4.3 using domain \( \vec{A} \) and a co-domain \( D(C) \) as formal parameters. Every instance of Decision Algebra (i.e., a specific type of decision models as presented in Chapter 2.1) can be specified by referencing this generic specification with fitting parameters. The parameters depends on a problem domain (i.e., a specific decision information captured). The parameters can also be extensions of a formal parameters specification (the specification for \( D(C) \).
Chapter 4. Accuracy of Decision Functions

\[ DF(D, \tilde{A}) = D+, \tilde{A}+, INT+, A_1+, \ldots, A_n+ \]

**sort**  
\( DF \)

**ops**  
\( cons : D \rightarrow DF \)
\( bind_A_i : DF \times A_i \rightarrow DF; \)
\( eval : DF \times \tilde{A} \rightarrow D; \)
\( arity : DF \rightarrow INT; \)
\( evert_A_i : DF \rightarrow DF; \)
\( decide : DF \times \tilde{A} \rightarrow C; \)
\( apply : (D \times D \times \cdots \times D) \times DF \times \cdots \times DF \rightarrow DF; \)
\( \equiv : DF \times DF \rightarrow BOOL; \)
\( \sqcup : DF \times DF \rightarrow DF; \)
\( approx_A_i : DF \rightarrow DF; \)

**eqns**  
\( \forall a_1, \ldots, a_n : A_1; b_1, \ldots, b_n : A_i; df_1, df_1', \ldots, df_k: DF; df_1^0, df_2^0 \in DF_0; \)
\( \tilde{a} : \tilde{A}_1 d_1, \ldots, d_k \in D; i : INT, i < |\tilde{A}_1|. \)
\( \text{arity}(\text{df}_i^0) = 0; \)
\( \text{arity}(x^i(\text{df}_1^1, \ldots, \text{df}_{|A_i|})) = 1 + \text{arity}(\text{df}_1^i); \)
\( x^i(\text{df}, \ldots, \text{df}) \equiv \text{df} \)
\( \text{bind}(1, x^i(\text{df}_1^1, \ldots, \text{df}_{|A_i|}), a) = \text{df} \text{ if } a = a_i \in A_1 \)
\( \text{bind}(i, \text{df}, \emptyset) = \text{df}; \)
\( \text{eval}(\text{df}, \tilde{a}) := \text{bind}(n, \ldots, \text{bind}(2, \text{bind}(1, \text{df}, a_1), a_2), \ldots, a_n), \)
\( \text{where } a_1, \ldots, a_n \in \tilde{a}; \)
\( \text{eval}(	ext{cons}(\text{df}), \emptyset) = \text{df}; \)
\( \text{evert}(i, \text{df}) = x^i(\text{bind}(i, \text{df}, b_1), \ldots, \text{bind}(i, \text{df}, b_A_i)); \)
\( \text{evert}(j, \text{df}) = \text{df} \text{ if } j \notin |\tilde{A}_1|; \)
\( \text{evert}(0, \text{df}) = \text{df}; \)
\( \text{evert}(1, x^i(\text{df}_1^1, \ldots, \text{df}_{|A_i|})) = x^i(\text{df}_1^1, \ldots, \text{df}_{|A_i|}); \)
\( \text{decide}(\text{df}, \tilde{a}) = \text{decide}(\text{bind}(1, \text{df}, a_1), \tilde{a}) \text{ where } \tilde{a} = (a_1, \tilde{a}) \)
\( \text{decide}(\text{cons}(\text{df}, \emptyset)) = \text{modet}(\text{df}); \)
\( \text{apply}(g_D, \text{df}_1, \ldots, \text{df}_k) := x^i(\text{apply}(g_D, \text{bind}(1, \text{df}, a_1), \ldots, \text{bind}(1, \text{df}, a_1)), \ldots, \text{apply}(g_D, \text{bind}(1, \text{df}, a_1), \ldots, \text{bind}(1, \text{df}, a_1)); \)
\( \text{apply}(g_D, \text{cons}(d_1), \ldots, \text{cons}(d_k)) = \text{cons}(g_D(d_1), \ldots, d_k)); \)
\( \equiv (\text{df}_1, \text{df}_2) = \text{apply}(\equiv, \text{df}_1, \text{df}_2) \equiv \text{df}_1; \)
\( \equiv (\text{cons}(d_1), \text{cons}(d_2)) = \text{cons}(d_1) \equiv \text{cons}(d_2); \)
\( \sqcup (\text{df}_1, \ldots, \text{df}_k) = \text{apply}(\sqcup, \text{df}_1, \ldots, \text{df}_k); \)
\( \sqcup (\text{df}, \text{df}^f) = x^i(\text{bind}(1, \text{df}, a_1) \sqcup \text{bind}(1, \text{df}, a_1), \ldots, \text{bind}(1, \text{df}, a_1), \ldots, \text{bind}(1, \text{df}, a_1)); \)
\( \sqcup (\text{df}_1^0, \text{df}_2^0) = \text{cons}(\sqcup(\text{eval}(\text{df}_1^0), \text{eval}(\text{df}_2^0))); \)
\( \text{apply}(\sqcup, \text{df}, \text{df}) = \text{df}; \)
\( \approx \text{merge}(\text{bind}(i, \text{df}, b_1), \ldots, \text{bind}(i, \text{df}, b_A_i)); \)
\( \approx \text{apply}(1, x^i(\text{df}_1, \ldots, \text{df}_{|A_i|})) = \text{apply}(\sqcup, \text{df}_1, \ldots, \text{df}_{|A_i|}) \)

**axioms**  
\( \forall df : DF, i : INT \)
\( \text{evert}(i, \text{df}) \equiv \text{df}; \)
\( \text{arity}(\text{evert}(i, \text{df})) = \text{arity}(\text{df}); \)
\( \text{arity}(\text{approx}(i, \text{df})) \leq \text{arity}(\text{df}); \)

Table 4.3: Parameterized Algebraic Specification of Decision Algebra
was introduced in Section 3.1). Note, that the definition \( x^i(d_{f_1}, \ldots, d_{f_n}) \) is not an operation of Decision Algebra, but a term representation as was discussed in Sections 3.1.

The specification of instantiations of DA may specify new symbols or axioms, or even required further properties. The equations provided by DA specification refer to general implementations of the operations discussed in the previous sections.

### 4.6 Summary

In this and previous chapters, we have defined Decision Algebra as a theoretical framework for capturing and manipulating decision information. Decision Algebra defines the notion of decision functions as a formal representation of decision models that can be used in different types of application domains.

Decision Algebra defines a set of operations over decision functions: a core operation \textit{bind}, which binds the decision function according to the attribute value \( a_i \in A_i \) and auxiliary operations, which are defined below.

- \textit{Decide} - decides based on an actual context \( \vec{a} \in \vec{A} \),
- \textit{evert} - changes the order in which attributes occur in a decision function,
- \textit{apply} - applies any function \( g \) to the co-domain of the decision function \( D(C) \),
- \textit{equals} - determines whether two decision functions are equivalent,
- \textit{merge} - combines two decision functions,
- \textit{approx} - approximates a decision function with regard to an attribute \( A_i \).

Abstract operation \textit{merge} defines a general way to combine different decision models. Basically, it can be applied to any decision model to implement a single core operation \textit{bind}. Furthermore, we showed the theoretical outcomes of combining, comparing and approximating decision functions in terms of accuracy. We also showed that combining of a series of probably accurate decision functions tendentially results in more accurate decision functions.

The uniform theory of Decision Algebra can be applied to formalizing different classification approaches. In fact, many classical decision models belonging to DM1 (decision trees, decision tables) and variants thereof can be seen as default Decision Algebra implementations. In addition, other decision models (DM2-DM6) can serve as Decision Algebra instances by varying the implementations of the Decision Algebra core operation (i.e., \textit{bind}). Even though other non-core operations have default implementation in Decision Algebra, a dedicated implementation can be found directly in these specific decision models.

Several decision models (decision trees, decision graphs, decision tables, and Naïve Bayesian classifiers) will be discussed in the next Chapter 5 as Decision Algebra instantiations. Chapter 6 will present the experimental
Chapter 4. Accuracy of Decision Functions

results that validate some of the theoretical implications discussed in this chapter.
Chapter 5

Instantiations of Decision Algebra

In the previous chapter, we defined DA as a parametrized algebraic specification of decision models with attributes domain $A$ and decisions co-domain $D(C)$ as parameters. DA provides a general representation of the decision information as an abstract decision function along with a set of operations. Such generalization requires the decision models to implement the core operation $bind$ of DA with its defined pre- and post-conditions.

In general, each category of decision models DM1 – DM6 (see Chapter 2) is a specific implementation of a decision function and, along with its operations, it forms an instance of DA. The core operation $bind$ has to be implemented in each decision model, introducing the default implementation of non-core operations (e.g., decide, evert, apply, merge, approximate). The default implementation of the other operations doesn’t prevent other efficient implementations of these operations in decision models. Notice that these implementations have to follow the algebraic specifications (e.g., signatures and axioms) defined in the previous chapter.

In this chapter, we introduce instantiations of DA with tree-based decision models (i.e., decision graphs and decision trees), referred to as DGA. Decision functions of DA are, thus, defined as decision graph functions (DGF).

5.1 Decision Graphs

In this section, we introduce an instantiation of DA with tree-based decision models (i.e., decision graphs and decision trees), referred to as DGA. Decision functions of DA are, thus, defined as decision graph functions (DGF). The difference between decision graphs and decision trees models is in the
5.1.1 Decision Trees

Decision trees (DTRs) are widely used and practical decision models for inductive inference [75]. The corresponding data structure implementing this model represents trees that encode context attributes in their inner nodes. Each outgoing edge of such a node refers to a value (or a value range) of the context attribute, and each path from the root node to a leaf represents actual context values leading to a classification result.

Let $df : A_1 \times \ldots \times A_n \rightarrow D(C)$ be an $n$-ary decision function. Every decision term representation of $df$ can be seen as a tree $G = (N, E, r)$. The root node $r \in N$ corresponds to the selection operator $x_1$ of an attribute $A_1$, returning the child $idx_1(a)$ for a given argument $a \in A_1$. The child is a sub-tree representing the $(n - 1)$-ary decision function $bind_{A_1}(df, a)$. The 0-ary decision functions $df^0$ are represented as leaves that capture the distribution $d(c) \in D(C)$.

DTRs are constructed (e.g., by the learning algorithm presented in Section 3.3 that interpolates decision information). The only difference between decision trees construction and decision graph construction is that in the decision tree construction every recursively constructed $df_v$ is immediately added to a node as a child (cf. lines 14–16). While, in graphs, the constructor first checks whether this node was created before. This difference occurs due to the specifics of the implementation of a tree node constructor $x^l(x)$ that constructs a new tree node corresponding to the selection operator $x^l$. As an argument $x$, the constructor takes either a decision distribution $d(C)$ and generates a leaf node, or an integer number $n$ and generates an inner node.
5.1. Decision Graphs

with empty slots for \( n \) children.

5.1.2 Decision Graphs

The main drawback of DTRs is that they grow exponentially with the number of context attributes and even a small decision problem requires relatively large DTRs [50]. The main concept of decision graphs (DGs) is to represent decision information in a compact way by eliminating redundancy. DGs were inspired by the \( \chi \)-term data structure, which captures context-sensitive information in static points-to analysis [107], and by Ordered Binary Decision Diagrams (OBDDs) [12, 11], which capture Boolean functions. In contrast to DGs, OBDDs contain binary decisions instead of multiple decisions due to the multiplicity of the context attributes. Also, decision values \( c \in \mathbb{C} \) are not binary. These more general DGs correspond to the multi-terminal multi-valued decision diagrams [76].

DGs serve as a concrete example of an instance of our DA framework. The implementation of DGA is a straightforward implementation of DA, where decision functions are implemented as rooted acyclic graphs with the inner nodes representing context attributes and the leaves representing decisions (the same as with the trees). The learning algorithm presented in Section 3.3 captures the decision information in DGs similarly as to how it is captured in DTRs. The advantage of DGs over DTRs is the redundancy elimination that leads to a decreased size and, therefore, lower memory consumption.

The co-domain of DGs is represented as a class distribution \( D(\mathbb{C}) \) as discussed above. The core operation for DGA is an operation of getting a child of the node:

\[
\text{child} : DFG(\vec{A}, D(\mathbb{C})) \times A \to DFG(\vec{A}', D(\mathbb{C})) \quad \vec{A}' = A_2 \times \cdots \times A_n
\]

While the rest of the discussed operations are derived:

\[
\text{bind}_{A_i}(dfg, a) := \text{cons}_{DF}(a_1, \text{bind}_{A_i}(\text{child}(dfg, a_1), a), \ldots, a|_{A_i|}, \text{bind}_{A_i}(\text{child}(dfg, a|_{A_i|}), a)) \quad a \in A_i, \quad a_1, \ldots, a|_{A_i|} \in A_i, \quad i \in [2 \ldots n]
\]

\[
\text{bind}_{A_1}(dfg, a) := \text{child}(dfg, a) \quad a \in A_1
\]

5.1.2.1 Implementation Details. Our implementation uses a repository that captures the decision term representations of \( df \). However, the repository guarantees that each term corresponding to a unique decision (sub-)function, reduced by redundancy elimination (cf. Section 3.1.1) is captured only once and gets a unique identifier. The children in the DGs of \( df \) only refer to the unique identifier of the corresponding sub-terms. As a consequence, we never store two equivalent (cf. Section 3.1.1) decision (sub-) functions in
Chapter 5. Instantiations of Decision Algebra

the repository and we never make use of two equivalent sub-terms in the same decision function. Since selection operators $x$ and $x'$ may point to the same children, our terms are represented by rooted directed acyclic graphs (instead of trees).

In order to help recognizing the equivalent sub-terms we maintain a "normal" order of all nodes. If an attribute $A_i$ is chosen for splitting by a classification algorithm before an attribute $A_j$, then $i < j$ and $x' < x'$ and operator $x'$ occurs before operator $x'$ on all paths from the root to the leaves in the corresponding decision graph.

All DA operations discussed in Chapter 3 are implemented in DGs as DGs inherit all decision function properties (e.g., discussed in Section 3.1.1). In what follows, we discuss the implementation of bind, apply and approximation as other operations are implemented in a straightforward manner.

5.1.2.2 Implementation of Bind. In Chapter 3.4, we showed that many DA operations can be defined using the core operation bind. For DGA, bind becomes a derived operation based on the core operation child. Our implementation of bind on DGs is presented in Algorithm 5. Basically, it recursively applies the bind operation over the children of the given graph node and returns a new decision term constructed from the results of the operation applied. Lines 1–6 handle the base cases where the decision terms are leaves (lines 1–3), or the attribute, which we bind to, corresponds to the given decision term (lines 4–6). In the former case, the algorithm returns the leaf. In the latter case, the binding is applied by getting $a$-the child $\text{child}(x^k, a)$ of the current decision term $x^k$, and then the bind is pushed further toward the leaves (since there can exist more than one decision term with the same corresponding attribute in the path due to the properties of the learning algorithm).

Lines 7–9 recursively bind the children of the decision term $x^k$. Note that

```
Algorithm 5 $\text{bind}_{A_i}(x^k : \text{DG}(D), a \in A_i) \rightarrow \text{DG}(D)$

1: if $x^k \in D$ then
2: return $x^k$
3: end if
4: if $k == i$ then
5: return $\text{bind}_{A_i}(\text{child}(x^k, a), a)$
6: end if
7: for each $j \in \{1 \ldots |\text{children}(x^k)|\}$ do
8: $y_j = \text{bind}_{A_i}(\text{children}(x^k)_j, a)$
9: end for
10: return $\text{cons}_D(a_1, y_1, \ldots, a_i A_k, y_{|\text{children}(x^k)|}), \text{ where } a_i \in A_k$
```
5.1. Decision Graphs

the children($x^k$) refers to the sub-terms of $x^k$, and $|\text{children}(x^k)|$ to the number of the sub-terms of $x^k$. Finally, in line 10, a new decision term is created if and only if it has not been created before.

5.1.2.3 Implementation of Evert. An implementation of the $\text{evert}$ operation on DGs is similar to the implementation of the Shannon expansion on OBDDs [11]. Evert was introduced in Section 3.4. Given an attribute $A_i$ and a root node $r$ as an input, the operation reorders nodes in the graph rooted by node $r$ corresponding to attribute $r$ in such a way that attribute $A_i$ represented by node $i$ becomes a new root (cf. Algorithm 6).

The $\text{evert}_{A_i}$ operation recursively binds a current root node $x^r$ to the $a_j$ attribute value of the attribute $A_i$ where $i$ ranges from 1 to $|A_i|$ (cf. lines 4–6).

5.1.2.4 Implementation of Apply. In Sections 3.4 and 4.3, we showed that many DA operations can be defined as special cases of the higher order function $\text{apply}$. Our implementation of $\text{apply}$ on DGs is outlined in Algorithm 7. It pushes a function $g$ to the leaves and applies it on them. Hence, the operations must be defined on the leaf values $D(C)$. The result is a new decision term recursively constructed from the results of the operation applied to the leaves.

Lines 1–3 handle the base case where the decision terms are leaves and the operation is applicable directly. The result is a new basic decision term $df^0 = g(x^k, x^l)$ (if it has not been created before). Lines 4–17 handle the case where one of the decision terms is a leaf. Lines 18–23 handle the case where the root selection operators (i.e., the first selection attributes) are identical. In all three cases (lines 9, 15 and 22), a new decision term is created if and only if it has not been created before. Line 24 handles the case when the selection operators differ and both decision terms are not leaves. In this case, we $\text{evert}$ the second term $x^l$ to $k$. An $\text{apply}$ of $op$ to the result leads to the case where the root selection operators are identical.
5.1.2.5 Implementation of Approximation. In order to save space or avoid overfitting in DGs (DTRs), pruning is applied [114]. Different pruning variants can easily be implemented using the `approx` operation, allowing it to ignore any attribute or even sub-terms of the decision function (defined in Section 4.4). Examples of different pruning implementations can be found in our previous work [21].

Here, we define a simple pruning method, referred to as `k-approx`, that approximates a decision graph $dg$ by all its attributes $A_i$ indexed higher than a certain index $k < i \leq n$. The $k$–`approx` is a repeated merge $\sqcup$ operation starting from the leaves and proceeding to the $k$-index of a decision function $dg$. The prerequisite for this operation is that a decision graph should be
5.1. Decision Graphs

Figure 5.2: Approximation and k-approximation of \(df^2\)

ordered. This operation can be formalized as follows:

\[
k\text{-}\text{approx}(k) : (A_1 \rightarrow \ldots \rightarrow A_k \rightarrow \ldots \rightarrow A_n \rightarrow D(C)) \rightarrow (A_1 \rightarrow \ldots \rightarrow A_k \rightarrow D(C))
\]

\[
k\text{-}\text{approx}(k, \text{cons}_{DF}(a_1, d_{g_1}, \ldots, a_{\lvert A_1 \rvert}, d_{g_{\lvert A_1 \rvert}})) = \text{cons}_{DF}(a_1, \text{approx}(k-1, d_{g_1}), \\
\ldots, \\
\qquad a_{\lvert A_1 \rvert}, \text{approx}(k-1, d_{g_{\lvert A_1 \rvert}}))
\]

\[
k\text{-}\text{approx}(0, \text{cons}_{DF}(a_1, d_{g_1}, \ldots, a_{\lvert A_1 \rvert}, d_{g_{\lvert A_1 \rvert}})) = \text{apply}(\sqcup_D, \text{cons}_{DF}(a_1, \text{approx}(0, d_{g_1}), \\
\ldots, \\
\qquad a_{\lvert A_1 \rvert}, \text{approx}(0, d_{g_{\lvert A_1 \rvert}}))
\]

where the k-approximated decision function \(df^a\) becomes a new decision function \(df^{a-k}\) (i.e., the arity of the decision function decreases by \(k\)). The base-case for \(k\text{-}\text{approx}\) with \(d_{g_0} \in DF_0[0, D]\) is defined as:

\[
k\text{-}\text{approx}(0, d_{g_0}) = d_{g_0}
\]

Figure 5.2 shows an example of approximation (b) and \(k\text{-}\text{approx}\) (a, b) of the decision graph \(df^2 = x^1(\text{na}, x^2(\text{na}, \text{na}, \text{a}, \text{a}), x^2(\text{na}, \text{na}, \text{a}, \text{a}, \text{a}), \text{a})\):

\[
\text{approx}_{A_2}(df^2) = x^1(\text{na}, \text{na}, \text{na}, \text{a}) \\
k\text{-}\text{approx}(2, df^2) = x^1(\text{na}, x^2(\text{na}, \text{na}, \text{a}, \text{a}), x^2(\text{na}, \text{na}, \text{a}, \text{a}, \text{a}), \text{a}) \\
k\text{-}\text{approx}(1, df^2) = x^1(\text{na}, \text{na}, \text{na}, \text{a})
\]

Notice that, in this particular case, the result of \(k\text{-}\text{approx}(1, df^2) = \text{approx}(2, df^2)\). This \(k\text{-}\text{approx}\) will also be used in our experiments in Chapter 5.
Chapter 5. Instantiations of Decision Algebra

Algorithm 8 $k$-approx($k \in \mathbb{N}, x^n \in DG(D)) \rightarrow DG(D)$

1: if $k = 0$ then
2: return new collapse($x^n$)
3: end if
4: for each $i \in [1 \ldots |children(x^n)|]$ do
5: $y_i = k$-approx($k - 1, children(x^n)_i$)
6: end for
7: return new cons$_D$(a$_1$, y$_1$, \ldots, a$_{|A_n|}$, y$_{|children(x^n)|}$), where $a_i \in A_n$

Algorithm 9 collapse($x^n \in DG(D)) \rightarrow D$

1: if $x^n \in D$ then
2: return $x^n$
3: end if
4: $v = \perp$
5: for each $i \in [1 \ldots |children(x^n)|]$ do
6: $v = \sqcup(v, collapse(children(x^n)_i))$
7: end for
8: return new $v$

The implementation of $k$-approx is shown in Algorithm 8. The $k$-approximation of the decision functions is easy to understand based on the graphs representations: all of the sub-graphs of depth $\geq k$ from the root are replaced with the merger $\sqcup$ of their leaves (recall from the standard graphs representation that every leaf keeps the distribution for each class). Therefore, the result is a new decision graph with depth $\leq k$. Algorithm 8 finds the sub-graphs at depth $k$, while Algorithm 9 merges their respective leaves.

After defining the operations of the DGF, we can specify its parameterized algebraic specification as follows in Table 5.1. It is worth mentioning that some operations defined in DGA can be also be applied to DA (e.g. $k$-approx). This property is one of the advantages of DA, when operations defined in one decision model can be transferred to other decision models through DA interface.

5.1.3 Non-Functional Properties

In the worst case scenario, the amount of memory required for capturing DG is quite large and also grows exponentially with the number of used attributes $n$. The decision information in this case has $k = |A_1| \times \ldots \times |A_n|$ decision tuples and the corresponding DG has $L = k$ leaves and (almost) $k$ inner nodes. Thus, the number of edges is $O(L) = O(|A_1| \times \ldots \times |A_n|)$. This size is reduced when the learning strategy interpolates the decisions and,
5.1. Decision Graphs

\[ DGF(\vec{A}, D) = DF[\vec{A}, D]^+, Array^+ \]

<table>
<thead>
<tr>
<th>sort</th>
<th>DFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>ops</td>
<td>( \text{child}: DFG \times A_i \rightarrow DFG; )</td>
</tr>
<tr>
<td></td>
<td>( \text{children}: DFG \rightarrow Array[DFG]; )</td>
</tr>
<tr>
<td></td>
<td>( \text{depth}: DFG \times \text{INT}; )</td>
</tr>
<tr>
<td></td>
<td>( k\text{-approx}: \text{INT} \times DFG \rightarrow DFG; )</td>
</tr>
<tr>
<td>eqns</td>
<td>( \forall a_1, \ldots, a_n: A_1; b_1, \ldots, b_n: A_i; dfg, dfg_1, \ldots, dfg_n: DFG; )</td>
</tr>
<tr>
<td></td>
<td>( d_1, \ldots, d_k \in D; i \in \text{INT}, i &lt;</td>
</tr>
<tr>
<td></td>
<td>( \text{child}(\text{cons}<em>{DF}(a_1, dfg_1, \ldots, a</em>{</td>
</tr>
<tr>
<td></td>
<td>( \text{children}(\text{cons}<em>{DF}(a_1, dfg_1, \ldots, a</em>{</td>
</tr>
<tr>
<td></td>
<td>( \text{bind}<em>i(dfg, a) = \text{cons}</em>{DF}(a_1 \text{bind}<em>a_i(\text{child}(dfg, a_1), a_i), \ldots, a</em>{</td>
</tr>
<tr>
<td></td>
<td>( \text{where } a \in A_i, i \in [2 \ldots n]; )</td>
</tr>
<tr>
<td></td>
<td>( \text{bind}_a_i(dfg, a) = \text{child}(dfg, a) ) where ( a \in A_1; )</td>
</tr>
<tr>
<td></td>
<td>( k\text{-approx}(i, dfg) = \text{cons}<em>{DF}(a_1, k\text{-approx}(i - 1, dfg_1), \ldots, a</em>{</td>
</tr>
<tr>
<td></td>
<td>( k\text{-approx}(i, dfg_0) = dfg_0 );</td>
</tr>
<tr>
<td></td>
<td>( k\text{-approx}(0, \text{cons}<em>{DF}(a_1, dfg_1, \ldots, a</em>{</td>
</tr>
<tr>
<td></td>
<td>( \text{depth}(\text{cons}<em>{DF}(b_1, d, \ldots, b</em>{</td>
</tr>
<tr>
<td></td>
<td>( \text{depth}(\text{cons}<em>{DF}(a_1, dfg_1, \ldots, a</em>{</td>
</tr>
<tr>
<td>axioms</td>
<td>( \forall dfg: DFG, i \in \text{INT} )</td>
</tr>
<tr>
<td></td>
<td>( \text{arity}(dfg) \leq \text{depth}(dfg) )</td>
</tr>
<tr>
<td></td>
<td>( \text{arity}(k\text{-approx}(k, dfg^n)) = n - k )</td>
</tr>
</tbody>
</table>

Table 5.1: Parameterized Algebraic Specification of Decision Graph Algebra
therefore, the number of edges becomes smaller. It can also be reduced if all paths from an inner node lead to the same decision (making this whole sub-tree redundant). Hence, the memory consumption is

\[ M = \text{size} \times \text{edges}, \]

where \( \text{edges} \) is the number of edges in the graph and \( \text{size} \) bytes are necessary and sufficient to encode all of the different nodes of the graphs. In addition, we expect that, due to the elimination of redundancies, the size of the DGs is expected to be considerably smaller in practice.

Capturing context \( \vec{a} = (a_1, \ldots, a_n) \) in DG is straightforward: in each inner node corresponding to an attribute \( A_i \) starting at the root node, the attribute value \( a_i \in A_i \) is used to decide which child to visit next. The number of tests necessary to reach a leaf is equal to the depth of the DG. This depth varies around the number \( n \) of the context attributes. For discrete context attributes, it is, at most \( n \). Continuous attributes can even occur several times on the path due to data partitioning [91]. So, generally, the decision time is \( \text{depth} \times T_{aa} \) and we approximate

\[ T(n) \approx n \times T_{aa} + c. \]

where \( T_{aa} \) is the array access time and \( c \) is a constant time used for small operations.

### 5.2 Decision Tables

A decision table (DTB) is a standard decision model that can be used in all application domains. It is a table-based representation of a decision function, where decision and formal context are represented by table columns sorted in a special order (usually: formal context, decision) and the actual context and decision are items in the table. Table 5.2 (a) presents the decision function \( df^2 = df^2 = x^1(\text{na}, x^2(\text{na}, \text{na}, a, a), x^2(\text{na}, \text{na}, a, a), a) \) as a decision table. Basically, DTB \( DT[D, \vec{A}] \) is a table of decision tuples \((\vec{a}, c)\), where \( \vec{a} \) is an actual context (i.e., a vector of attribute values \((a_1, \ldots, a_n)\) with \( a_i \in A_i \)) that leads to a certain decision \( d(c) \in D(C) \) (usually the co-domain of DTB is a class \( C \), so it does not capture any distributions or probabilities). Each attribute \( A_i \in \vec{A} \) and a formal decision \( D(C) \) correspond to a table column. Thus, each attribute value \( a_i \in A_i \) and decision \( d(c) \in D(C) \) are entries in the table. When an attribute \( A_i \in \vec{A} \) is of a continuous type, the DTB captures the variants at the sample points. The constructor of a DTB \( dt^n \) takes decision tuples \((\vec{a}, c)^k\), with \( k \) being a number of tuples in the decision information, sorts them according
5.2. Decision Tables

Table 5.2: Example of (a) a decision table of $df^2$, (b) a decision table of $bind_{A_2}(df^2, vhigh)$ and (c) a decision table of $approx_{A_2}(df^2, vhigh)$
Chapter 5. Instantiations of Decision Algebra

to the order of the attributes specified in a DTB, and inserts the into the table:

\[ dt^n : (A_1 \times \ldots \times A_n \times D(C)) \rightarrow DT[D(C), \vec{A}] \]

k-decision tuples

Even though distributions \( d(c) \in D(C) \) of decisions \( c_i \in C \) in a DT are often neglected, it still can be computed if the construction of the DTB uses the simple algorithm presented in Section chapter 4: simpleLA.

The core operation of DT is \( \text{bind} \) (the same as in DA). It is implemented as follows:

\[
\text{bind}_{A_i}(dt^n((a_1, \ldots, a_n, c_1)^1, \ldots, (a_1, \ldots, a_n, c_k)^k), a) = dt^{n-1}((a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c_1)^1, \ldots, (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c_k)^k)
\]

where \( a_i = a \land a \in A_i \). The \( \text{bind}_{A_i}(dt^n, a) \) returns a new decision table \( dt^{n-1} \) decreased by one column of the attribute \( A_i \) and with tuples where the attribute value \( a_i \in A_i \) is equal to \( a \). For instance, Table 5.2 (b) presents the decision function \( \text{bind}_{A_2}(df^2, \text{vhigh}) \).

Most of the DTB derived operations (i.e., decide, approx, equals and merge) use the implementation given in DA. However, DTB has a different structure than the default decision functions representation, therefore, it has an alternative implementation of \( \text{evert} \) and \( \text{apply} \).

5.2.1 Evert and Apply Operations of Decision Tables

The \( \text{evert} \) operation simply changes the order of the columns in DT by putting the i-th attributes first in the constructor:

\[
\text{evert}_{A_i}(dt^n((a_1, \ldots, a_n, c_1)^1, \ldots, (a_1, \ldots, a_n, c_k)^k)) = dt^n(a_i, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c_1)^1, \ldots, (a_i, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, c_k)^k)
\]

In order to \( \text{apply} \) a function \( g \) on a set of \( m \) decision tables \( dt_1, \ldots, dt_m \in DT[D, \vec{A}] \), this function has to be defined over the co-domain of DTBs (either \( D(C) \) or \( C \)). The prerequisite for this alternative implementation is that DTBs have the same domain \( \vec{A} \):

\[
\text{apply}(g, dt_1, \ldots, dt_m) = dt^n(\vec{a}_1, \text{apply}(g, \text{eval}(dt_1, \vec{a}_1)), \ldots, \text{eval}(dt_m, \vec{a}_1)), \ldots, \text{eval}(dt^n, \vec{a}) = \text{bind}(\ldots \text{bind}(\text{bind}(dt, a_1), a_2), \ldots, a_n)
\]

where \( \vec{a}_1, \ldots, \vec{a}_n \in \vec{A} \).
5.2. Decision Tables

\[ DFT(\vec{A}, D) = DF[\vec{A}, D] + \]

sort \text{DFT}

\[ dt: (A_1 \times \cdots \times A_n \times D \times \cdots \times A_1 \times \cdots \times A_n \times D) \rightarrow DT; \]

children: DFG \rightarrow \text{Array}[DFG];

\[ \text{depth} : \text{DFG} \times \text{INT}; \]

\[ k-\text{approx} : \text{INT} \times \text{DFG} \rightarrow \text{DFG}; \]

eqns \forall a_1, \ldots, a_n; A_i; a \in A_i; dtf_1, \ldots, dtf_m; DT; d_1, \ldots, d_k \in D; \vec{a}_1, \ldots, \vec{a}_k \in \vec{A}; \vec{b}_1, \ldots, \vec{b}_k \in \vec{A}

\[ \text{bind}_{A_i}(dt(a_1, \ldots, a_n, d_1, \ldots, d_1, \ldots, a_n, d_k)), a) = \]

\[ = dt(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, d_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, d_k); \]

\[ \text{apply}(g, dtf_1, \ldots, dtf_m) = dt(\vec{a}_1, \text{apply}(g, \text{eval}(dtf_1, \vec{a}_1), \ldots, \text{eval}(dtf_m, \vec{a}_1)), \ldots, \vec{a}_k, \text{apply}(g, \text{eval}(dtf_1, \vec{a}_k), \ldots, \text{eval}(dtf_m, \vec{a}_k))); \]

\[ \sqcup(dt_1(\vec{a}_1, d_1, \ldots, \vec{a}_k, d_k), dt_1(\vec{b}_1, d_1, \ldots, \vec{b}_k, d_k)) = \]

\[ = dt(\vec{a}_1, d_1, \ldots, \vec{a}_k, d_k, \vec{b}_1, d_1, \ldots, \vec{b}_k, d_k); \]

end

Table 5.3: Parameterized Algebraic Specification of Decision Table

An example of the approximation \text{approx} operation as defined on the \text{apply} operation is given in Figure 5.2 (c), where the decision table representation of \text{approx}_{A_2}(df)_2^2$ is presented. Notice that a merge operation \( \sqcup_D \) over the co-domain C should be defined.

In addition, a \text{merge} operation over two DTBs can be implemented in a simpler way by transferring decision tuples \((\vec{a}, c)\) from one DTB to the other (non-redundancy property still works).

After defining all of the operations over DTBs, we can now specify a parameterized algebraic specification of \( DT[D, \vec{A}] \) as an instantiation of \( DF[\vec{A}, D] \), shown in Table 5.3.

5.2.2 Non-Functional Properties

DTBs are implemented as \( n \)-dimensional arrays, \( n \) being the number of context attributes. Each dimension \( i \) contains entries corresponding to the sample values of the context attribute \( A_i \). Thus, the memory consumption \( M \) of the DT grows exponentially with the number of context attributes and can be approximated from below by

\[ M = \text{size} \times m^n, \]

where the \text{size} bytes are necessary to encode all of the variants in \( C \) and \( m \) is the minimum number of values of any of the context attributes.

\text{Decide} corresponds to a lookup in the table based on the indices of input context attributes, where an index for a continuous attribute has to be
computed with a logarithmic distance. Additionally, each access to an \( n \)-dimensional array is basically an access to a 1-dimensional array requiring some offset calculations:

\[
\text{offset} = \text{base	extunderscore address} + (d_1 \times |A_2| + d_2 \times \ldots + |A_{n-1}| + d_{n-1}) \times |A_n| + d_n) \times \text{size},
\]

where \( \text{base	extunderscore address} \) is the address of the first element of the array, \( d_i \) is the index of the element in the dimension \( i \) and \( |A_i| \) is the number of the values of the context attribute \( A_i \). Therefore, the decision time for an \( n \)-dimensional DT can be estimated as

\[
T(n) = (log \times k + n) \times T_{\text{flops}} + (n - k + 1) \times T_{\text{aa}} + c,
\]

where \( k \) is a number of continuous attributes, \( \text{log} \) is a number of the floating point operations used to calculate the logarithm\(^1\), \( T_{\text{flops}} \) is the time for a flop, \( T_{\text{aa}} \) is the array access time and \( c \) is a constant time used for small operations.

### 5.3 Naïve Bayesian Classifier

As an example of how DA can be instantiated with decision models other than tree-based decision models, we discuss Naïve Bayesian classifiers. Naïve Bayes (NB) is a simple probabilistic classifier based on Bayesian statistics [54]. It naïvely assumes a conditional independence of the attributes from each other (i.e., the influence of an attribute \( A_i \) on a given class \( c \in C \) is independent of other attributes [75]).

NB calculates the conditional probabilities (i.e., the probability of an actual class \( c \in C \) given an actual context \( \vec{a} \in \vec{A} \)):

\[
\forall \vec{a} \in \vec{A}, c \in C : P(C = c|\vec{A} = \vec{a}),
\]

Class \( c \in C \) is considered the most probable class for an actual context \( \vec{a} = (a_1, \ldots, a_n) \) iff:

\[
P(C = c) \times P(A_1 = a_1|C = c) \times \ldots \times P(A_n = a_n|C = c)
\]

is maximum among all classes in \( C \) [75, 54]. Therefore, the constructor \( nb^n \) of NB needs to capture the absolute probability distribution of each actual class \( \forall c \in C : D(C) \) and the probability distribution functions \( PD_i = PD(A_i|C = c_j) \) for each attribute \( A_i \in \vec{A} = A_1 \times \ldots \times A_n \) given the class \( c_j \in C = \{c_1, \ldots, c_k\} \):

\[
nb^n : D(C) \times PD_1^1 \times \ldots \times PD_n^1 \times \ldots \times PD_1^k \times \ldots \times PD_n^k \rightarrow NB[\vec{A}, D].
\]

\(^1\)Many processors provide the integer \( \text{log}2 \) in a single instruction in the hardware. In our Java implementation, we need 21 flops.
5.3. Naïve Bayesian Classifier

where \( n = |\vec{A}| \) and \( k = |C| \). Each \( PD_i \) is a discrete probability distribution over attribute \( A_i \). If it cannot be approximated with a parametric distribution, then the constructor of such a discrete probability distribution takes \( l = |A_i| \) pairs \((a, p)\) of probabilities \( p \in P \) and corresponding actual attributes \( a \in A_i \), and returns a distribution \( d(A_i) \in D(A_i) \):

\[
\text{cons}_D : (A_i \times P)^l \to D(A_i)
\]

where \((a, p) \in d(A_i)\) denotes that attribute value \( a \) is supported with a probability of \( p \).

As an example, we take a data set "Car Evaluation" that was introduced in Section 2.2 and construct a Naïve Bayesian classifier. First, we calculate the absolute probability distribution \( D(C) \) for each class of "Car acceptability" ("not acceptable", "acceptable", "good" and "very good") and then a set of discrete probability distributions \( PD_i \) over each attribute \( A_i \in \{A_1, A_2, A_3, A_4, A_5, A_6\} \) for each class \( c_j \in C \). The result, presented in Table 5.4, forms a new decision function \( dfnb^6 \) represented as NB. Alternatively, \( PD_i \) can be constructed from a parametric distribution (e.g., Normal distribution) with the respective parameters (e.g., \( \mu \) and \( \sigma \)). The corresponding operation \( prob \) is defined as:

\[
\text{prob} : D(A_i) \times A_i \to P
\]

\[
\text{prob}(d(A_i), a) = p \quad \text{iff} \quad (a, p) \in d(A_i).
\]

Thus, an algebraic specification of a Naïve Bayesian co-domain \( PD \) should be defined as \( PD(A) = \{D(A), A^+\} \) (i.e., it is based on a PAS of \( D(C) \) specified in the beginning of Section 3.1).

The core operations of NB can simply be defined based on the information captured by the constructor: getting the class distribution \( D(C) \) and getting the (conditional) probabilities \( D(A_i) \) of the attribute \( A_i \) given a class \( c \in C \), respectively:

\[
\text{dist}:\text{NB}[\vec{A}, D] \to PD(C)
\]

\[
\text{pd}_{A_i}:\text{NB}[\vec{A}, D] \times C \to PD(A_i)
\]

Based on these two operations, the \textit{bind} operation can be implemented. An \( n \)-ary decision function \( df^n \in \text{NB}[\vec{A}, D] \) bound to \( a \in A_i \) returns a new \((n - 1)\)-ary decision function \( df^{n-1} \in \text{NB}[\vec{A}, D] \), with \( \vec{A}' = A_1 \times \cdots \times A_{i-1} \times A_{i+1} \times \cdots \times A_n \). For each class \( c \), the \textit{bind} operation multiplies the class probability with the probability of \( a \) in the conditional probability \( PD_i \) given \( c \), this gives a new class distribution \( D(C) \).
Chapter 5. Instantiations of Decision Algebra

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$a \in A_i$</th>
<th>$PD_i^1$</th>
<th>$PD_i^2$</th>
<th>$PD_i^3$</th>
<th>$PD_i^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;very high&quot;</td>
<td>0.31</td>
<td>0.18</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$A_1$</td>
<td>&quot;high&quot;</td>
<td>0.28</td>
<td>0.32</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>&quot;medium&quot;</td>
<td>0.21</td>
<td>0.28</td>
<td>0.38</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>&quot;low&quot;</td>
<td>0.2</td>
<td>0.23</td>
<td>0.62</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>&quot;very high&quot;</td>
<td>0.31</td>
<td>0.14</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$A_2$</td>
<td>&quot;high&quot;</td>
<td>0.27</td>
<td>0.32</td>
<td>0.0</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>&quot;medium&quot;</td>
<td>0.2</td>
<td>0.26</td>
<td>0.31</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>&quot;low&quot;</td>
<td>0.22</td>
<td>0.29</td>
<td>0.69</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>&quot;2&quot;</td>
<td>0.26</td>
<td>0.19</td>
<td>0.19</td>
<td>0.2</td>
</tr>
<tr>
<td>$A_3$</td>
<td>&quot;3&quot;</td>
<td>0.26</td>
<td>0.19</td>
<td>0.19</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>&quot;4&quot;</td>
<td>0.23</td>
<td>0.28</td>
<td>0.31</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>&quot;5, more&quot;</td>
<td>0.25</td>
<td>0.31</td>
<td>0.19</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>&quot;2&quot;</td>
<td>0.46</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$A_4$</td>
<td>&quot;4&quot;</td>
<td>0.24</td>
<td>0.53</td>
<td>0.5</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>&quot;more&quot;</td>
<td>0.29</td>
<td>0.47</td>
<td>0.5</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>&quot;small&quot;</td>
<td>0.39</td>
<td>0.27</td>
<td>0.31</td>
<td>0.0</td>
</tr>
<tr>
<td>$A_5$</td>
<td>&quot;medium&quot;</td>
<td>0.34</td>
<td>0.35</td>
<td>0.5</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>&quot;big&quot;</td>
<td>0.27</td>
<td>0.39</td>
<td>0.19</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>&quot;low&quot;</td>
<td>0.42</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$A_6$</td>
<td>&quot;medium&quot;</td>
<td>0.34</td>
<td>0.43</td>
<td>0.56</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>&quot;high&quot;</td>
<td>0.24</td>
<td>0.57</td>
<td>0.44</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>$D(C)$</td>
<td>0.69</td>
<td>0.23</td>
<td>0.04</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 5.4: Naïve Bayesian classifier for "Car Evaluation" data set
5.3. Naïve Bayesian Classifier

For a decision function \( df \in NB[\vec{A}, D] \) and an attribute value \( a \in A_i \) let \( \text{prob}_{c,A_i}(df, a) = \text{prob}(\text{dist}(df), c) \cdot \text{prob}(pd_{A_i}(df, c), a) \).

\[
\begin{align*}
\text{bind}_{A_i} : NB[\vec{A}, D] \times A_i &\rightarrow NB[\vec{A}', D] \\
\text{bind}_{A_i}(df^a, a) &= nb^{a-1}(\text{cons}_D((c_1, \text{prob}_{c_1,A_i}(df^a, a)), \ldots (c_n, \text{prob}_{c_n,A_i}(df^a, a))), \\
pd_1(df^a, c_1), \ldots, pd_{i-1}(df^a, c_i), pd_{i+1}(df^a, c_1), \ldots, pd_n(df^a, c_1), \\
\ldots \\
pd_1(df^a, c_k), \ldots, pd_{i-1}(df^a, c_k), pd_{i+1}(df^a, c_k), \ldots, pd_n(df^a, c_k))
\end{align*}
\]

For instance, let us calculate \( \text{bind}_{A_1}(dfnb^b, "low") \):

\[
\text{bind}_{A_1}(dfnb^b, "low") = nb^5(\text{cons}_D((c_1, \text{prob}_{c_1,A_1}(dfnb^b, "low")),
\text{prob}_{c_2,A_1}(dfnb^b, "low"),
\text{prob}_{c_3,A_1}(dfnb^b, "low"),
\text{prob}_{c_4,A_1}(dfnb^b, "low"),
 pd_2(dfnb^b, c_1), \ldots, pd_6(dfnb^b, c_1), \\
\ldots \\
pd_1(dfnb^b, c_4), \ldots, pd_6(dfnb^b, c_4))
\]

\[
\text{prob}_{c_1,A_1}(dfnb^b, "low") = 0.69 \cdot 0.2 \\
\text{prob}_{c_2,A_1}(dfnb^b, "low") = 0.23 \cdot 0.23 \\
\text{prob}_{c_3,A_1}(dfnb^b, "low") = 0.04 \cdot 0.62 \\
\text{prob}_{c_4,A_1}(dfnb^b, "low") = 0.03 \cdot 0.4
\]

The result of this \( \text{bind} \) is a modified Naïve Bayesian classifier with an updated \( D(C) = (0.14, 0.05, 0.02, 0.01) \) and that excludes attribute \( PD^j_1 \) for attribute \( A_1 \).

The operations \( \text{evert}, \text{decide}, \text{apply} \) and \( \text{approx} \) are derived and, hence, defined based on \( \text{bind} \). However, it is more efficient to redefine some of them as core operations of NB. The \( \text{evert} \) operation simply changes the order of the PDs of the attribute values, while putting the set of the \( i \)-th attribute first in the constructor:

\[
\text{evert}_{A_i} : DF[\vec{A}, D] \rightarrow DF[\vec{A}', D]  \quad \vec{A}' = A_i \times A_1 \times \ldots \times A_{i-1} \times A_{i+1} \times \ldots \times A_n \\
\text{evert}_{A_i}(dfnb) = nb(\text{dist}(dfnb), pd_{A_i}(dfnb, c_1), \text{pd}_{A_i}(dfnb, c_1), \ldots, pd_{A_{i-1}}(dfnb, c_1), \\
pd_{i+1}(dfnb, c_1), \ldots, pd_n(dfnb, c_1), \\
\ldots \\
pd_{A_i}(dfnb, c_k), \text{pd}_{A_i}(dfnb, c_k), \ldots, \text{pd}_{A_{i-1}}(dfnb, c_k), \\
pd_{A_{i+1}}(dfnb, c_k), \ldots, pd(n, dfnb, c_k))
\]

79
Chapter 5. Instantiations of Decision Algebra

The approx operation constructs a decision function that ignores the attribute $A_i$ by simply "forgetting" the PDs of this attribute:

$$\text{approx}_{A_i} : DF[\vec{A}, D] \rightarrow DF[\vec{A}', D] \quad \vec{A}' = A_1 \times \cdots \times A_{i-1} \times A_{i+1} \times \cdots \times A_n$$

$$i \in [1 \ldots n]$$

$$\text{approx}_{A_i}(dfnb) := \text{nb}(\text{dist}(dfnb), \text{pd}_{A_1}(dfnb, c_1), \ldots, \text{pd}_{A_i}(dfnb, c_1),$$

$$\text{pd}_{A_{i+1}}(dfnb, c_1), \ldots, \text{pd}_n(dfnb, c_1),$$

$$\ldots$$

$$\text{pd}_{A_1}(dfnb, c_k), \ldots, \text{pd}_{A_i}(dfnb, c_k),$$

$$\text{pd}_{i+1}(dfnb, c_k), \ldots, \text{pd}_n(dfnb, c_k))$$

The decide on abstract level would bind one attribute after the other until the result is a constant decision function (with one class distribution) and select the most frequent class of this distribution. This implementation is almost identical to a direct implementation of decide on NB:

$$\text{decide} : DF[\vec{A}, D] \times \vec{A} \rightarrow C$$

$$\text{decide}(dfnb, \vec{a}) := \arg \max_{c \in C} P(C = c) \times P(A_1 = a_1 | C = c) \times \cdots \times P(A_n = a_n | C = c)$$

For instance, let us compute $\text{decide}(dfnb^6, \vec{a})$, where $\vec{a} = \{1 : \text{low}; 2 : \text{low}; 3 : \text{5, more}; 4 : \text{more}; 5 : \text{big}; 6 : \text{high}\}$:

$$\text{prob}(\text{dist}(dfnb^0), c) = \text{prob}(\text{pd}_1(dfnb^6, c), \text{low}) \cdot \text{prob}(\text{pd}_2(dfnb^6, c), \text{low})$$

$$\cdot \text{prob}(\text{pd}_3(dfnb^6, c), \text{5more}) \cdot \text{prob}(\text{pd}_4(dfnb^6, c), \text{more})$$

$$\cdot \text{prob}(\text{pd}_5(dfnb^6, c), \text{big}) \cdot \text{prob}(\text{pd}_6(dfnb^6, c), \text{high})$$

$$\text{dist}(dfnb^0) = \{(na, 0.0001), (a, 0.00049),$$

$$(g, 0.0001), (vg, 0.00045)\}$$

$$\text{decide}(dfnb^6, \vec{a}) = \text{acceptable}$$

In order to apply a function $g$ on a set of $k$ DFNBS, this function has to be defined over $D(C)$. The default definition of apply as a derived operation (shown in Section 3.4.2). In addition, the merge operation over NBs $df \in \text{NB}[\vec{A}, D], df' \in \text{NB}[\vec{A}', D]$ can be defined at the abstract level based on bind (as defined in Section 4.3). This implementation means that the two NBs have to be unfolded to a default representation of decision functions: DGs. However, merge over $df, df'$ can also be defined directly as another core operation.

Let $A_1, \ldots, A_i$ be the exclusive attributes of $\vec{A}$ (not shared by $\vec{A}'$) and let $A'_1, \ldots, A'_j$ be the exclusive attributes of $\vec{A}'$ (not shared by $\vec{A}$) and $A''_1, \ldots, A''_k$ the common attributes of both $\vec{A}$ and $\vec{A}'$. Finally, $\vec{A}'' = A_1 \times \cdots \times A_i \times A'_1 \times$
5.3. Naïve Bayesian Classifier

\[ \ldots \times A'_j \times A''_1 \times \ldots \times A''_l \] and \( n = i + j + l \). Then

\[
\therefore_{NB} : NB[\vec{A}, D] \times NB[\vec{A}', D]
\rightarrow NB[\vec{A}'', D]
\]

\[
\therefore_{NB}(df, df') := nb^b(\therefore_D(dist(df), dist(df')),
\]

\[
pd_{A_1}(df, c_1), \ldots, pd_{A_i}(df, c_1), pd_{A_i}(df', c_1), \ldots,
\]

\[
pd_{A_i}(df, c_k), \ldots, pd_{A_i}(df, c_k), pd_{A_i}(df', c_k), \ldots,
\]

\[
\therefore_D(pd_{A_i}(df, c_k), pd_{A_i}(df', c_k), \ldots, \therefore_D(pd_{A_i}(df, c_k), pd_{A_i}(df', c_k))
\]

where the merge operator \( \therefore_D \) is applied to the class distributions \( dist(df), dist(df') \) and to the conditional distributions \( pd_{A_i}(df), pd_{A_i}(df') \) of the common attributes.\(^2\) The examples of the merge operation will be shown in the next Section 5.4.

The decide operation can be implemented based on the implementation of the class operation in \( D(C) \) or in a more straightforward way:

\[
decide : DF[\vec{A}, D] \times \vec{A} \rightarrow C
\]

\[
decide(df_{nb}, \vec{a}) := \arg \max_{c \in C} P(C = c) \times P(A_1 = a_1|C = c) \times \ldots \times P(A_n = a_n|C = c)
\]

The PAS of the NB can be specified as shown in Table 5.5.

5.3.1 Non-Functional Properties

NBs use the decision information and compute all of the possible probabilities for each attribute vector \( \vec{a}_i \) over all class values \( C \). As a result, a NB is just a set of probabilities that are accessed during classification for computing the most probable variant. For a discrete attribute \( A_i \) the probability is stored in an array with \( |C| \times |A_i| \) elements, where \( C \) is a set of all possible class values. For a continuous attribute, a mean and a variance are computed and stored in two arrays of size \( |C| \). So, the memory consumption is

\[
M = |C| \times size \times (2k + 1 + (n - k) \sum_{i=1}^{(n-k)} |A_i|),
\]

with \( n \) being the number of all of the attributes and \( k \) being the number of continuous attributes.

\(^2\)The result does not yield probability distributions, but it is decision equivalent if the data sets of \( df \) and \( df' \) are of the same size.
\[ NB[\tilde{A}, D] = DF[\tilde{A}, D] \]

**opns**

- \( nb^c : D_{\text{times}}(PD \times \cdots \times PD)^{\text{km}} \to NB; \)
- \( \text{dist} : NB \to D; \)
- \( pd_{A_i} : NB \times C \to PD(A_i); \)
- \( \text{prob}_{C,A_i} : NB \times A \to \mathbb{P}; \)

**eqns**

- \( \forall a \in A_i; nb_f, nb_f' \in NB[\tilde{A}, D]; c, c_1, \ldots, c_k \in C; a \in A_i; p_1, \ldots, p_k \in P; \)
- \( pd^1, \ldots, pd^m, pd_{A_i}^1, \ldots, pd_{A_i}^m \in D; i \in \text{INT}; \)
- \( nb_f = nb^c(\text{cons}_D((c_1, p_1), \ldots, (c_k, p_k)), pd^1, \ldots, pd^m, p_1^k, \ldots, p_k^m); \)
- \( nb_f' = nb^c(\text{cons}_D((c_1, p_1), \ldots, (c_k, p_k)), pd^1, \ldots, pd^m, p_1^k, \ldots, p_k^m); \)
- \( \text{prob}_{A_i}(nb_f, a) = \text{prob}(\text{dist}(nb_f), c) \cdot \text{prob}(pd_{A_i}(nb_f, c), a); \)
- \( \text{bind}_{A_i}(nb_f, a) = nb^{m-1}(\text{cons}_D((c_1, \text{prob}_{C,A_i}(nb_f, a)), \ldots, (c_k, \text{prob}_{C,A_i}(nb_f, a))); \)
- \( pd_{A_i}(nb_f, c_1) \ldots \text{pd}_{A_{i-1}}(nb_f, c_1), \)
- \( \text{pd}_{A_{i-1}}(nb_f, c_1) \ldots \text{pd}_{A_1}(nb_f, c_1), \)
- \( \text{pd}_{A_1}(nb_f, c_1) \ldots \text{pd}_{A_{i-1}}(nb_f, c_1), \)\ldots
- \( \text{approx}_{A_i}(nb_f) = nb^{m-1}(\text{dist}(nb_f), pd_{A_{i-1}}(nb_f, c_1), \ldots, pd_{A_i}(nb_f, c_1), \ldots \text{pd}_{A_{i-1}}(nb_f, c_1), \ldots \text{pd}_{A_i}(nb_f, c_1)); \)
- \( \text{approx}_{A_i}(nb_f) = nb^{m-1}(\text{dist}(nb_f), pd_{A_{i-1}}(nb_f, c_1), \ldots, pd_{A_i}(nb_f, c_1), \ldots \text{pd}_{A_{i-1}}(nb_f, c_1), \ldots \text{pd}_{A_i}(nb_f, c_1)); \)

\[ \bigcup(nbf, nb^f) = nb^c(\bigcup_0(\text{dist}(nb_f), \text{dist}(nb^f))), \]

\[ \bigcup_0(pd_{A_1}(nb_f, c_1), pd_{A_1}(nb^f, c_1)), \ldots, \bigcup_0(pd_{A_1}(nb_f, c_1), pd_{A_1}(nb^f, c_1)), \ldots \]

\[ \bigcup_0(pd_{A_1}(nb_f, c_1), pd_{A_1}(nb^f, c_1)), \ldots, \bigcup_0(pd_{A_1}(nb_f, c_1), pd_{A_1}(nb^f, c_1)); \]

Table 5.5: Parameterized Algebraic Specification of the Naïve Bayesian classifier
5.4 Merging Different Decision Functions

A *decide* operation of NB takes quite some time: it requires 4 flops for each discrete attribute and 88 flops for each continuous attribute (including mathematical operations computing Gaussian) for each possible class. Thus, the decision time is estimated as

\[ T(n) = (4n + 84k) \times T_{\text{flops}} + (2n + k) \times T_{\text{aa}} + c \]

where \( T_{\text{flops}} \) is the time for a flop, \( T_{\text{aa}} \) is the array access time, and \( c \) is a constant time used for small operations.

### 5.4 Merging Different Decision Functions

In the Section 4.3, we defined a general *merge* operation for combining decision functions. This *merge* operation was based on a core operation *bind*. Hence, whenever a decision model provides an implementation of *bind* (i.e., implements the DA), our *merge* operation would work as well. This implementation holds not only for tree-based decision models but also for others. In what follows we show how the decision models of two different classification approaches can be merged. We also show that *merge* can be implemented more efficiently if the merging scenario and the actual classification approach are known.

For each scenario discussed below, we constructed two example decision functions: decision graphs (learning algorithm FC4.5 [42]) and Naïve Bayesian classifiers (NB learning algorithm [54]), respectively. The NB for each scenario is not shown due to the space limits, but they are available for download\(^3\).

#### 5.4.1 Scenario one

Scenario one assumes the formal contexts of the decision functions \( df, df' \in DF[\vec{A}, D] \) to be equivalent. If both \( df, df' \in NB[\vec{A}, D] \), then the special merge operator \( \sqcup_{\text{NB}} \) is applicable, otherwise the general abstract merge \( \sqcup \) implemented in Algorithm 2 in Section 4.3.3.1 should be used.

For example, Figure 5.3 shows a base-level \( df_{dg} \in DG \) (a.1) and the resulting DG (a.2) of merging it with another base-level \( df_{nb} \in NB \). The result of a merging is a decision function that is naturally represented as a decision graph. In order to compute the merger, we recursively applied Algorithm 2 over DG and NB using *bind* as defined in DG and NB, respectively. For instance, let us compute the result of a path \{ 6 : high, 4 : more : 1 : med, 2 : low \}. First, we bind step-by-step the attributes of the actual contexts in \( df_{dg} \) and \( df_{nb} \), resp. On each step, bind of \( df_{nb} \in NB \) integrates the unconditional

\(^3\)https://dl.dropboxusercontent.com/u/16037459/CombClassifiersCode.zip
Figure 5.3: Three scenarios of merging DG and NB
5.4. Merging Different Decision Functions

class probability distribution for this path. More specifically, we multiply the probabilities of the attributes \{6: high, 4: more: 1: med, 2: low\} given each class \(c \in C\) to the respective unconditional class probability of \(c\). The result on this path is the constant decision function with a distribution \(\text{dist}(df_{nb}^0) \in D(C)\) that is merged with the corresponding constant decision function \(\text{dist}(df_{df}^0) \in D(C)\):

\[
\text{prob}(\text{dist}(df_{nb}^0), c) = \text{prob}(pd_6(df_{nb}, c), \text{high}) \cdot \text{prob}(pd_4(df_{nb}, c), \text{more}) \cdot \text{prob}(pd_1(df_{nb}, c), \text{med}) \cdot \text{prob}(pd_2(df_{nb}, c), \text{low})
\]

\[
\text{dist}(df_{nb}^0) = \{(\text{na}, 0.002), (\text{a}, 0.005), (\text{g}, 0.002), (\text{vg}, 0.003)\}
\]

\[
\text{dist}(df_{df}^0) = \{(\text{na}, 0), (\text{a}, 0), (\text{g}, 0), (\text{vg}, 0.007)\}
\]

\[
\text{dist}(\sqcup(df_{dg}^0, df_{nb}^0)) = \{(\text{na}, 0.002), (\text{a}, 0.005), (\text{g}, 0.002), (\text{vg}, 0.01)\}
\]

5.4.2 Scenario two

Scenario two assumes the formal contexts of the decision functions \(df \in DF[\vec{A}, D], df' \in DF[\vec{A}', D]\) to be disjointed. The default merge operation of \(df, df'\) directly applies Algorithm 3. However, we can implement a more efficient merge operation in this scenario. The result is a hybrid implementation of \(DA\) with NBs as the leaves of a DG. Since the bind operation is defined on both DG and NB, it will apply the right implementation depending on the attribute being part of the DG or the NB, respectively, of the hybrid implementation. Since the other operations are derived (i.e., based on the bind), they will work as well. Only the core exert operation on NB will not continue to work any longer, but it was just an optimization.

In order to construct the hybrid merger of DG and NB, we recursively apply the Algorithm 3 to DG until we reach its leaves (constant decision functions). However, we substitute the base case in line 5 with a hybrid merge function \(\sqcup_H\) as defined below. It propagates the probability distribution over classes from the leaves in DG \(\text{dist}(df_{dg}^0) \in D(C)\) toward the probability distribution over the classes in NB \(\text{dist}(df_{nb}^0) \in D(C)\). It is defined as:

\[
\sqcup_H : DF^0[\vec{1}, D] \times NB[\vec{A}, D] \rightarrow NB[\vec{A}, D]
\]

\[
df_{dg}^0 \sqcup_H df_{nb}^0 := \text{nb}(\text{dist}(df_{dg}^0) \sqcup_D \text{dist}(df_{nb}^0),
\]

\[
pd_{A_1}(df, c_1), \ldots, pd_{A_n}(df, c_1) \ldots
\]

\[
pd_{A_1}(df, c_k), \ldots, pd_{A_n}(df, c_k))
\]

Figure 5.3 (b.1) presents the base-level DG that is merged with the NB for this scenario leading to a merged hybrid DG with NBs as the leaves,
as depicted in (b.2). For each leaf, we apply the hybrid merge operation presented above. For instance, the resulting leaf NB of the path \{5 : med, 1 : low, 3 : 4\} computes a new distribution \( dist(d_{f_{nb}}) \cup'_H dist(d_{f_0}) \in D(C) \):

\[
\begin{align*}
\text{dist}(d_{f_{nb}}) & = \{(na, 0.71), (a, 0.23), (g, 0.03), (vg, 0.03)\} \\
\text{dist}(d_{f_0}) & = \{(na, 0.02), (a, 0.12), (g, 0.02), (vg, 0)\} \\
\text{dist}(\cup'_H(d_{f_{d_0}}, d_{f_{nb}})) & = \{(na, 0.73), (a, 0.24), (g, 0.32), (vg, 0.03)\}
\end{align*}
\]

### 5.4.3 Scenario three

Scenario three allows the formal contexts of the decision functions \( df \in DF[A, D], df' \in DF[A', D] \) to be non-disjointed. Algorithm 4 works as the default merge operation, but a more efficient merge operation uses the same modification as scenario two. We use Algorithm 4, but substitute the base case in line 5 with \( \cup_H \) as defined above. The result is again a hybrid decision function.

Merging a DG and an NB in this scenario would, first, recursively bind the attribute values of the attributes in the DG (Lines 10-12). These attributes are either joint attributes, which are processed according to the scenario one, or disjointed attributes, which leave the respective NBs unchanged (recall that \( \text{evert} \) with regard to an unknown attribute leaves the decision function unchanged). Recursion stops when it reaches the leaves of the DG with its constant decision functions. They are now merged with the (partially bounded) NBs using \( \cup_H \).

Figure 5.3 (c.1) presents the base-level DG and the DG that is the result of merging with the NB for this scenario. For example, let us compute the result of the path of \{1 : high, 3 : more, 6 : high, : 5 : med\}. The joint attributes are 6 and 5. Therefore, we bind NB with regard to these common attributes:

\[
\begin{align*}
\text{prob}(\text{dist}(d_{f_{nb}}), c) & \equiv \text{prob}(pd_6(d_{f_{nb}}, c), \text{high}) \cdot \text{prob}(pd_5(d_{f_{nb}}, c), \text{med}) \\
\text{dist}(d_{f_{nb}}) & = \{(na, 0.052), (a, 0.041), (g, 0.003), (vg, 0.025)\} \\
\text{dist}(d_{f_0}) & = \{(na, 0.002), (a, 0.007), (g, 0), (vg, 0)\} \\
\text{dist}(\cup_H(d_{f_{d_0}}, d_{f_{nb}})) & = \{(na, 0.054), (a, 0.048), (g, 0.003), (vg, 0.025)\}
\end{align*}
\]

### 5.5 Summary

In this chapter, we presented four decision models as possible instantiations of DA: DGs, DTRs, DTBs (as tree-based decision models) and Naïve Bayesian classifiers (as probability-based models). For each decision model, we discussed how it could be constructed; its operations in terms of DA; and its non-functional properties, such as memory consumption and decision-making strategies.
In addition to the general DA operations, certain operations can be common to some, but not all decision models. This case can be modelled with common subtypes of DA. For instance, DTRs, graphs and tables share operations of DGA, while NBs and other probability models may share operations of a Naive Bayesian Algebra, etc. In addition, we showed how two different types of decision models (basically different sub-types of DA) can be merged. Our implications are confirmed via the experiments in the following Chapter 5. Due to the careful assessment of each decision model implemented in term of DA interface, a specific model can be chosen for an application domain with certain non-functional requirements.
Chapter 6

Experiments over Decision Algebra

In Chapter 5, we presented several existing decision models as DA instances: decision graphs, decision trees, decision tables, and Naïve Bayesian classifier, along with their implementation details.

This chapter presents the results of two experiments: (1) comparison of decision graphs (DGs) and decision trees (DTRs) (commonly-used default implementation), which are tree-based decision models constructed from the same decision information; (2) accuracy assessment of merged decision functions represented and decision graphs and Naïve Bayesian classifiers (NBs).

In the first experiment, discussed in Section 6.1, the comparison is based on the evaluation of a set of metrics: memory consumption, computational costs, and accuracy. The experiment consists of two parts: the first part (see Section 6.1.3) compares memory and time consumption used for learning and deciding for both decision models; the second part evaluates the accuracy of approximated decision models, where DGs are approximated with k-approximation (discussed in Section 5.1.2) and DTRs are approximated with standard post-pruning [88]. It compares the time required for both approximations.

In the Chapter 4 we showed that merging a series of probably accurate decision functions gives a new decision function that is tendentiously more accurate. In the second experiment (see Section 6.2) we merge a series of decision functions for scenario one (see Section 4.3.3.1) which confirms our theoretical observations. The experiment is two-fold: first (in Section 6.2.2) we merge a series of decision models with only DG representation, and then (in Section 6.2.3) we merge a series of decision models represented as DGs and NBs.

Finally, Section 6.3 concludes this chapter with a short summary of the results of both experiments.
6.1 Decision Graphs versus Decision Trees

In the previous Section 5.1.2, we presented DGs as a straight-forward instantiation of DA. Here we compare our DG implementation of decision function with a default well-known DTR implementation (which also is a possible instantiation of DA) [86]. In both cases decision functions are learned by the C4.5 algorithm that captures the same decision information (in this section referred to as dataset).

6.1.1 Data Selection

Our experiments are performed on 16 different benchmark datasets from the UCI Machine Learning Repository [34]. We were interested in a classification problems and, therefore, selected the 14 largest with both categorical and continuous attributes. We also added two large datasets with only continuous attributes to show the applicability of our theoretical framework even on these cases.

The used datasets are presented in Table 6.1 in ascending order of the number of training instances. In addition to the dataset names, the table also reports on the number of training and test instances, and the number of nodes/the depths of the generated DTRs. The final column (Attributes) describes what type of attributes each dataset is using. In all bar charts we present our experimental results for these datasets in the same order as in this table.

6.1.2 Implementation Details

We used the DTRs generated by the FC4.5 learning algorithm [42, 93] as a baseline to which we compare our graph-based implementation. FC4.5 is a fast implementation of the C4.5 learning algorithm outlined in Section 3.3. We adopted the FC4.5 algorithm to directly learn both DTRs and Decision Graphs. As a result, both representations have exactly the same classification accuracy when no additional pruning is applied. In order to make a fair comparison between the two representations, we had to make a few minor adjustments, though:

1. Each internal node of the DTR constructed by FC4.5 keeps a training weight, a distribution, and a possible classification, information that is later used for decision making. In our DG implementation, the internal nodes do not contain any such information. They just contain information about the attribute they represent. It is only the leaves that keep a classification distribution $d(C)$. Both representations use the

---

1This is confirmed experimentally as well.
<table>
<thead>
<tr>
<th>Id</th>
<th>Dataset</th>
<th>Training ins.</th>
<th>Test ins.</th>
<th>Tree Size/Depth</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ionosphere</td>
<td>309</td>
<td>42</td>
<td>21/7</td>
<td>34-categorical</td>
</tr>
<tr>
<td>2</td>
<td>cancer-wisconsin</td>
<td>500</td>
<td>199</td>
<td>125/4</td>
<td>1-cont., 9-categorical</td>
</tr>
<tr>
<td>3</td>
<td>australian</td>
<td>552</td>
<td>138</td>
<td>143/9</td>
<td>6-cont., 8-categorical</td>
</tr>
<tr>
<td>4</td>
<td>crx</td>
<td>600</td>
<td>119</td>
<td>174/9</td>
<td>6-cont., 9-categorical</td>
</tr>
<tr>
<td>5</td>
<td>dibetes</td>
<td>688</td>
<td>100</td>
<td>27/7</td>
<td>8-cont.</td>
</tr>
<tr>
<td>6</td>
<td>anneal</td>
<td>798</td>
<td>133</td>
<td>151/14</td>
<td>6-cont., 32-categorical</td>
</tr>
<tr>
<td>7</td>
<td>german</td>
<td>800</td>
<td>200</td>
<td>416/11</td>
<td>7-cont., 13-categorical</td>
</tr>
<tr>
<td>8</td>
<td>hypothyroid</td>
<td>2527</td>
<td>636</td>
<td>19/8</td>
<td>7-cont., 18-categorical</td>
</tr>
<tr>
<td>9</td>
<td>ad</td>
<td>3057</td>
<td>420</td>
<td>153/55</td>
<td>3-cont., 1556-categorical</td>
</tr>
<tr>
<td>10</td>
<td>waveform</td>
<td>4000</td>
<td>1000</td>
<td>515/18</td>
<td>21-cont.</td>
</tr>
<tr>
<td>11</td>
<td>nursery</td>
<td>11664</td>
<td>1294</td>
<td>905/7</td>
<td>8-categorical</td>
</tr>
<tr>
<td>12</td>
<td>chess</td>
<td>28042</td>
<td>4886</td>
<td>10001/5</td>
<td>6-categorical</td>
</tr>
<tr>
<td>13</td>
<td>adult</td>
<td>32561</td>
<td>16281</td>
<td>8124/20</td>
<td>6-cont., 8-categorical</td>
</tr>
<tr>
<td>14</td>
<td>connect-4</td>
<td>67557</td>
<td>13994</td>
<td>15940/22</td>
<td>42-categorical</td>
</tr>
<tr>
<td>15</td>
<td>census-income</td>
<td>159617</td>
<td>39906</td>
<td>46363/26</td>
<td>8-cont., 33-categorical</td>
</tr>
<tr>
<td>16</td>
<td>covtype</td>
<td>409985</td>
<td>171027</td>
<td>28389/63</td>
<td>10-cont., 45-categorical</td>
</tr>
</tbody>
</table>

Table 6.1: Dataset Characteristics for Comparison Decision Graphs and Decision Trees
same type of distributions, the frequency-based distribution presented in Section 3.1. Additionally, we take into account unknown attribute values by using counts less than one in the distributions [93]. Moreover, in cases when the distribution is zero, we use a bottom distribution where all classes have a frequency equal to the classification probability $1/|C|$. Due to these simplification, the repository is able to identify (and reuse) equivalent sub-graphs without losing any information.

(2) As a consequence of (1), the decide operation used for both DGs and DTs is a simplified version of decide as implemented in the FC4.5 algorithm. For example, when a test attribute value is missing, the test data is passed to all the children of the current node without taking into account the partitioned weight of the children. Because of this simplification, we expected to lose classification accuracy when comparing the results with the pure, unmodified FC4.5 implementation. However, the accuracy remained the same in all experiments. This observation justifies the simplifications presented in (1) where we neglect additional information kept in the internal nodes.

(3) A continuous attribute $A_i$ can be partitioned into different intervals in different branches of the tree. In these cases, we consider each new partitioning as a new categorical attribute and also coming with a new selection operator. As we will see when we discuss our experimental results, this drastically reduces the chance for the repository to identify redundancy due to equivalent sub-graphs in datasets where many continuous attributes are used.

These implementation details will be of importance when we later on discuss our experimental results.

### 6.1.3 Assessment Sizes and Times of Decision Trees and Graphs

In this sub-section, we present the result of the first sub-experiment. Our DG implementation recognizes identical subtrees and makes sure that we only keep one such instance. Thus, we expect the DGs to contain fewer nodes than the corresponding DTRs.

Figure 6.1 displays two bars for each dataset for comparing the number of nodes in the DGs (right) with the number of nodes in the DTs (left, always scaled to 100%). For example, Bars 1 (ionosphere) show that the DG has the same size as a corresponding DT, whereas Bars 2 (cancer-wisconsin) show that the DG contains only 36% of the nodes of the corresponding DT, indicating a 64% node reduction. The overall result, an average node
6.1. Decision Graphs versus Decision Trees

Figure 6.1: The percentage of reduced internal nodes and leaves compared to the total tree size (100%)

Figure 6.2: Times of learning and deciding based on a DG as % of DT (100%)
Chapter 6. Experiments over Decision Algebra

reduction of 44%, indicates that much memory can be saved by using DGs as representation.

Each bar in the chart is also divided into two parts, separating internal nodes from leaves. When comparing the number of internal nodes in the trees with the corresponding graphs, we see that, in most cases, the numbers are almost the same, indicating that a majority (98%) of the reduced nodes are leaves. This has two reasons:

- A large part of the removed leaf nodes are bottom leaves representing attribute value combinations not covered by any instances in the dataset. Although associated with different weights, a majority of these leaves could be removed. The remaining part of the removed leaves are due to non-empty, but identical, distributions.

- The internal node reduction is quite small (4%). The major reason is that we treat different interval partitions of continuous attributes as entirely different attributes. Hence, the possibility of identifying identical subtrees in these cases is very low. This explanation is supported by the observation that in datasets where we are using mainly categorical attributes (2,11,12,14), we have a larger reduction of internal nodes. For example, case 11 (nursery) has a reduction of 37%.

Additionally, we have measured the time for construction the DTs (DT) and DGs (DG) and also the time used for classifying (decide) a fixed number of instances for each dataset. The time was measured in milliseconds, but we have used a relative measure (DG.Time/DT.Time) in order to simplify the comparison. The results presented in Figure 6.2 show that the DG implementation is faster in almost all cases. The only exception is the graph construction in case 13. The average construction and classification time for DGs is about 19 and 20% less than for DTs, respectively. The reduced classification time for DGs is at first glance a bit surprising, given that the number of selections in both cases are the same. However, this is likely due to a reduced strain on disk caches and the hierarchy of memory caches due to the reduced memory usage in the smaller graphs.

6.1.4 Assessment of Accuracy and Times of Approximated Decision Trees and Graphs

In a second sub-experiment, we compare the accuracy of $k$-approximated DGs with post-pruned DTs. The accuracy was measured with cross-validation (where the number of the test instances is presented in Table 6.1). We also compare the time required for learning followed by pruning of DTs with the time required to learn directly to $k$-approximated DGs.
6.1. Decision Graphs versus Decision Trees

The post-pruning in the FC4.5 implementation uses a so-called reduced error pruning strategy \cite{88}, where internal nodes of a fully grown tree are removed one at a time as long as the error is decreasing.

Our $k$-approximation is, in contrast, a very simple process, where we merge the leaves of all sub-trees below a certain depth $k$, see Section 5.1.2. Notice that, although the pruning takes place during the learning process, it is some kind of post-pruning since we merge the leaves of fully grown branches, unlike pre-pruning which can suffer from premature termination of a tree-growing process.

In our experiments, we used the depths of the pruned DTs to decide on the $k$ used in the $k$-approximations. Figure 6.3 shows the results of the accuracy comparison.

For each dataset (1–16) we have four bars. The first three show: 1) the accuracy (%) of DTR before pruning (denoted DT), 2) the accuracy of the pruned DTR (denoted PDT), and 3) the accuracy of the $k$-approximated DG (denoted KDG). On top of each PDT bar, we show the depth $k$ of the pruned DT; this is also the depth of the corresponding KDG. Finally, the fourth bar shows the results of an improved $k$-approximation referred to as KDG2: the depth $k$ is decremented step-by-step as long as the error decreases further. The fourth bar shows the KDG2 accuracy along with the finally reached depth as the number on top of that bar.

First, the results for the two approaches (PDT and KDG) are quite similar. On average, the pruned DTs outperform the $k$-approximated graphs by only 0.04%. This is a bit surprising given the difference in complexity of the two approaches.

Second, in a number of datasets (1,10,11,12) the depths of the pruned trees remain unchanged. Consequently, no $k$-approximation is applied and the results for DT and KDG are the same. At least for dataset 10 (waveform), the second approximation strategy leads to improvements: it reduces the size of the DG considerably and reaches the accuracy of the pruned Decision Tree, cf. PDT vs. KDG2.

Finally, there are cases where KDG and KDG2 are more accurate than PDT (3, 12, 14). For example, dataset 14 (connect-4) indicates that the accuracy of the DT before pruning was 92%, after pruning 88%, the $k$-approximated accuracy 92% with $k=19$, and the $k$-approximated accuracy 91% with $k = 16$. There are also other cases where pruning is not improving accuracy at all and cases where PDT is more accurate than KDG and KDG2.

Regarding the learning and pruning/approximation times, the results are, again, non-ambiguous. In the time measurements, we have used the same relative metric as in Section 6.1.3. Figure 6.4 shows that the learning of $k$-approximated DGs clearly outperforms the joint procedure of learning and pruning the DTs by up to 50%. On average, the $k$-approximated approach requires about 21% less time than the tree pruning approach.
Chapter 6. Experiments over Decision Algebra

Figure 6.3: The accuracy gained by pruning DTs and using k-approximated DGs.
6.2. Accuracy of Merged Decision Functions

In Section 4.2.3 we showed that merging of a series of “probably accurate” decision functions gives a new decision function that is tendentiously more accurate. In this section we present an experiment that confirms our theoretical implications. The experiment first merges a series of DGs (see Section 6.2.2) for scenario one of merging the decision functions of the same classification approach, and then a mixed series of DGs and NBs (see Section 6.2.3) for the scenario one of merging the decision functions of different classification approach.

6.2.1 Data Selection

This experiment is performed on 8 different benchmark datasets from the UCI Machine Learning Repository [34]. Notice, that the datasets chosen for this experiment differ from those we selected before. This is because we were only interested in datasets with categorical attributes. Therefore, we took two datasets from previous experiments (nursery and chess) and added six new datasets. We only selected datasets with a number of learning instances ≥ 200.

The selected datasets are presented in Table 6.2 sorted in ascending order of the number of training instances. In addition to the dataset names, the table also gives the number of training instances and the number of attributes. The last column (Accuracy %) shows the accuracy gained by learning over a complete dataset using FC4.5 algorithm for DGs.

Figure 6.4: Learning and approximation times of DGs as % of DTs (100%)
6.2.2 Assessment of Accuracy of Merged Decision Graphs

For this experiment, each training set was divided into eight smaller sets. For each small set (1/8 of a complete dataset), a DG was learned, refer to as regular DG. The accuracy of all regular DGs was measured with regard to the same common test set, which was randomly chosen from the complete dataset.

For each training set, we merged the regular DGs step by step using the apply algorithm apply discussed in Section 5.1 which, in turn, uses the merge operator $\sqcup$ over the distributions on the leaves explained in Section 4.2.3. The accuracy of the merged decision function depends on the order in which the regular DGs are merged. For each training set, we computed all permutations of its 8 regular DGs (40 320 permutations for each training set) and regular DGs in the order of their occurrence in the permutations, and measured the accuracy for each step and an average accuracy after each merging.
6.2. Accuracy of Merged Decision Functions

<table>
<thead>
<tr>
<th>Id</th>
<th>Dataset</th>
<th>Training Instances</th>
<th>Attributes</th>
<th>Accuracy(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>audiology</td>
<td>200</td>
<td>69</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>monks</td>
<td>415</td>
<td>6</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>balance-scale</td>
<td>438</td>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>tic-tac</td>
<td>749</td>
<td>9</td>
<td>83</td>
</tr>
<tr>
<td>5</td>
<td>car</td>
<td>1728</td>
<td>6</td>
<td>95</td>
</tr>
<tr>
<td>6</td>
<td>mushroom</td>
<td>6494</td>
<td>22</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>nursery</td>
<td>11164</td>
<td>8</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>chess</td>
<td>28042</td>
<td>6</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 6.2: Dataset Characteristics for Merging Decision Functions

Figure 6.5 displays for each dataset the average accuracy growth of the merged DGs (blue plot) after each step and the average accuracy of the regular DG (green line) in relation to the accuracy gained by learning over the complete dataset. That is the merged accuracy and regular accuracy calculated at each step is divided by the complete set accuracy. As predicted by Theorem 4.3.1, all charts show that on each step the accuracy of the merged DG tendentiously grows. For example, the accuracy growth for "audiology" is 5.5% (from 62.5% till 68%), while the accuracy of the regular DG is around 62.5%. Notice, that the plot shows the ratio of merged/step accuracy and the accuracy of a complete dataset. The highest accuracy growth is 16% for the "chess" dataset, and the lowest is 2.5% for "nursery" dataset. Moreover, for all datasets we can see that the merged accuracy slowly growth towards 1 (red line), i.e. towards the result that a decision model gives when learned over a complete dataset. For each dataset, we calculated the average number of accuracy growth steps per permutation (the number of steps per permutation when the accuracy of the merged decision function grows by merging) and the permutations with accuracy growth (percent of permutations with more than half of the steps are accuracy growth steps).²

Figure 6.7 shows the results for each dataset, cf. dataset identifiers in Table 6.2. Numbers above each bar give the permutations with accuracy growth and the accuracy growth steps (in phases). For instance, Bar 8 ("chess") show that 100% of the permutations have at least 4 merging steps leading to accuracy growth; on average about 5 merging steps per permutation lead to accuracy growth.

The overall result indicates that merging decision functions gives a new decision function that is tendentiously more accurate and, therefore, we could define a general online learning approach based on merging decision functions.

²Notice, that the total amount of merging steps is seven, since on the first step we do not merge the DG.
6.2.3 Assessment of Accuracy of Merged Decision Graphs and Naïve Bayesian Classifiers

Here we present the results of merging a series of DGs and NBs for scenario one.

For this experiment, each data set was randomly divided into eight smaller sets (1/8 of a complete dataset). For a half of the small sets a DG was learned, referred to as base-level DG, for another half an NB, referred to as base-level NB (we learned them alternately, first DG then NB as so on). The accuracies of all base-level DGs and NBs were measured using same common test set.

For each data set, we merged the base-level DGs with NBs step by step using Algorithm 2 discussed in Section 4.2.3, which works unchanged in
6.2. Accuracy of Merged Decision Functions

Scenario one. Notice, that the result of a merger of DG\textsuperscript{3} and NB is a DG, referred to as merged DG.

The accuracy of the merged DG depends on the order in which the base-level DGs and NBs are merged. Therefore, for each data set, we computed all permutations of its 8 parts (40 320 permutations for each data set) consisting of 4 base-level DGs and 4 base-level NBs. Then, we measured the accuracy for each step and computed an average accuracy after each merging step.

Figure 6.6 displays, for each dataset, the average accuracy growth of the merged DG (straight black line) after each step of merging, the average accuracy of the base-level DGs (DG(1/8)) and NBs (NB(1/8)), respectively. As claimed by Theorem 4.3.1, all charts show that on each step the accuracy of the merged DG grows. For example, the accuracy growth for “Audiology” is 6% (from 62.5% till 68.5%), while the average accuracy of the base-level DG is around 62.5%, and the average base-level NB is around 50%. The highest accuracy growth is 17% for the “Chess” dataset, and the lowest is 2.7% for the “Nursery” dataset. Moreover, for all datasets we can see that the merged accuracy slowly grows towards 1 (optimum value), and towards the accuracy learned over a complete dataset, cf. Table 6.2, column “Accuracy”.

The overall result indicates that merging decision functions gives a new decision function that is tendentiously more accurate.

\textsuperscript{3}When the probability of a certain class with a certain attribute value is computed as 0, then for the computation of the final probability we take a default \textit{epsilon} value.
Chapter 6. Experiments over Decision Algebra

6.3 Summary

In the first part of this chapter we compared DGs and DTRs as alternative decision models. On the theoretical side, the experiments showed that DGs and DTRs become comparable, and the learning algorithm usually used by DTRs becomes applicable to DGs as well. On the practical side, our experiments showed benefits of Decision Graphs compared to Decision Trees regarding memory resource and time utilization as detailed below.

First, the experiments show that, without losing accuracy, Decision Graphs reduce the memory consumption of Decision Trees by 44% (on average over a number of standard datasets). The reduction is mostly caused by the elimination of redundant leaves, but replicated and redundant decision (sub-)trees also contribute to the memory reduction. Redundancy (and with it the potential saving of our redundancy elimination) increases as Decision Graphs avoid keeping additional information in internal nodes. Such internal information can be different for different (otherwise identical) subtrees. The absence of such information does not influence the classification accuracy, as proved by our experiments. Also worth mentioning is that the reduction appears to grow with the number of categorical attributes and with the size of (number of instances in) the dataset.

Second, simple k-approximated Decision Graphs and pruned Decision Trees have almost equivalent accuracy. As opposed to, e.g., error-based post-pruning, the k-approximated Decision Graphs do not apply any complex statistical calculations in the leaves and simply merge classes in a fully grown tree branch. This means that using approximated Decision Graphs allows avoiding additional costly post-pruning.

Third, the time for Decision Graphs construction shows a decrease by 19% compared to the time for Decision Tree construction. Furthermore, the time measured for classification using Decision Graphs was 20% less than in the corresponding Decision Trees. This is, likely, the result of less strain on caches due to the memory reduction in Decision Graphs. The time for learning followed by pruning the Decision Tree compared to learning directly to the k-approximated Decision Graphs decreased by around 21%. This result supports our statement about avoiding post-pruning operations mentioned above.

In the second part of our experiments we confirmed our theoretical implications, and showed that merging of a series of probably accurate decision functions gives a new decision function that is tendentiously more accurate. The experiment result of merging: (1) DGs showed lowest accuracy growth of 2.5% and highest of 16%; (2) DGs and NBs showed an accuracy growth of 2.7% up to 17%.

By this chapter we conclude the first part of this thesis, where we presented a main theory of Decision Algebra. We defined an abstract parameter-
ized decision model as a decision function along with a set of operations over this function. Also, as possible instances of Decision Algebra we focused on tree-based and probability-based decision models, each coming with defined core and derived non-core operations that follow Decision Algebra interface. Finally, this part was concluded by experiments that compare and merge decision functions represented by different decision models implementation.

The next part of the thesis (Chapters 7 and 8) discuss the application of DA to context-aware composition, in particular, how context-aware applications can benefit from Decision Algebra. In Chapter 7 we introduce a DA component in context-aware composition, and suggest an approach to integrate a context-aware Decision Algebra component into legacy applications. In Chapter 8 we assess the performance of context-aware application with DA component.
Chapter 7

Decision Algebra in Context-Aware Composition

Context-aware computation is an essential part of a wide range of application domains, where an application should adapt its behavior according to a potentially changing context or environment during its execution. Context-oriented programming is a technique used for the design of such applications [43, 20].

Context-aware composition (CAC) is a special case of context-aware computation that aims at adapting applications to changing call contexts and available resources in the system environment. Its goal is to improve performance (or other optimization goals) of such applications. Among others, CAC allows developers to develop performance-portable programs for modern multi-core hardware.

CAC separates the concerns of defining component variants from the decision in favor of any of these variants in a composition context. The former is completed by component designers who develop variants that might be advantageous in certain composition contexts that are possibly supported by variant generators, which generate, for instance, different schedules. The latter is fully automated. CAC applications evaluate their call contexts and system environments at runtime and, depending on that, select between different alternative sequential and parallel component variants. The selected decisions are based on previous experiences from offline profile executions or from online execution monitoring, which is often abstracted and aggregated using dispatch (decision) tables. Therefore, CAC can improve the performance of software systems considerably as it dynamically composes with the prospected best variant of alternative algorithms, data structures and resource allocations for each dynamic composition context. In order to exploit the benefits of the recent hardware development toward multi-core processors, possibly supported by hardware accelerators such as GPUs, applications have to adapt to their respective system environments. For this problem domain DA is capable to provide such a solution based on the possibility of reusing decision models for selecting best-fit component variant that can improve the application performance even more.
Chapter 7. Decision Algebra in Context-Aware Composition

This chapter is structured as follows. Sections 7.1 and 7.2 give an overview of CAC and application of DA in CAC. Section 7.3 sketches the common manual approach of implementing CAC with built-in DA component in applications. In Section 7.4 we discuss a practical application of CAC with built-in DA component to the development process as a context-aware recommender system for non-functional requirements. Section 7.5 provides an automated way by which to adapt an existing application to context-awareness using Aspect-Oriented Programming. In addition, this section discusses the main principles of aspect-oriented context-awareness and how it could be applied to the adaptation of legacy applications. Additionally, this section provides the key idea of the possible implementation approach. Finally, Section 7.6 concludes the chapter and guides the reader to the next chapter.

### 7.1 Context-Aware Composition

CAC is a technique used for the automatic selection of optimal variants of program components at runtime. Creating new CAC applications is well understood [23, 72]. However, introducing CAC into existing applications usually requires a high re-engineering and implementation effort and, therefore, can be time-consuming and error-prone.

Regardless of the concrete applications, adoption of CAC needs to address some common concerns: offline profiling and/or runtime monitoring, an approach to extrapolate or interpolate the profiling/monitoring results and runtime variant selection and dynamic composition for each call context and system environment. These concerns can be separated from the actual application.

The scalability of context-aware composition generally depends on the scalability of the variants-modeling technique that, in turn, depends on the actual number of contexts. However, dispatch tables are not very scalable. The process of capturing decision information into dispatch tables can be memory consumptive as the tables grow exponentially with the number of context attributes (as was shown in Chapter 5). Hence, capturing these tables for every composition point (i.e., all calls to a function \( f \) of a component with several implementation variants) might lead to problems with the overhead in memory consumption, implying that the amount of required memory for the execution of a certain applications might be dramatically increased.

In order to remedy this problem, in this chapter we enhance the scalability of the variants-modeling technique by introducing the Decision Algebra component. In fact, we learn the decision functions \( df : A_1, ..., A_n \to D(C) \), with a table dimension \( A_i \) (i.e., a context attribute) and a table entry \( C \) (i.e., the presumably best algorithm variant of \( f \)). The actual composition (dynamic
7.2 Decision Algebra in Context-Aware Composition

dispatch) then evaluates the function \( df \) for the actual context attributes and invokes the best variant. The Decision Algebra component provides alternative decision functions representations of different classification approaches. Therefore, we need to bias:

- Accuracy of the decision: does \( df \) always decide for the optimal variant, and what is the impact of suboptimal decisions on the overall performance;
- Decision time and its impact on the overall performance; and
- Memory size for capturing \( df \).

For this work, we assume offline learning and hence ignore the learning time.

7.2 Decision Algebra in Context-Aware Composition

CAC is the runtime context-dependent binding of a call in one component to one best-fit callee out of a set of matching callee variants defined in other components. For this chapter, we assume that CAC is used to improve application runtime performance (while other optimization goals, such as decreasing energy or memory consumption are possible, too). Optimization is achieved by binding the caller with the currently best-fit callee variant depending on the actual runtime context. Context-aware composition operates on the concepts introduced by Decision Algebra in Chapter 2:

**Formal context** is a call site with formal context parameters. Formal context parameters can include (abstractions and selections of) the formal call parameters (e.g., the problem size), the locally assessable system state, and even the system environment (e.g., the number of available processors).

**Actual context** is a valuation of the formal context parameters at runtime before a corresponding call.

**Formal decision** (a component interface) is a callee interface that can be bound to the call site of a context. Formally, it is an abstract method with pre-conditions and post-conditions that are always implied by and always satisfying the call site.

**Actual decision** (a component variant) is a callee implementation, formally, a co-variant subtype of the component interface. The implementation
variants can range from alternative algorithms and data structures to alternative schedulings or exploitations of hardware accelerators. Note that not all implementation variants need to fit all actual call contexts (co-variance) as long as there is always one variant matching each possible actual call context.

**Decision function** (a utility function) is an optimization goal function that maps an actual context and a component variant to numerical values representing the variant’s performance, footprint and memory consumption among other variables (or combinations thereof) in that context.

Context-aware composition requires a **learning** phase, where the component variants are tested for different actual context and the champion variant of each context is determined. This information can be extrapolated and interpolated using different decision models of common DA interface from a total mapping of contexts to champion variants. The decision function is used in the actual **composition** phase for selecting the presumably optimal variant for each actual context.

While in principle learning and composition may happen offline or online (depending on whether the formal context parameters allow for a static actual context evaluation and binding), offline learning and online composition are the most common cases, which are also the basis of the present work.

### 7.2.1 Decision Information in Context-Aware Composition

In a **learning** phase, variants are tested in different contexts, and the champion variant of each context is usually captured in a generalized dispatch table that is later used to compose with this variant in the actual composition context. During the actual **composition** phase, when a *decision* in a favor of a best-fit variant has to be made, the dispatch table gets a champion variant for the actual composition context ($\vec{a} \in \vec{A}$).

The composition context can range from domain-specific to technology-dependent attributes, and even include properties that may be based on hardware or software (e.g., the number of available processors, problem size and available memory, etc.). Hence, the number of attributes in the dispatch table can be very large. The dispatch table is a decision table.

When a problem domain is relatively small (e.g., the number of attributes and classes is small), a dispatch table may have an advantage over other decision models due to a compact and easy problem description. However, it is usually time- and memory-consuming to construct and maintain such a table when the number of attributes, attribute values and decision tuples is large.
7.3 Object-Oriented Design and Concerns

The inefficient representation of decision information in CAC leads to the low performance of the application in terms of memory consumption and execution time. Different classification approaches might be employed depending on the strategy of the problem to solve, resulting in decision models that differ in regard to the speed of learning and deciding, memory consumption and decision accuracy. Additionally, decision models need operations that can be used to process context-aware decision information in order to adjust and improve the application behavior with respect to the specific requirements of the given application domain. Thus, having a built-in DA block in CAC may remedy these problems and increase the overall performance of applications with context-aware adaptation.

In Section 7.3, we discuss the prerequisites and common CAC concerns required in the object-oriented design of the application for its adaptation to context-awareness.

7.3 Object-Oriented Design and Concerns

When adapting legacy applications to CAC, we require a good object-oriented design following appropriate design patterns. If an application is not designed accordingly, then the design must be established in a manual refactoring/reengineering step.

We need to separate component variants from component interfaces and to distinguish the stateful from the stateless components in such a way that they both could be changed independently. Therefore, we assume the Strategy design pattern [36] as depicted in Figure 7.1: each abstract data type ADT (component) encapsulates states and algorithms operating on that state. This pattern separates the abstract state from its representation and the abstract algorithms from their implementations. All algorithm implementations need access to the state representations (general or special). Besides setting and getting the state or invoking the algorithms, users of the ADT can also control the implementations.

More specifically, ADT is configured with concrete Strategy objects determining the Algorithm Variant and Representation Variant. The ADT maintains a reference to these objects and defines an interface that lets the algorithm variants uniformly access the data. The algorithm and representation variants are implemented separately by subclasses of the abstract Representation and Algorithm classes. The ADT exposes the means by which to change the data representation variant (changeReprTo) and the algorithm variant (setAlgVariantTo). Calls to an abstract algorithm are redirected to the current algorithm variant (callAlgVariant) using the current representation.
Chapter 7. Decision Algebra in Context-Aware Composition

7.3.1 Concerns of Context-Aware Composition

Once the prerequisites are established, the following concerns have to be defined: the formal context affecting the choice of the implementation variants and functions assessing the actual context at each corresponding call site. These concerns cannot be guessed automatically and have to be implemented manually for each ADT. The call sites still bind to the ADT operations on the interface level; they neither invoke the functions used to assess the actual context nor explicitly control the implementations of the representations or algorithms.

For the offline learning phase, the test data needs to be generated for each sample of the actual contexts. This test data generator needs to be implemented manually for each abstract representation. The learning in-
7.4 Context-Aware Recommender Systems

In order to develop a fully-featured system, a system designer has to make sure that the system meets both the necessary functional and non-functional requirements. The functional requirements define what a system is supposed to do, while the non-functional requirements specify how a system is supposed to behave [63].

Non-functional requirements, such as performance, dynamic memory consumption and memory footprint of the code, are usually considered during all of the phases of the project development and are highly dependent upon the architecture and design decisions. For large systems, it becomes relatively complicated for the system designer to deal with all of the non-functional requirements, while also focusing on the core functionality of the system, especially when the requirements quickly emerge during the development, as normally is the case in the Agile development processes. As such, handling all of the requirements at once can easily get out of control. For example, in order to write efficient software that involves solving a computational problem, the software designer must carefully choose and tune the algorithms to keep the number of operations as low as possible so as to eventually lead to a high application efficiency. Therefore, it is beneficial to postpone the architectural decisions until a system designer meets the non-functional efficiency requirements so as to save time and effort on these decisions.

On the other hand, an attempt to work out non-functional requirements at the end of the development phase can lead to a significant system rework or even to a failure of time and effort estimations [70]. At each step of the
development process, a danger exists that a design decision will have undesirable side-effects that will eventually lead to either inaccurate or inefficient application performances. It is, therefore, important to make sure that architectural decisions regarding functionality do not later inhibit the ability of the designer to meet some of the non-functional requirements.

This section suggests a context-aware recommender system with a built-in DA block that will help system designers and developers create fully-featured software with less effort and within a shorter amount of time. This recommender system uses context-aware composition as the basic technique and this approach intends to postpone the architectural decisions to be made in order to meet the non-functional requirements and allow a system designer to focus on developing the functionality of the software first. The recommendation system, similar to CAC, may recommend for a best component variant in a certain application context (a composition context is evaluated, e.g., the static system environment or the runtime resource utilization). The recommendations can be used by a software developer who writes the composition code, which statically binds to the best-fit variant in a static context, or dynamically dispatches to the best-fit variant in a dynamic context. Alternatively, this composition code can be generated automatically (e.g., using Aspect-Oriented Programming) as will be shown in the next section.

In the following section, we will give an overview of our suggested recommender system and sketch the main ideas of how this system can be applied in the development process using context-aware composition techniques.

7.4.1 Main Principles of CA Recommender Systems

Figure 7.2 depicts the conceptual building blocks of the recommender system and the entire development process. As discussed previously, profiling tests alternative component variants in different contexts and captures the champion variant of each context. A learner extrapolates and interpolates the information received from the profiler (using decision function) from a total mapping of the contexts to champion variants and generates a recommender (a decision model). A recommender predicts the presumably optimal variant for each actual context.

Figure 7.2 shows the entire process of applying the recommender in the software development process. A system designer is required to define formal contexts that affect the choice of the component variants that might be advantageous in certain actual contexts. Based on this information, a software developer implements the functions that assess the actual contexts at each corresponding call expression and alternative component variants. After profiling and learning, the recommender system outputs a recommen-
7.4. Context-Aware Recommender Systems

Figure 7.2: The recommender system in a software development process.

dation plan that can be used either by a software developer (system designer) or composition code generator.

Based on the given recommendation, the software developer (code generator) writes (generates) a connector that binds to the best-fit component, statically or dynamically. More specifically, the connector can be a static monomorphic call or a complex dynamic dispatch pattern. The former is preferred when the actual context is static or the recommendation for different dynamic contexts is always the same. The latter is more expensive in terms of runtime overhead, but necessary when the actual context changes at runtime and the best-fit component variant also may change. Then the connector implements a dynamical adaptation of the application behavior in an actual context. The (static or dynamic) composition eventually puts together the alternative component variants using the respective connector, and binds a caller in one component to a matching callee in another component.

The profiling and learning phases can happen offline, when the application runs a test regime under the supervision of a system developer, or online, at the system runtime (depending on whether the actual contexts and the variants’ utility can realistically be assessed statically or not). The recommendation and composition phases can happen offline or online as well, depending on whether the formal context parameters allow for a static actual context evaluation and composition or not. Possible usage scenarios
Table 7.2: Usage scenarios.

<table>
<thead>
<tr>
<th>Scenario (1)</th>
<th>Scenario (2)</th>
<th>Scenario (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>profiling</td>
<td>profiling</td>
</tr>
<tr>
<td></td>
<td>learning</td>
<td>learning</td>
</tr>
<tr>
<td></td>
<td>recommendation</td>
<td>composition</td>
</tr>
<tr>
<td>Online</td>
<td>recommendation</td>
<td>profiling</td>
</tr>
<tr>
<td></td>
<td>composition</td>
<td>learning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recommendation</td>
</tr>
</tbody>
</table>

are summarized in Table 7.2 and detailed below.

**Scenario (1).** If the formal context (e.g., target hardware) and the variant utility (e.g., memory footprint) can be evaluated statically or if the same variant is recommended in all of the sampled actual contexts, then the composition can be done statically, i.e., offline. Composition is implemented with a static monomorphic call.

**Scenario (2).** If the formal context needs to be evaluated dynamically (e.g., the problem size) and different variants are recommended for different actual contexts, then the recommendation and composition are performed online, and the connector is adaptive. The profiling and learning phases can still be done offline if the actual contexts occurring online can be simulated offline.

**Scenario (3)** If, however, uncertainty exists about the actual contexts occurring online (e.g., the target system or the actual parameter profile are unknown or changing), it might be beneficial to postpone even profiling and learning until the runtime. A special adaptive connector composes according to the (initial random) recommendation of the recommender, but it deliberately deviates from time to time from the presumably best-fit component variant. The profiler monitors the selected component variant in the deployed and running application. A learner runs in parallel to the application, aggregates the monitoring results, and constantly updates the recommender. The recommender provides (increasingly better) recommendations to the special adaptive connector.

Notice that in all three cases, the connector can be written by a software developer or generated by a code generator as will be shown in Section 7.5.

### 7.5 Aspect-Oriented Context-Aware Composition

It has already been shown that context-awareness is in principle a cross-cutting concern and, therefore, can be treated as an *aspect* [102]. By means
of a so-called advice, aspects can be applied to certain program points called join points. Application of the aspects can be controlled by the actual context of the join point, its program state, and the state of the system environment, which has to be determined at runtime.

In this section we suggest an approach to adapt legacy applications to context-awareness based on Aspect-Oriented Programming (AOP) and Reflective Programming that separates the concerns of offline profiling and runtime composition and reuses them. The context-awareness aspect becomes reusable regardless of the legacy application. It selects the optimum component variant dynamically for each actual call context, including recursive calls. Assuming good object-oriented design, presented in Section 7.3, adaptation does not require any changes in the legacy codes, which enables the (re)-engineering of self-adaptive and performance-portable (legacy) applications and makes them run efficiently on modern hardware.

The reminder of this section is structured as follows. Section 7.5.1 demonstrates how to implement the CAC concerns (presented in Section 7.3.1) with reusable aspects, while Section 7.5.2 presents key ideas of this implementation approach.

### 7.5.1 Concerns of AOP Composition

In order to separate the profiling/learning from the composition phases and to avoid changing legacy applications (except for refactoring the code to guarantee the prerequisites described in Section 7.3.1), we use the Template design pattern [36]. The pattern implements the common behavior of profiling/learning and composition once and for all applications that have established prerequisites for adaptation to CAC. The common behavior of profiling/learning is implemented in a template method train of the Profiler class (cf. Figure 7.3). The train method profiles all of the algorithm variants with different data representations and captures the best-fit variant. The algorithm variants are defined as a skeleton algorithm deferring the implementation of some steps to subclasses that are specific for concrete applications. These steps are generateContext and generateTrainData. The former operation samples actual contexts while the second operation generates the input data for the algorithm variants based on these samples. Context is an abstract class with an abstract getCurrentContext method that provides the actual context. This method implementation is again deferred to application-specific implementations. Finally, the best-fit variant for each actual context is captured and abstracted to a final decision function using different classification approaches. Due to the common DA interface, the actual representation of the decision function can be exchanged as well, based for example, on decision tables, decision graphs and Naïve Bayesian classifiers, etc.
Chapter 7. Decision Algebra in Context-Aware Composition

The common behavior of composition is separated (cf. the Composition block of Figure 7.3 and Figure 7.4 for details). Composition only needs access to evaluate the actual context via \texttt{getCurrentContext} and to decide on the corresponding best-fit variant via \texttt{decide} operation in the decision function. Other operations of DA are also included into the abstract decision function, and can be used for online learning or a combination of different decision models (e.g., \texttt{merge} and \texttt{approx}).

The program points of interposition, where the method calls to alternative algorithm variants occur, are theoretically known. During the learning phase, the method interposition occurs in recursive calls of the algorithm implementation variants. In contrast to the learning phase, at runtime, it occurs at every call site to a method with implementation variants. However, regardless of the phase, the call sites, and the target method, the same composition mechanism applies (i.e., the same code fragments have to be inserted at these call sites).

As the set of interposition points cuts across a number of application module boundaries, they can be defined as cross-cutting concerns. AOP is designed to handle these concerns by providing a mechanism, called \textit{aspect}, for expressing and automatically incorporating them into the application’s code\footnote{http://www.ibm.com/developerworks/rational/library/2782.html} [59]. That is, the points of method interposition are defined once, at one place, making them easy to understand and maintain.

Using the

---

Figure 7.3: Design of the profiling/learning phase in CAC
terminology of AOP, we refer to an interposition program point as a **join point**, a composition of joint points as a **pointcut**, a set of actions to be executed at a join point as an **advice** and the unit of a program that encapsulates these constructs as an **aspect**.

One of the important properties for adapting existing legacy codes to CAC is that the program modules require no modifications to be advised by the aspects. This property is achieved by a process called **weaving** that occurs at the build or runtime. It adds the ability to replace method bodies with new implementations, inserts code before, after, or around method calls, and, what is most important, associates new state and behaviors with existing classes. Therefore, expressing context dependencies in aspects follows naturally while separating the definitions of join points, the definition of actual context, and the definition of advice.

Figure 7.4 complements the profiling/learning phase presented in Figure 7.3 and shows the AOP-based adaptation to the CAC. In order to model the aspects of the UML diagram, we used the bottom-up approach presented in [52].

The aspect Composition modelled as a class defines the pointcut methodCall and the advice method contextCall. The pointcut picks the set of join points where the set of actions defined by the advice method are executed. In order
to actually perform the advice, the aspect requires a concrete Context class to evaluate the actual context, and a Decision Function class to decide the best-fit variant. The pointcut methodCall as part of the aspect Composition binds the aspect to occurrences of execute operations that occur in some Caller or even recursively in an Algorithm Variant.

7.5.2 Implementation Details

In our implementation, we used Java/JDK 1.6. as the main programming language for both for the AOP adaptation system and for the exiting legacy codes. In the implementation of the CAC aspect (cf. Fig. 7.4), we used AspectJ 1.6.12\(^2\). AspectJ is an AOP environment for the Java language that implements join points, pointcuts, advices, and aspects. At compilation time, the AspectJ compiler inserts the codes defined in an aspect to the existing Java codes and a standard Java compiler compiles these codes to the final class files of the program.

For our experiments, we only used algorithm variants, while neglecting the data representation variants. In order to model the algorithm variants and to minimize the change of legacy codes, we complemented the AOP with used reflective programming, more specifically, the Java Reflection API\(^3\), which allows CAC to accessing class definitions at execution time.

In order to specify the methods to be intercepted and the programming points where the interception has to occur, we exploit Java annotations. We created an @CACMethod annotation by identifying the methods with alternative algorithm variants. During training when the algorithm performance for different actual contexts was measured, the alternative method variants’ objects were obtained by reflection instantiated from algorithm classes containing these methods. Training invoked these methods using invoke of java.lang.reflect.Method.

Join points in AspectJ can be represented by methods, constructor calls, field accesses and class initialization, etc. CAC requires the interception of certain method calls to algorithm variants. These calls can be represented by the set of join points that are eventually picked up by the pointcut methodCall. It is worth mentioning that not every call to an algorithm variant has to be intercepted when measuring the algorithm performance, only the recursive calls, which allows CAC component to select a new best-fit variant because of the new actual context (e.g. a smaller problem size). Accessing the right variant is achieved by the call pointcut that matches all calls from within a method with @CACAMethod annotation.

pointcut methodCall(): call(@CACAMethod * *(..));

\(^2\)http://www.eclipse.org/aspectj/
\(^3\)http://www.ibm.com/developerworks/library/j-dyn0603/
7.6. Summary

@Around(methodCall())
public Object callContext(ProceedingJoinPoint pjp){
(1) Object [] params = context.getCurrentContext(pjp.getArgs());
(2) Method m = context.decider.decide(params);
(3) return m.invoke(getMethodInstance(m), pjp.getArgs());}

An AspectJ advice can be executed @Before, @After or (@Around) a certain join point. The advice contextCall used the @Around annotation. In our case, this decision makes sure that the original method does not call execute at all. The arguments for this method are extracted from the first parameter of the callContext (of type ProceedingJoinPoint) using getArgs. Thereafter, the actual context is received (line 1) and used to determine a best-fit method for this context from a decider instance, in our case, a dispatch table (cf. line 2). The dispatch table was implemented as a simple two-dimensional array capturing the corresponding number of algorithm variant. The aspect aware of all alternative method variants objects and class instances to reflectively invoke the best-fit variant method during the execution of the program. The value returned by the aspect advice is the return value of best-fit method execution and is the return value seen by the caller of the initial method.

7.6 Summary

Context-aware composition allows for the improvement of the performance of applications on modern multi-core hardware by separating the concerns of design, deployment and execution in regard to the component adaptations. It provides reusable, performance-portable systems that select (sequential and parallel) component variants for actual (call and hardware) contexts. Some of these concerns are general regardless of the application domain.

The performance of the context-aware application depends upon the scalability of the CAC applied, which, in turn, depends upon the variants-modelling technique. We enhanced the scalability of the variants-modelling technique by employing a Decision Algebra component that allowed for plugging different decision models, their combinations and exchangeability. Chapter 8 assesses memory consumption and decision time, accuracy, and the speed-ups of the CAC using different decision function implementations.

Manually introducing CAC into legacy applications can be time-consuming since it requires additional efforts in regard to changing application designs and adapting the existing code. To automate this process, we proposed an AOP-based adaptation approach. Due to the strong encapsulation of AOP programs, general CAC concerns can be reused easily for different applications and may also be used in the development of new CAC applications from scratch, allowing developers to focus on their own application design.
and add the CAC aspect later rather than focus on implementation of the CAC design pattern manually. Our main objectives were to impose as little execution overhead as possible and to require as few changes as possible to the (legacy or core) application. These objectives are confirmed by our experiments in Chapter 8.

Moreover, the problem of handling non-functional efficiency requirements during the software development process can be solved using a context-aware recommender system that intends to postpone the architectural decisions and allows designers to focus on the design of the core system functionality. This approach can help improve the software development process (especially in regard to developing computational software) by decreasing the burden on a system designer or a software developer in terms of time and effort spent on the project.
Chapter 8

Experiments over Context-Aware Composition

In the previous Chapter 7 we presented CAC as an application domain that processes decision information with corresponding actual contexts (call, hardware, or locally assemble system contexts) and a set of actual decisions (component variants, such as algorithms or data structures). Moreover, we suggested an approach based on AOP for adapting legacy applications to CAC, such that the context-awareness aspect becomes reusable regardless of the legacy application. To make CAC scale, we suggested the use of DA component which allows for plugging, comparing and combining different implementations of decision function as an alternative to commonly used dispatch table.

This chapter presents two experiments: (1) In Section 8.1 we assess memory consumption and decision overhead gained by replacing dispatch tables (DTB) with a DA component providing several alternative decision models discussed in Chapter 4: decision trees (DTR), decision graphs (DGs), and Naïve Bayesian classifier (NB); (2) In Section 8.2 we assess AOP-based and manual code adaptation approaches by speed-up, and performance overhead of the AOP-based application, and the lines of code required by a programmer. For the first experiment, we also use support vector machines (SVM) as maximum-margin decision model. Moreover, it compares the accuracy of different classification approaches with decision table as the baseline, and finally shows the speed-ups of CAC using these decision models. Finally, Section 8.3 concludes the chapter.

8.1 Evaluation of Decision Models in Contact-Aware Composition

In Chapter 4 we introduced several decision models (tree-based and probability-based) as possible instantiations of DA. We defined their implementations of DA core and non-core operations, and gave a theoretical assessment of memory consumption and decision time. In this section we first shortly in-
introduce a maximum-margin decision model SVM as a possible instantiation of DA, which later on will be used in our experiments. Then, we practically assess memory and decision overhead of these models plugged into the DA component in CAC application. Moreover, we compare the accuracy of these models with decision tables as the baseline. Finally, we show the speed-ups of CAC application with our DA component.

8.1.1 Support Vector Machines

SVM is a maximum-margin decision model that is widely used in pattern recognition [19, 4]. The basic idea of SVM is to find an optimal linear hyperplane in a vector space that can maximize a margin between two groups of decision tuples representing two different classes ($c_1 = -1, c_2 = +1 \in C$). The main principles of SVMs can be found in [4, 40, 111, 114].

SVM can be implemented as an instantiation of DA. Theoretical interpretations of such operations as decide, apply, merge and approx over SVMs are explained in our previous work [21]. Notice, that we didn’t provide a detailed specification as was presented for DGs, trees, tables, and NB, since it requires additional deep research into machine learning algorithms, which is not the goal of this thesis.

However, for our experiments and assessing non-function properties we adapted a LIBSVM [16] library (concrete SVM implementation) to our DA interface. This library is basically a black-box with a radial basis function for the kernel function with normalization and parameter finding suggestions from [48].

8.1.1.1 Non-Functional Properties. SVM can be trained to be very accurate classifier. However, the training time for SVMs can be very long, especially on large sets of decision tuples, as it takes $O(l^3)$ computations, where $l$ is the number of decision tuples. The encoded SVM classifier in the LIBSVM library has a quite high memory consumption for capturing the support vectors, approximated with

$$M = 12s + 24|C| + 4|C|^2 + 100$$

double precision values, where $s$ is the number of support vectors and $C$ is a set of class values.

More importantly, deciding requires an evaluation of the kernel function that takes for the selected kernel approximately $20 + (n + |C|)$ flops for each support vector and a number of additional flops for finding the actual variant, where $n$ is the number of attributes. The number $s$ of support vectors is usually proportional to the number of decision tuples $l$, so we can approximate

$$T(l, n) \approx T(s, n) \approx (20 + n + 2|C| - 1) \times s \times T_{\text{flops}} + s \times (2|C| - 1) \times T_{\text{ass}} + c$$
8.1.2 Experimental Setup

Algorithm Variants: We implemented the well-known sorting algorithms Selection sort, Quicksort, and Merge sort along with two parallel versions of Quicksort and Merge sort, which spawn a new thread for one of the sub-problems in each divide step. We used textbook implementations [18] and did not optimize the variants. In particular, we greedily create new threads regardless of the number of available cores. All algorithms are implemented in Java/JDK 1.6.

Platforms: All practical experiments are executed on three different multi-core machines:

M1 a 2 core Dell Latitude PC running Windox XP (2002, SP 3) on an Intel Dual Core T2300 at 1.66GHz and 2GB RAM,

M2 an 8 core IBM Blade Server running Linux (CentOS) on an Intel 8 Core 5450 at 3GHz and 16GB RAM, and

M3 a 2 core MacBook Pro running Mac OS X (10.6.5) on an Intel Core i5 at 2.4 Ghz and 8GB RAM.

All tests run on the respective native JVMs with -Xms128m -Xmx1g as the virtual machine parameters.

8.1.3 Memory Overhead of the Approaches

To compare different classification approaches, we constructed three Dispatch (decision) Tables for our sorting problem for different multi-core machines M1, M2, M3. The dispatch technique attempts to speed up sorting by selecting the best algorithm (class) \( Y \) for the current context (problem size \( N \), processor availability \( P \)), where \( N \) is a continuous integer sampled at powers of two between \( 2^0 \ldots 2^{16} \), \( P \) is boolean with 0 and 1 as possible values (encoding whether or not processors are available). \( Y \) is a discrete integer in the range 1 \ldots 5, each representing one of the algorithm variants. The memory needed for storing 34 (\( 2 \times 17 \)) entries is rather small, i.e., \( M = 4 \times 34 = 136 \) Bytes, cf. Table 8.1 for the size of the DTable and the alternative classifiers constructed.

To encode a DTB in a DTR or a DG, we used the FC4.5 learning algorithm. DTRs have a moderate learning effort. The memory compression rate is quite high: \( \approx 82\% \) for tables M1, M2 (the number of edges equals 6, requiring 24 Bytes) and \( \approx 71\% \) for M3 (edges = 10, 40 Bytes).

The DGs are a redundancy-free encoding of the DTRs and, hence, the depth does not change. However, DTB compression does not improve compared to DTRs (the number of edges is the same). In our example, the graphs use up to 27% fewer nodes than the tree.
Table 8.1: Memory overhead of different classification models.

<table>
<thead>
<tr>
<th>Model</th>
<th>in Bytes</th>
<th>SVM</th>
<th>in Bytes</th>
<th>Bayes</th>
<th>in Bytes</th>
<th>Graph</th>
<th>in Bytes</th>
<th>Tree</th>
<th>in Bytes</th>
<th>Table</th>
<th>in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>140</td>
<td></td>
<td>5</td>
<td></td>
<td>2.5</td>
<td></td>
<td>2.5</td>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td></td>
<td>40</td>
<td>0</td>
<td>40</td>
<td>0</td>
<td>40</td>
<td></td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 8.2: Decision overhead of different classification models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Array access</th>
<th>SVM</th>
<th>Operation</th>
<th>Bayes</th>
<th>Operation</th>
<th>Tree</th>
<th>Operation</th>
<th>Table</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flops</td>
<td>23</td>
<td>2.5</td>
<td>140</td>
<td>140</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.1. Evaluation of Decision Models in Contact-Aware Composition

The NB was the fastest to construct and the encoded DTBs take only 80 Bytes giving a 41% of reduction immediately without any additional optimizations, since the size of the classifier only depends on the context attributes, not their values.

In order to construct an accurate prediction model based on Support Vector Machines, a learning phase requires an optimization of kernel function parameters specific to each problem domain and input sample data. Although done automatically, finding parameters giving an acceptable decision accuracy requires a cross-validation pre-learning over the large range of parameter values. In our example, (pre-)learning time was still acceptable in range of a few seconds. The memory required to encode the SVM classifier is 544 Bytes (based on the LIBSVM library implementation). This is the highest memory overhead and even 75% larger than DTB memory consumption.

8.1.4 Decision Accuracy of the Approaches

In this section, we compare the accuracy of the different decision approaches. As we do not know the right decision for each actual context (processors available and problem size), but only for the sample points measured in the training phase, we can assess accuracy only approximatively: (1) by comparing the decisions of the different approaches at these sample points, and (2) by comparing their decisions with the decisions of the table approach as the baseline. We define a decision error as (1) the ratio of decisions not suggesting the best algorithm variant in the sample points over all decisions, and (2) the ratio of decisions diverging from the suggestion of the decision table over all decisions. The DTB captures the best implementation for the sample points. Hence, its error is 0 in the measure (1). However, we do not know if the DTB suggests the best variant between sampled problem sizes. Hence, (2) is an accuracy measure relative to the baseline implementation of Context-Aware Composition using DTBs.

For the different platforms, Table 8.4 shows the error (1) at the sample points and the error (2) for selected sizes of arrays to sort. All ratios are given in %.

DTBs (trivially), DTRs, DGs, and SVMs (cf. rows “Sample points”) are to 100% accurate according to measure (1) and, generally, all decision approaches perform very accurate at the sample points with an error of at most 15%.

The error is somewhat higher for the accuracy measure (2). Independent of the problem size, DTRs and DGs suggest in 21% to 25% of the cases different algorithm variants than the DTB (cf. columns “Tree” and “Graph”). For the NB and SVMs, the error (2) increases with the problem size. However, it is in most cases smaller than for DTRs and DGs. Note that the error of 100% for the NB for the very large problems is due to its first (wrong) decision in
Chapter 8. Experiments over Context-Aware Composition

<table>
<thead>
<tr>
<th>Platform</th>
<th>Problem</th>
<th>DTR</th>
<th>DG</th>
<th>NB</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>in %</td>
<td>in %</td>
<td>in %</td>
<td>in %</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>Sample points</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>21</td>
<td>21</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>21</td>
<td>21</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1,000,000</td>
<td>21</td>
<td>21</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>Sample points</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1,000,000</td>
<td>25</td>
<td>5</td>
<td>100</td>
<td>25</td>
</tr>
<tr>
<td>MAC 2 cores</td>
<td>Sample points</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>21</td>
<td>21</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>21</td>
<td>21</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1,000,000</td>
<td>21</td>
<td>21</td>
<td>100</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 8.3: Errors of different decision approaches.

As a conclusion, none of the approaches ought to be dropped from further evaluations since their accuracy is comparable with the accuracy of decision tables.

8.1.5 Decision Overhead of the Approaches

In this sub-section, we compare the overhead of the different approaches for decision making. We compare them first based on our theoretical assumptions and then experimentally.

For the 2-dimensional DTB based on one continuous and one discrete attribute, the prediction time includes 23 floating point operations and one array access, that is $T_{\text{flops}} \times 23 + T_{\text{aa}}$. It is worth mentioning that in our cost model a look-up for any 2-dimensional DTB is constant and does not depend its number of entries, which is a simplification ignoring caching effects. Table 8.2 shows the floating point operations and the number of array accesses for the classifiers.

The decision time required by DTRs and DTRs depends on the trees and graphs depth, resp. For each of the three DRTs (DGs) constructed, the decision requires at most three indirect array accesses, since the maximum depth is three for all trees (the average depth is 2.5). Thus, the expected overhead can be estimated as $T_{\text{aa}} \times 2.5$.

As discussed, a NB is quite effective in terms of memory consumption, but on the downside, it has quite a high runtime overhead for making decisions.
In our setting, it requires 46 floating point operations and 5 array accesses, 
\(T_{flops} \times 46 + T_{aa} \times 5\). This is a factor of two when it comes to flops and a factor of five when it comes to array accesses, compared to DTBs, which makes it a slow classifier for our problem.

A decision in SVM requires 21 flops that correspond to a kernel function computation calculated for each of the \(l = 28\) support vectors and then additional 56 flops for classification. Altogether, the decision phase takes 644 flops and the whole decision time is \(T_{flops} \times 644 + T_{aa} \times 140\). This is the highest look-up overhead among all classification models.

Experiments confirm these overhead predictions. For these experiments, we use the fastest homogeneous solution – Quicksort – as the baseline. The third column of Table 8.4 shows the time for Quicksort for the different platforms and three selected array sizes to sort.

On each recursive invocation of Quicksort (with a certain number of processors still available and a certain sub-problem size), we look up the best algorithm variant in the different decision repositories. However, in order to compare the decision overhead in a fair way, we always invoke Quicksort regardless of the best algorithm variant suggested. Hence, all compared solutions essentially implement Quicksort.

Table 8.4 shows the execution times of the Context-Aware Composition based on the different decision functions relative to the execution time of Quicksort on the same platform and architecture. For instance, the DTB (cf. column "Table") introduces an overhead of 43% on the PC with 2 cores and the problem size of 10,000 array elements, as it requires \(1.43 \times 3.62\)msec = 5.18msec of the corresponding Quicksort execution times without table lookup (3.62msec).

There are \(O(N^2)\) expected lookups for problems of size \(N\) in Quicksort; the expected work of Quicksort is \(O(N \log N)\). The lookup time is \(O(1)\); it only depends on the number of algorithm variants, the number of context attributes, and the decision repository variant, but it does not grow with the problem size. Hence, for all decision repository variants, the overhead gets smaller with increasing problem size. This is confirmed by our measurements. The DG (column “Graph”) introduces the lowest overhead in almost all cases, with the DTR (column “Tree”) not far behind. These overheads are between 6% and 36% depending on problem size and platform. The DTB comes with an overhead between 15% and 55%. Algorithm variant selection using NBs and SVMs (columns “Bayes” and “SVM”, respectively) slows down the execution by factors between 2.4 and almost 42.

Quicksort (and other recursive sorting algorithms) are extreme in the sense that the ratio between decision points (recursive calls) and workload is rather high. Therefore, we observed a rather high overhead for context-aware composition compared to the homogeneous Quicksort variant. In conclusion, DTBs, DTRs, and DGs introduce an overhead that is still accept-
Table 8.4: Time overhead (in %) of different decision approaches.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Problem size</th>
<th>Quicksort</th>
<th>Graph</th>
<th>Bayes</th>
<th>SVM</th>
<th>Bayes</th>
<th>Graph</th>
<th>Quicksort</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores</td>
<td>10,000</td>
<td>3.62%</td>
<td>25%</td>
<td>22%</td>
<td>11%</td>
<td>1.87%</td>
<td>34%</td>
<td>42%</td>
<td>13%</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>100,000</td>
<td>51.91%</td>
<td>28%</td>
<td>14%</td>
<td>10%</td>
<td>553%</td>
<td>29%</td>
<td>30%</td>
<td>12%</td>
</tr>
<tr>
<td>MAC 2 cores</td>
<td>1,000,000</td>
<td>998.36%</td>
<td>15%</td>
<td>8%</td>
<td>6%</td>
<td>423%</td>
<td>25%</td>
<td>16%</td>
<td>15%</td>
</tr>
</tbody>
</table>
Figure 8.1: Homogeneous Quicksort and Context-Aware Sorting using DGs (“Opt Graph”) and DTables (“Opt Table”). The x-axis displays the array size, the y-axis the time in msec.
able as it can be expected that a clever algorithm variant selection compensates for the overhead. At least for recursive sorting problems, this is not the case for NB and SVM.

As conclusion, we disregard NB and SVM as decision repositories for improving sorting using context-aware composition due to their (too) high decision overhead.

As DTRs and DGs by construction always suggest the same variant, and DGs are smaller in size and have a slightly smaller overhead, we disregard DTRs in the final overall assessment.

### 8.1.6 Overall Performance

Now we are ready to assess the overall performance of context-aware composition using DGs vs. DTBs. Figure 8.1 shows the experimental results on the different platforms. As a reference, it also shows how the fastest homogeneous implementation variant (sequential Quicksort) performs on the three platforms.

On the PC with two cores M1, the optimized version using DGs gains a speed-up of 1.47 over sequential Quicksort, on average over all array sizes from 10,000 – 1,000,000 (step 10,000), while the optimized version using DTBs gains a speed-up of 1.46. On the server with 8 cores M2, the difference between the two implementations is even more pronounced: an average speed-up of 1.92 for the graph-based solution vs. 1.79 for the table-based solution. On the MAC with 2 cores M3, the speed-up results are 1.39 vs. 1.37, again in favor of the graph-based solution.

Obviously, the decisions of the DTBs are (slightly) more accurate, but this is more than compensated by the lower runtime overhead of the Decision graphs. Altogether, the experiments showed that the DG is not only smaller by a factor of five than the DTB, but also (slightly) faster when used as a decision repository in Context-Aware Composition.

### 8.2 Context-Aware Composition with AOP

For this experiment, we picked two applications: Sorting and Matrix-Multiplication. For both, implementations with the required object-oriented design as well as manual CAC implementations existed already.

Algorithm variants use textbook implementations [18] and we did not further optimize them. Especially, parallelization of these algorithms did greedily create new threads when admitted by the essential program dependencies regardless of the actual number of cores available. Sorting comes in the well-known variants of Selection Sort, QuickSort, and MergeSort and two parallel versions of QuickSort and MergeSort. The latter fork a new thread.
for one of the two recursive sub-problem calls in each divide step. Matrix-Multiplication implementation variants include the classic based on three nested loops (referred to as Product Inlined later on), a variant reducing the problem to matrix-vector and then further to vector-vector multiplications, a recursive variant based on eight multiplications of sub-matrixes of one quarter the size of the original, and the famous Strassen algorithm with only seven sub-matrix multiplications. Additionally, the recursive variant was parallelized; it forks new threads for seven of the eight recursive sub-matrix multiplications in each recursion step.

Sorting is an extreme problem for CAC in the sense that the ratio of dynamic variant selection and payload is rather high. Therefore, the performance overhead in this application gives insights on expected upper bounds of this overhead in more common cases like Matrix-Multiplication.

All experiments are executed on two different multi-core machines (M1, M2) on their respective native JVMs with virtual machine parameters -Xms40m -Xmx384m.

M1 is a 2 core Fujitsu Siemens Esprimo Mobile v5515 PC running Windows XP (2002, SP 3) on an Intel Dual Core T5300 at 1.73GHz and 1.75GB RAM.

M2 is an 8 core Server Dell Precision WorkStation T7400 running Windows 7 Enterprise 32-bit on an Intel 8 Core Xeon E5410 at 2.33GHz and 8GB RAM (3GB RAM usable).

8.2.1 Overall Performance

In this section we compare the overall performance of the CAC applications using AOP-based and manual-based adaptation approaches. In the deployment phase, we constructed DTBs for our sorting and matrix-multiplication problems for the different multi-core machines (M1, M2). The formal contexts are problem size $N$ and cores availability $P$. For sorting, the actual context of $N$ was the array size sampled at powers of two between $2^0$ ... $2^{16}$; For matrix-multiplication, the actual context of $N$ is the number of rows (columns) of the (square) matrixes and sampled 1, 16 up to 256 with step 16th. For both problems, the actual context of $P$ is a Boolean encoding whether or not a free core is available. For both scenarios the formal decision is an algorithm (sorting or matrix-multiplication) to be applied. Therefore, actual decisions will be such algorithms as Selection Sort, QuickSort, MergeSort, Product Inlined, etc., that were mentioned above.

For a fair evaluation, we compared the performance with the manual approach using both the DTB produced by aspect-oriented CAC Manual Adaptation (Aspect) and the DTB generated by manual CAC Manual Adaptation (Aspect) and with pure automated AOP-based approach Aspect. In
Table 8.5: Speed-up of different CAC approaches for Sorting.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores</td>
<td>50.000</td>
<td>34.5</td>
<td>1.46</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>500.000</td>
<td>661.1</td>
<td>2</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>1.000.000</td>
<td>1531.2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 8.6: Speed-up of different CAC approaches for Matrix-Multiplication.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 8 cores</td>
<td>100</td>
<td>294.9</td>
<td>1.07</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>500</td>
<td>41119</td>
<td>2.97</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>800</td>
<td>172820</td>
<td>3.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores</td>
<td>100</td>
<td>198.8</td>
<td>2</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>500</td>
<td>29276</td>
<td>2.98</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>800</td>
<td>124722</td>
<td>3.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 8 cores</td>
<td>100</td>
<td>21.7</td>
<td>1.2</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>500</td>
<td>41119</td>
<td>2.97</td>
</tr>
<tr>
<td>Server 8 cores</td>
<td>800</td>
<td>172820</td>
<td>3.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores</td>
<td>100</td>
<td>21.7</td>
<td>1.2</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>500</td>
<td>29276</td>
<td>2.98</td>
</tr>
<tr>
<td>PC 2 cores</td>
<td>800</td>
<td>124722</td>
<td>3.4</td>
</tr>
</tbody>
</table>
### 8.2. Context-Aware Composition with AOP

In the experiments, we compare performance with the fastest context-unaware solutions: the sequential QuickSort and ProductInline, respectively. The third column of Tables 8.5 and 8.6 shows the time for QuickSort and ProductInline, resp., for two platforms and three selected problem sizes. The columns 4–6 show the speed-up of the CAC approaches relative to the execution time of QuickSort and ProductInline, resp., on the same platform and architecture. For instance, in Table 8.5 *Aspect* gives a speed-up of about 1.46 on the PC with 2 cores and a problem size of 50.000 array elements.
Chapter 8. Experiments over Context-Aware Composition

as it requires only 34.5\text{msec}/1.46 = 23.6\text{msec} of the corresponding QuickSort execution time. The speed-up can be higher then the number of processors, due to the applied context-awareness, where the best-fit algorithm is selected for the current set-up (problem size and number of processors available).

Altogether, for sorting, the pure AOP-based approach gives a speed-up of 1.2 and 2.2 depending on the problem size and the platform. Manual adaptation (Aspect) has a bit higher speed-up that is between 1.5 and 2.3. Manual Adaptation (Manual) provides the highest speed-up, a factor of up to 2.8. Figure 8.2(a) shows the experimental results from the sorting problem on the two platforms over a wider range of problem sizes. On the PC with 2 cores, the AOP-based approach has an average speed-up of 2.05 over sequential QuickSort over all array sizes from 50.000 – 1.000.000 (step 50.000). The manual approaches have a speed-up of 2.17 and 2.5 for aspect and manually constructed DTB, respectively. For 8 core Server, the difference between AOP-based and manual adaptation is even smaller: an average speed-ups of 1.6 versus 1.7 and 1.9, resp.

For Matrix-Multiplication, Table 8.6 shows that AOP-based adaptation speeds up the baseline by factors of between 1.07 and 3.4 for the 2 cores PC and the 8 cores Server, respectively. Similar to sorting, the manual approach are slightly faster and speed-up the baseline by factors up to 3.5. Figure 8.2(b) shows again the performance for a wider range of problem sizes. The average speed-up of the AOP-based approach over matrix sizes of 1 – 800 rows (step 50) is on average 1.74 and 2.5 for M1 and M2, respectively. However, unlike in sorting, the difference between this AOP based and and manual approaches is minimal: the Manual Adaptation (Aspect) has an average speed-up of 1.76 and 2.61, and Manual Adaptation (Manual) an speed-up of 1.9 and 2.6 for M1 and M2, resp.

8.2.2 Performance Overhead

The differences in the speed-ups of AOP and manual approaches can be explained by the additional overhead due to AOP and reflection of the AOP-based approach. In order to measure this overhead in a fair way, we compared the execution time of CAC using the AOP-based approach with the execution time of the manual approach where both use the same DTB. Hence, both approaches select exactly the same variants all the time.

For sorting, the AOP-based approach introduces a average overhead of 5% and 9% over the corresponding time of Manual adaptation (Aspect) approach for the 2 core and 8 core machines, respectively, as it requires on average 1.05 and 1.09, respectively, of the corresponding Manual adaptation (Aspect) execution time. The overhead is considerably smaller for Matrix-Multiplication; on average about 3% for M1 and 2% for M2.

The overhead of the AOP-based approach is the result of reflection calls
and aspect call interceptions. For sorting problem, this happens frequently in the recursive algorithms (QuickSort, MergeSort and their parallel implementations). These algorithms are extreme in their ratio between decision points (recursive calls) and workload. This explains the difference in the speed-up of the AOP-based and the manual approaches. In contrast, Matrix-Multiplication shows a rather low overhead since. In the recursive algorithms, the ratio between decision points and workload is rather low as can be expected from many other applications.

In both example application the overhead of CAC is more than compensated by the speed-up compared to the context-unaware variants.

### 8.2.3 Lines of Code

The slightly higher performance of the manual CAC approaches are paid by a higher programming effort. In order to measure this programming effort required to adapt a given legacy application to CAC, we used the lines of code (LOC) metrics. It measures the number of lines a program's source code. Specifically, we measured LOC required to manually adapt to CAC, referred to as $LOC_M$, and the size of the code required for the AOP-based adaptation, referred to as $LOC_A$. The manual adaptation may reuse some parts of the existing legacy application and change only some lines of the code. $LOC_M$ counts only lines that require additional changes. The AOP-based approach can reuse the code of the aspects implementing general CAC concerns. $LOC_A$ counts only the code to be written for the application-specific CAC concerns, cf. Table 7.1. Finally, the programming effort improvement due to the AOP-based approach is simply assessed with a metrics $PI = LOC_M / LOC_A$.

For the sorting problem, $LOC_M = 154$ and $LOC_A = 115$ leading to quite a productivity improvement of $PI = 1.4$. For Matrix-Multiplication, $LOC_M = 383$ and $LOC_A = 134$ leading to $PI = 2.8$.

However, it is worth pointing out that the code reused in manual adaptation is actually a duplication of existing legacy code and the number of lines of this code is rather high. It can even require some time and effort to identify which lines have to be changed for CAC adaptation. Therefore, the LOC metrics used above becomes unfair when it comes to measuring of the locality of changes for further maintenance of the application code. If to calculate all the lines of code to be added by manual adaptation it even becomes up to four times more then for AOP-based adaptation.

### 8.2.4 Data representation Conversion

As was mentioned in Section 7.3, the variant components may also include data representation. For instance, for the sorting problem possible data rep-
Chapter 8. Experiments over Context-Aware Composition

representations are Arrays, Lists, Collections, etc. This also influences the overall performance of CAC application.

Another set of experiments, that includes data representation variants, was conducted separately by a our master student. Therefore, implementation, design and configuration of profiling, learning and composition blocks can be found in the master thesis [98]. Here we only present the experiment results which confirm our observations made in the previous Section 8.2.1.

The experiment takes Matrix Multiplication problem (discussed above), and is executed on two different multi-core machines: (M1) a 2 core Samsung PC with Ubuntu 12.10, Intel Dual Core i5 M560, 2.67GHz, 4GB RAM; and (M2) 4 core MacBook Pro, OS V 10.8.2, Intel Dual Core i7, 2.3 GHz, 8GB RAM. Notice, that the experimental set-up differs from the one given above, therefore, the experiment results cannot be simply compared.

For this problem, five previous algorithm variants were taken, and are used with different data representations, called dense matrices and sparse matrices. Dense matrix has the ratio of non-zero elements of $O(N^2)$ and sparse matrix has ratio of $O(N)$. This gives us four representation variants of multiplication of two matrices: (R1) first and second are dense matrices (dense-dense), (R2) sparse-sparse, (R3) sparse-dense, (R4) dense-sparse, resp. The context parameters for this application are size of the matrix, and processors available.

The performance of AOP-based approach with exchangeable data representation (Aspect (representation)) was compared with previous AOP-based approach with non-changeable representation (Aspect Adaptation), with manual approaches that use the DTable generated by manual CAC and with the fastest context-unaware solution ProductInline. Also, manual adaptation is two-fold, the first Manual doesn’t take into account different variants of data representations, and the second Manual(representation) makes an adaptation with data representation variant. Both Manual and Aspect Adaptation use dense-dense data representation variant.

Table 8.7 shows the speed-up of AOP-based adaptations in comparison with ProductInline algorithm.

In terms of lines of code for adaptation with data representation: $LOC_m = 589$, $LOC_a = 175$, thus $PI = 3.3$. 

136
<table>
<thead>
<tr>
<th>Platform</th>
<th>Problem size</th>
<th>Inlined in msec</th>
<th>Manual</th>
<th>Manual (representation)</th>
<th>Aspect</th>
<th>Aspect (representation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 2 cores (M1)</td>
<td>176</td>
<td>150</td>
<td>1.01</td>
<td>1.7</td>
<td>1.01</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>224</td>
<td>389</td>
<td>1.4</td>
<td>2.0</td>
<td>1.4</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>501</td>
<td>1.3</td>
<td>1.9</td>
<td>1.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Mac 4 cores (M2)</td>
<td>176</td>
<td>196</td>
<td>1.8</td>
<td>2.9</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td></td>
<td>224</td>
<td>295</td>
<td>1.4</td>
<td>1.8</td>
<td>1.4</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>345</td>
<td>1.2</td>
<td>1.6</td>
<td>1.2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 8.7: Speed-up of CAC approach with Exchangeable Data Representation for Matrix-Multiplication.
Altogether, the experiments showed that our approach can effectively adapt existing legacy codes to CAC which makes them running efficiently even on multi-core machines. Even though the speed-ups of applications with CAC adapted manually than with the AOP-based approach, the performance overhead quite small. Moreover, AOP-based adaptation requires a smaller programming effort and the resulting systems still significantly outperform the fastest heterogeneous applications. Based on these observations, we claim that the AOP-based approach can be widely used by software developers to adapt legacy applications to CAC.

8.3 Summary

In the first part of this chapter we evaluated our AOP-based approach for introducing CAC to the legacy applications. It was evaluated on Sorting and Matrix-Multiplication applications in terms of performance and programming effort relative to the manual adaptation approach. The experiments showed that applications derived by our AOP-based adaptation speeds up the execution time by factors of up to 2.2 and 3.4 for Sorting and Matrix-Multiplication, respectively, on multi-core machines with two and eight cores. The application manually adapted and optimized to CAC achieved a slightly higher speed-up (2-9% higher). However, development and maintenance effort for the manual adaptation is higher than for the AOP-based approach. Manual adaptation requires almost three times the code of the AOP-based approach and is additionally spread throughout the application and not nicely encapsulated as in the AOP-based adaptation. These results suggest that an AOP-based adaptation to CAC can effectively improve the performance of existing applications with a moderate transition effort.

Finally, the second part of this chapter contributes with:
8.3. Summary

- A platform for plugging classifiers into Context-Aware Composition. It was instantiated with 5 classifier variants: DTBs (baseline), DTs, DGs, NB and SVMs.

- A practical assessment on three different hardware platforms using sorting as a running context-aware composition example.

DGs introduced the lowest overhead in almost all cases for our sorting problem (different problem sizes and hardware platforms). These overheads are between 6% and 36% depending on problem size and platform. On the other hand, SVM and NB slow down the execution by factors between 2.4 and almost 42 due to the high decision overhead. For the sorting problem in context-aware composition, DGs reduced the memory consumption by a factor of five and increased overall performance by a few percent. Therefore, this experiment also suggests DGs to be a preferred implementation of decision functions.
Chapter 9

Related Work

This chapter expounds upon worked related to this thesis. Section 9.1 summarizes the current research related to the generalization and efficient implementation of decision models. It gives an overview of the approaches used that were similar to our decision graphs implementation in Section 9.1.1 and, in Section 9.1.2, gives an overview of the various methods used to combine different decision models.

Section 9.2 discusses related work for using Decision Algebra in context-aware composition applications and for adapting legacy codes to context-awareness.

9.1 Decision Algebra

To the best of our knowledge, no common unifying theory exists that specifies the common abstract decision model that allows for knowledge combination and reuse between different application domains. Although there exists a variety of systems that provide analysis engines used to capture and process large volumes of decision information, they are usually available as stand-alone applications for data analysis and as data mining or machine learning engines that can be integrated into the third-party applications. Examples of such tools are Weka Toolbox [39] and Rapid Miner [46]. Weka is a widespread collection of machine learning algorithms for data mining analysis. Rapid Miner is a commercial system for data and text mining that contains methods for web mining, opinion mining and sentiment analysis. The main scientific benefit of such systems is the clean, object-oriented class hierarchy that provides the common general interface from which different decision models can be instantiated. Such an interface specifies the operations necessary for data processing, including the routines that generate a decision model from decision information, and to test it on an unseen dataset. The level of abstraction of such tools is quite low and, even though each tool introduces a general implementation interface, the tools still vary in regard to their sets of operations, which are usually limited to the construction and evaluation of the decision model. The combination of decision models, learning (approximating) them using different algorithms (e.g., learning of
decision trees learns Naïve Bayesian classifier) are not supported by these tools, which are required for introducing Composability (1.3).

In contrast, our Decision Algebra is a theoretical framework that provides a high level of abstraction, which allows for combining, learning and manipulation of decision models regardless specific implementation details. Therefore, tools, such as Weka or Rapid Miner, could instantiate our framework with efficient implementations for different decision models.

At the same time, there exists a great variety of learning algorithms, data structures and applications that capture classification information (see Section 2.1). Generally, a modification of these data structures comes with a modifications of a corresponding learning algorithm. In this related work we will address the decision trees and decision tables and their modifications (category (DM1)), since they come out as a natural representation of our Decision Algebra (see chapter (Chapter 5).

Several approaches to the so-called fragmentation problem, a result of replications as discussed in [81]. One suggested approach uses decision tree nodes to switch on combinations of attributes. For instance, Lam and Lee [65] presented a method for building classification models by using correlation analysis of attributes (identifying so-called functionally dependent attributes). Similar ideas are presented in [84, 51]. Vilalta et al. [109] investigated top-down decision tree construction and proved theoretically and empirically the significance of the fragmentation problem in the learning process. In order to overcome this problem, they choose the best out of a number of possible attribute orderings by assessing their results against all of the training examples and, thereby, avoided a misclassification of examples, for which little support is found.

Friedman et al. [35] presented lazy learning, an algorithm that tries to construct the best decision tree for a given decision domain by basically keeping the information of each training instance. However, this algorithm requires a lot of memory when using the classical decision tree structure. A solution that avoids storing redundant information might be more memory efficient.

Oliver [82] presented decision graphs as modifications of decision trees. However, Oliver had to invent a completely new learning algorithm. This approach might benefit from the general decision model, which is independent from the learning algorithm and, thus, can substitute decision trees as used by any tree construction algorithm.

Stiglic et al. [99] presented an extension of the machine learning algorithm J48 implemented by Weka and a study on the visual tuning of decision tree classifiers. By constraining certain parameters used in the construction process of decision trees (e.g., a minimal number of objects in a single leaf), their final decision tree obtained good performance in terms of size and complexity. However, the tuned tree constructed by Stiglic et al. can be effective
only in cases where the number of classes is not high. Thus, it can benefit from a more memory efficient decision model that can be substituted for decision trees and, at the same time, keep its properties. In contrast, our approach performs the compression of decision graphs during its construction process and shows its applicability for datasets with more than two classes.

Quinlan [87] merged different decision trees and extracted proposition rules from an already generated decision model in order to eliminate unused conditions replicated in different paths of the tree. Sets of decision rules for the same data domain are merged in order to increase the accuracy of a classifier. However, the rules have to be extracted from the decision trees and their merger has to be implemented somehow (e.g., in a decision tree again). It might be more efficient to apply a more general merge operator directly to the decision trees or graphs. On the other hand, such an action is not a guarantee of an improvement in accuracy.

Perner [85] addressed the issue of the comparison of decision trees that represents the classification models of the same problem domain. It arises when two different data sets for one problem are available or when the data set is collected in temporal sequence. In order to compare different decision trees, the author proposed an approach of decision rules extraction followed by computing the similarity measure between several sets of rules. In fact, this study requires a general representation of decision models with general properties that can allow for identifying similarities in model implementations. In contrast to our work focusing on the unification of operations over different decision models, Perner put more effort into identifying the specific steps for classifiers having a concrete tree representation.

Bohatec et al. [5] addressed the problem of simplifying decision trees, possibly at the expense of accuracy, so that the simplified decision tree still represented the problem domain "sufficiently" well. The chosen simplification method was tree pruning, where the approach was to find the smallest pruned tree with some specified accuracy rate. This goal was achieved by generating a dense sequence of the pruned trees, decreasing in size, where each tree had the highest accuracy among the pruned trees with the same size. Bohatec et al. traded accuracy for the simplicity of a decision model (represented in their case as a decision tree). However, they developed a new algorithm, which had to be an add-on to the current pruning algorithm, and specified a certain accuracy threshold. In order to resolve this issue, one needs an approach that does not require any adopted pruning algorithms and is executed during the learning process with the parameter of the required depth of the resulting tree. This approach can be achieved by having a generalization of decision models with certain properties, which reduces memory consumption.

A theoretical framework used for system model checking of non-finite aspects of a system was presented by Mokhtari et al. [76]. Similar to ours, it
Chapter 9. Related Work

is based on higher order functions for defining Multiway Decision Graphs (MDG). The goal was to overcome the OBDD binary representation limitations for a certain class of many-sorted first-order logic formulae. Essentially, it is a BDD generalization with signatures for MDG construction, evert, merge and pruning operations; however, it was tailored to applications in system model checking.

In addition, there are many approaches suggesting different modifications of to decision trees and tables data structures [53, 38, 101, 24, 80].

9.1.1 Decision Graphs

The idea of DGs is not new. Several studies have presented DGs as an alternative decision model that overcomes several issues of decision trees. For instance, Oliver [82] presented DGs as being similar to $\chi$-terms as a modification of decision trees. However, Oliver had to invent a completely new learning algorithm that built a decision graph based on the so-called Minimum Description Length Principle, which minimized the number of encoding bits used for graph implementation.

Another type of DGs was presented by Jensen et al. [50]. These DGs were built on the concept of the Bayesian Networks, which form the basis for the so-called Influence Diagram Framework. This framework denotes a number of properties for DGs, where the core conception for all properties is based on the approach of collapsing identical sub-trees in decision trees [21]. The identity of sub-trees in Influence Diagrams is defined similarly to the identity of decision functions: if both sub-trees give the same decision in the end, then they are pronounces to be the same regardless of the order of the attributes. However, the purpose of introducing DGs is to build normative systems, namely Decision Support Systems. Hence, the Influence Diagram Framework is concentrated on solving problems related to normative systems and is rather heavy in regard to applying it to different applications dealing with simple classification tasks.

Finally, our decision graphs are a generalization of $\chi$-terms [107], capturing context-sensitive program analysis results, and Ordered Binary Decision Diagrams (OBDDs) [12, 11] representing propositional logics formulas in a compact way. $\chi$-terms define merge and approx but not evert, while OBDDs (and their generalizations to multi-valued and multi-target decision diagrams) lack a natural definition of merge and approx.

9.1.2 Combining Decision Models

Various methods of combining classifiers, often referred to as base-level classifiers, have been proposed [113, 27]. However, most of them combine the actual results of the classification rather than producing a new, possibly more accurate, classifier capturing the merged classification information.
There is, for example, the commonly used approach to combining classifiers called aggregation that weights several individual classifiers and then combines their predictions to obtain a new classifier [96, 32, 49, 62]. The aggregation can be complete by the majority voting, where each base classifier has some vote weight and the final prediction is the result of the voting, or by the aggregation of the class-probabilities of all of the base-level classifiers [26, 64]. Sometimes, the ensemble of base classifiers can be chosen by a special selection method [13].

However, most of this research is concerned with using a single classification approach [27], such as a decision tree [106, 45] or neural network training [17]. For instance, the combining method called Decision Forest [44, 106, 45, 9] combines a set of decision trees learned over the distinct samples of one data set with completely disjointed formal contexts. The final classification is usually done using a similar merge operator over the class-probability predictions of all of the classifiers (see Section 4.3). The best accuracy for this method is archived when the data set involves many redundant attributes. As the forest grows (in regard to the number of trees), the classification efficiency decreases, which is not a problem in our case.

Similar to our combining approach, the aggregation approaches combine the classifiers predictions and concern the disjointed set of formal contexts. However, they are less general since they use only one classification approach for base-level classifiers.

The major drawback of the aggregation approach is that it uses a fixed combining rule and a complex multi-classifiers structure, which may produce inferior classification performance in terms of accuracy and classification time compared to the best classifier in the ensemble [73]. Therefore, a balance between the diversity and accuracy of the combining method has to be found [64].

Another approach to combining classifiers is a form of meta-learning, called stacking, which constructs a new classifier over the predictions of a series of base-level classifiers [73, 115, 74, 103, 15]. The input to the meta-model is the discrete decision tuples output from the given classifiers and the meta-features of these classifier outputs (e.g., properties of the class probability distributions, such as entropy or the maximum class probability). A set of aggregated features is often computed for each instance in the data set.

For instance, similar to our approach, Todorovski et al. [105] introduced meta decision trees (MDT) for combining multiple classification models. MDT specifies which classifier should be used for classification in an actual context. The final decision is made based on the class probability distribution predicted by base-level models. MDT are understandable as ordinary decision trees and are usually small so that they can easily be inspected. Another work presented by Menahem et al. [73] provided a meta-learning
algorithm for combining one-class classifiers, which, similar to the decision function, processes them as black-box components. It extracts the meta-attributes from the outputs of the base-level classifiers and assigns them to each decision tuple of the given classification information. Similar to our method, meta-learning takes into account properties of the classifiers they combine. However, they are usually designed for combining classifiers with the same formal context and rely on new (offline) learning of data sets.

Aggregation and meta-learning approaches, in principle, pursue a different aim that we did as they aim for more accurate classifiers learned from data sets, while we aimed for the re-use of classifiers when data sets are not available (with a result classifier that is more accurate than the input classifiers). The majority of these approaches are designed for combining different classifiers learned over samples of the same classification problem or constructing a single classifier learned on different variations of the same classification problem [94, 8], therefore, prohibiting that the contexts can differ.

To the best of our knowledge, the work on combining classifiers learned from different data sets with partly or completely disjointed contexts mostly focuses on the aggregation approach [106, 45, 17] only using one classification approach for base-level classifiers. Meta-learning approaches, in such scenarios, need to be rebuilt using newly collected classification information that becomes expensive or impossible in many real world applications. Moreover, these approaches may lead to drawbacks in accuracy and diversity since most do not directly exploit the internal information captured in the models (e.g., the exact distributions and probabilities).

9.2 Decision Algebra in Context-Aware Composition

Context-aware composition is gaining importance as a tool for performance optimization, in particular since the stagnation of CPU clock rates puts an urgent need to exploit new sources for performance improvements. In general, context-aware computing allows applications to change depending upon the location of the user, hosts and accessible devices [95]. Variants differ in the applied problem domain and in the applied composition technology. Problem domains include mobile, Web, e-services and applications [69, 25, 3]. Context-awareness can be achieved with Context-Oriented Programming (COP) [110, 43, 20]. CAC aims at dynamically optimizing applications in changing call contexts and available resources in the system environment [1, 56, 58].

The optimization of domain-specific libraries for, for example, linear algebra or signal processing is a natural target for optimized composition,
which is also referred to as autotuning. As the domain and code base is limited and statically known, computations can often be described in a restricted domain-specific language from which variants can be generated and tuned automatically. Well-known examples include the library generators ATLAS for basic linear algebra computations and FFTW and SPIRAL [78, 79] for transformations in signal processing. Even though these approaches are similar to CAC, they do not have any separate concerns for introducing autotuning to any other domains than the one they were defined for.

Optimized composition has been proposed as an optimization technique in the more general context of component-based systems, where the programmer is responsible for annotating components so that their composition can be optimized for performance. However, only a few approaches consider recursive components with deep composition and only a few consider the co-optimization of the selection of the implementation variants with other variation possibilities, such as the layout and data structure of operands or scheduling [68, 10, 104, 2, 83, 112]. For instance, Andersson et al. used CAC to compose and optimize implementation variants of data structures and recursive algorithms, considering Matrix-Multiplication as a case study [1]. Li et al. [68] non-recursively applied dynamic algorithm selection for sorting and matrix computations. Three different learning algorithms were used: a decision tree learner based on Quinlan’s ID3 algorithm with different pruning strategies was found to perform best in the experimental evaluation, a standard feed-forward neural network with back-propagation learning and a Naïve Bayesian classifier found to be inferior in its classification accuracy.

STAPL [104] non-recursively applies dynamic algorithm selection for sorting and matrix computations. Three different learning algorithms are used: a Decision Tree learner based on Quinlan’s ID3 algorithm with different pruning strategies, found to perform best in the experimental evaluation, a standard feed-forward neural network with back-propagation learning, and a Naïve Bayesian classifier found to be inferior in its classification accuracy.

Yu and Rauchwerger [117] took a similar approach for reductions. Using the measurements of the individual implementation variants, they constructed predictor functions using two steps. First, they selected a small number of polynomial terms in the context and machine model parameters from a term pool to build a generic prediction function. Second, they calibrated the coefficients from the training data using general linear regression and Decision Tree learning. For each call, the calibrated run-time prediction functions were then evaluated and the decision was memorized so it could be reused if the same parameter configuration should occur again. This way, the overhead of the dispatch at each call is reduced.

Kessler and Löwe [56, 57, 58] considered optimized composition at the level of annotated user-defined components (i.e., not limited to closed libraries) together with scheduling, resource allocation and other optimiza-
tions, which allowed for simultaneous optimization. For the off-line search and optimization phase, the approach used an interleaved dynamic programming algorithm to construct a component variant Dispatch Table (V-table) for each component-provided functionality $f$ and a resource allocation and schedule dispatch table (S-table) for the independent calls. The tables were constructed simultaneously bottom-up to increase the problem sizes and resource assignments.

Olszewski and Voss [83] proposed a dynamic adaptive algorithm selection framework for divide-and-conquer sorting algorithms in a fork-join parallel setup. Their approach was divided into two phases. First, they used a dynamic programming algorithm to select the best sequential algorithm for different problem sizes. Then, they determined the threshold problem sizes for when to submit the sub-problems to a shared work queue and execute in parallel rather than to execute them sequentially. For the construction of a classifier, they used the C4.5 algorithm to generate a Decision Tree. While apparently sufficient prediction accuracy was achieved in the considered examples, the resulting sizes and overheads were not discussed.

PetaBricks [2] applied a similar approach where, in the offline search phase, the variant choice functions for recursive components were not computed using dynamic programming, but rather were computed using a genetic algorithm, essentially applying heuristic cuts to the optimization space. There appears to be no further compression of the variant selection function. Schedules and resource allocation were not co-optimized with variant selection, but delegated to a run-time system with a work-stealing dynamic scheduler. Numerical accuracy was considered an additional algorithmic property in composition. The paper by Wernsing and Stitt [112] presented an approach that builds upon PetaBricks, but targets multicore CPU systems with FPGA-based accelerators. Like PetaBricks, this approach relied on dynamic scheduling and the learning is done offline. Similar to the approach by Kessler and Löwe [56, 57], the learned execution plan is stored in table form.

However, regardless of all of the benefits of the CAC, it still requires considerable effort to design and implement the applications, which becomes even more complicated when adapting existing complex legacy applications to CAC. Therefore, the current work presented in this chapter is orthogonal to the previous work as is suggests a general way for introducing CAC to legacy applications.

The notion of a context-aware aspect with behavior depending upon the context was first presented by Tanter et al. in [102]. They analyzed the appropriateness of the support of AOP languages for expressing aspects used to access the information associated with the current application contexts. Moreover, the authors proposed an open framework for the context-aware aspects that supported the definition of the context-awareness constructs
9.2. Decision Algebra in Context-Aware Composition

for aspect. It also included the ability to refer to past contexts and provide domain- and applications-specific contexts. CAC applications additionally required learning from past context experiences about the best-fit component variants to be executed in an actual context.

There are other ongoing efforts supporting the development of context-aware computing systems. For instance, David et al. [23] presented the WildCAT system, which is a general context awareness toolkit. It provides a way for Java developers to make their software context-aware, by providing API's (along with shared low level code) to maintain various events (contexts) occurring at program runtime. Using this system in our AOP-based approach could avoid application-specific context implementation.

Delicato et al. [25] proposed a framework for the development of context-aware applications for mobile computing. The framework is aspect-oriented and it is implemented in AspectJ. It provides a set of default adaptive concerns common to mobile applications along with concrete aspects implementing these concerns. However, besides implementing a GUI interface, the developer has to specify which concerns have to be used at which program points, which is application-specific. Li et al. [67] presented an AOP-based approach to address context-aware Web service composition. The approach semantically composes with different Web services whenever the context changes. This work shows that context weaving is suitable for the implementation of CAC services when the context is dynamic and hard to predict. Although, these approaches allow for the implementation of new CAC systems, they still remain domain-specific and, in contrast to our work, do not consider the adaptation of existing legacy applications to CAC for the improvement of application performance.

9.2.1 Context-Aware Recommender Systems

Recent approaches to recommender systems in software development have focused on the information overload problem that occurs due to the growing number of software artifacts that a programmer should consider when working on a software project [7, 41]. Several approaches improve programmer productivity by mechanisms that filter and recommend relevant software artifacts to a particular development task. For instance, Kesten and Murphy [55] provided a task context model that shows the artifacts by ranking and filtering the information presented in the development environment. Čubanić et. al [108] developed a tool for recommending artifacts, including electronic media, bug report and test plans, created as a part of the development process.

Another possible recommendation context can be a particular source code that the developer is working on. For instance, the works in [47] and [116] present approaches and tools for providing relevant source code examples
using the structure of the source code and for reusing the software components by actively recommending methods that are suitable in a context, respectively. Similarly, McCarey et al. [71] suggested a recommender system that recommends reusable software components for agile reuse.

Fluri et al. [33] suggested a recommender system for software developers that provides context change recommendations by applying automated code adaptation support leveraging changes applied in the past. Felfernig et al. [30], in turn, related their work to requirement engineering that could improve the overall quality of decision processes and requirement development of individual stakeholders that deal with continuously changing and quickly ageing requirements.

In contrast to these works, our approach concentrates only on non-functional efficiency requirements. However, it also intends to improve the quality of software processes and to reduce the overall costs of software projects by improving developers’ productivity.
Chapter 10

Conclusions and Future Work

The aim of this chapter is to: 1) summarize the findings from this thesis, 2) argue the fulfilment of the research goals presented in Section 1.2, 3) highlight the most important conclusions based on the research presented in this thesis and 4) present possible directions for future work that might utilize this thesis as an input.

The thesis consists of two parts. The first part presents a generalized theory of Decision Algebra with supporting experiments (Chapters 2-6), while the second part discusses the application of Decision Algebra in context-aware composition (Chapters 7, 8). The second part proves the applicability and effectivity of Decision Algebra in a context-aware composition where the choice in favor of a concrete decision model for a particular problem domain can give performance and memory consumption improvements.

This first part of this thesis explores the generalized theory (i.e., Decision Algebra) used to capture and manipulate decision information. This information is usually captured via decision models using different learning strategies. The decision models may have different values for non-functional requirements (e.g., memory consumption, accuracy and decision time) and different sets of operations that can be applied in order to manipulate the decision information captured. Chapter 2 presents a literature study on prior research papers that have used decision models in different application domains of Computer Science. The main conclusion of this study is that the selection of a decision model for a certain application domain is mostly ad-hoc and only a few studies have examined the over functional and non-functional properties of decision models before choosing the decision model. Later on, based on this study, we introduced a set of basic notations that characterized the decision information used in the different application domains. This Chapter 2 justifies our research goals and motivates the Decision Algebra concept.

In Chapter 3, we presented Decision Algebra based on higher-order decision functions. We formally defined the notion of a decision function along with a basic core operation bind and a set of auxiliary operations: decide, evert, apply. Short reminder: bind binds an attribute captured in a decision function to a concrete attribute value, decide determines a concrete decision based on
Chapter 10. Conclusions and Future Work

a given actual context, every reorders the attributes captured in a decision function, and apply applies a general function defined on the co-domain of a decision function to a set of decision functions. We also showed an example of how a decision function could be constructed by using known learning strategies.

Chapter 4 discusses how decision functions can be compared, combined and approximated. For these operations we defined a merge operation over the decision functions, which combines the decision function by applying a defined merge operator over the co-domain of the decision functions, and an approx operation, which approximates a decision function by ignoring one (or more) attributes. In this chapter, we also showed how these operations influenced the accuracy of the decision functions (e.g., we showed that the merger of probably accurate decision functions constantly gives us a more accurate decision function).

In Chapter 5, we presented four existing decision models as instantiations of Decision Algebra (i.e., decision trees, decision graphs, decision tables, Naïve Bayesian classifiers) using implementations of the core-operation bind. We also showed how more efficient auxiliary operations could be implemented on these decision models. Additionally, for each decision model, we analyzed memory consumption and decision time non-functional properties. This chapter showed that the core operation of Decision Algebra can be implemented in each decision model providing a default implementation of auxiliary operations. Since the auxiliary operations can be defined regardless of the specific decision model implementation, comparing the decision models becomes rather fair as it shows the advantages and disadvantages of the decision models instead of the different implementations thereof. In Chapter 6, we conducted experiments of comparing decision graphs (straight-forward Decision Algebra implementation) with decision trees. In this experiment, we considered memory consumption, learning and deciding time, and the accuracy of the decision models. The experiments showed that decision graphs outperformed decision trees in all measurements as decision graphs gained learning and classification speed-up ups of up to 20% without accuracy loss, and reduced memory consumption by 44%. Additionally, we assessed the accuracy of merged decision functions after merging decision graphs (the highest accuracy growth of the merged decision graph was 16% and the lowest was 2.5%), and on merging decision graphs and Naïve Bayesian classifiers (the highest accuracy growth of the merged decision graph was 17% and the lowest was 2.7%).

The second part of the thesis begins in Chapter 7 where we generally discuss the context-aware composition domain and how it can be integrated into applications containing a Decision Algebra component. In this chapter, we showed the manual and automated (based on Aspect-Oriented Programming) integration of context-awareness using built-in Decision Algebra into
In Chapter 8, we experimentally show that using a best-fit decision model in a context-aware composition can improve the performance of the application. In this chapter we compared decision and memory overheads of five decision models (i.e., decision trees, decision graphs, decision tables, Naïve Bayesian classifiers and support vector machines) as instantiations of Decision Algebra. The results showed that using decision graphs leads to better speed and memory scalability of the context-aware composition (i.e., the decision graphs introduced the lowest decision overhead of 6% to 34% over all other decision models) and reduced the memory consumption by a factor of five.

Finally, this thesis shows that the Decision Algebra as a generalized theoretical framework for decision information can be applied to formalizing classical classification approaches in Computer Science fields (cf. Chapter 2) that address the problems of manipulating and capturing decision information as well as the typical classification problems of fragmentation, replication, and model overfitting. In fact, classical decision models, such as decision trees, decision graphs, decision tables, Naïve Bayesian classifiers (and eventually support vector machines) and variants thereof, can be represented as instances of Decision Algebra. Having the general representation of decision information can lead to further insights in regard to selecting an appropriate decision model, developing a new set of algorithms and improving the performance of decision models (i.e., increasing decision accuracy, reducing memory and reducing decision overheads).

### 10.1 Review of the Goals and Goal Criteria

In Section 1.2, we presented the goal of the thesis, which was to define a unified abstraction for decision models. This goal was motivated by the many application domains in which different decision models were used for classification purposes, and the right choice of a decision model actually influenced the application’s performance. The goal was divided into two sub-goals:

1. Provide a unified theoretical formalization for classification approaches and
2. Create a context-aware component for the integration of this formalization in applications in order to improve application’s performance.

In Section 1.3, we also presented the criteria to be used to fulfil our research goals. We can say that all of the criteria were fulfilled with regard to the defined scope of DA (the decision models of DM1 and DM2 categories). We do not exclude that certain changes to the algebra can be required if to further instantiate it to other categories of the decision models.
Chapter 10. Conclusions and Future Work

10.1.1 Goal 1

The goal criteria for our first goal was:

1. Completeness: Decision Algebra shall be algebraically complete;
2. Soundness: Decision Algebra shall be sound; and
3. Composability: Decision Algebra shall be composable.

Criterion 1 was fulfilled: Chapters 3 and 4 defined the base sets of Decision Algebra as decision functions, constant properties and operations over these decision functions. They also provided signatures and parameterized algebraic specification for Decision Algebra.

Criterion 2 was fulfilled: Chapter 5 discussed the existing decision models (i.e., decision trees, decision graphs, decision tables and Naïve Bayesian classifiers) as possible instantiations of Decision Algebra. By defining a core Decision Algebra operation over these models, we showed that the default implementation of the auxiliary operations and properties of Decision Algebra hold for these models. The instantiations of DA were also proven experimentally in Chapter 6.

Criterion 3 was fulfilled: Chapter 4 defined a general combining operation merge that combined any instantiations of Decision Algebra if the merge operator was implemented on the co-domain of the instantiated decision functions. It also provided a general approximating operation approx that approximated the decision models regardless of their specific implementations. Additionally, in Chapter 5, we showed how other types of approximation algorithms (e.g., k-approx and pruning) could be applied using the Decision Algebra auxiliary operations. Decision Algebra was evaluated experimentally in Chapter 6.

10.1.2 Goal 2

The goal criteria for our second goal were:

1. Integrability: Decision Algebra shall be integrated into existing context-aware legacy applications based on a well-defined integration steps; and
2. Performance efficiency: the context-awareness component with integrated Decision Algebra shall improve application performance by allowing the application to switch between different decision models.

Criterion 1 was fulfilled: Chapter 7 showed how Decision Algebra should be integrated in context-aware composition and presented manual and automated integration of context-awareness into legacy applications. For the
10.2 Future Work

In this section, we will present future work that should be completed toward the goal of the further improvement of our Decision Algebra formalization and its implementation. Basically, this section should be seen as a summary of possible research directions if one were to take this thesis as a first step. We believe that the following ways could be used to further broaden the scope of Decision Algebra:

1. Instantiate and, if necessary, generalize, Decision Algebra over other types of decision models mentioned in Section 2.1.3, such as maximum-margin (e.g., support vector machines and artificial neural networks) and regression models (e.g., linear and logic regressions);

2. Study and analyze common problems and their solutions in different application domains where decision models are used and check how these solutions can be transferred from one domain to another using Decision Algebra;

3. Apply and evaluate the general combining decision models’ operation merge in incremental classifications, which become a central concern in applications whose main goal is to deal with information that varies over time; and

4. In a conjunction with [3], apply and evaluate context-aware composition approaches in dynamically changing applications where the conditions can evolve over time and, thus, an online learning approach has to be used. Additionally, due to the evolved changes in the application environment (software as well as hardware), the best-fit decision model can also change.
Appendix A

Generalized Weak Law of Large Numbers

For our main result, the convergence of mergers of decision functions towards the accurate one, we need to establish the following

**Lemma A.1.** Let $X_1, \ldots, X_n$ be a series of independent, identically distributed random variables with $E(X_1) = \mu$ and finite variance $\text{Var}(X_1) \leq \sigma^2$. Define the weighted average of the $X_i$:

$$A_n = \frac{\sum_{i=1}^n X_i \times N_i}{\sum_{i=1}^n N_i}, N_i > 0.$$

It holds for the expected value and the variance, resp., of these weighted averages:

$$E(A_n)=\mu \quad \text{(A.1)}$$
$$\text{Var}(A_n) \leq \sigma^2 \quad \text{(A.2)}$$
$$\lim_{n \to \infty} \text{Var}(A_n)=0. \quad \text{(A.3)}$$

**Proof.**

$$E(A_n)=E\left(\frac{\sum_{i=1}^n X_i \times N_i}{\sum_{i=1}^n N_i}\right)$$

$$= \frac{1}{\sum_{i=1}^n N_i} \sum_{i=1}^n E(X_i) \times N_i$$

$$= \frac{1}{\sum_{i=1}^n N_i} \sum_{i=1}^n \mu \times N_i$$

$$= \frac{\mu}{\sum_{i=1}^n N_i} \sum_{i=1}^n N_i$$

$$= \mu$$
Appendix A. Generalized Weak Law of Large Numbers

which proves [Equation A.1]. Further

\[
Var(A_n) = Var\left( \frac{\sum_{i=1}^{n} X_i \times N_i}{\sum_{i=1}^{n} N_i} \right) \\
= \frac{1}{(\sum_{i=1}^{n} N_i)^2} Var\left( \sum_{i=1}^{n} X_i \times N_i \right) \\
= \frac{1}{(\sum_{i=1}^{n} N_i)^2} \sum_{i=1}^{n} Var(X_i) \times N_i^2 \\
\leq \frac{1}{(\sum_{i=1}^{n} N_i)^2} \sum_{i=1}^{n} \sigma^2 \times N_i^2 \\
\leq \frac{\sum_{i=1}^{n} N_i^2}{(\sum_{i=1}^{n} N_i)^2} \sigma^2
\]

To see that this term is less or equal \( \sigma^2 \) [see Equation A.2], and approaches zero for large \( n \), [see Equation A.3], we rewrite its first factor:

\[
\frac{\sum_{i=1}^{n} N_i^2}{(\sum_{i=1}^{n} N_i)^2} = \frac{\sum_{k=1}^{n} N_k^2}{(N_k + \sum_{i=1, i \neq k}^{n} N_i)^2} = \sum_{k=1}^{n} \frac{N_k^2}{N_k^2 + 2N_k \sum_{i=1, i \neq k}^{n} N_i + \left( \sum_{i=1, i \neq k}^{n} N_i \right)^2}
\]

and note that for each \( k \), it holds \( \sum_{i=1, i \neq k}^{n} N_i > 0 \) for \( N_i > 0 \), proving [Equation A.2], and \( \sum_{i=1, i \neq k}^{n} N_i \) approaches infinity for large \( n \), proving [Equation A.3].

From Lemma A.1, it immediately follows that the weighted averages \( A_n \) converge in probability to their expected values \( \mu \):

**Lemma A.2.** Let \( A_n \) be weighted average of a series \( X_1, \ldots, X_n \) of independent, identically distributed random variables with \( E(X_1) = \mu \) and finite variance \( Var(X_1) \leq \sigma^2 \). Then for any \( \epsilon > 0 \)

\[
\lim_{n \to \infty} P(|A_n - \mu| \geq \epsilon) = 0.
\]

**Proof.** Due to Chebyshev’s inequality, \( P(|X - \mu| \geq k\sigma) \leq 1/k^2 \), or \( P(|X - \mu| \geq \epsilon) \leq 1/\epsilon^2 \sigma^2 \) (when choosing \( \epsilon = k\sigma \)), we have:

\[
\lim_{n \to \infty} P(|A_{n,\epsilon} - \mu| \geq \epsilon) \leq \lim_{n \to \infty} \frac{1}{\epsilon^2} Var(A_n),
\]

when choosing \( X = A_n, \sigma^2 = Var(A_n) \). This converges to zero for large \( n \) as \( Var(A_n) \) does according to Lemma A.1. 

\[\square\]
Lemmata A.1, A.2 and their proofs are similar to the weak law of large numbers stating that under the same conditions the unweighted sample average of real valued random variables converges in probability towards the expected value. In fact, it is a special case with weights $N_i = 1$, which we use in the following (obvious, hence unproved) corollary.

**Corollary A.1.** Let $D_1, \ldots, D_n$ be a series of independent decisions, identically distributed from a classification distribution $d : C \rightarrow \mathbb{R}$ with expected values $E[D_1 = c] = p_c$ and variances $\text{Var}[D_1 = c] = p_c(1 - p_c)$ for any decision $c \in C$. Let $A_{n,c}$ be the (un-)weighted average of $D_1, \ldots, D_n$ of these decisions with equal weights $N_1 = 1$. Then it holds:

\[
\begin{align*}
E(A_{n,c}) &= p_c \\
\text{Var}(A_n) &\leq p_c(1 - p_c) \\
\lim_{n \to \infty} \text{Var}(A_{n,c}) &= 0 \\
\lim_{n \to \infty} P(|A_{n,c} - p_c| \geq \epsilon) &= 0 \text{ for any } \epsilon > 0.
\end{align*}
\]

Overall, this Generalized Weak Law of Large Numbers can be used as a supplementary material to the text presented in Chapter 4 in Section 4.3 regarding the accuracy of learning by merging decision functions.

---

1Expectation and variance of standard classification distributions; $[\cdot]$ the Iverson bracket with $[\text{cond}] = \begin{cases} 1 & \text{if } \text{cond} \\ 0 & \text{otherwise} \end{cases}$
Bibliography


161
Bibliography


[36] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.
Bibliography


M. M. Kandé, J. Kienzle, and A. Strohmeier. From aop to uml – a bottom-up approach.


Bibliography


Bibliography


Linnaeus University Dissertations

Below please find a list of recent publications in the series Linnaeus University Dissertations. For a full list and more information: Lnu.se


197. Olga Maskenskaya 2014. **Abundance and fractionation of rare earth elements in calcite and other secondary minerals in fractures in the upper kilometre of crystal-