UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire ICube – UMR 7357

THÈSE présentée par :

Clément CHARNAY

soutenue le : **30 juin 2016**

pour obtenir le grade de : **Docteur de l'université de Strasbourg**

Discipline / Spécialité : Informatique

Enhancing Supervised Learning with Complex Aggregate Features and Context Sensitivity

THÈSE dirigée par :

M. LACHICHE Nicolas	Directeur, Maître de Conférences HDR, Université de Strasbourg
M^{me} BRAUD Agnès	Co-encadrante, Maître de Conférences, Université de Strasbourg

RAPPORTEURS :

M. BLOCKEEL Hendrik	Professeur, Katholieke Universiteit Leuven
M^{me} VRAIN Christel	Professeur, Université d'Orléans

AUTRES MEMBRES DU JURY :

M. FERRI Cèsar	Professeur associé, Universitat Politècnica de València

This thesis has been prepared at

ICube: Engineering science, computer science and imaging laboratory - UMR 7357

Télécom Physique Strasbourg
300 Boulevard Sébastien Brant
CS 10413
F-67412 Illkirch Cedex

☎ +33 (0)3 68 85 45 54
✉ `contact@icube.unistra.fr`
Web Site `http://icube.unistra.fr`

Enhancing Supervised Learning with Complex Aggregate Features and Context Sensitivity

Abstract

In this thesis, we study model adaptation in supervised learning. Firstly, we adapt existing learning algorithms to the relational representation of data. Secondly, we adapt learned prediction models to context change.

In the relational setting, data is modeled by multiples entities linked with relationships. We handle these relationships using complex aggregate features. We propose stochastic optimization heuristics to include complex aggregates in relational decision trees and Random Forests, and assess their predictive performance on real-world datasets.

We adapt prediction models to two kinds of context change. Firstly, we propose an algorithm to tune thresholds on pairwise scoring models to adapt to a change of misclassification costs. Secondly, we reframe numerical attributes with affine transformations to adapt to a change of attribute distribution between a learning and a deployment context. Finally, we extend these transformations to complex aggregates.

Keywords: relational data mining, reframing, complex aggregation, stochastic optimization, cost-sensitive classification, model adaptation, machine learning, artificial intelligence

Amélioration de l'apprentissage supervisé par l'utilisation d'agrégats complexes et la prise en compte du contexte

Résumé

Dans cette thèse, nous étudions l'adaptation de modèles en apprentissage supervisé. Nous adaptons des algorithmes d'apprentissage existants à une représentation relationnelle. Puis, nous adaptons des modèles de prédiction aux changements de contexte.

En représentation relationnelle, les données sont modélisées par plusieurs entités liées par des relations. Nous tirons parti de ces relations avec des agrégats complexes. Nous proposons des heuristiques d'optimisation stochastique pour inclure des agrégats complexes dans des arbres de décisions relationnels et des forêts, et les évaluons sur des jeux de données réelles.

Nous adaptons des modèles de prédiction à deux types de changements de contexte. Nous proposons une optimisation de seuils sur des modèles à scores pour s'adapter à un changement de coûts. Puis, nous utilisons des transformations affines pour adapter les attributs numériques à un changement de distribution. Enfin, nous étendons ces transformations aux agrégats complexes.

Mots clés : fouille de données relationnelles, reframing, agrégation complexe, optimisation stochastique, classification sensible au coût, adaptation de modèles, apprentissage automatique, intelligence artificielle

ICube: Engineering science, computer science and imaging laboratory - UMR 7357

Télécom Physique Strasbourg – 300 Boulevard Sébastien Brant – CS 10413 – F-67412 Illkirch Cedex

Remerciements

Presque quatre années se sont écoulées depuis octobre 2012. Une page va se tourner et il est grand temps de remercier les personnes qui ont contribué à mon équilibre au cours de cette phase de ma vie qu'a constituée le doctorat.

Tout d'abord, je remercie Nicolas Lachiche pour avoir dirigé cette thèse. La confiance qu'il m'a accordée dès le début, sa présence et son soutien continus tout au long de ces quelques années, m'ont permis de mener à bien cette aventure. Plus particulièrement, nos *brainstorming* réguliers m'ont été d'un grand secours pour organiser et développer les idées de cette thèse, ainsi que sa relecture du présent « tapuscrit ».

Je remercie également Agnès Braud pour avoir co-encadré cette thèse, pour son appui et son sens du détail, ainsi que pour ses relectures d'articles et commentaires précieux lors de la rédaction de cette thèse.

Je remercie Hendrik Blockeel, Christel Vrain, Cèsar Ferri et Marc Boullé pour avoir accepté de relire cette thèse et de faire partie du jury.

Je remercie encore Hendrik Blockeel ainsi que Jan Ramon, qui m'ont permis d'effectuer mon stage de Master dans leur laboratoire de Louvain, me mettant ainsi le pied à l'étrier du merveilleux monde des données et de l'apprentissage automatique.

Je remercie tous les participants du projet Reframe : Peter Flach, José Hernandez-Orallo, César Ferri encore, María José Ramírez-Quintana, Meelis Kull, Adolfo Martínez Usó, Reem Al-Otaibi, Chowdhury Farhan Ahmed, Nicolas et Agnès encore, ainsi que les autres, pour m'avoir permis de participer à un projet scientifique très intéressant, auquel une partie conséquente de cette thèse est consacrée.

Je remercie le laboratoire ICube et les permanents de l'équipe BFO, devenue SDC, pour leur accueil. Je remercie plus particulièrement les doctorants, post-doctorants, stagiaires et ingénieurs que j'ai eu l'occasion de côtoyer : Bruno, Andrés, Karim, Manuela, Igor, Carlos, Cristina, Ali, Alain, Caroline, Alexandre, Djawad, et les autres, pour l'ambiance au sein de l'équipe, et les discussions animées sur des sujets aussi divers que le football ou la religion lors des repas, même si j'ai fini par décrocher du RU.

Je remercie spécialement Gaëlle pour son soutien lors de ces derniers mois de thèse. J'espère pouvoir te procurer ce même soutien pour tes propres derniers mois.

Enfin, je remercie mes parents pour le soutien logistique et humain apporté durant ces années de thèse, sans lequel cette thèse n'aurait pas été possible. Malgré les occurrences répétées des deux questions qu'aucun doctorant n'a jamais envie d'entendre, je n'en serais pas ici sans votre amour ces vingt-cinq dernières années !

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter, we describe the scientific context of our work, from the general field of artificial intelligence to our concrete field of interest: machine learning. We provide an overview of these domains, motivate our work, and briefly list our contributions.

1.1 Scientific Context

This thesis is concerned with learning prediction models. These models are computer programs, trained for the specific prediction task. Thus, our work relates to the field of artificial intelligence, which aims at learning "intelligent" computer programs. More specifically, the scientific area in which we place our work, called machine learning, is concerned with computer programs able to "learn by themselves". In other words, these computer programs can learn to be efficient on a certain category of tasks without being explicitly programmed for it. In our case, we are interested in supervised machine learning, which aims at learning programs for predictive purposes.

1.1.1 Artificial Intelligence

Artificial intelligence received attention from the general public for some of its famous achievements. Recently, the victory of Google DeepMind's AlphaGo program over professional Go player Lee Sedol, brought artificial intelligence into light. The Go board game has always been considered a challenge for artificial intelligence, in the sense that from a human point of view it requires strategy, intuition and creativity, abilities that are considered difficult to program. Artificial intelligence was also popularized by fictional works, for instance the HAL 9000 computer from Arthur C. Clarke's novel and Stanley Kubrick's movie *2001: A Space Odyssey*, which raises both the positive aspects of Artificial Intelligence, assistance to human beings by discharging them of some tasks, and the potential threat they represent when they combine intelligence and access to sensitive controls. It also addresses the question of a possible consciousness of machines. More recently, in the television series *Person of Interest*, the Machine is designed to watch citizens in order to detect acts of terror in advance, but is actually able to detect ordinary crimes and prevent threats to its own safety and the one of its human interfaces, raising the ethical question of global surveillance. It also suggests the ability for an artificial intelligence program to evolve to adapt to situations endangering its integrity, and depicts the possible consequences of a clash between two artificial intelligence programs.

Formally, artificial intelligence is the discipline which studies and creates computer programs capable of intelligent behavior. John McCarthy, who introduced the term, is one of the founders of the field and co-organized the seminal Dartmouth Summer Research Project on Artificial Intelligence in 1956, gave the following definition in his website:

“Artificial intelligence is the science and engineering of making intelligent machines, especially intelligent computer programs.” (John McCarthy (November 12th, 2007), *What is Artificial Intelligence?*)

Applications of artificial intelligence include robotics, fraud detection in banking, clinical decision support for medical diagnosis. The two latter applications, based on prediction models, either for defining and detecting a fraud or for finding a possible disease in a patient, belong to a specific sub-field of artificial intelligence: machine learning. This sub-field will be our main topic of interest.

1.1.2 Machine Learning

Machine learning studies the ability for a computer program to learn from data. More formally, Tom Mitchell gives the following definition:

“A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.” (Mitchell 1997)

For instance, let us imagine we want to design an algorithm to play chess. The program learns to play chess (the task T) if its ability to win against other players (the performance P) improves through playing practice games against itself (the training experience E).

Machine learning covers a wide range of tasks, divided in two main families:

Supervised learning consists in building predictive models. For instance, let us consider a diagnosis application: the aim is to predict the nature of the disease (or an absence of disease) in a patient. This prediction will be based on predictive characteristics such as the vital constants of the patient, e.g. blood pressure, internal temperature… The experience used to learn the model is a bank of diagnosis, i.e. records of patients with the value of their vital constants and the actual disease he/she suffers (or not). Classification and regression tasks are the two main categories of supervised learning tasks.

Unsupervised learning consists in discovering structures in unlabeled data. As opposed to prediction models, it builds descriptive models. The two main description models are given by clustering, where groups of similar data are defined according to a given distance, and association rule learning, where the aim is to find relationships between variables, based on how frequently they occur together in data. Community discovery for market analysis is an application: groups of similar customers can be discovered according to characteristics such as their age, gender, or the items they buy…

This thesis focuses on supervised learning, more specifically on building and adapting predictive models.

1.1.3 Data Representation and Relational Setting

In a supervised learning context, data is commonly represented by a simple table structure. Columns represent the characteristics of data, i.e. the input characteristics and the outcome. Generally, there is a single outcome to predict, a single target column called the output, while the other columns represent the input characteristics to use for prediction. Then, a training example is instantiated by a row of the table, i.e. a tuple of values corresponding to the different columns. This setting, where data is represented as a single table, is called the **attribute-value** setting. An example from the diagnosis application described above is provided in Table 1.1.

Table 1.1 – Data in the attribute-value setting.

Temp	BloodP	Other vital constants	State
36	mid	···	sick
39	high	···	sane
37	mid	···	sick
37.5	low	···	sane
···	···	···	···

Another possibility is to represent data through a multi-table scheme. In the attribute-value setting, the table represents one type of objects, along with its associated characteristics. This setting reaches its limit when the task at hand involves several kinds of objects. As an example, let us consider a molecule classification task, where the aim is to predict mutagenicity, i.e. if the molecule is likely or not to provoke genetic mutations in living organisms. The molecule has its own chemical characteristics, that are modeled in a table along with a boolean column indicating its mutagenicity. A molecule is composed of atoms, which have their own characteristics, such as their charge, or the chemical element they are. Classification of a molecule may depend not only on the global, chemical, characteristics of the molecule, but also on properties of its atoms. Thus, a two-table representation, with a table of molecules and a table of atoms, related in such a way that molecules are associated to their respective atoms, is more relevant. This representation, shown in Table 1.2, is called the **relational** setting, and is our first domain of interest.

Such a representation is powerful because it keeps the "natural" structure of relational data by not embedding the atoms at the molecule level. This representation

Table 1.2 – Data in the relational setting.

Main table - Molecules

mol_id	lumo	logp	mutagenic
m_1	-1.246	4.23	yes
m_2	-1.034	2.68	no
m_3	-1.598	6.26	yes
...	...	...	...

1

0..N

atom_id	symb	nucl	charge	mol_id
a_{11}	c	22	-0.117	m_1
a_{12}	o	40	-0.388	m_1
a_{13}	h	3	0.142	m_1
...	...	...	...	...
a_{21}	c	22	-0.128	m_2
a_{22}	n	38	0.801	m_2
...	...	...	...	...

Secondary table - Atoms

allows for great expressivity, since characteristics of a molecule based on characteristics of atoms can be constructed in many ways. This expressivity also induces a great complexity: many characteristics could be constructed in this multi-table setting, possibly billions. This order of magnitude implies that it is impossible to construct all atom-based characteristics of the molecules, and add them as columns of the molecules table. Most state-of-the-art relational supervised learning algorithms are based on a language bias which allows to control the size of the characteristics vocabulary. In this work, we will focus on a certain family of relational characteristics, **complex aggregates**, whose vocabulary is very large, and design optimization algorithms to search for the best features to use for prediction.

1.1.4 Adaptation to Context Change

Our second domain of interest is model reuse: in data-driven learning applications, a common process is to train a model for prediction, use it in production with no evolution over time, and finally discard it when it does not perform well anymore. A possible reason is a change in data characteristics: training examples characteristics values were lying in a certain range of values which is no longer valid in the production context. We refer to such a change as a **context change**.

The most straightforward way to deal with context change is, as suggested above, to discard the old model and to train a new one, adapted to current data. This approach is called *retraining*. We propose another approach, based on reusing the old model and *adapting* it so that it performs well on new data. This approach is called **reframing**.

As a concrete example, let us consider a very simple pullover sales prediction task based on temperature as the single input. A prediction model is trained over daily data from one city. Now let us consider the deployment of this model in a second city. Pullover sales increase when the weather becomes cold, but the definition of *cold* in the second city may not be the same as in the first city. For instance, if the first city is located in southern Europe, a cold temperature may be defined as lying below 10°C. If the second city is located in Northern Europe, the threshold for the cold temperature may be 0°C. The pullover sales will increase when the temperature decreases in both cities, but the range of temperatures for which the sales will increase will not be the same. In this case, the model trained in Southern Europe will not be useful for a deployment in Northern Europe. A possible reframing transformation would be, when deploying in this second context, to add 10 to all input temperatures so that the temperatures from Northern Europe “look like” temperatures from Southern Europe, the original model is then usable on this transformed data. Our goal will be to design algorithms able to learn such a reframing transformation.

1.2 Motivation and Contributions

The relational representation of data is relevant for many learning tasks. Indeed, if the task involves several related entities, the relational representation becomes a useful option. The great interest of the relational paradigm is that many domains can be naturally modeled in this setting, a composition relationship in data is enough to consider using this representation. A non-exhaustive list of application domains is:

- As mentioned above, chemistry data, with a prediction task on molecules based on properties of their atoms.
- We will focus on a geographical application, where the aim is to predict the kind of a urban block, i.e. a set of buildings, which can be blocks of individual houses, an ensemble of tall buildings for collective housing, a mix of both, an industrial area... This prediction will be made according to geometrical properties of the buildings, such as their areas.
- Image content recognition, where the prediction task is to recognize what an image represents, according to properties of segments the image was decomposed into.
- Handwritten character recognition, according to an image representing the character decomposed in a matrix of filled or empty pixels, or according to the sampled trajectory used to write the character.

The expressivity of the relational representation implies that the number of possible relational characteristics is very high, often impossible to be fully considered. In particular, we will focus on the complex aggregates language bias, which is very expressive and easy to interpret. Nevertheless, the number of possibilities to form a complex aggregate in a given task is very high, intractable for an exhaustive exploration. Moreover, their great expressivity means they are also very specific, which implies they are very sensitive to data changes.

Thus, our contributions in the relational supervised learning domain are the following:

- We develop heuristics to explore efficiently, yet not exhaustively, the complex aggregate search space, for inclusion in a prediction model.
- We adapt these models so that the specificity of complex aggregates, i.e. its tendency to overfit the training data, is not an issue.

We are interested in reframing because it promotes the key concept of model reuse: with the increasing trend of *big data*, prediction models tend to be more complex to adapt to the quantity and complexity of data. Thus, model training will become more and more time-consuming, and adapting models rather than fully retrain new models becomes a valid alternative. In this context, we propose several approaches to reframing, based on adapting data from the current context to data from the previous context the model was trained with in order to make the model useful for the current context. We also propose an approach to adapt the structure of the model in order to deal with such context changes.

Our contributions in the reframing, adaptation to context change, domain are the following:

- We design an algorithm to adapt numerical input and output characteristics of data through affine transformations, optimized thanks to stochastic hill-climbing heuristics.
- We extend these affine transformations to the relational setting and complex aggregates, which mostly involve numerical characteristics, binding the two axis of the thesis.
- We introduce a structural reframing algorithm to adapt decision thresholds of a set of binary, one-versus-one, prediction models for multi-class cost-sensitive classification.

Finally, we present, as an application of relational supervised learning, a possible way to deal with spatio-temporal data using complex aggregates.

1.3 Structure of the Manuscript

This manuscript is structured as follows. **Chapter 2** develops background and useful notions on the fields in which this thesis takes place: supervised learning, relational data and reframing.

Part I presents our works on the relational setting and complex aggregation. **Chapter 3** formalizes the notion of complex aggregate, and suggests a first hill-climbing algorithm to generate them on-the-fly for inclusion in decision tree splitting conditions. **Chapter 4** introduces a stochastic optimization algorithm, based on random restart hill-climbing, to search efficiently through the complex aggregate space, for inclusion in a decision tree, and presents its extension to a Random Forest learner using less exhaustive hill-climbing algorithms to perform the search for complex aggregates.

Part II exposes our works on reframing and context change. **Chapter 5** introduces a pairwise threshold optimization algorithm to tackle multi-class cost-sensitive classification tasks, which can be used for structural reframing. **Chapter 6** presents stochastic hill-climbing algorithms to find appropriate affine transformations of numerical inputs to deal with context change and binds together the two axis of this thesis by extending these affine transformations of numerical inputs to complex aggregates and the relational setting.

Finally, **Chapter 7** summarizes our contributions, and suggests an application of our works to prediction on spatio-temporal data.

Chapter 2

Background

In this chapter, we give an introduction to key notions from the scientific fields relevant for the thesis. Section 2.1 presents supervised learning concepts and two associated families of prediction models we will focus on. Section 2.2 deals with the particular sub-domain of supervised learning in the relational setting, positioning our approach with respect to the two main families of approaches, Inductive Logic Programming-based methods and propositionalization algorithms. Finally, Section 2.3 introduces the notions of context change and reframing along with related fields.

2.1 Supervised Learning and Prediction Models

This section describes the field of supervised learning concepts, i.e. the kind of tasks it covers and performance measures. Then, it describes in more detail two families of models falling in this category: decision tree models and naive bayesian classifiers.

2.1.1 Supervised Learning

Supervised learning refers to a kind of machine learning tasks focusing on building predictive models. More formally, the aim is to learn a model, or function, that maps a vector of inputs to a vector of outputs, given a set of training examples (or training instances) which associate a vector of inputs to its desired outputs. This training set constitutes the training experience. Finally, the performance is measured by a loss function, related to the error made by the model when predicting the output value of test examples, different from the training examples.

In the attribute-value setting, data is usually represented by a table, that we denote by T, where columns represent the inputs and outputs. Columns of the table are denoted by $T.A = (X_1, X_2, \cdots, X_{a(T)}, Y_1, Y_2, \cdots, Y_{o(T)})$, where $X_i, 1 \leqslant i \leqslant a(T)$ is an input, and $Y_j, 1 \leqslant j \leqslant o(T)$ is an output. Thus, $a(T)$ is the number of inputs in table T, while $o(T)$ is the number of outputs. The most common case in supervised learning is the prediction of a single output, i.e. $o(T) = 1$ and the output is denoted by Y.

Columns of a table will be referred to as *attributes*. More generally we will refer to characteristics of objects of table T with the term *feature*. An attribute is considered a *direct* feature of objects of the table, i.e. a feature explicitly present in the database, as opposed to constructed features. The domain of a feature F will be denoted by $domain(F)$. Features can be of different nature depending on the nature of their domain, the two main families are:

categorical: if the feature takes a finite, unordered, number of values, i.e. its domain is a set of values.

numerical: if the feature is a number, integer or real, semantically representing an ordering.

A row represents an example, i.e. a tuple of values for inputs and outputs. We denote an example by $t = (v_{X_1}(t), v_{X_2}(t), \cdots, v_{X_{a(T)}}(t), v_{Y_1}(t), v_{Y_2}(t), \cdots, v_{Y_{o(T)}}(t))$, where $v_Z(t)$ is the value of column Z for example t.

In our case, we will focus on the following supervised learning task: mapping a vector of inputs to a single output. Formally, **given a table** T with columns $T.A = (X_1, X_2, \cdots, X_{a(T)}, Y)$, denoting by $\mathbb{Z} = domain(Z)$ the domain of column $Z \in T.A$, **and a training set** $Train = \{t_1, t_2, \cdots, t_n\} \subset T$, with example $t_i = (v_{X_1}(t_i),\ v_{X_2}(t_i),\ \cdots,\ v_{X_{a(T)}}(t_i),\ v_Y(t_i))$, **the aim is to find a function** $F : \mathbb{X}_1 \times \mathbb{X}_2 \times \cdots \times \mathbb{X}_{a(T)} \longmapsto \mathbb{Y}$ **approximating the relationship between the inputs and the output**, based on the observations from $Train$. The quality of the approximation is measured with a loss function.

This task has different names depending on the nature of the output:

classification: if the output attribute is categorical. In this case, performance measures on a test set containing n examples are derived from the confusion matrix. The output attribute takes C distinct values, indexed from 1 to C and denoted by $domain(Y) = \{y_1, \cdots, y_C\}$. The confusion matrix has size $C \times C$ and element (i, j) is the count of test examples that actually belong to class y_i and are predicted by the model as belonging to class y_j. Its standard shape is given in Table 2.1.

Table 2.1 – Confusion matrix for C classes.

Predicted Actual	y_1	y_2	$\cdots$	y_C
y_1	🟩	🟥	🟥 $\cdots$	🟥
y_2	🟥	🟩	🟥 $\cdots$	🟥
$\vdots$	🟥 $\vdots$	🟥 $\vdots$	🟩 $\ddots$	🟥 $\vdots$
y_C	🟥	🟥	🟥 $\cdots$	🟩

By construction, well-classified examples are counted in the diagonal elements of the confusion matrix, while misclassified examples are counted out of the diagonal. The most common loss measure is the error rate, defined as the proportion of misclassified examples, i.e. the sum of non-diagonal elements over the sum of all elements in the matrix (trivially n). Its complementary, the performance measure called accuracy, is the proportion of well-classified examples, i.e. the sum of diagonal elements over n.

$$ErrorRate = \frac{1}{n} \sum_{1 \leqslant i \neq j \leqslant C} Confusion(i,j)$$

$$Accuracy = \frac{1}{n} \sum_{1 \leqslant i \leqslant C} Confusion(i,i) = 1 - ErrorRate$$

regression: if the output feature is numerical. In this case, numerical distances between predictions and actual values are used as loss measures. Two classic error measures for regression are the mean absolute error (MAE) and the root mean squared error (RMSE). For test example k, we define y_k the actual output value of example k, and p_k the prediction made by the model for example k, the errors are then defined as:

$$MAE = \frac{1}{n} \sum_{1 \leqslant k \leqslant n} |y_k - p_k|$$

$$RMSE = \sqrt{\frac{1}{n} \sum_{1 \leqslant k \leqslant n} (y_k - p_k)^2}$$

On most datasets, no test set is clearly defined. In this case, learners are evaluated using a method called cross-validation. A k-fold cross-validation consists in partitioning the examples set into k folds of approximately equal size. Then k models are built, leaving aside one fold for testing purposes, and using the $k-1$ remaining for training a model. This gives k experiments, which can be considered separately, or as a single one. For instance, in a classification task, the k confusion matrices can be used to compute independent statistics, which is useful to estimate variance of a performance metric, or be fused to compute global statistics for the k experiments. The most common value of k is 10, we will refer to this evaluation process as 10-fold cross-validation.

Example 2.1. The Postop dataset.

The Post-Operative Patient dataset is publicly available on the UCI Machine Learning Repository (Lichman 2013). A sample of the dataset is given in Table 2.2. It is a classification task, where the aim is to predict where patients recovering from surgery in a post-operative recovery area of an hospital should be next sent to. The 3 possibilities constitute the 3 classes of the task: they can be sent back home (*home*), to the general hospital floor (*GHF*), or to the intensive care unit (*ICU*). It is the output attribute *Decision*.

Eight attributes constitute the input vector: internal temperature "CORE", surface temperature "SURF", blood pressure "BP", stability associated with those three param-

Table 2.2 – Sample of the Postop dataset.

CORE	SURF	BP	CORE-STBL	SURF-STBL	BP-STBL	O2	COMF	Decision
mid	low	mid	stable	stable	stable	excellent	15	GHF
mid	high	high	stable	stable	stable	excellent	10	home
mid	low	mid	stable	stable	unstable	good	10	ICU
mid	mid	mid	mod-stable	stable	unstable	excellent	15	GHF
mid	mid	mid	unstable	stable	unstable	good	10	home
low	mid	high	unstable	mod-stable	mod-stable	good	10	ICU
mid	high	low	unstable	stable	stable	good	10	GHF
mid	low	mid	stable	stable	unstable	excellent	10	home
...	...	...	...	...	...	...	...	...

eters “CORE-STBL”, “SURF-STBL” and “BP-STBL”, and oxygen saturation “O2” are categorical attributes, since they take a finite number of values indicating their level.

Note that “CORE”, “SURF”, “BP” and “O2” could be numerical attributes, since actual temperature values could be used for instance. In this dataset, they have been transformed into categorical attributes through an operation called **discretization**: using thresholds on the numerical value of the attribute, we can define intervals with the thresholds as cutpoints; these intervals define categories, and labeling these categories finally gives values for a new, categorical attribute. For instance, the CORE attribute has value *high* if the actual internal temperature is higher than 37°C, *mid* if it is between 36 and 37, and *low* if it is below 36. ■

Several algorithms have been developed to learn prediction models in attribute-value supervised learning. We will present two families of models that will be used in this work: decision trees, and naive bayesian models.

2.1.2 Decision Tree Induction

A decision tree is a machine learning model which relies on partitioning the input feature space to form homogeneous groups of data with respect to the output attribute. Popular decision tree learning algorithms include Breiman’s Classification and Regression Trees (CART) (Breiman et al. 1984) and Quinlan’s Iterative Dichotomiser 3 (ID3) (Quinlan 1986) and its extension C4.5 (Quinlan 1993). It uses a tree structure, where internal nodes are splits on input attributes, categorical as well as numerical. The outcomes of a split define children branches to the node, which are partitions of the input space based on the attribute involved in the split. Leaves are labeled with decisions on the output attribute: one of its possible values or a probability distribution in a classification task, a real value in a regression task. Predictions on unseen data are made by descending data down the tree: starting from the root, it will follow a path in the tree depending on the outcomes of the tests at internal nodes, until it reaches a leaf which will give a prediction.

Unlike most prediction models, the representation of decision trees makes it easy for an external human user to read and understand. The graphical tree representation is nothing more than a flowchart, with test nodes indicating the path to follow and ending up with a final outcome. Example 2.2 shows an example of decision tree built on the Postop dataset presented above, along with the performance such a model achieves on this dataset.

Example 2.2. Decision tree building on the Postop dataset.

Figure 2.1 shows a decision tree built using a training set of 90 examples, i.e. the whole Postop dataset.

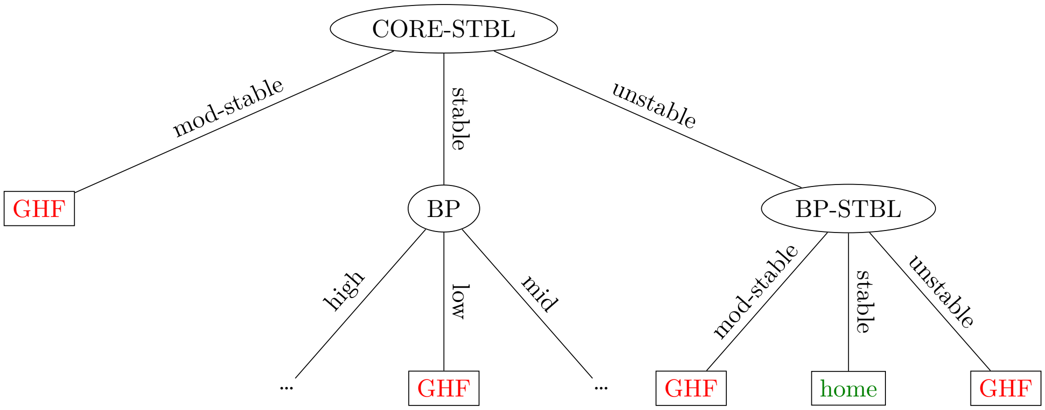


Figure 2.1 – Upper part of the decision tree built on the Postop dataset.

Table 2.3 shows the confusion matrix of a 10-fold cross-validation performed on the Postop dataset with Quinlan's ID3 learner. 54 examples out of the 90 are well classified, which corresponds to a classification accuracy of 60%.

Table 2.3 – Confusion matrix for 10-fold cross-validation in the Postop dataset.

Predicted Actual	GHF	ICU	home
GHF	52	0	12
ICU	1	0	1
home	21	1	2

■

Decision tree induction is usually achieved in a top-down way: it starts from the root by partitioning the whole training set, and recursively splits the resulting partitions until they are homogeneous or a stopping criterion is reached, for instance if the number of examples to split is below a given minimum. In both cases, the set of instances is used to create a leaf. This procedure is called top-down induction of decision trees, the general algorithm is given in Algorithm 2.1.

Algorithm 2.1 BuildDecisionTree

```
Input: train: set of training examples, feats: set of possible split features, target: the target attribute
Output: tree: a decision tree

if StopCriterion(train) then
  tree $\leftarrow$ Leaf(train)
else
  bestScore $\leftarrow$ WORST_SCORE_FOR_METRIC
  bestSplits $\leftarrow$ []
  for all f $\in$ feats do
    if f is categorical then
      spl $\leftarrow$ EvaluateCategoricalFeature(train, f, target)
    else
      if f is numerical then
        spl $\leftarrow$ EvaluateNumericalFeature(train, f, target)
      end if
    end if
    score $\leftarrow$ spl.score
    if score $\geq$ bestScore then
      if score $>$ bestScore then
        bestScore $\leftarrow$ score
        bestSplits $\leftarrow$ []
      end if
      bestSplits.Add(spl)
    end if
  end for
  bestSpl $\leftarrow$ bestSplits.OneRandomElement()
  tree $\leftarrow$ InternalNode(bestSpl.condition)
  partitions $\leftarrow$ bestSpl.partitions
  for all part $\in$ partitions do
    childNode $\leftarrow$ BuildDecisionTree(part, feats, target)
    tree.AddChild(childNode)
  end for
end if
return tree
```

An internal node is a test on one of the input attributes. Tests are different depending on the nature, categorical or numerical, of the involved attribute. Let us denote by *Attr* an input attribute the decision tree learner considers for a split.

If *Attr* is categorical, let us denote by N_{vals} the size of $Vals = domain(Attr)$, i.e. the number of values *Attr* can take. Then, splits based on *Attr* have two possible shapes.

It can be a binary test $\mathit{Attr} \in Vals$ where $Vals \subset \mathit{domain}(\mathit{Attr})$: it is either true or false so the node will have two children branches. Splits on categorical attributes can also yield as many branches as possible values of the attribute, i.e. N_{vals}, unseen data will then follow the branch corresponding to its value for the considered attribute.

In this work, we will consider the first shape, binary split based on subset belonging. The number of possible splits is then $2^{N_{vals}} - 2$, which is the number of subsets of $Vals$ excluding the empty set $\emptyset$ and the full set $Vals$ itself. Indeed, these two splits would yield no gain, since one of the two branches would be empty: $\mathit{Attr} \in \emptyset$ always fails, while $\mathit{Attr} \in Vals$ always succeeds. In order not to consider all splits, and achieve linear complexity with respect to the number of possible values instead of exponential, we proceed as explained in pseudo-code in Algorithm 2.2. First, only splits with the form $\mathit{Attr} == \mathit{val}$, or $\mathit{Attr} \in \{\mathit{val}\}$, are evaluated, as shown in lines 6 to 9. This gives a ranking $r = [v_1, v_2, \cdots, v_{N_{vals}}]$ of the elements of $Vals$, according to the score achieved by split $\mathit{Attr} == v_k$, from the best split to the worst. Then, the split chosen is of the form $\mathit{Attr} \in \{v_k | 1 \leq k \leq k_{max}\}$. The right-hand side of the test is a set of size k_{max} with $0 < k_{max} < N_{vals}$. It is built by adding elements to the subset one by one, according to the ranking, as shown in the loop from line 12 to 26. The first split evaluated is $\mathit{Split}_1 = \mathit{Attr} \in \{v_1\}$, which will be the reference. Then v_2 is added to the set if it yields an improvement of the split, i.e. if $\mathit{Split}_2 = \mathit{Attr} \in \{v_1, v_2\}$ is a better split than Split_1, otherwise the search stops and Split_1 is chosen as the best split achievable from Attr. In other words, we keep adding elements to the subset following the ranking until adding one element does not improve the quality of the split. The number of splits tested in this phase is at most $N_{vals} - 1$, since all values cannot be added to the set. Including the first, singleton evaluation, phase, the overall number of splits evaluations is lower than $2 \cdot N_{vals} - 1$.

If Attr is numerical, splits based on Attr will be of the following form: $\mathit{Attr} \geq \mathit{threshold}$ where $\mathit{threshold}$ is a constant. This split is binary, and yields two branches. Numerical attributes can also be discretized into several intervals, which turns them into categorical attributes, and yields one branch per interval of discretization. In this work, as Quinlan's C4.5, we will consider the first kind. Every possible value of Attr on the current training instances will be tested as *threshold*. The number of splits tested is then of the order of the number of training instances. Algorithm 2.3 details this in pseudo-code.

To achieve decision tree induction, a metric is needed to evaluate and compare splits. Such a metric should separate training examples to create homogeneous partitions with respect to the output attribute. In a classification task, this is materialized by partitions that contain training examples from only one class, i.e. training instances with the same value for the output attribute.

Several metrics exist to evaluate the purity of a set of instances in a classification setting. Let us denote by D the set of instances, C the number of classes, i.e. the number of possible values of the target attribute, and p_i, with $1 \leqslant i \leqslant C$, the proportion of instances in D with the i$^{\text{th}}$ class. Here are three possibilities for impurity measure $\mathit{Imp}(D)$ of the set of instances D:

Algorithm 2.2 EvaluateCategoricalFeature

```
Input: feature: feature to evaluate splits, train: labeled training set, target: target attribute
Output: bestSplit: best split found for feature

bestScore $\leftarrow$ WORST_SCORE_FOR_METRIC
bestSplits $\leftarrow$ []
map $\leftarrow$ InitEmptyMap()
for all value $\in$ feature.domain do
    score $\leftarrow$ EvaluateSplit(feature == value)
    map.Put(value, score)
end for
keepGoing $\leftarrow$ true
valueSet $\leftarrow$ InitEmptySet()
for all value $\in$ map keys ordered by decreasing score and keepGoing do
    valueSet.Add(value)
    spl $\leftarrow$ CreateSplit(feature $\in$ valueSet, train)
    EvaluateSplit(spl, target)
    if spl.score $\geqslant$ bestScore then
        if spl.score $>$ bestScore then
            bestScore $\leftarrow$ score
            bestSplits $\leftarrow$ []
        end if
        bestSplits.Add(spl)
    else
        keepGoing $\leftarrow$ false
        valueSet.Remove(value)
    end if
end for
bestSplit $\leftarrow$ bestSplits.OneRandomElement()
return bestSplit
```

$$p_i = \frac{|\{d \in D | v_Y(d) = i\}|}{|D|}$$

$$MajorityClass : Imp(D) = 1 - \max_{1 \leqslant i \leqslant C} p_i$$

$$Gini : Imp(D) = 1 - \sum_{i=1}^{C} p_i^2$$

$$Entropy : Imp(D) = \sum_{i=1}^{C} -p_i \cdot \log_2 p_i$$

Algorithm 2.3 EvaluateNumericalFeature

```
Input: feature: feature to evaluate splits, train: labeled training set, target: target attribute
Output: bestSplit: best split found for feature

bestScore ← WORST_SCORE_FOR_METRIC
bestSplits ← []
for all threshold ∈ feature.PossibleValuesForInsts(train) do
    spl ← CreateSplit(feature $\geq$ threshold, train)
    score ← EvaluateSplit(spl, target)
    if score $\geqslant$ bestScore then
        if score $>$ bestScore then
            bestScore ← score
            bestSplits ← []
        end if
        bestSplits.Add(spl)
    end if
end for
bestSplit ← bestSplits.OneRandomElement()
return bestSplit
```

For a binary (two-class) classification task, there is only one degree of freedom in the class proportions. Indeed, the class proportions must add up to 1 over all classes, so we have $p_2 = 1 - p_1$, and all formulas can be written with respect to p_1. The evolution of the three measures is shown in Figure 2.2. We show evolution of the actual metrics value on the left, i.e. according to the formulas given above, and normalized so that the maximum values of each metric coincide on the right.

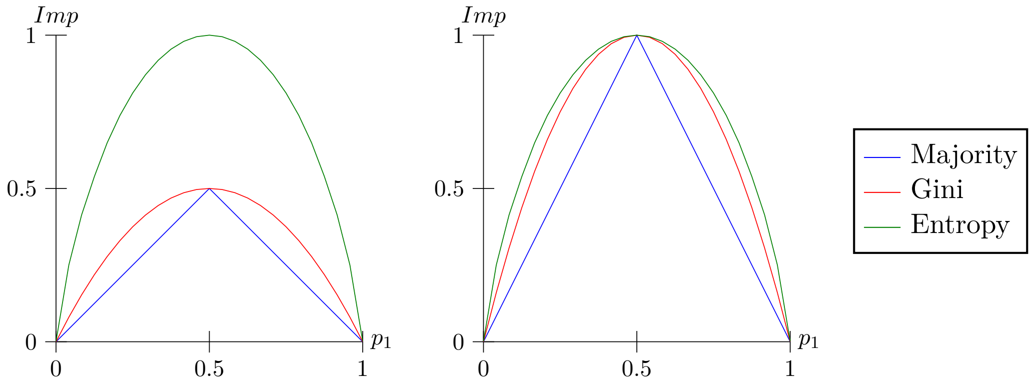


Figure 2.2 – Impurity score in a binary classification task with respect to the prior probability, using the original formulas (left), and normalized (right).

The majority class measure is the proportion of instances in the set that do not belong to the majority class. The proportion of instances that belong to the majority class is the maximum of all p_i proportions. Thus, the proportion of instances that do not belong to the majority class is the complementary to 1 of this maximum.

The Gini impurity, used by Breiman's Classification and Regression Trees (CART) (Breiman et al. 1984), measures the probability for an instance from the set to be classified wrong by a random classifier using the current class probabilities. In probabilistic terms, it is the complementary of the probability for an instance to be classified accurately by a random classifier. Let us consider an instance from the training set. It belongs to class i with probability p_i. Assuming that the instance is of class i, a random classifier using class probabilities from the training set will classify it accurately with probability p_i. Summing up over all classes, a random instance from the training set will be classified accurately with a probability equal to the sum of squares of class probabilities. The impurity measure is the complementary, which gives the formula shown above.

Finally, the Shannon entropy, derived from information theory, measures the number of bits needed to encode the class information of an instance. If only one class is present in the instance set, i.e. the set is pure, no encoding is needed since the class of an instance is known for sure, and Shannon entropy reaches its minimum. The opposite case is when all classes have the same probability, in this case the same encoding is needed for each class, the encoding length is maximal and Shannon entropy reaches its maximum.

The three measures equal zero when the instance set is pure with respect to the output attribute, and reach their maximum when the set is the most undecided, i.e. when all classes have the same probability.

For a regression task, relying on the strict value for the output attribute may be too drastic, since the value of the output attribute is numerical and may be unique for every example. This would lead to an overly big decision tree, with nearly one leaf per instance. A criteria for partition homogeneity is in this case a low variance of the output attribute among examples in the same partition. The impurity $Imp(D)$ is then the variance of the output value $v_Y(d)$ over D:

$$Imp(D) = \left(\frac{1}{|D|} \cdot \sum_{d \in D} v_Y(d)^2\right) - \left(\frac{1}{|D|} \cdot \sum_{d \in D} v_Y(d)\right)^2$$

For both tasks, the quality of a split is given by the impurity reduction induced by the split. Let us denote by n the number of children branches induced by the split, and D_k, $1 \leqslant k \leqslant n$, the partitions of D induced by the split, i.e. D_k is the subset of D which goes down the k$^{\text{th}}$ child branch after the split. The gain is measured by the difference between the original impurity $Imp(D)$ and the after-split impurity which is the average of the resulting $Imp(D_k)$, weighted by the proportion of examples in the k$^{\text{th}}$ branch. Formally:

$$Gain(Split) = Imp(D) - \sum_{k=1}^{n} \frac{|D_k|}{|D|} \cdot Imp(D_k)$$

For a given node of the tree, the split chosen is the one maximizing the gain among all possible splits.

When the impurity measure is entropy, the gain is called information gain. It is the foundation of Quinlan's Iterative Dichotomiser 3 (ID3) (Quinlan 1986). ID3 has then been extended by Quinlan to C4.5 (Quinlan 1993), which introduces splits on numerical attributes through comparison to a threshold.

Another split quality measure has been derived from information gain, called gain ratio. The aim is to penalize splits that induce a lot of small children branches, by taking the number and size of the branches into account. This split information is defined as:

$$SplitInfo(Split) = \sum_{k=1}^{n} - \frac{|D_k|}{|D|} \cdot log_2\left(\frac{|D_k|}{|D|}\right)$$

The information gain ratio is then defined as:

$$GainRatio(Split) = \frac{Gain(Split)}{SplitInfo(Split)}$$

2.1.3 Naive Bayesian Classifiers

Naive Bayesian classifiers are probabilistic classifiers for classification tasks. Contrary to decision trees, which usually return a "hard" prediction, i.e. only a label as class prediction, a Naive Bayesian classifier returns a score for each class, i.e. for each possible output value. First, we introduce useful probabilities notations:

- X: the vector of input attributes.
- Y: the output attribute.
- $P(X) = P(X = x)$: the probability distribution of the inputs.
- $P(Y) = P(Y = y)$: the probability distribution of the output, also called prior probability distribution.
- $P(Y|X) = P(Y = y|X = x)$: the conditional probability distribution, functional relation between the inputs and the output.
- $P(X|Y) = P(X = x|Y = y)$: the conditional probability distribution, functional relation between the output and the inputs.
- $P(X, Y) = P(Y|X) \cdot P(X) = P(X|Y) \cdot P(Y)$: the joint probability distribution, actual distribution of the data.

We consider all features, both inputs and output, to be categorical. Therefore all probability distributions are discrete. We use the following notations:

- $\{y_1, y_2, \cdots, y_C\}$: the C possible values of the output attribute Y.
- $X = (X_1, X_2, \cdots, X_a)$: the a input attributes.
- $\{x_{i1}, x_{i2}, \cdots, x_{in_i}\}$: the n_i possible values of input attribute X_i, i.e. the domain of attribute X_i.

The idea behind the Naive Bayes algorithm is to compute a score for each class to make a prediction. This score of a given class is related to the probability of the example to be of this class given the values of the input attribute. The greater the score, the more likely the example is of the given class. This conditional probability can be rewritten using Bayes' theorem:

$$P(Y = y_i | X_1, X_2, \cdots, X_a) = P(Y = y_i) \cdot \frac{P(X_1, X_2, \cdots, X_a | Y = y_i)}{P(X_1, X_2, \cdots, X_a)}$$

We observe that the probability at the denominator of the right-hand side simply is the probability of the input combination of values. It is not related to the class for which we want to compute the score. Therefore, we can drop it and define for class y_i the score s_i as:

$$s_i = P(Y = y_i) \cdot P(X_1, X_2, \cdots, X_a | Y = y_i)$$

The Naive Bayes classifier is called “naive” because of the strong independence assumption it is based upon. It considers the individual input attributes distributions to be independent. Therefore, we can rewrite the previous formula:

$$s_i = P(Y = y_i) \cdot \prod_{j=1}^{a} P(X_j | Y = y_i)$$

All elements of this formula are easy to compute from the training set:

- $P(Y = y_i)$ is the prior probability of class y_i, it is estimated as the proportion of examples in the training set of class y_i.
- $P(X_j | Y = y_i) = P(X_j = x_{jl} | Y = y_i)$ is the conditional probability of the input attribute value given the output attribute value, which is the proportion of examples in the training set of class y_i that have value x_{jl} for attribute X_j.

Scores s_i are computed for each class and, for a usual classification task, the predicted class is the one achieving maximum score, i.e.:

$$\begin{aligned} Prediction &= y_c \\ \text{with } c &= \underset{1 \leqslant i \leqslant C}{argmax}(s_i) \end{aligned}$$

These supervised learning models are designed for the attribute-value setting, i.e. prediction on a single table. In relational data mining, data is represented by several tables, thus there is more information than the sole attributes of the target table for prediction that can be used to learn a model. Therefore, these models will need adaptation to be used in a relational setting.

2.2 Relational Representation of Data and Learning Methods

This section introduces notations and the state of the art about relational data mining, illustrated by the example of an urban block dataset.

Contrary to the attribute-value setting, data in the relational setting is represented across several tables, corresponding to different kinds of objects linked by relationships. The relationships can have the following cardinalities: one-to-one, one-to-many, or many-to-many. In relational data mining, we are interested in making predictions on one kind of object in particular, that we call the main object. The prediction is based upon information on other kinds of objects linked to the main one, the secondary objects. This leads to a rich representation, where many features can be expressed. It implies a big feature space, which often cannot be searched exhaustively. However, several families of methods have been developed to construct relevant features for relational data mining, along with extensions of classic attribute-value learning algorithms.

2.2.1 Relational Database

The notations for relational databases are inspired by those introduced in (Getoor 2001). A relational database, noted *DB*, is seen as a set of N tables $DB = \{T_1, T_2, \ldots, T_N\}$.

A table T, following the notations from Section 2.1, has:

- At least one primary key attribute, denoted by T.K.
- Descriptive attributes, their number in table T is denoted by *a(T)*, their set is denoted by $T.A = \{T.A_1, T.A_2, \ldots, T.A_{a(T)}\}$. The definition domain of a descriptive attribute is denoted by $domain(T.A_k)$. If the attribute is categorical, then it is a set. If it is numerical, then it is an interval of $\mathbb{R}$.
- Foreign keys, their number in table T is denoted by *f(T)*, their set is denoted by $T.F = \{T.F_1, T.F_2, \ldots, T.F_{f(T)}\}$. Each foreign key references the primary key of another table. The primary key referenced by the foreign key $T.F_k$ is denoted by $key\_referenced(T.F_k)$. The referenced table is denoted by $table\_referenced(T.F_k)$.

For an object t in table T_i, the value of attribute $T_i.A_j$ is denoted by $v_{T_i.A_j}(t)$. In supervised learning, the value of one descriptive attribute of a specific table is to be predicted. This table will be referred to as the main table.

Example 2.3. The Urban Block dataset.

We introduce an example with two tables: a table of urban blocks and a table of buildings. Those two tables are linked through an aggregation relationship, i.e. a one-to-many relationship: one urban block is linked to several buildings. The aim is to predict the kind of block in presence: collective housing, individual housing, mixed housing, specialized area, empty area, or urban tissue. The database is represented in Figure 2.3.

The table of urban blocks, i.e. the main table, has:

- A primary key: *block_id*.
- Five descriptive attributes: the class attribute *class*, and 4 numerical attributes *density*, *convexity*, *elongation* and *area*.
- No foreign key.

The table of buildings, i.e. the secondary table, has:

- A primary key: *building_id*.
- Three descriptive attributes: *convexity*, *elongation* and *area*.
- One foreign key: *block_id* which references the table of urban blocks.

■

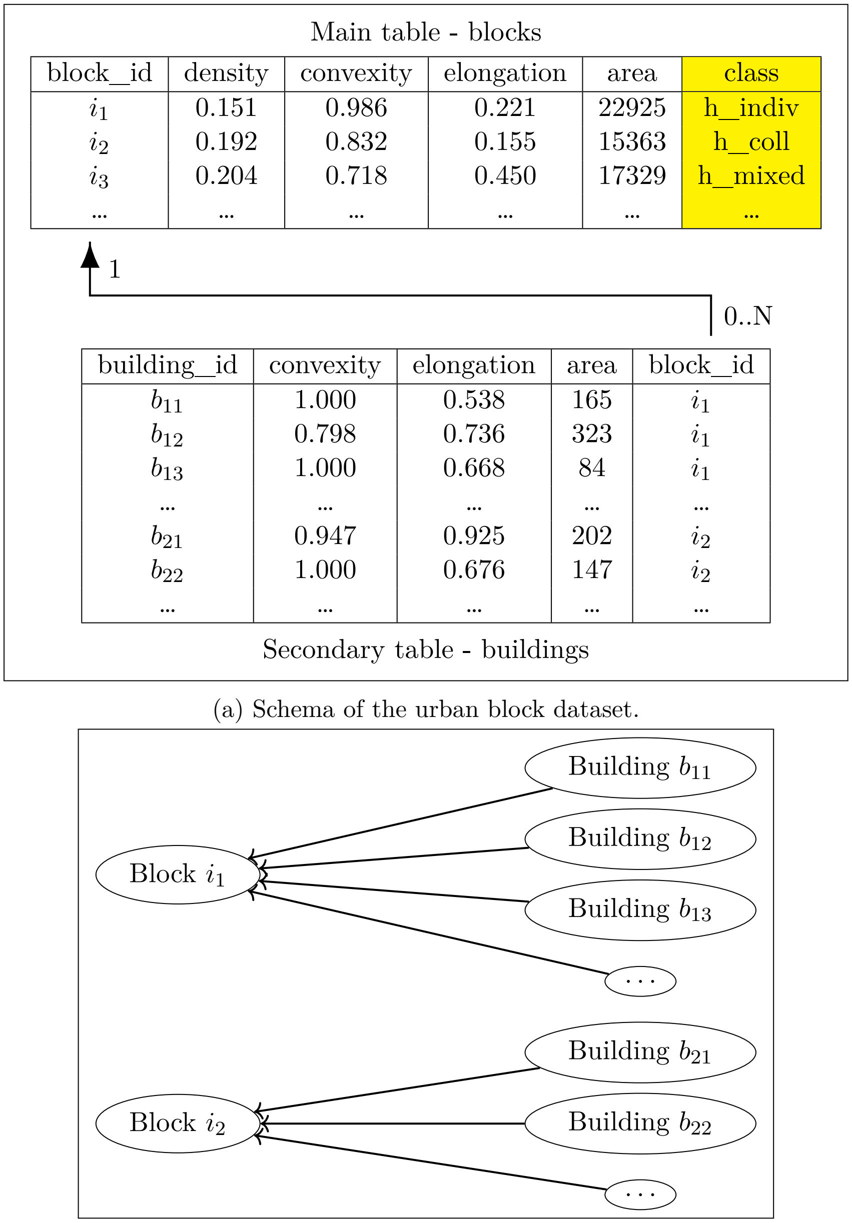
In terms of supervised learning, the introduction of secondary objects having their own attributes causes a surge in the number of possible features for the main objects, since the secondary objects along with their attributes can be used to create such features. Contrary to attribute-value learning, where the number of features matches the number of attributes, features of the main objects in the relational setting can be constructed in many different ways and thus their number is combinatorial, making a full consideration of all features impossible. Approaches to relational data mining all have in common the use of a language bias, in other words a vocabulary guiding the construction of features, a control structure to avoid combinatorial explosion.

2.2.2 Learning Paradigms for the Relational Setting

Relational data mining is addressed by several paradigms. Multiple-instance learning is a similar concept, although it is usually not classified as a relational paradigm. Usual relational data mining algorithms can be divided into two main categories: Inductive Logic Programming-based, and propositionalization methods. The three paradigms will be presented in this subsection.

Multiple-Instance Learning

Multiple-instance learning, for which an overview is given in (Amores 2013), consists in a classification task of bags of instances. Therefore, to relate to relational data mining



Main table - blocks

block_id	density	convexity	elongation	area	class
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixed
...	...	...	...	...	...

building_id	convexity	elongation	area	block_id
b_{11}	1.000	0.538	165	i_1
b_{12}	0.798	0.736	323	i_1
b_{13}	1.000	0.668	84	i_1
...	...	...	...	...
b_{21}	0.947	0.925	202	i_2
b_{22}	1.000	0.676	147	i_2
...	...	...	...	...

Secondary table - buildings

(a) Schema of the urban block dataset.

(b) One-to-many relationship in the urban block dataset.

Figure 2.3 – Illustration of the relational setting with the urban block dataset.

concepts, the bags are main objects, while the instances are secondary objects. The difference is that all predictive information is contained at the instance-level. Only the label, the target for prediction, is originally at the bag-level. There is also no relationship chaining, it considers only the bags and the instances. It can be seen as a relational data mining task with two tables with a one-to-many relationship from bags to instances. The associated classification task is also usually binary, i.e. with two classes usually labeled as “positive” and “negative”. Formally, the aim is to classify as positive or negative a bag $B = \{t_1, t_2, \cdots, t_k\}$, with $t_i, 1 \leqslant i \leqslant k$ the k instances of bag B.

Three main categories of learning exist for the multiple-instance setting. First, the Instance-Space paradigm considers learning a model at the instance-level. Since instances do not have a class label, two possible assumptions can be made. One is that a bag is positive if all its instances are positive, and negative bags contain only negative instances, in other words the instances get the label of their associated bag. Then, a scoring classifier is built to predict how likely an instance is to belong to a positive bag, and how likely it is to belong to a negative bag. The bag-level score is then obtained by averaging the instance-level scores, and a final prediction for the bag can be made based on this bag-level score. Another possibility is to consider that a positive bag contains at least one positive instance which makes it a positive bag, and that negative bags are negative because they do not contain any positive instance, i.e. they contain only negative instances. In this approach, a label has to be assigned to every instance based on these assumptions, labels of instances from negative bags are straightforward, but positive bags contain at least one positive instance, which means they may also contain negative instances. A common approach to do so is the Axis-Parallel Rectangles (Dietterich, Lathrop, and Lozano-Pérez 1997). Given the instance space $I = \{A_1, A_2, \ldots, A_n\}$ where $A_j, 1 \leqslant j \leqslant n$ the attributes of the instances, the aim is to find an axis-parallel rectangle such that, if an instance belongs to the rectangle, it is predicted as positive, and negative otherwise. Formally,

$$R = \{t = (a_1, a_2, \ldots, a_n) \in I | l_1 \leqslant a_1 \leqslant u_1 \wedge l_2 \leqslant a_2 \leqslant u_2 \wedge \cdots \wedge l_n \leqslant a_n \leqslant u_n\}$$
$$\text{with } a_j = v_{A_j}(t)$$

The instance-level score $f(t)$ is defined as:

$$f(t) = \begin{cases} 1 & \text{if } t \in R, \\ 0 & \text{otherwise.} \end{cases}$$

Rectangle R is optimized so that a maximum of positive bags contain at least one positive instance. Then, the bag-level score $F(B)$ is defined using a maximum rule:

$$F(B) = \overset{k}{\underset{i=1}{max}} \left(f(t_i) \right)$$

This maximum rule complies with the definition of positive and negative bags: if there is at least one positive instance in the bag, it has a score of 1, and since it is the maximum score achievable by an instance, it will be the score of the bag which will be predicted positive. On the other hand, if there are only negative instances in the bag, scores of all instances will be 0, so will be the score of the bag and thus it will be predicted negative.

Secondly, the Bag-Space paradigm focuses on the bag-level, it is based on the definition of a comparison measure between two bags, a distance function or a kernel-based comparison. A distance-based model, such as the K-Nearest Neighbors or a Support Vector Machine, can then be used.

Since an instance is a point in the instance space, and a bag is a set of instances, any distance between two sets of points can be used. Let us consider two bags $B_1 = \{t_{11}, t_{12}, \cdots, t_{1n_1}\}$ and $B_2 = \{t_{21}, t_{22}, \cdots, t_{2n_2}\}$. A possible distance function which can be used is the minimal Hausdorff distance defined by:

$$D(B_1, B_2) = \underset{1 \leqslant i \leqslant n_1, 1 \leqslant j \leqslant n_2}{min} (d(t_{1i} - t_{2j}))$$

where $d(t_1, t_2)$ is a distance between two points in the instance space, e.g. euclidean distance.

The distance can be converted into a kernel, e.g. a Gaussian kernel defined by:

$$K(B_1, B_2) = exp\left(-\gamma \cdot D(B_1, B_2)\right)$$

with γ the scale parameter of the Gaussian kernel.

Finally, the Embedded-Space paradigm consists in a transformation of the bag-space into an attribute-value representation of the bags. Then, any attribute-value learning algorithm can be used. This approach follows the same idea as propositionalization algorithms which we will present later and also consists in mapping the relational representation into an attribute-value representation.

There are two main possible approaches to this transformation. The first one is based on aggregating the set of instances corresponding to each bag. This can be achieved by taking the average of all instances in the set, and use the attributes values as features of the bag, in other words we summarize the bag to its average instance.

The other possible approach is to use a vocabulary-based method. It consists in labeling the instances, i.e. assigning classes to instances. Since labels are originally present only at the bag-level, this is done in an unsupervised way, by clustering the instances. Let us consider the clustering algorithm discovered C clusters of instances, the “vocabulary” contains description of each individual cluster, for instance its mean vector and covariance matrix in terms of the attributes from the original instance space.

With this vocabulary, a cluster membership likelihood of instance $t = (a_1, a_2, \dots, a_n)$, denoted $f_j(t), 1 \leqslant j \leqslant C$, can be defined for each cluster. If cluster j is represented by a mean vector μ_j of length n and a covariance matrix Σ_j with size $n \times n$, we can define this likelihood according to the multivariate normal law:

$$f_j(t) = \frac{1}{(2\pi)^{C/2} det(\Sigma_j)^{1/2}} exp\left(-\frac{1}{2}(t - \mu_j)^T \Sigma_j^{-1} (t - \mu_j)\right)$$

Then, the bag of instances $B = \{t_1, t_2, \cdots, t_k\}$ is represented as $B = \{v_1, v_2, \cdots, v_C, y\}$ with y its class label and one feature per cluster, which is the sum for the given cluster of membership likelihoods for this cluster over all instances in the bag. If we denote by

v_j this feature for cluster j:

$$v_j = \sum_{i=1}^{k} f_j(t_i)$$

With this representation, a regular attribute-value learning algorithm can be used. The idea behind the Embedded-Space paradigm is the same as behind propositionalization in the relational setting. It is one of the two main families of algorithms used in relational data mining, which we will now focus on.

Propositionalization

Propositionalization methods consist in transforming relational data into attribute-value data, in other words going from several tables to a single one. After this transformation, regular attribute-value learners can be used to build a model. This is interesting, since there are many more learners in the literature for attribute-value learning than for relational data mining. Their second advantage is that the features generated for attribute-value learning are either predefined by the language bias, or constructed in the context of a language bias and kept to a small number, contrary to full-fledged relational data mining approaches.

Recent propositionalization approaches include:

- RelF (Kuzelka and Zelezný 2009) finds a relevant feature set, and reduces it to small conjunctions allowing to compose the original feature set, using relevancy monotonicity as opposed to the frequency monotonicity generally used in rule learning.
- LBP (Dinh, Vrain, and Exbrayat 2012) builds boolean features as chains of variable literals.
- BCP (Ontañón and Plaza 2015) transforms examples into sets of properties. A vocabulary of properties is a common set of boolean features obtained through a desintegration operation on the relational examples.
- Wordification (Perovsek et al. 2015) transforms a relational example into a bag of words, and applies text mining techniques such as document frequencies to obtain numerical features from the set of words.
- Relaggs (Krogel and Wrobel 2003) and Polka (Knobbe, Haas, and Siebes 2001) build numerical aggregates to achieve propositionalization.

Among these approaches, only Relaggs and Polka deal with numerical attributes without discretizing them beforehand. They build simple aggregates: for a given main object, they aggregate all secondary objects related to it. However, all the secondary objects may not be relevant for prediction. For instance, the average area of buildings with low elongation may be more relevant for prediction than the average area of all buildings. The introduction of these relevant secondary objects is possible using the Inductive Logic Programming-based approaches.

Inductive Logic Programming and Existential Quantifier-based Methods

The other main family of methods to deal with relationships in relational data mining is the use of the existential quantifier, popularized by Inductive Logic Programming (ILP) (Lavrac and Dzeroski 1994). In the ILP formalism, data as well as models are represented as logic programs, using first-order predicates. This is usually written using Prolog formalism. First, let us introduce basic ILP terminology:

- The most basic elements of a logic program are *constants* and *variables*. In Prolog formalism, they are distinguished by their first character: a constant starts with a lowercase letter, while a variable starts with uppercase. For instance, *a* and *blue* denote constants, while *X* and *Elem* denote variables.

- A *functor* is a constant immediately followed by a tuple delimited by round brackets. The number of elements in the tuple is called the *arity* of the functor. For instance, in *f(X, g(Y,a), b, h(Z))*, functor *g* has arity 2, functor *h* has arity 1, and functor *f* has arity 4. A function name followed by a tuple is a *predicate*. The arity of a predicate is the arity of its functor.

- A *rule* is denoted *Head :- Body* in Prolog, which is read as an implication from the right-hand side to the left-hand side, in other words “Head is true if Body is true”. For instance, *animal(X) :- cat(X)* means that if X is a cat, then X is an animal. The body of a rule consists in conjunctions and disjunctions of predicates, the conjunction operator, i.e. the logical *AND*, is denoted by “,”, while the disjunction operator, i.e. the logical *OR*, is denoted by “;”.

- A rule with an empty body, i.e. whose head is considered as true, is called a *fact*. For instance *cat(tom)* means that the constant *tom* is a cat. This notation is equivalent to *cat(tom) :- true*. According to the previous rule, *animal(tom)* holds, explicit statement of this fact is not needed.

- Facts and rules are referred to as *clauses*, and a set of clauses constitutes a *logic program*.

Listing 2.1 shows the urban block dataset from Figure 2.3 written as a logic program using the Prolog formalism.

```
block(i1, 0.151, 0.986, 0.221, 22925, h_indiv).
block(i2, 0.192, 0.832, 0.155, 15363, h_coll).
block(i3, 0.204, 0.718, 0.450, 17329, h_mixed).

building(b11, 1.000, 0.538, 165, 1).
building(b12, 0.798, 0.736, 323, 1).
building(b13, 1.000, 0.668, 84, 1).
building(b21, 0.947, 0.925, 202, 2).
building(b22, 1.000, 0.676, 147, 2).

block_is_not_empty(IdBlock) :- block(IdBlock, _, _, _, _, _), building(_, _,
    _, _, IdBlock).
```

```
12 wide_area_building(IdBuild) :- building(IdBuild, _, _, AreaBuild, _),
       AreaBuild > 200.
```

Listing 2.1 – A logic program in Prolog for the urban block dataset.

Lines 1 to 3 are facts that define 3 blocks, through the predicate *block* of arity 6. Arguments of the predicate correspond to the columns of the table of blocks in the database: they respectively match with the id of the block, its density, its convexity, its elongation, its area, and its class. Lines 5 to 9 are facts that define 5 buildings, through the predicate *building* of arity 5. Its arguments respectively correspond to the id of the building, its convexity, its elongation, its area, and the id of the block the building is a part of.

Line 11 defines the predicate *block_is_not_empty* with a rule: id *IdBlock* is the id of a non-empty block if there is at least one building contained in the block of id *IdBlock*. In Prolog, the "_" symbol acts as wildcard, it usually replaces an argument whose value is used nowhere else in the rule. Thus, we are only interested in the existence of a building in the block; we are not interested in which building exactly is in the block, but only in the fact *building(_, _, _, _, IdBlock)* being present at least once in the logic program. For instance, *block_is_not_empty(i1)* holds because *block(i1,_,_,_,_,_)* holds according to line 1, and *building(_,_,_,_,i1)* holds three times (once would be enough), for instance in line 5. The same reasoning holds with block *i2*. However, *block_is_not_empty(i3)* does not hold: no fact matching *building(_,_,_,_,i3)* is present in the logic program, which means there is no building in block *i3*, thus it is empty.

Line 12 defines the predicate *wide_area_building*: *IdBuild* is the id of a building with wide area if the area of the building *IdBuild*, denoted by variable *AreaBuild*, is greater than 200. According to this rule and the facts available in the logic program, buildings of id *b12* and *b21* are wide-area buildings.

Many approaches are based on the ILP formalism. Commonly used approaches include:

- Foil (Quinlan 1990) is a rule learner. Given a set of positive and negative examples and a set of background knowledge predicates, it learns the concept represented by the positive examples, i.e. it looks for a rule that covers positive examples and no negative examples. This is achieved through an incremental hill-climbing method: starting from an empty clause, at each step of the algorithm it looks for the best literal to add to the body of the rule according to an information-theory-based metric. The positive examples covered by the rule are removed from the training set until all positive examples are covered.

- Progol (Muggleton 1993) looks for a list of clauses covering the examples. Starting from the most specific clause covering a given example, it reduces the clause to minimize Occam compression, adds the result to the background knowledge, and repeats this process until all examples are covered.

- Tilde (Blockeel and De Raedt 1998) is a decision tree learner, generalizing C4.5 (Quinlan 1993) to the relational setting. It builds decision trees using a top-down approach, having the possibility to introduce new variables, i.e. new objects, through the existential quantifier using a language bias indicating which refinements are allowed. The variables must be linked to already existing ones, and can be reused in deeper levels of the tree.

Among these approaches, only TILDE naturally deals with numerical attributes. Moreover, even though the ILP formalism is powerful and allows for advanced reasoning, it is limited by the use of the existential quantifier: secondary objects that are introduced verify individual properties, i.e. there is at least one secondary object with such properties. But relevant features can be built with the existence of at least two objects with the same properties, more generally the existence of at least n objects with the same properties where the value of n is to be determined by the learning algorithm. One of our aims will be to go beyond the existential quantifier to handle such properties. The idea of introducing the relevant secondary objects on-the-fly, when needed in the model, will be kept, and associated to the more expressive framework of aggregation. This combination is called **complex aggregation**, or aggregation of relevant subsets of secondary objects. Our work on complex aggregation will be presented in Part I.

2.3 Reframing

Knowledge reuse is a machine learning trending issue. A common process, in data-driven prediction applications, is the following:

1. Learn a prediction model, using training data.
2. Deploy it, for use in production, with no change over time.
3. When the model does not work anymore, go back to step 1, and retrain a model.

Typically, step 3 may occur when *something has changed in the data*, compared to the data used originally to train the model. This *something* can be a change of distribution of an attribute in the data. For instance, if a temperature-based model is used to predict clothes sales, and was trained using data from shops in a first city, it is fair to assume it will not be reliable when making predictions for a second city with a different climate. One could then retrain another sales prediction model for this second city. As opposed to this approach, based on retraining models for specific contexts (here: the different cities), we propose an alternative, based on learning one general, versatile, model, that can be used in different contexts, through a certain adaptation procedure, less expensive than full retraining.

This idea is called *reframing*, and is illustrated in Figure 2.4. A *versatile* model is trained in a first context, denoted by A, using training data from this context. We are now in presence of data from a new context, denoted by B. We have few labeled data available, not enough to retrain a model, so we want to reuse the model learned for context A. Reframing is the procedure which uses the few labeled data from context B, called *deployment* data, to adapt the model from context A so that it can be used in context B. The versatile nature of the model does not limit it to just two contexts: the full model from context A can be adapted not only to context B, but to any other context for which deployment data is available, context C from the schema for instance.

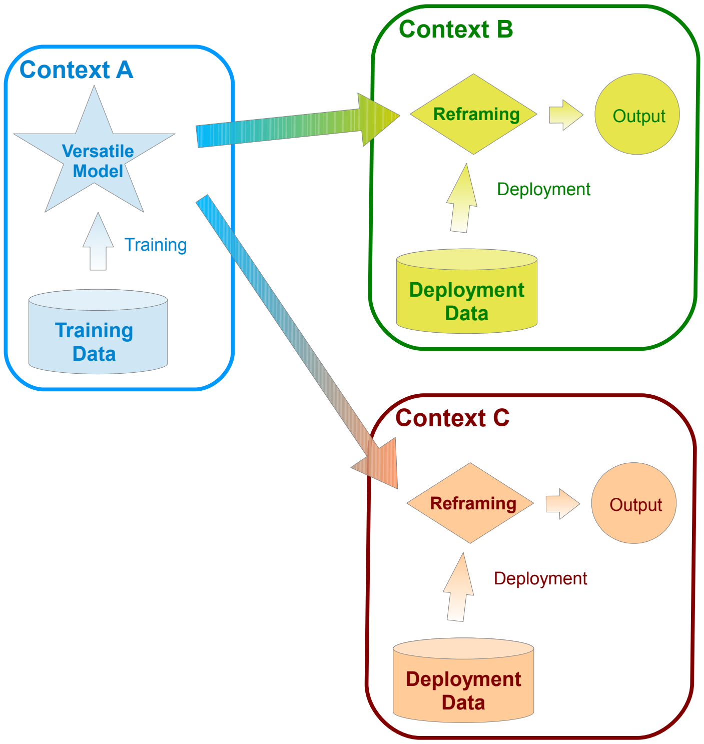


Figure 2.4 – Schema of the reframing[1]process.

In this section, we will characterize more precisely reframing and the associated notion of context. We will also provide an illustrative example.

2.3.1 Operating Context

In reframing terminology, a context is an ensemble of characteristics associated to a dataset. Two examples of this notion are:

Data distributions: the probability distribution of attributes of the dataset. For instance, mean and standard deviation of a numerical attribute over the dataset define a context, as they allow to model the attribute through a Gaussian distribution. This can concern both input and output attributes. A change of distribution between two contexts may cause a drop of performance of a model trained in a first context when deployed in a second context where attributes do not lie in the

[1] I am thankful to the REFRAME project, granted by CHIST-ERA, for this figure.

same range. Similarly to the example given above about clothing sales, if we train a model to predict ice-cream sales in Helsinki, Finland, and apply it in Barcelona, Spain, where temperatures are higher, the original model will not perform well.

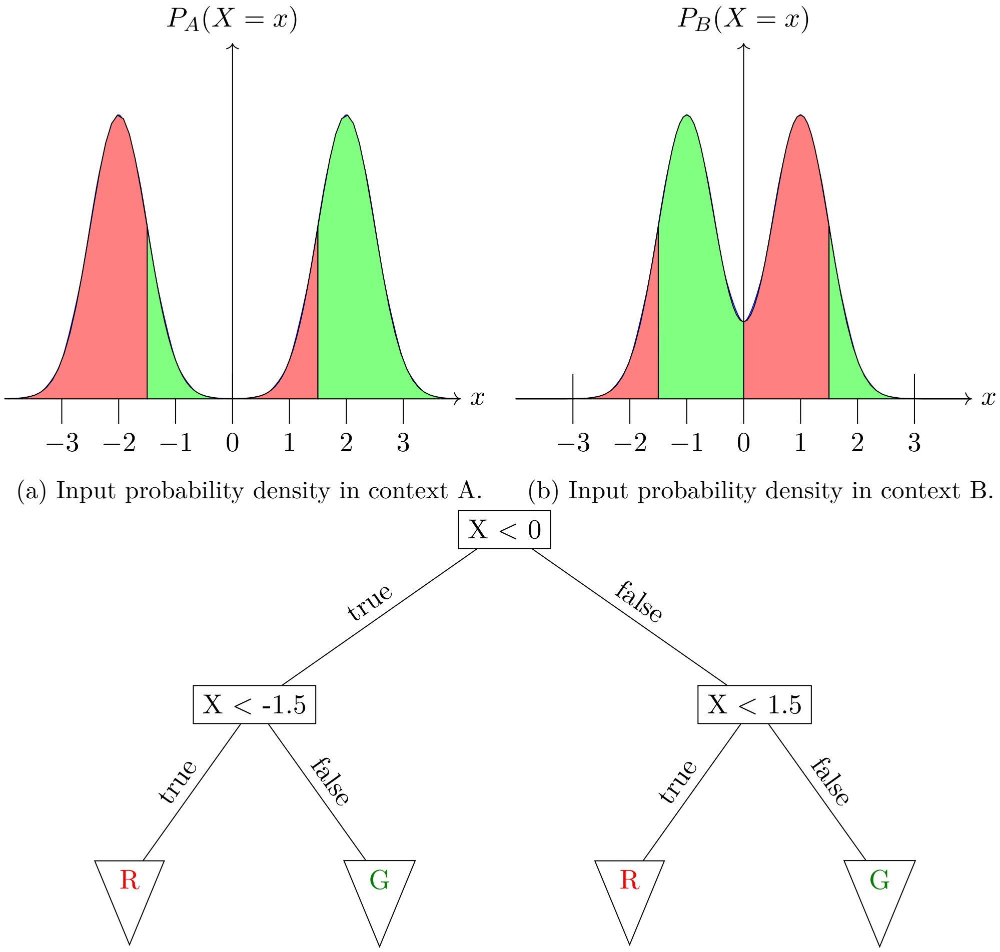
Loss functions: in some learning tasks, performance of learning may not be evaluated the same way depending on the context. For instance, in a medical application, classifying a patient as sane when he/she is actually sick does not have the same impact as classifying as sick an actually sane patient. Classification errors do not have the same impact and are assigned different costs, which gives birth to the field of cost-sensitive learning. These costs, which bias the accuracy measure towards penalizing the most costly errors, define a context. Costs can also be defined attribute-wise, in case attribute evaluation is more difficult for some attributes, e.g. in medical tests. This all relates more generally to the notion of loss introduced in Section 2.1.

In both cases, the aim is to reuse the original model, and adapt it to overcome the context change. Formally, this notion of context change between datasets A and B corresponds to the change of a characteristic between the two datasets. A special case is the notion of dataset shift, explained in (Moreno-Torres, Raeder, et al. 2012), which covers changes in data distribution. We reuse probability notations from Section 2.1. The main difference with the Naive Bayesian presentation is that probability distributions may be continuous since attributes are possibly numerical, as in the illustrative example we will provide.

Dataset shift is defined as a change in the joint probability distribution between A and B. More formally, dataset shift happens when $P_A(X, Y) \neq P_B(X, Y)$. This covers several types of changes detailed in (Moreno-Torres, Raeder, et al. 2012):

Covariate shift: This denotes a change in the input probability distribution while the functional relation, conditional distribution, remains the same. Formally, $P_A(X) \neq P_B(X)$ and $P_A(Y|X) = P_B(Y|X)$. We illustrate this in Figure 2.5. We take the example of one input X, related to one binary output Y, the two possible values being "R" and "G". The relation to the input is defined by the decision tree model given in Figure 2.5c. Figure 2.5a and 2.5b show the density of probability of the input X respectively in contexts A and B, along with the functional relation to the output: value G in green areas ($X \in [-1.5; 0[\cup[1.5; +\infty[$), and value R in red areas ($X \in [-\infty; -1.5[\cup[0; 1.5[$). This does not change with the context. The change resides in the distribution of the input X: it is in both cases the union of two Gaussian distributions with standard deviation 0.5, centered on -2 and 2 in context A, and on -1 and 1 in context B. Graphically, the shape of the curves is different, but the borders between colored areas are the same.

Prior probability shift: This denotes a change in the output probability distribution while the functional relation, conditional distribution, remains the same. Formally, $P_A(Y) \neq P_B(Y)$ and $P_A(X|Y) = P_B(X|Y)$. We illustrate it in Figure 2.6. In the



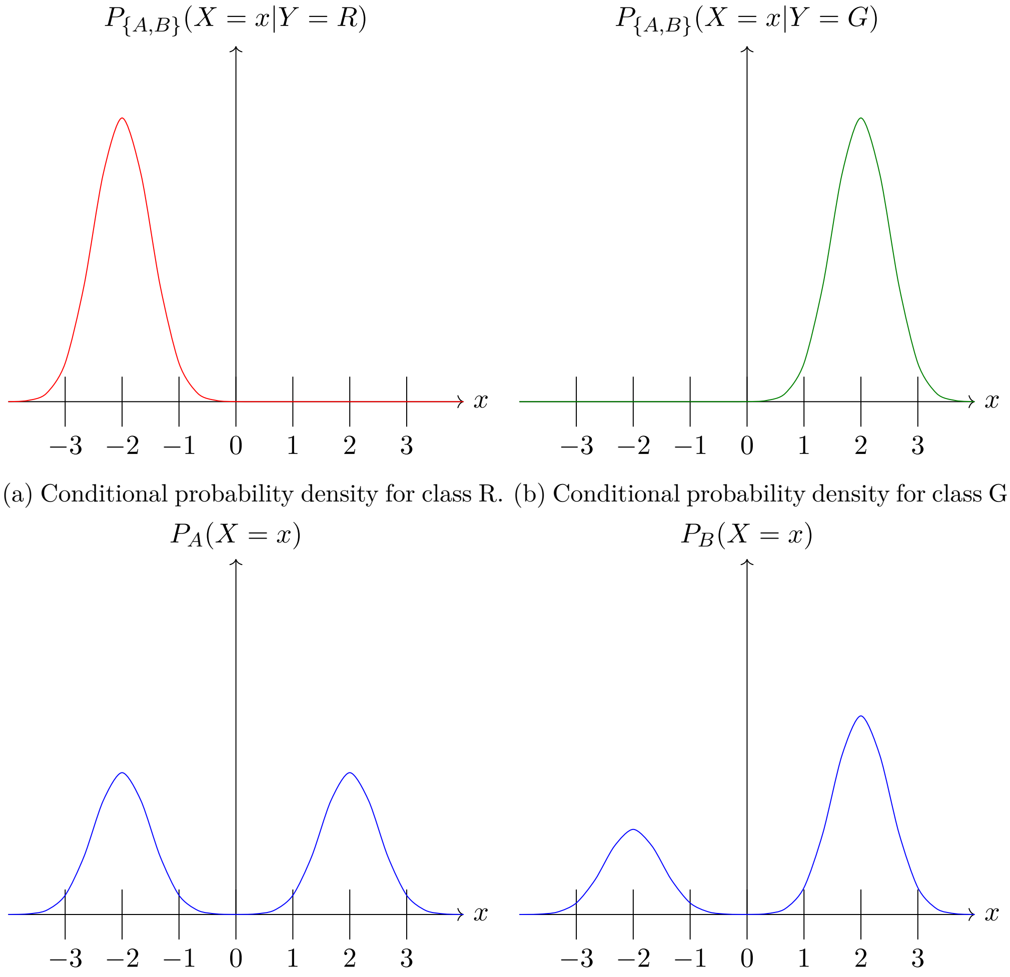
(a) Input probability density in context A.

(b) Input probability density in context B.

(c) Target model (conditional probability density) for both contexts.

Figure 2.5 – Illustration of covariate shift.

same binary classification task as before, we tackle the inverse problem: X is generated from Y, in other words a distribution is associated to each value of Y from which we sample to get the value of X. For value R, it is a Gaussian distribution with mean -2 and standard deviation 0.5, while for G, it has mean 2. These densities are shown in Figures 2.6a and 2.6b respectively. This does not change between contexts. The prior probabilities associated to R and G change, they are equal in context A, meaning both classes are present in the same proportions ($P(Y = R) = P(Y = G) = 0.5$), while in context B, class G is more present than class R, with priors $P(Y = R) = 0.3$ and $P(Y = G) = 0.7$. This change has an influence on the X distribution: in context A, both Gaussians have the same importance as shown on Figure 2.6c, while in context B, the Gaussian associated to class G has more importance in the final X distribution, as shown on Figure 2.6d.



(a) Conditional probability density for class R.

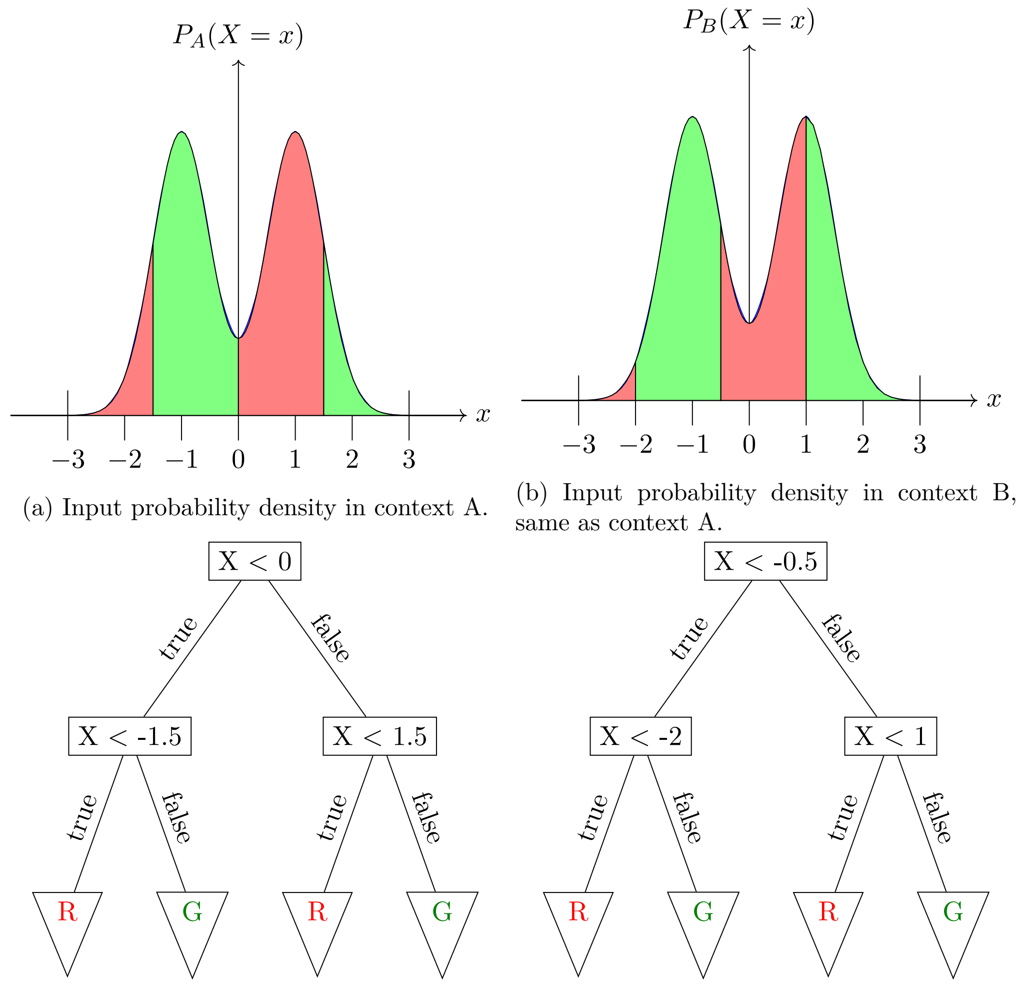
(b) Conditional probability density for class G.

(c) Input probability density in context A ($P(Y = R) = 0.5$ and $P(Y = G) = 0.5$).

(d) Input probability density in context B ($P(Y = R) = 0.3$ and $P(Y = G) = 0.7$).

Figure 2.6 – Illustration of prior probability shift.

Concept shift: This denotes a change in the functional relation, while the distribution of input/output remains the same. In other words, we have either $P_A(X) = P_B(X)$ and $P_A(Y|X) \neq P_B(Y|X)$, or we have $P_A(Y) = P_B(Y)$ and $P_A(X|Y) \neq P_B(X|Y)$. We illustrate it in Figure 2.7. Contrary to covariate shift, the input distribution of X does not change, it is the union of Gaussian distributions with standard deviation 0.5, centered on -1 and 1, thus curves in Figures 2.7a and 2.7b have the same shape. However, the relational function is different, as shown on decision trees from Figures 2.7c and 2.7d. The borders between classes in context B have been shifted by -0.5 with respect to the borders in context A, i.e. in context A, $Y = R \Leftrightarrow X \in [-\infty; -1.5[\cup [0; 1.5[$ and $Y = G \Leftrightarrow X \in [-1.5; 0[\cup [1.5; +\infty[$, while in context B, $Y = R \Leftrightarrow X \in [-\infty; -2[\cup [-0.5; 1[$ and $Y = G \Leftrightarrow X \in [-2; -0.5[\cup [1; +\infty[$.



(a) Input probability density in context A.

(b) Input probability density in context B, same as context A.

(c) Target model (conditional probability density) for context A.

(d) Target model (conditional probability density) for context B.

Figure 2.7 – Illustration of concept shift.

In presence of dataset shift, and in reframing tasks in general, the original model cannot be used as such. From this observation, there are two possible approaches: discarding completely the original model, or considering it bears useful information which can be reused.

2.3.2 Retraining VS Reframing

Given a training dataset in a first context A, and a deployment dataset in a second context B differing from A, we oppose two approaches that can be used to adapt to context B.

The first one is *retraining*: each time we want to adapt to a new context B, we collect enough data from that context to train a new model, suitable for the context. Two possible variants arise from the use or not of the original training data from context A. Its use is standard in transfer learning, which allows the use of the same training data in different ways, depending on the context information. If the original training data is not meant to be used, because it is not accessible anymore for instance, retraining is performed only using deployment data from the new context, and there may not be enough data to do so, or the data available may not be labeled.

To overcome these limitations, we introduce the *reframing* approach. The aim is to reuse the model learned on training data from context A, and to apply a transformation, the *reframing* procedure, that takes into account the deployment context, to make the original model usable in the deployment context. The main idea is the training of a *versatile model*, which captures at training additional reusable information. Then, the reframing procedure combines this reusable knowledge and contextual information to adapt the model.

We distinguish three kinds of reframing procedures, which are not mutually exclusive and can be combined:

Input reframing: The input X of the deployment data are pre-processed, with a numerical transformation for instance, and the model is used as such, using the transformed input.

Output reframing: The model is used as such on the original input, but its prediction for the output Y is altered to fit the deployment context.

Structural reframing: The structure of the model is modified, either through a systematic transformation or instantiation which takes the context into account, or by the use of only part of the model depending on the context.

Part II of this work introduces methods for input and output reframing of numerical features in attribute-value learning, and extends them to adapt complex aggregates for reframing in the relational setting. We also present an output reframing approach to multi-class cost-sensitive tasks.

Part I

Stochastic Optimization Heuristics for Complex Aggregation in Relational Learning

Chapter 3

Relational Learning Paradigms and Complex Aggregation

As presented in Section 2.2, data in the relational setting is modeled by several tables, representing different kinds of objects, linked by relationships. This setting opposes the attribute-value setting, in which data are represented by a single table associated to the object targeted for prediction. In relational learning, the target object has its own set of properties, including the target property for prediction, but it is also related to other kinds of objects, having their own sets of properties which may be used to perform prediction on the target object. The purpose of relational learning is to take advantage of these "secondary" properties to perform prediction on the main kind of object.

To achieve this goal, two main paradigms are used: Inductive Logic Programming, and propositionalization. The former focuses on the introduction of relevant secondary objects related to the main object on which prediction is to be performed, usually through the existential quantifier, i.e. the existence of at least one secondary object related to the main object verifying some desired properties. The latter flattens the relational, multi-table, representation, into an attribute-value representation with a single table. Aggregation is a sub-family of propositionalization methods, which summarizes the set of secondary objects related to one main object into properties of this main object, e.g. the number of secondary objects related to the main one.

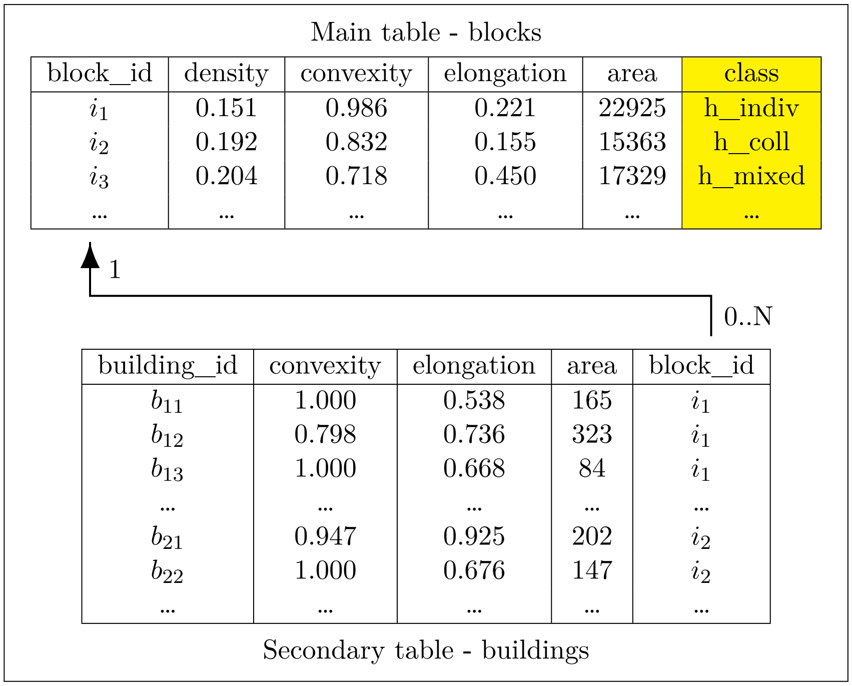
In this context, complex aggregation mixes both ideas: it consists in aggregating only a subset of relevant secondary objects related to the main object. This is done by filtering the secondary objects with a conjunction of conditions. This selection part induces an explosion of the size of the feature space. Thus, our objective is to elaborate heuristics to explore this feature space in a non-exhaustive way, to introduce complex aggregate features in a decision tree model.

The aim of this chapter is mostly to formalize complex aggregation and situate it with respect to state of the art in relational data mining. We also make a first algorithm proposal which will be improved in the next chapter. In Section 3.1, we detail the relational learning algorithms we compare to. In Section 3.2, we define formally complex aggregate features, and describe directly related work. In Section 3.3, we sketch a first, novel, hill-climbing algorithm to explore the complex aggregate search space, and consider ways to deal with some difficulties related to the introduction of complex aggregates. Finally, in Section 3.4, we draw first conclusions about this algorithm.

3.1 Relational Decision Trees and Propositionalization by Aggregation

In this section, we detail the two approaches we will compare to. The first is the aggregation-based propositionalization approach Relaggs, and the second is the relational decision tree learner based on the Inductive Logic Programming formalism Tilde.

3.1.1 RELAGGS: Propositionalization by Aggregation



Main table - blocks

block_id	density	convexity	elongation	area	class
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixed
...	...	...	...	...	...

building_id	convexity	elongation	area	block_id
b_{11}	1.000	0.538	165	i_1
b_{12}	0.798	0.736	323	i_1
b_{13}	1.000	0.668	84	i_1
...	...	...	...	...
b_{21}	0.947	0.925	202	i_2
b_{22}	1.000	0.676	147	i_2
...	...	...	...	...

Secondary table - buildings

Figure 3.1 – Schema of the urban block dataset.

Relaggs (Krogel and Wrobel 2003) and Polka (Knobbe, Haas, and Siebes 2001) build numerical aggregates to achieve propositionalization. For each main object, they aggregate the set of related secondary objects using standard aggregation functions: *count* of the secondary objects, *average*, *minimum*, *maximum*, *sum* and *standard deviation* of numerical attributes, *proportion of objects with a particular value* for categorical attributes. Formally, given a set S of secondary objects related to a given main object, f

an aggregation function among the ones enumerated above, and A a numerical attribute of the secondary objects, and denoting by $v_A(t)$ the value of attribute A for a secondary object t, the aggregation operator is defined as:

$$aggregation(f, A, S) = f(\{v_A(t), t \in S\})$$

The difference between Polka and Relaggs is the handling of nested relationships: when the secondary table is linked in a one-to-many relationship to another table, which can itself be linked to another table..., Polka applies aggregation operators to the deepest relationships first, up to the main table, while Relaggs joins the secondary table to the lower level tables depending on it, and then applies aggregation operators to the columns of the joined table.

For each attribute in the secondary table, every relevant aggregation function with respect to the type of attribute (categorical or numerical) is applied, and this result in an enriched main table, which contains the original columns of the main tables and the ones resulting from the aggregation of secondary objects, as shown in Table 3.1.

block_id	class	density	area	elongation	convexity		cnt	min_area	max_area	sum_area	avg_area	stddev_area		avg_elong	...	sum_elong	min_conv	...	max_conv
i_1	h_indiv	0.160	34657	0.178	0.737		43	45	242	5562	129	41.1		0.800	...	34.387	0.861	...	1.000
i_2	h_coll	0.129	12860	0.280	0.911	...	9	97	232	1659	184	49.3	...	0.626	...	5.632	0.999	...	1.000
i_3	h_mixed	0.276	4088	0.148	0.990		7	105	340	1126	161	81.1		0.609	...	4.264	0.782	...	1.000
...	...	...	...	...	...		...	...	...	...	...	...		...	...	...	...	...	...

Table 3.1 – Urban block dataset propositionalized with Relaggs.

For instance, the value of column *avg_area* for block of id i_1 is the average of the values of the area attribute over the set of buildings in block i_1, i.e. buildings b_{11}, b_{12}, b_{13}...

Propositionalization algorithms themselves are not learning algorithms: they just modify the representation of data. A model is learned using data in this new representation using a proper learning algorithm. For experimental comparison, we will consider the use of J48 implementation of C4.5 decision tree learner (Quinlan 1993) in Weka (Hall et al. 2009), after propositionalization with Relaggs.

```
block(IdBlock, Density, ConvBlock, ElongBlock, AreaBlock, h_indiv) :-
    ElongBlock > 0.2, !.
block(IdBlock, Density, ConvBlock, ElongBlock, AreaBlock, h_coll) :- building
    (IdBuild, ConvBuild, ElongBuild, AreaBuild, IdBlock), ConvBuild > 0.95,
    !.
block(IdBlock, Density, ConvBlock, ElongBlock, AreaBlock, h_mixed).
```

Listing 3.1 – Decision tree for the urban block dataset, in Prolog formalism.

3.1.2 TILDE: Relational Decision Tree Learning Using Inductive Logic Programming

Top-Down Induction of Logical Decision Trees (Tilde) (Blockeel and De Raedt 1998) is an extension of C4.5 decision tree learner to relational learning using ILP. It is a first-order, logical, decision tree learner, based on the ILP formalism. Indeed, data is represented as a set of facts, as defined in logic programming, and background knowledge can be provided as a set of rules. In particular, this background knowledge can extend the information available in the set of facts, by allowing the inference of facts that are not explicitly mentioned in the original set, thus reducing the size of the database. The decision tree model is returned as a set of logical rules, according to the logic programming formalism. An example of such a model on the urban blocks dataset is given in Listing 3.1 as a set of rules in Prolog formalism, and in Figure 3.2 as a decision tree.

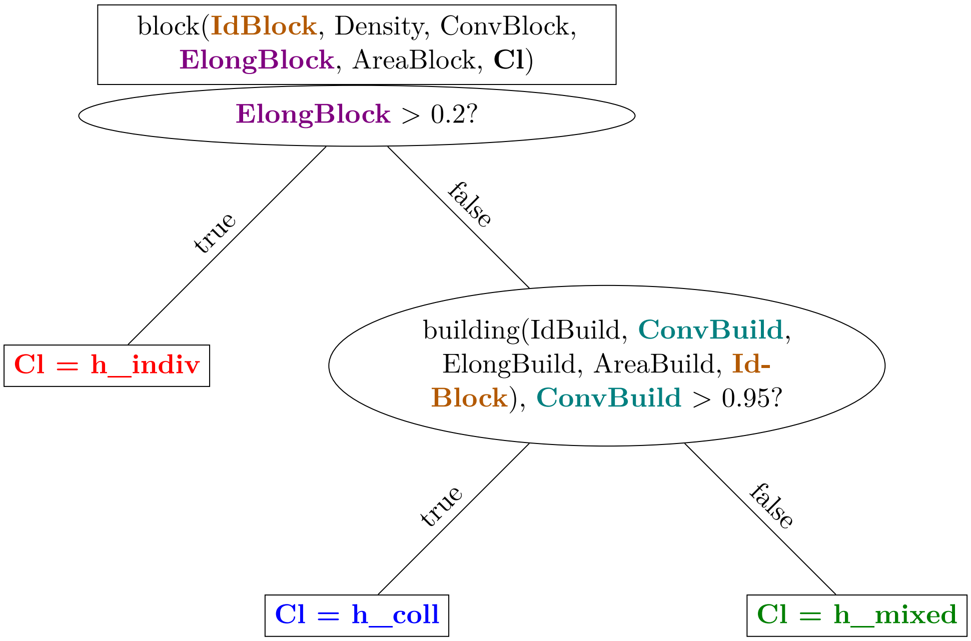

block(IdBlock, Density, ConvBlock, ElongBlock, AreaBlock, Cl)
ElongBlock > 0.2?
true
false
Cl = h_indiv
building(IdBuild, ConvBuild, ElongBuild, AreaBuild, IdBlock), ConvBuild > 0.95?
true
false
Cl = h_coll
Cl = h_mixed

Figure 3.2 – Set of rules for the urban block dataset, represented as a decision tree.

Nodes in such relational decision trees contain conjunctions of first-order predicates. The root element introduces the class predicate, with variables as arguments, as shown in the upper node of Figure 3.2. Then, at each internal node, predicates involving an already existing variable can be added to the conjunction of a node. These predicates may themselves introduce new variables. The leaves of the tree instantiate, i.e. assign a value to, the class variable introduced in the class predicate at the root of the tree.

Classification of an example by the tree is achieved as follows: the example is first defined by the declaration of a fact using the class predicate. All attributes have a value for the example. The corresponding variables in the class predicate at the root node of the decision tree are bounded with these atomic values. Then, at each internal node, an example will follow the left branch if it matches the conjunction of predicates, and the right branch otherwise. To determine if the example matches the conjunction of predicates, all variables in the conjunction are instantiated, i.e. they receive an atomic value in agreement with the facts relative to the example. If at a given node, such an instantiation exists, i.e. all variables can be replaced with a constant value matching the facts, the example is considered to match the conjunction of predicates, and follows the left child branch of the node. If such a substitution cannot be found, the example does not match the conjunction of predicates and follows the right child branch. A leaf of the tree instantiates the variable associated to the class attribute, thus giving a prediction for the example.

For instance, let us consider the classification by the decision tree from Figure 3.2 of urban block i_2 introduced in Figure 3.1. The arguments of the class predicate are instantiated as follows:

	block(IdBlock, Density, ConvBlock, ElongBlock, AreaBlock, Cl)
→	block(i2, 0.192, 0.832, 0.155, 15363, Cl)

The variable *ElongBlock* receives value *0.155*. The condition to match at the root node is *ElongBlock* $>$ *0.2*, thus the substitution according to the set of facts does not allow to match the predicate, and the example i_2 continues down the right child branch. This second node introduces new variables, related to buildings. According to the upper node, the variable *IdBlock* is already instantiated with value *i2*. Then, it is possible to replace the other variables so that the instantiation matches the set of facts:

	building(IdBuild, ConvBuild, ElongBuild, AreaBuild, IdBlock)
→	building(IdBuild, ConvBuild, ElongBuild, AreaBuild, i2)
→	building(b22, 1.0, 0.676, 147, i2)

Indeed, the condition to match is *ConvBuild* $>$ *0.95*, which is verified by building b_{22}. Thus, there is an instantiation matching the conjunction of predicates, and the example continues down the left child branch. The corresponding leaf classifies block i_2 as collective housing, i.e. class "h_coll", which is a right prediction.

The strength of Tilde lies in the introduction of new variables through the existential quantifier. These new variables represent secondary objects verifying specific conditions, the existence or not of these objects being relevant to the classification process. As a relational decision tree learner based on the introduction of relevant secondary objects, Tilde will be our second baseline in experimental comparisons.

With respect to Relaggs propositionalization by aggregation and Tilde logical decision tree learning algorithm, we consider a third approach mixing both ideas. This third idea, complex aggregation, introduces aggregation over a subset of relevant secondary objects. This allows to not perform aggregation over the whole set of secondary objects associated to a main object, taking into consideration more local tendencies inside the set of secondary objects than Relaggs aggregation over the whole set. On the other hand, complex aggregation also allows to go beyond the existential quantification used to deal with secondary objects in Tilde, to take into account global properties of the set of secondary objects rather than individual secondary objects properties.

3.2 Complex Aggregation

As stated previously, complex aggregate features in the relational setting are aggregates of a subset of the secondary objects associated to a main object, this subset being defined by a conjunction of conditions selecting the secondary objects to aggregate. This section formalizes complex aggregation, explains associated challenges and details related work on the matter.

3.2.1 Formalization of Complex Aggregate Features

A complex aggregate feature of an object from table M can be seen as a pair *(AggregationProcess, Selection)*, where *Selection* is a list of relationships and associated conditions defining which secondary objects related to a main object are selected for aggregation, and *AggregationProcess* refers to the process used to map the set of selected secondary objects to a single value.

AggregationProcess can be seen as a pair *(Function[, Feature])*, that is the association of an aggregation function, denoted by *Function* and, depending on this function, a feature of the aggregated objects, denoted by *Feature*. More precisely:

- *Function* is the aggregation function to apply to the set of feature values for the selected objects. In this work we will consider the following aggregation functions: *count* of secondary objects, *minimum*, *maximum*, *sum*, *average*, *standard deviation*, *median*, *first and third quartiles*, *first and ninth deciles*, *interquartile range* for functions operating on numerical features, and *proportion of possible values* for functions operating on categorical features.

  Median, *first and third quartiles*, *first and ninth deciles* and *interquartile range* functions rely on quantiles. They are defined as follows: given a set V of numerical values, the *k-th n-quantile* $Q_{k/n}(V)$ of set V, with $k \in \mathbb{N}$, $n \in \mathbb{N}$ and $1 \leqslant k < n$, is defined as:

$$Q_{k/n}(V) = \underset{v \in V}{argmax} \left(\frac{|\{x \in V | x \leqslant v\}|}{|V|} \leqslant \frac{k}{n} \right)$$

In other words, the *k-th n-quantile* of V is the greatest element $v \in V$ such that the proportion of elements in V lower than v is lower than k/n. For instance, the median is the first 2-quantile, i.e. it is the element of V that cuts V into two halves of equal size, one with elements below the median, the other with elements above the median. Similarly, the first quartile is the first 4-quantile, i.e. a quarter of the elements in V are below it, and three quarters above. The effect of the third quartile, i.e. the third 3-quantile, is the opposite: three quarters of the elements in V are below, while one quarter is above. From these quantiles, we define the interquartile range, which measures the dispersion of the values in the set similarly to standard deviation, as the difference between the third and the first quartile. Finally, first and ninth deciles correspond respectively to $Q_{1/10}$ and $Q_{9/10}$, i.e. the first and the last 10-quantile.

For a categorical feature $S.A$ of a secondary table S, related to table M by a one-to-many relationship from M to S, let us denote by $D = domain(S.A)$ the set of possible values of A. Then, for a given main object $m \in M$, we denote by $Sec \subseteq S$ the set of secondary objects from S related to m. Finally, let us denote by $Vals = \{v_{S.A}(s) | s \in Sec\}$, the set of values of feature $S.A$ for all objects in Sec. For every possible value v of feature $S.A$, we define the *proportion of objects with value* v aggregation function, denoted by $ratio_v(Vals)$, as:

$$ratio_v(Vals) = \frac{|\{x \in Vals | x = v\}|}{|Vals|}$$

In other words, it is the frequency of value v in the set $Vals$.

- *Feature* can be:
  - either a feature of the aggregated objects. In this case, it is either a descriptive attribute of the corresponding secondary table, or a complex aggregate feature of these objects.
  - or nothing, if the aggregation function aggregates the set of objects itself rather than the set of values of a feature of the secondary objects. Among the aggregation functions we consider, only the *count* function does not need a feature to aggregate, since it returns the number of secondary elements in the set.

The *Selection* element selects the objects to aggregate. It is a list of s pairs (Rel_k, $Cond_k$). In each pair, Rel_k selects a table using an existing relationship with the table from the previous pair ($Rel_{k-1}, Cond_{k-1}$), while $Cond_k$ introduces a conjunction of conditions on the objects of this table. Formally, they are defined as follows:

- Rel_k is a foreign key, referencing the table to which belonged the foreign key from the previous couple, i.e. Rel_{k-1}. The list of the Rel_k elements defines a chain of relationships, in which the first referenced table, the table referenced by Rel_1, is the table of the main object. On the other hand, the last introduced table, the table to which the foreign key Rel_s belongs, corresponds to the secondary objects that will be aggregated using the process defined by *AggregationProcess*. Formally, the following holds: $\forall k \in [\![2; s]\!], table\_referencing(Rel_{k-1}) = table\_referenced(Rel_k)$, and the table defined by $table\_referenced(Rel_1)$ is called the main level of the aggregate, since it corresponds to the main object, i.e. the object the complex aggregate is a feature of. In other words, for two consecutive pairs, the table associated with the second pair references the table of the first pair. The first pair of the list references the main table of the aggregate. The table associated to the last couple of the list is the table whose objects will be aggregated.

- $Cond_k$ is a conjunction of c_{kn_k} conditions, i.e. $Cond_k = \bigwedge_{1 \leqslant i \leqslant n_k} c_{ki}$, where c_{ki} is a condition on a single feature of $table\_referencing(Link_k)$. These basic conditions, for a given feature A, are:
  - $A \in Vals$, with $Vals \subset domain(A)$ if A is categorical. We reuse the same set belonging type of condition as in a decision tree internal node split condition.
  - $A \in [v_l; v_u[$, with $-\infty \leqslant v_l < v_u \leqslant +\infty$, if A is numerical. Instead of a single inequality as in a decision tree internal node split condition, we introduce interval belonging conditions in selection conditions of complex aggregate features.

The complex aggregation process is illustrated in Figure 3.3. Starting from a main object, the pairs (Rel_k, $Cond_k$) of the *Selection* element are considered sequentially. Firstly, the objects related to the main object through the relationship Rel_1 are considered. Then, these objects are filtered, and only the ones verifying the conjunction of conditions $Cond_1$ are kept. From these remaining objects, secondary objects related to them through relationship Rel_2 are considered, these objects are filtered to keep only objects verifying the conjunction of conditions $Cond_2$, and so on. This selection process ends with the filtering of the final secondary objects by conjunction of conditions $Cond_s$: remaining objects after this filtering are considered for aggregation. The aggregation process defined by the aggregation function and a possible feature of these objects is then applied to this set of remaining objects, to obtain the value of the complex aggregate for the main object.

Let us consider the two-table setting, with a main table M containing the target feature for prediction, and a secondary table S in a one-to-many relationship with M. The *Selection* part of a complex aggregate feature of table M can only be a chain with a single element, since there is only one foreign key in the schema of the database, i.e. only one relationship between tables. Thus, the search for the best complex aggregate feature boils down to optimize the pair $(AggregationProcess, Cond_1)$. Assuming there

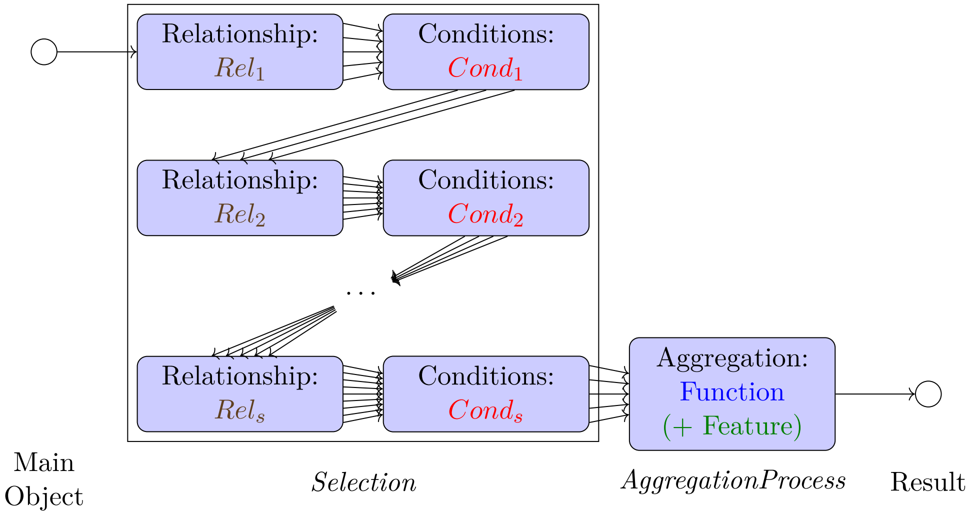


Figure 3.3 – Schema of the complex aggregation process.

is no one-to-many relationship from S to any other table, no complex aggregate feature of S can be built. Then

- The *Feature* element of the *AggregationProcess* part of the aggregate is either nothing, or an element of $S.A$. In other words, if the aggregation function of the aggregation process requires to be applied on a feature, this feature is a descriptive attribute of table S.

- Similarly, every condition in conjunction $Cond_1$ is a condition on a descriptive attribute of table S. For $Attr \in S.A$, a simple condition is either $Attr \in Vals$ where $Vals \subset domain(Attr)$ if $Attr$ is categorical, or $Attr \in [v_l; v_u[$, with $[val_1; val_2[$ an interval of $\mathbb{R}$.

In this two-table setting, we will denote a complex aggregate using the notation $Function([Feature,]S, Cond_1)$.

Example 3.1. Complex aggregate features on the urban block dataset.

Let us give three examples of complex aggregate features that can be considered on the urban block dataset. Since the dataset consists in two tables, with a one-to-many relationship from the table of blocks to the table of buildings, we can build complex aggregate features of blocks by aggregating buildings. As shown in Figure 3.1, the buildings table from the urban block dataset has three numerical descriptive attributes: area, elongation, and convexity. For each example, we explicitly separate the aggregation process and the selection process which, in this two-table setting, consists in a single conjunction of conditions. Implicitly, the relationship used to select secondary objects is the relationship between the table of blocks and the table of buildings.

- *count(buildings, area $\geqslant$ 450)* computes the number of buildings in the block with area greater than or equal to 450.

  Aggregation Process: count of secondary objects

  Selection: buildings with area $\geqslant 450$

- *average(elongation, buildings, area $\geqslant$ 450)* computes the average elongation of buildings in the block with area greater than or equal to 450.

  Aggregation Process: average of elongation of secondary objects

  Selection: buildings with area $\geqslant 450$

- *maximum(convexity, buildings, area $\geqslant$ 450 $\wedge$ elongation $<$ 0.7)* computes the maximum of convexity of buildings in the block with area greater than or equal to 450 and elongation lower than 0.7.

  Aggregation Process: maximum of convexity of secondary objects

  Selection: buildings with area $\geqslant 450$ and elongation < 0.7

■

3.2.2 Combinatorial Explosion of the Complex Aggregate Search Space

Complex aggregation is a rich feature representation: many features can be expressed in this setting. However, this huge expressive power induces a critical number of complex aggregate features, in particular when dealing with numerical attributes. Indeed, a basic selection condition on a numerical attribute is an interval belonging condition, the bounds of the interval having to be optimized. If all combinations of possible values for these bounds are considered, the number of complex aggregates with a given aggregation process and a selection conjunction consisting of a single basic condition on this attribute already surges quickly with the number of possible values for the numerical attribute, reaching more than a thousand with only 50 possible values.

Moreover, the selection part of the aggregate is a conjunction of such basic conditions. Thus, the number of possible selection conjunctions increases exponentially with the number of numerical attributes, reaching a million with only 3 numerical attributes and 50 possible values for the attributes. Multiplying by the number of aggregation processes, there are millions of complex aggregate features to consider for learning. In an attribute-value setting, this would be represented as a single table with the corresponding millions of columns, which cannot be handled by any learning algorithm in a reasonable duration.

We estimate in more detail the size of the complex aggregate search space in a two-table setting. Reusing previous notations, we have:

- For each categorical attribute $S.A_{cat}$ of the secondary table S, if we denote by N_{vals} the number of possible values of the categorical attribute, we have $2^{N_{vals}}$ possible conditions on this attribute. Indeed, the possible conditions are defined by the possible subsets of values of the attribute, which gives the previous result.

- For each numerical attribute $S.A_{num}$ of the secondary table S, we approximate the number of possible values of the attribute by the total number of secondary objects in the dataset, i.e. $N_{vals} = |S|$. It is a worst-case scenario, but it is plausible if the attribute is continuous. Then, the number of conditions is given by the number of possible intervals, which is of the order of magnitude of $N_{vals}^2/2$.

The number of possible values of a given categorical attribute is in most cases low, under 10. Thus, the number of possible conditions on a categorical attribute rarely exceeds 1000. This number of conditions for a numerical attribute is reached when the attribute has 50 possible values. For a continuous attribute, this can only occur in small datasets. Thus, we consider a worst-case scenario with only numerical attributes in the secondary table, which happens in practice in many real-world datasets we encountered. The number of possible conjunctions of conditions in this setting is the product over all attributes in table S of the number of possible basic conditions on one attribute, i.e. with $a(S)$ attributes

$$|Conjunctions| = \left(\frac{|S|^2}{2}\right)^{a(S)}$$

To obtain the number of possible complex aggregates, we multiply by the number of aggregation processes, i.e. the number of pairs $(Function, Feature)$ allowed by the dataset. Since most aggregation functions we consider handle numerical attributes, it is roughly $numFunctions \cdot a(S)$, and the number of possible complex aggregates is:

$$\begin{aligned} |ComplexAggregates| &= |AggregationProcess| \cdot |Conjunctions| \\ &= |Functions| \cdot a(S) \cdot \left(\frac{|S|^2}{2}\right)^{a(S)} \end{aligned}$$

The urban block dataset has 3 numerical attributes and 7694 lines in the buildings table. We use the following 6 aggregation functions : *count, min, max, sum, average* and *standard deviation*. The five last functions can be used with any of the three attributes, while *count* alone defines one aggregation process. We then have $3 * 5 + 1 = 16$ possible aggregation processes. This brings a total number of complex aggregates bounded by $16 \cdot \left(\frac{7692^2}{2}\right)^3 \approx 4.14 \cdot 10^{23}$, while the urban block dataset does not have many secondary attributes, nor many lines in the secondary table. This example shows how much the size of the feature space explodes with the use of complex aggregates. This motivates the need for heuristics able to explore this space in a non-exhaustive way.

3.2.3 Related Work on Complex Aggregates

Complex aggregates can be used as propositionalization features, in a similar way as the simple aggregate features from Relaggs. For instance, the method introduced in (Boullé 2014) builds complex aggregate features for use in a Bayesian classifier, guided by minimum description length principle to avoid overfitting and a heuristic sampling based on prior distribution of constructed features.

An alternative approach, potentially more powerful, is to generate relevant complex aggregates for subsets of training data, rather than only aggregates relevant on the whole training set. This can be achieved using decision trees: a complex aggregate feature relevant to split the whole data will be generated for the root of the tree. Then, aggregates relevant on more specific parts of the data can be learned on the partitions of the original training set in the sub-branches.

Following this idea, complex aggregates have been integrated in the relational decision tree learner TILDE, along with an heuristic to prune the search space. Detailed description of the heuristic can be found in (Vens, Ramon, and Blockeel 2006). We describe this work in this subsection. The heuristic is based on refining the aggregate-based splits along monotonic paths in a refinement cube. The split to refine is *Function*(Feature, *SecondaryObjects*) $<$ *operator* $>$ *th*. The feature to aggregate and the numerical comparison operator, either $\geqslant$ or $\leqslant$, are fixed. Thus, there are three degrees of freedom in the split, which define the three dimensions of the refinement cube, as illustrated in Figure 3.4. In these three dimensions, an ordering can be defined according to the number of main objects verifying the split condition. Let us consider the urban block example dataset, with *area* as the feature of secondary objects *buildings* to aggregate, and $\geqslant$ as the operator. The monotonicity, according to the evolution of the set of main objects, is defined along one dimension as follows.

- The aggregation function F: an ordering on the aggregation functions can be defined: e.g. application of the *minimum* function on a given attribute for a given set of objects is lower than the result of the *average* of the same attribute on the same set. Thus, if the comparison threshold does not change, less main objects will verify the condition *average*(*area*, *SecondaryObjects*) $\geqslant$ *th* than *minimum*(*area*, *SecondaryObjects*) $\geqslant$ *th*, since *average*(*area*, *SecondaryObjects*) $\geqslant$ *minimum*(*area*, *SecondaryObjects*). The same relationship exists with the *maximum* function, which returns a higher value than the *average* function on a given set of values. Thus, the ordering $minimum \leqslant average \leqslant maximum$ can be defined on aggregation functions.

- The set S of objects to aggregate: a set can be reduced or enlarged by specializing or generalizing the selection condition. Let us consider two sets of buildings S_1 and S_2 with $S_1 \subset S_2$. Trivially, *minimum*(*area*, S_1) $\geqslant$ *minimum*(*area*, S_2), since the second takes the minimum of a super-set of the set considered in the first expression. Thus, if *minimum*(*area*, S_2) $\geqslant$ *th*, then we have *minimum*(*area*, S_1) $\geqslant$ *th*, and decreasing the set to aggregate, i.e. removing elements from it, increases the value of the minimum function, such that more main objects verify the split condition. Thus, on the plane of the cube defined by $F = minimum$, if the set to aggregate is enlarged, then more main objects are likely to verify the split condition. If the aggregation function is *maximum*, the opposite happens, since $maximum(area, S_1) \leqslant maximum(area, S_2)$. Thus, the specializing move in plane

$F = maximum$ will consist in reducing the set to aggregate. If the aggregation function is *average*, then there is no monotonicity property, since the evolution of the average value of the set depends on the relative position of the current average value and the value of the element added or removed. The two bounds of the cube along this dimension are the two extreme cases where all secondary objects associated to a main object are selected for aggregation, or none.

- The constant part of the split: i.e. in case of a numerical aggregate, the threshold *th* to compare to. Let us consider two threshold values $th_1 < th_2$. Independently of the aggregation function we have $Function(area, S) > th_2$, we also have $Function(area, S) > th_1$. Thus, decreasing the threshold value induces a generalization of a split condition, which is more likely to be verified by main objects. In this dimension, decreasing the threshold leads to a more general split condition, while increasing the threshold leads to a specialization split condition. The three aggregation functions considered return a value in the same range as the values from the set they aggregate, so we can defined a lower limit th_{min} and an upper limit th_{max} for the comparison threshold, corresponding to the lower and upper bounds of the aggregated feature. In our example, we would consider the lowest and highest area of the available buildings.

Starting from the most general condition, the aim of the split optimization is to follow paths in the cube that specialize the split condition, i.e. paths that lead to conditions that are verified by fewer and fewer main objects. This is done until none verifies the condition, in which case further exploration of the path is not needed. In our example, the most general condition is $maximum(area, Every) \geqslant th_{min}$, i.e. *the maximum of area over all buildings in the block is greater than or equal to the lowest area possible for a building*, which all blocks verify. Then, the set of buildings selected for aggregation can be reduced, for instance with a condition that selects only *buildings with elongation greater than 0.4*, which will decrease the maximum area value, and the resulting split condition will be verified by fewer blocks.

This effect is also achieved by increasing the comparison threshold, and modifying the aggregation function from *maximum* to *average*, which returns a lower value. Once the aggregation function is *average*, only the threshold can be modified, since there is no monotonicity property along the object selection dimension, i.e. there is no rule on the evolution of the value returned by the *average* function when we decrease or increase the set to aggregate, thus no rule on if such a move will specialize of generalize the split condition.

Finally, when the function becomes *minimum*, specializing the aggregate can be achieved by enlarging the set to aggregate, contrary to the *maximum* function, while increasing the threshold is still possible. The search stops at the most specific condition possible, i.e. $minimum(area, Every) \geqslant th_{max}$, when *the minimum of area over all buildings in the block is greater than the highest area possible for a building*, which is achieved for a minimum of blocks.

Likewise, our aim is to avoid the extensive search of the complex aggregate space,

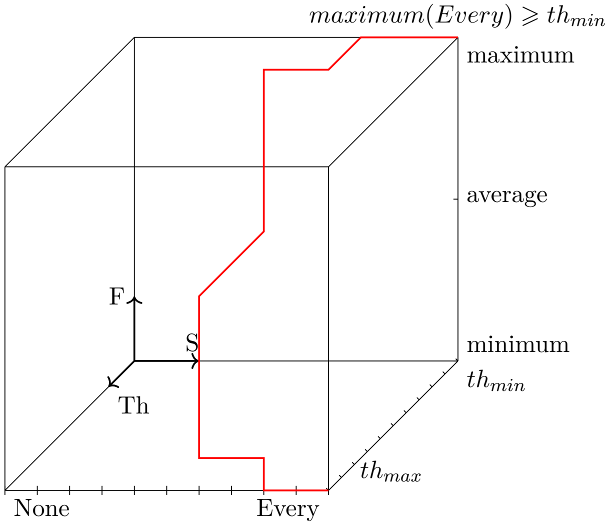


Figure 3.4 – Refinement cube for complex aggregates, from (Vens, Ramon, and Blockeel 2006).

and to propose methods to perform the search in an efficient way. Unlike the refinement cube method, which prunes the search space when possible but is exhaustive in a worst-case scenario, the methods we propose will be hill-climbing-based, thus avoiding an exhaustive search.

3.3 Incremental Construction of Complex Aggregates

As explained in the previous section, exhaustive consideration of complex aggregate features is not feasible because of their number. An attribute-value decision tree learning algorithm would consider split conditions on all available attributes, which is not possible in the relational setting when considering complex aggregates. Thus, there is a need for heuristics to explore the complex aggregates space in an efficient way. We intend to generate complex aggregate features for use in a decision tree model. More precisely, splits at internal nodes of a decision tree will be conditions on complex aggregate features. Thus, at a given node, we have to generate a complex aggregate that will produce a relevant split. An example of such a decision tree on the urban block dataset is given in Figure 3.5.

The heuristics we propose are based on hill-climbing optimization. Considering a space of candidate points and a metric to evaluate the quality of a candidate, hill-climbing optimization consists in, starting from an arbitrary candidate point, to explore candidates in a neighborhood of this point looking for a better candidate, in terms of quality, than the starting point. If such a candidate can be found in the neighborhood, the search continues in the neighborhood of this better candidate. Otherwise, the

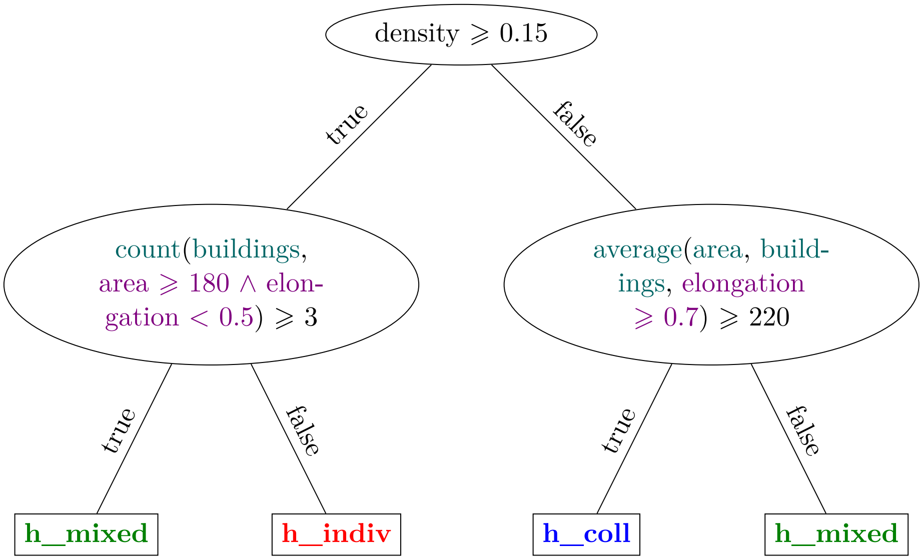

density ⩾ 0.15
true
false
count(buildings, area ⩾ 180 ∧ elongation < 0.5) ⩾ 3
average(area, buildings, elongation ⩾ 0.7) ⩾ 220
true
false
true
false
h_mixed
h_indiv
h_coll
h_mixed

Figure 3.5 – Example of complex-aggregate-based decision tree.

starting point is considered as the optimal solution and the search stops.

In this section, we first describe the implementation of a hill-climbing algorithm for complex aggregate feature optimization, including technical details about handling the empty sets to aggregate, which is an issue to consider with attention in complex aggregation. Finally, we detail possible implementations of complex aggregate feature generation in a multi-table setting, with several nested one-to-many relationships.

3.3.1 Hill-Climbing of Complex Aggregates

As observed above, the exhaustive consideration of complex aggregate features for learning in the relational setting is computationally impossible. Thus, there is a need for optimization techniques to search for relevant complex aggregates. An optimization technique such as hill-climbing relies on a quality metric, used to evaluate each candidate point. In our case, the quality of a complex aggregate corresponds to the quality of the splits based on it in the decision tree. Formally, from every complex aggregate *ComplAgg*, a set *Splits*(*ComplAgg*) of split conditions based on *ComplAgg* can be formed. In the decision tree learning process, when a split on the set of examples E is to be found, the set of candidate splits using a numerical aggregate is the set of comparisons of the aggregate value to a set of candidate thresholds defined by the possible values of the aggregate on the examples in E, i.e.

$$Splits(ComplAgg) = \{ComplAgg \geqslant th \mid th \in \{v_{ComplAgg}(e) \mid e \in E\}\}$$

Then, the quality of a complex aggregate is defined by the maximum quality achieved by a split based on it, on the set of examples E, i.e.

$$Quality(ComplAgg, E) = \max_{Split \in Splits(ComplAgg)} (Gain(Split, E))$$

The second crucial element of a hill-climbing algorithm is the definition of the neighborhood of a candidate point. In our case, a complex aggregate is defined by its aggregation process, i.e. an aggregation function and optionally a feature to aggregate, and the conjunction of conditions selecting secondary objects to aggregate, in a two-table setting. In this search space with three dimensions, the neighborhood of a given complex aggregate is defined by a set of allowed moves. A move in the space corresponds to a change in one of the three dimensions, i.e. one parameter only of the aggregate, among the function, the feature to aggregate, and the conjunction of conditions, changes between the original aggregate and its neighbor. As an example, we consider the original aggregate to be $maximum(convexity, buildings, area \geqslant 450)$, in the context of the urban block dataset. Valid moves families in the complex aggregate search space are:

Modification of the aggregation function: The aggregated feature and selection conjunction of conditions do not change, while the aggregation function is modified. From the original aggregate taken as example, possible neighbors with respect to this family of moves include:

- $\mathbf{average}(convexity, buildings, area \geqslant 450)$
- $\mathbf{minimum}(convexity, buildings, area \geqslant 450)$
- $\mathbf{count}(buildings, area \geqslant 450)$

Modification of the feature to aggregate (if present): If the aggregation function is different from the count of secondary objects, the aggregated feature can be modified. On our example, possible moves are:

- $maximum(\mathbf{area}, buildings, area \geqslant 450)$
- $maximum(\mathbf{elongation}, buildings, area \geqslant 450)$

Addition of one basic condition to the selection conjunction: The selection conjunction contains at most one condition on a given feature. Thus, in a two-table setting with secondary table S, the conjunction contains at most $a(S)$ basic conditions, i.e. at most one per secondary attribute. If an attribute is not present in a basic condition of the conjunction yet, a basic condition on it can be added. Only one basic condition at a time can be added to the original aggregate to obtain a neighbor aggregate. On our example, possible moves include:

- $maximum(convexity, buildings, area \geqslant 450 \wedge \mathbf{elongation} \geqslant \mathbf{0.7})$
- $maximum(convexity, buildings, area \geqslant 450 \wedge \mathbf{elongation} \leqslant \mathbf{0.9})$
- $maximum(convexity, buildings, area \geqslant 450 \wedge \mathbf{convexity} \geqslant \mathbf{0.88})$
- $maximum(convexity, buildings, area \geqslant 450 \wedge \mathbf{convexity} \leqslant \mathbf{0.95})$

Deletion of one basic condition from the selection conjunction: One basic condition can be removed from the selection conjunction. Similarly to the addition move, only one basic condition at a time can be removed from the conjunction to obtain a neighbor aggregate. Since there is only one basic condition in the selection conjunction of our example, the only possible move is:

- *maximum*(*convexity*, *buildings*)

Modification of one basic condition from the selection conjunction: One of the basic conditions in the selection conjunction can be modified. For instance, for a condition on a numerical secondary attribute, the comparison threshold can be increased or decreased to modify the set of selected secondary objects. Again, modification of only one basic condition at a time is allowed to obtain a neighbor. On our example, possible moves include:

- *maximum*(*convexity*, *buildings*, *area* $\geqslant$ **500**)
- *maximum*(*convexity*, *buildings*, *area* $\geqslant$ **400**)
- *maximum*(*convexity*, *buildings*, *area* $\leqslant$ **450**)

These possible moves are detailed in pseudo-code in Algorithm 3.1.

Example 3.2. Convergence of the selection conjunction of conditions.

Let us illustrate the use of the three kinds of moves on the selection conjunction, i.e. addition, removal and modification of a basic condition, to optimize a complex aggregate feature as finely as possible. On the urban block dataset schema, let us consider an artificial binary classification task where a block is of class:

- *yes* if and only if it verifies *maximum*(*convexity*, *buildings*, *area* $\geqslant 450 \wedge$ ***elongation*** $<$ **0.7**) $\geqslant 0.5$

- *no* otherwise, i.e. either *maximum*(*convexity*, *buildings*, *area* $\geqslant 450 \wedge$ ***elongation*** $<$ **0.7**) < 0.5, or there is no building in the block verifying both *area* $\geqslant 450$ and *elongation* < 0.7.

Let us consider the starting aggregate of the hill-climbing is *maximum*(*convexity*, *buildings*), i.e. the aggregation process is already the one used to generate data and discriminate between both classes. Through the hill-climbing process, only the selection conjunction of conditions needs optimization.

We observe that the condition introduced first uses a threshold close, but not exactly equal, to the target. For instance, the best neighbor of the original aggregate may be *maximum*(*convexity*, *buildings*, *area* $\geqslant 400$). Then, the best neighbor of this aggregate introduces in the conjunction a basic condition on *elongation*, which corresponds to the target aggregate. However, the best neighbor is *maximum*(*convexity*, *buildings*, *area* $\geqslant 400 \wedge$ *elongation* < 0.6), which does not match exactly the target aggregate. This is caused by the lone introduction of the condition on *area*: without the introduction of the condition on *elongation*, the comparison threshold introduced does not match the

Algorithm 3.1 EnumerateNeighborsExhaustive

```
Input: aggregate: original complex aggregate feature, aggProcs: available aggregation processes (couples of function and feature)
Output: allNeighbors: array of complex aggregate features, neighbors of aggregate

allNeighbors $\leftarrow$ []
for all aggProc $\in$ aggProcs do
  nextAggregate $\leftarrow$ CreateAggregate(aggProc.function, aggProc.feature, aggregate.selection)
  allNeighbors.Add(nextAggregate)
end for
for all attr $\in$ secondary attributes not present in aggregate.selection do
  for all cond $\in$ set of conditions achievable on attr do
    nextAggregate $\leftarrow$ CreateAggregate(aggregate.function, aggregate.feature, aggregate.selection and cond)
    allNeighbors.Add(nextAggregate)
  end for
end for
for all secCond $\in$ set of conditions present in aggregate.selection do
  nextAggregate $\leftarrow$ CreateAggregate(aggregate.function, aggregate.feature, aggregate.selection - cond)
  allNeighbors.Add(nextAggregate)
end for
for all secCond $\in$ set of conditions present in aggregate.selection do
  attr $\leftarrow$ secCond.feature
  for all cond $\in$ set of conditions achievable on attr do
    nextAggregate $\leftarrow$ CreateAggregate(aggregate.function, aggregate.feature, (aggregate.selection - secCond) and cond)
    allNeighbors.Add(nextAggregate)
  end for
end for
return allNeighbors
```

threshold of the target aggregate, corresponding to a use in conjunction with a threshold on *elongation*.

This is the reason why we allow the algorithm to reconsider its choice of threshold on the *area*, the best neighbor may then be: $maximum(convexity, buildings, area \geqslant 430 \ \wedge \ elongation < 0.6)$. Then the choice of threshold on *elongation* can also be reconsidered and the best neighbor is $maximum(convexity, buildings, area \geqslant 430 \ \wedge \ elongation < 0.7)$. Again, the threshold on *area* can be modified and we obtain the target aggregate: $maximum(convexity, buildings, area \geqslant 450 \ \wedge \ elongation < 0.7)$.

■

Finally, the starting point of the hill-climbing algorithm is the complex aggregate feature defined by one of the available aggregation processes chosen randomly, and an empty conjunction of conditions as selection, thus selecting all secondary objects for aggregation. At each step of the hill-climbing algorithm, all possible neighbors, as defined above, are considered. When adding a basic condition on a numerical secondary attribute, all possible interval conditions are considered, i.e. all pairs of possible values of the corresponding attribute. Pseudo-code for this hill-climbing algorithm is given in Algorithm 3.2.

Algorithm 3.2 First Hill-Climbing Algorithm

```
Input: functions: list of aggregation functions, features: list of attributes of the
  secondary table, train: labeled training set, target: attribute target for prediction
Output: split: best complex aggregate found through hill-climbing
--------------------------------------------------------------------------------
aggProcs ← InitializeProcesses(functions, features)
aggregate ← CreateAggregate(count, NULL, TRUE)
firstSpl ← EvaluateFeature(aggregate, train, target)
bestSplits ← [firstSpl]
bestScore ← firstSpl.score
keepGoing ← true
while keepGoing do
  allNeighbors ← EnumerateNeighborsExhaustive(aggregate, aggProcs)
  hasImproved ← false
  for all neighbor ∈ allNeighbors do
    spl ← EvaluateFeature(neighbor, train, target)
    if spl.score ≥ bestScore then
      if spl.score > bestScore then
        bestScore ← spl.score
        bestSplits ← []
        hasImproved ← TRUE
        aggregate ← neighbor
      end if
      bestSplits.Add(spl)
    end if
  end for
  keepGoing ← hasImproved
end while
split ← bestSplits.OneRandomElement()
return split
```

3.3.2 Dealing with Empty Sets

We discuss the general issue of aggregating the feature values when the set to aggregate is empty. This is a common problem in aggregation, and even more in complex aggregation since the set to aggregate is reduced. Indeed, for a given main object, there may be no secondary object related to the main object verifying the conjunction of conditions. In this case, the set to aggregate is empty, and the aggregation process may not be applicable. For instance, on the urban block dataset, the average area of buildings in a block with elongation greater than 0.7 cannot be computed when the block does not contain at least one building with elongation greater than 0.7. Only the *count* function can deal in a natural way with empty sets, while another solution has to be chosen for numerical aggregation functions. Some possibilities to deal with that issue have been discussed in (Vens 2007):

- Fixing an arbitrary value as the result.
- Using a value depending on the aggregation condition, as close as possible to the values for examples for which the aggregation selection conjunction does not result in an empty set, or as far as possible.
- Failing the aggregate when the aggregation function cannot be applied.
- Not considering the aggregate if the aggregation function cannot be applied for at least one example.

In our opinion, the first two options are not easy to apply for every function. Indeed, for functions *minimum* or *maximum*, one can choose threshold values as low or as high as possible so that the condition on the aggregate always succeeds or fails if the function cannot be applied directly, but for the *average* function, using a fixed value such as zero is not relevant if the attribute can take both positive and negative values, nor is choosing positive or negative infinity. Moreover, an empty set to aggregate is meaningful as such, and by assigning a result to the aggregate function we lose that significance. The third option also assigns the empty set a meaning we do not necessarily want it to have: the failure of the aggregate would mean the inequality between the result of the aggregation and the threshold is wrong. However, this is not the meaning of the empty set, which indicates the failure of the existential quantifier to find secondary objects, i.e. there does not exist any secondary object verifying the selection conjunction of conditions.

We propose two ways to address the issue of empty sets. Firstly, to consider them as a third branch in the internal nodes of our decision trees, since they correspond neither to a success nor to a failure of the inequality, they constitute a third possibility. The second method we propose, preserving the binary structure of our trees, is to introduce an internal node testing the existence of the secondary objects used by the complex aggregate feature, as parent of the node with the split condition on the complex aggregate. In other words, we create a three-branch split using two levels of a binary decision tree. An example is shown in Figure 3.6, where the aggregate *the average area of buildings* *with elongation greater than 0.7 is greater than 220* is “protected” by the existential quantifier, which tests the presence of at least one building in the block with elongation greater than 0.7. If the latter succeeds, then the former can be evaluated because it is meaningful to compute the average value of a non-empty set. If the existential quantifier fails, then the average area of the buildings with elongation greater than 0.7 cannot be computed, thus defining the third branch.

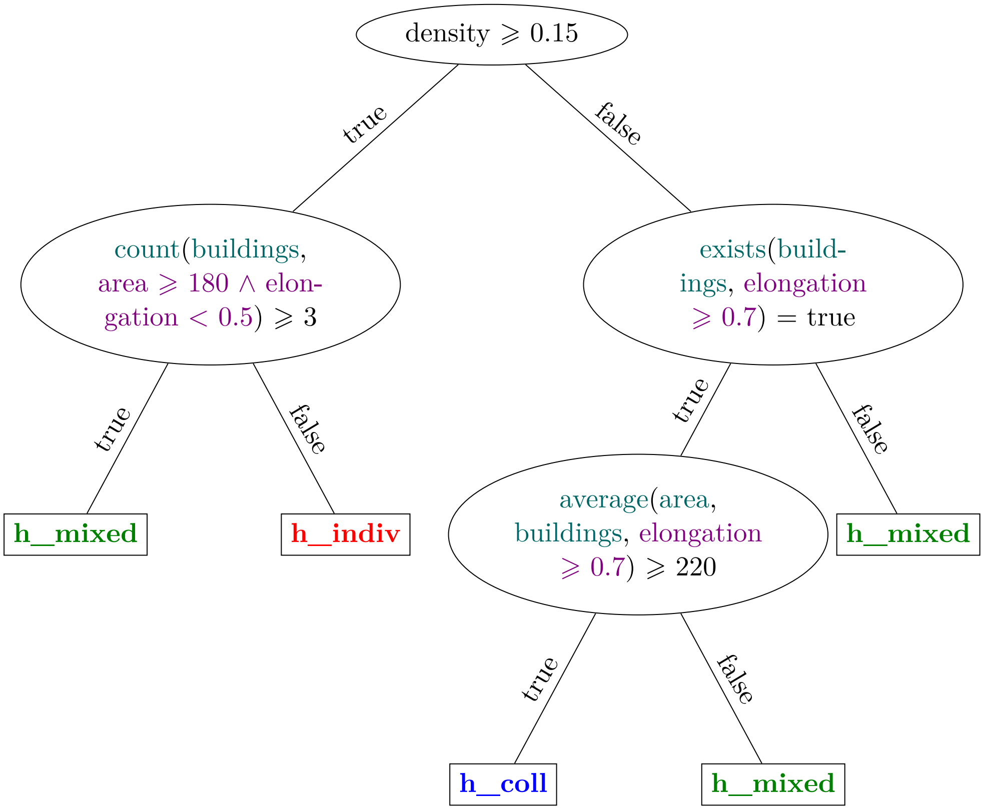
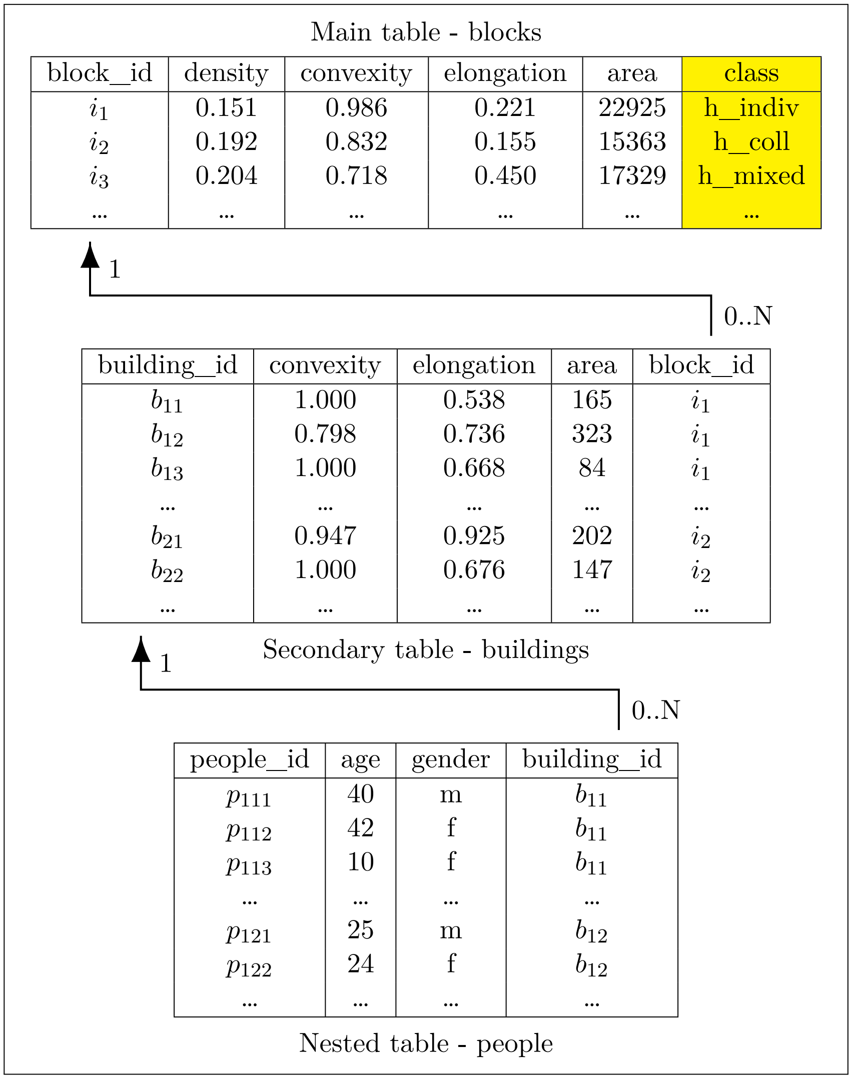

density ⩾ 0.15
true
false
count(buildings, area ⩾ 180 ∧ elongation < 0.5) ⩾ 3
exists(buildings, elongation ⩾ 0.7) = true
true
false
true
false
h_mixed
h_indiv
average(area, buildings, elongation ⩾ 0.7) ⩾ 220
h_mixed
true
false
h_coll
h_mixed

Figure 3.6 – Example of complex-aggregate-based decision tree, handling the empty set case.

3.3.3 Addressing the Two-Table Schema Limitation

We limit ourselves to a table schema with two levels, i.e. one main table with secondary tables directly related to it. This is motivated by the necessity to avoid introducing nested complex aggregates, or complex aggregates inside complex aggregates. Indeed, such an introduction would increase even more the already combinatorial size of the complex aggregate search space. To illustrate this case, we consider a third table linked through a one-to-many relationship to the secondary table. On the urban block dataset, we take the example of a table of people living in the buildings, as shown in Figure 3.7.



Main table - blocks

block_id	density	convexity	elongation	area	class
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixed
...	...	...	...	...	...

building_id	convexity	elongation	area	block_id
b_{11}	1.000	0.538	165	i_1
b_{12}	0.798	0.736	323	i_1
b_{13}	1.000	0.668	84	i_1
...	...	...	...	...
b_{21}	0.947	0.925	202	i_2
b_{22}	1.000	0.676	147	i_2
...	...	...	...	...

Secondary table - buildings

people_id	age	gender	building_id
p_{111}	40	m	b_{11}
p_{112}	42	f	b_{11}
p_{113}	10	f	b_{11}
...	...	...	...
p_{121}	25	m	b_{12}
p_{122}	24	f	b_{12}
...	...	...	...

Nested table - people

Figure 3.7 – Schema of the urban block dataset, with nested people table.

This second relationship between buildings and people also induces a third relationship between blocks and people. Thus, complex aggregate features can be created using any of these three relationships. A complex aggregate used in a split condition has to be a feature of the block table, but it can now aggregate either the buildings or the people. Moreover, complex aggregate features of the buildings can be obtained by aggregating the people, and used like any secondary attribute of the buildings, e.g. in selection conjunction of conditions to select the buildings. Thus, this new relationship induces three new possibilities for complex aggregation:

- We can create complex aggregate features related to buildings by aggregating people building-wise, and use this new feature in a complex aggregate related to blocks. The aggregate on people can be used in two ways:
  - In the aggregation condition of the higher-level aggregate (aggregating buildings at the block level): $\mathit{average}(\mathit{area},\ \mathit{buildings},\ \boldsymbol{\mathit{count}(\mathit{people},\ \mathrm{age} > 18) \leqslant 5})$ corresponds to the average area of buildings inhabited by fewer than 5 adults.
  - As the aggregated feature of the higher-level aggregate (aggregating buildings at the block level): $\mathit{average}(\boldsymbol{\mathit{count}(\mathit{people},\ \mathrm{age} > 18)},\ \mathit{buildings},\ \mathrm{area} \leqslant 220)$ corresponds to the average number of adults in buildings with area lower than 220.
- We can chain relationships and aggregate the objects from the deepest level at the main level, e.g. aggregate people at the block level: $\mathit{average}(\mathit{age},\ \mathit{people},\ \mathit{buildings}(\mathrm{area} \leqslant 220) \wedge \mathrm{gender} = \mathrm{male})$ corresponds to the average age of male people in the urban block living in buildings with area lower than 220.

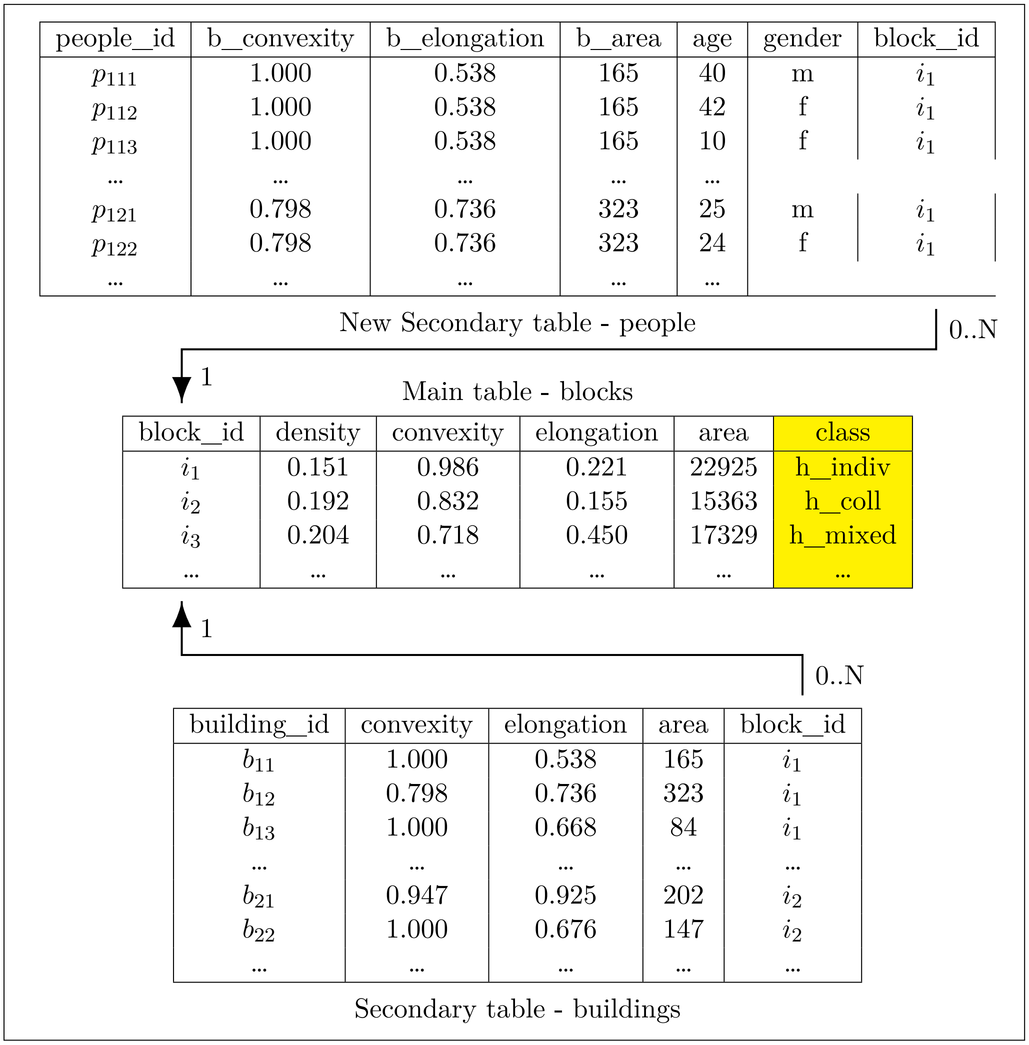
The size of the complex aggregate search space surges exponentially with the introduction of new relationships. Thus, we propose two methods to deal with this rising complexity.

The first option is to turn nested tables into secondary tables, by relating them directly to the main table. For instance, the table of people living in a building can become a table of people living in the block, by joining the table of buildings and the table of people on the key *building_id*, according to relational database terminology. This does not allow nested complex aggregates, i.e. the two first kinds we presented, but it does allow the use of the last kind of complex aggregate features, where people are aggregated with conditions on them and the building they live in. The result of the join operation can be visualized in Figure 3.8.

The second option is to propositionalize the sub-databases induced by choosing the secondary tables as main tables. For instance, we can propositionalize the sub-database induced by the buildings and the people. More specifically, if we use Relaggs, we would create simple aggregates of the people at the building level. This allows the introduction of simple aggregates at the secondary level, i.e. nested simple aggregates, either in the selection condition of a main aggregate, or as the aggregated feature of a main aggregate, which constitute the two first kinds of nested complex aggregates we presented above. This resulting database is sketched in Figure 3.9.

3.4 Conclusion

The structure of the complex aggregate search space is a basis for an intuitive implementation of hill-climbing. However, the implementation described in this chapter considers too many possibilities for the neighborhood of an aggregate. Indeed, when introducing or modifying a basic condition on a numerical attribute, the hill-climbing algorithm



people_id	b_convexity	b_elongation	b_area	age	gender	block_id
p_{111}	1.000	0.538	165	40	m	i_1
p_{112}	1.000	0.538	165	42	f	i_1
p_{113}	1.000	0.538	165	10	f	i_1
...	...	...	...	...		
p_{121}	0.798	0.736	323	25	m	i_1
p_{122}	0.798	0.736	323	24	f	i_1
...	...	...	...	...		

New Secondary table - people

Main table - blocks

block_id	density	convexity	elongation	area	class
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixed
...	...	...	...	...	...

building_id	convexity	elongation	area	block_id
b_{11}	1.000	0.538	165	i_1
b_{12}	0.798	0.736	323	i_1
b_{13}	1.000	0.668	84	i_1
...	...	...	...	...
b_{21}	0.947	0.925	202	i_2
b_{22}	1.000	0.676	147	i_2
...	...	...	...	...

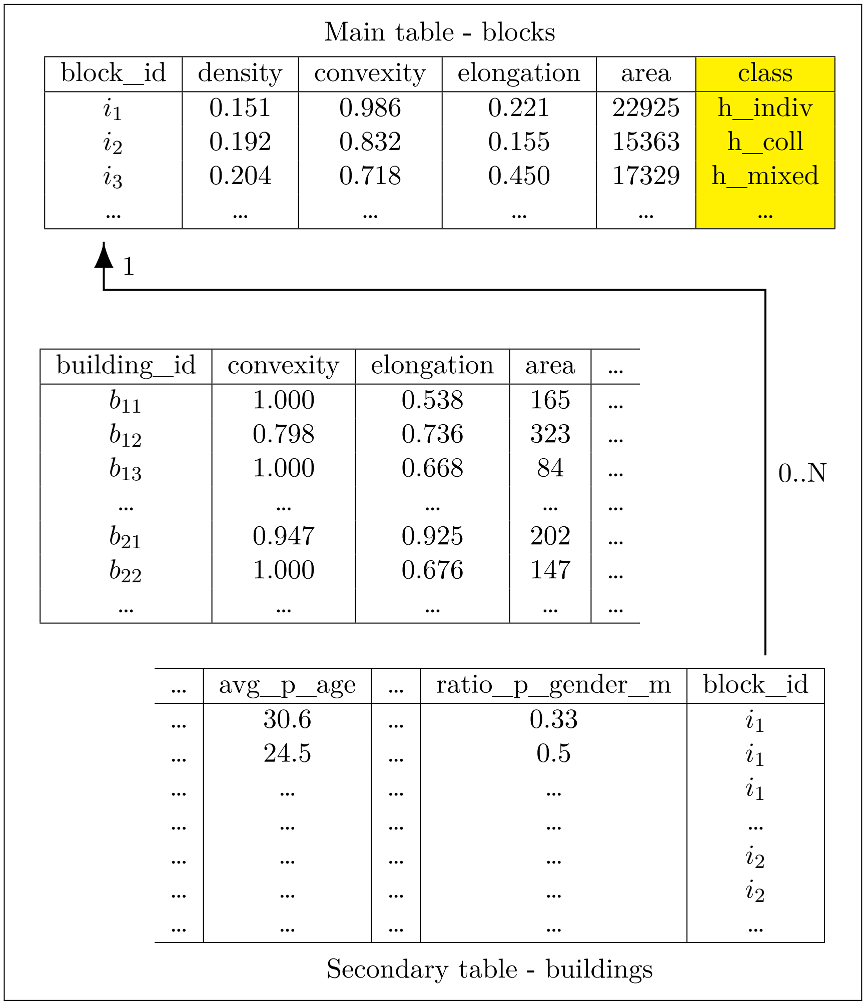
Secondary table - buildings

Figure 3.8 – Schema of the unnested urban block dataset with option 1.

considers all possible conditions on this attribute, i.e. the number of possible intervals, which can be approximated by $|S|^2/2$ with $|S|$ the number of secondary objects. Since we consider all $a(S)$ secondary attributes, and the aggregation process can be modified to obtain a neighbor, the size of such a neighborhood is

$$|Neighbors| \approx |AggregationProcesses| + \frac{|S|^2}{2} \cdot a(S)$$

In practice, this remains computationally expensive. The main challenge will be to reduce the size of the considered neighborhood. This will be achieved using stochastic hill-climbing.



block_id	density	convexity	elongation	area	class
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixed
...	...	...	...	...	...

building_id	convexity	elongation	area	...
b_{11}	1.000	0.538	165	...
b_{12}	0.798	0.736	323	...
b_{13}	1.000	0.668	84	...
...	...	...	...	...
b_{21}	0.947	0.925	202	...
b_{22}	1.000	0.676	147	...
...	...	...	...	...

...	avg_p_age	...	ratio_p_gender_m	block_id
...	30.6	...	0.33	i_1
...	24.5	...	0.5	i_1
...	...	...	...	i_1
...	...	...	...	...
...	...	...	...	i_2
...	...	...	...	i_2
...	...	...	...	...

Figure 3.9 – Schema of the unnested urban block dataset with option 2.

A second conclusion we draw is related to the possible moves. Indeed, the change of aggregation process in a hill-climbing step with an unchanged selection conjunction of conditions is too drastic. The appropriate selection conjunction of conditions seems to be related to the aggregation process in use. Thus, the approaches we will propose in the next chapter will consider searching for a suitable selection conjunction of conditions for a given aggregation process, rather than trying to optimize both in a coupled fashion.

Chapter 4

Stochastic Heuristics for Complex Aggregates Learning

As observed in the previous chapter, the introduction of complex aggregate features in the relational setting induces a large feature space to explore. In particular, in a dataset composed of many numerical attributes, exhaustive consideration of complex aggregates becomes impossible in terms of runtime. Thus, there is a need for non-exhaustive optimization heuristics to explore the complex aggregate search space.

In this chapter, we adapt existing stochastic heuristics to the complex aggregates search space. Randomization is introduced at two levels of the learning process: firstly, the hill-climbing optimization of complex aggregates is made stochastic through the use of randomized hill-climbing. Secondly, complex aggregate features are introduced in a Random Forest model, which combines several decision trees built using different random sub-samples of data and features.

In Section 4.1, we present and evaluate a random restart hill-climbing algorithm for complex aggregate inclusion in a single decision tree model. In Section 4.2, we introduce the Caraf system, which includes complex aggregate features in a Random Forest model, using random hill-climbing optimization algorithms. We evaluate it over several real-world relational datasets. In Section 4.3, we present extensions of our work on complex aggregates within Random Forests. Finally, in Section 4.4, we conclude on our work on complex aggregate features in relational data mining.

4.1 Random Restart Hill-Climbing of Complex Aggregate Selection Conditions

Random restart hill-climbing is an optimization heuristic which consists in performing several hill-climbing processes, each starting from a different, randomly chosen, candidate point. It differs from basic climbing in which only one process, starting from a predefined point, would be performed. In this section, we present an implementation of random restart hill-climbing to optimize the selection conjunction of conditions of a complex aggregate feature, given its aggregation process, for inclusion of a complex aggregate-based split in a decision tree model. We name this implementation "Random Restart Hill-Climbing of Complex Aggregates", or Rrhcca. We evaluate this approach on artificial data, and compare it to the original, existential version, of the Tilde relational decision tree learner, and to the aggregation-based propositionalization algorithm Relaggs, used in conjunction with a standard attribute-value decision tree learner.

4.1.1 Details of the Algorithm

As introduced in Chapter 3, a complex aggregate feature in a two-table setting is composed of two parts: a selection part, which consists in a conjunction of basic conditions to select the subset of secondary objects to aggregate, and an aggregation process, to summarize the set of selected secondary objects into one single value. The aim of our stochastic heuristics is to *optimize the selection conjunction of conditions, given the aggregation process*. In other words, there will be one optimization process, in this context a hill-climbing process, per aggregation process available.

For a given aggregation process, a *random restart hill-climbing* is performed. Firstly, a complex aggregate feature with the given aggregation process and an empty conjunction of conditions, i.e. selecting all secondary objects, is used as the first starting point. Similarly to the method introduced in Section 3.3, hill-climbing is performed, passing from an aggregate to a better neighbor, until a local optimum is reached. However, the search does not necessarily stop at this local optimum, and a new starting point can be generated using the given aggregation process and a randomly generated selection conjunction of conditions. With $a(S)$ attributes in the secondary table S, a condition on an attribute is introduced in the conjunction with a probability $1/a(S)$. If the attribute A is selected to be part of the conjunction, a random condition on this attribute is generated as follows:

- if A is categorical, the condition is $A \in Vals$, and for each possible value $v \in domain(A)$ of the attribute, there is a probability 0.5 that $v \in Vals$.
- if A is numerical, the condition is $A \in [v_l; v_u[$, with probability 0.25 to be bounded only on the left, 0.25 to be bounded only on the right, and 0.5 to be bounded on both sides.

The search resumes from this new starting point, and continues until a local optimum is reached. Then, the search resumes from a newly generated starting point, and so on, until a stopping criterion, detailed below, is reached. Pseudo-code for one hill-climbing step for an aggregation process is given in Algorithm 4.1, while Algorithm 4.2 shows a useful procedure to update the best split found by the process.

The definition of the neighborhood of an aggregate has been narrowed from Section 3.3. The considered neighborhood of a given complex aggregate feature is defined as shown in Algorithm 4.3, i.e.:

Algorithm 4.1 Process.Grow: Hill-Climbing Algorithm for One Aggregation Process

```
Input: train: labeled training set
Output: hasImproved: boolean indicating if the step of the hill-climbing has improved the best split found in the current hill-climbing of the aggregation process
hasImproved ← false
allNeighbors ← EnumerateNeighbors(this.aggregate, this.range)
for all neighbor ∈ allNeighbors do
  aggregateToTry ← CreateAggregate(this.aggregate.function, this.aggregate.feature, neighbor)
  spl ← EvaluateAggregate(aggregateToTry, train)
  hasImproved ← hasImproved or UpdateBestSplit(spl)
end for
if not hasImproved then
  for i = 1 to this.range.size do
    this.range[i] ← this.range[i]/2
  end for
end if
return hasImproved
```

Algorithm 4.2 Process.UpdateBestSplit

```
Input: split: candidate split
Output: hasImproved: boolean indicating if the split is better in terms of split metric than the best split found so far
hasImproved ← false
if spl.score > this.split.score then
  this.split ← spl
  this.aggregate ← nextAggregate
  hasImproved ← true
end if
return hasImproved
```

Addition of one basic condition to the selection conjunction: For each secondary attribute, if the conjunction does not contain a basic condition on the attribute, five random conditions on this attribute are randomly generated as described above. Each of the five basic conditions is tentatively added to the conjunction, defining five neighbors of the current aggregate. We consider five neighbors to match the number of attribute-wise neighbors defined by the modification and deletion operations presented below. Indeed, the deletion of a basic conditions defines one neighbor, while we define four possible modifications of conditions on secondary attributes already present in the conjunction.

Deletion of one basic condition from the selection conjunction: As in the previous chapter, the deletion of one basic condition from the conjunction defines a neighbor aggregate.

Modification of one basic condition from the selection conjunction: For each basic condition already present in the conjunction, modifications of this basic condition are considered to form four neighbor aggregates. These modifications depend on the kind of attribute concerned by the condition:

- if the concerned attribute A is categorical, the condition has the form $A \in Vals$ with $Vals \subset domain(A)$. For each value $v \in domain(A)$, if $v \in Vals$, a neighbor condition is defined by the replacement of $A \in Vals$ by $A \in Vals \setminus \{v\}$ in the conjunction; otherwise, the neighbor is defined by the replacement by $A \in Vals \cup \{v\}$.
- if the concerned attribute A is numerical, the condition has the form $A \in [v_l; v_u[$. Then, four modifications of this basic conditions are tested, defined by increasing/decreasing the upper/lower bound of the interval. Since these modifications can be achieved in several ways, i.e. there are many possibilities to modify the interval, we introduce a range parameter $r(A)$. This parameter controls how far the bounds of the interval will be modified. Initially, for each numerical secondary attribute A, $r(A) = 0.5$. Let us denote by $Vals = domain(A)$ the ordered set of possible values of attribute A. An increase of interval bound $v \in \{v_l, v_u\}$ is defined as follows: if v is at index i in *Vals*, i.e. $Vals[i] = v$, then the next candidate bound is $Vals[i + n(A) \cdot r(A)]$ with $n(A)$ the number of possible values of attribute A. If $i + n(A) \cdot r(A) > n(A)$, it is replaced by the maximum value of A, i.e. $Vals[n(A)]$. Similarly, the candidate decreased bound is $Vals[i - n(A) \cdot r(A)]$, if $i - n(A) \cdot r(A) > 1$, and $Vals[1]$, the minimum value of A, otherwise. If no neighbor aggregate improved over the current one, the range $r(A)$ is divided by 2 for all numerical attributes. The hill-climbing process stops when decreasing ranges becomes useless, i.e. for every numerical attribute A, $n(A) \cdot r(A) < 1$. In this case, the restart of the hill-climbing process can be performed.

There are as many hill-climbing processes as aggregation processes available. Thus, a global structure controls which hill-climbing process advances and when. It proceeds as follows: a hill-climbing process advances of one step at a time, i.e. only one neighborhood is explored at every turn, and the corresponding current aggregate is updated if one neighbor leads to an improvement of the quality metric. The search will resume from this aggregate when the hill-climbing process is allowed to advance again. At each turn, the hill-climbing process to advance is determined randomly through biased wheel selection. Each process is weighted by the best quality it achieved, i.e. the quality metric value of the best split it found so far, so the most promising processes have a higher probability of being selected. The pseudo-code of this main loop of the algorithm is shown in Algorithm 4.4.

Algorithm 4.3 EnumerateNeighbors

```
Input: aggregate: original complex aggregate feature, range array of range parameters for numerical attributes
Output: allNeighbors: array of complex aggregate features, neighbors of aggregate

allNeighbors ← []
for all attr ∈ secondary attributes not present in aggregate.selection do
  for i = 1 to 5 do
    if attr is categorical then
      vals ← random subset of domain(attr)
      baseCondition ← (attr ∈ vals)
    else if attr is numerical then
      $[v_1; v_2[$ ← random interval of domain(attr)
      baseCondition ← (attr ∈ $[v_1; v_2[$)
    end if
    allNeighbors.Add(CreateAggregate(aggregate.function, aggregate.feature, aggregate.selection and baseCondition))
  end for
end for
for all secCond ∈ set of conditions present in aggregate.selection do
  allNeighbors.Add(CreateAggregate(aggregate.function, aggregate.feature, aggregate.selection - secCond))
  attr ← secCond.feature
  if attr is categorical then
    vals ← secCond.rhs
    for all v ∈ domain(attr) do
      nextCond ← (attr ∈ vals.Switch(v))
      allNeighbors.Add(CreateAggregate(aggregate.function, aggregate.feature, (aggregate.selection - secCond) and nextCond))
    end for
  else if attr is numerical then
    $[v_1; v_2[$ ← secCond.rhs
    for all $[v_3; v_4[$ ←∈ intervals from domain(attr) obtained by decreasing or increasing $v_1$ or $v_2$ according to $range[attr]$ do
      nextCond ← (attr ∈ $[v_3; v_4[$)
      allNeighbors.Add(CreateAggregate(aggregate.function, aggregate.feature, (aggregate.selection - secCond) and nextCond))
    end for
  end if
end for
return allNeighbors
```

Algorithm 4.4 Random Restart Hill-Climbing Algorithm (Rrhcca)

```
Input: functions: list of aggregation functions, features: list of attributes of the
  secondary table, train: labelled training set
Output: split: best complex aggregate found through hill-climbing
-----------------------------------------------------------------
wheel ← InitializeProcesses(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
for i = 1 to MAX_ITERATIONS and wheel contains at least one process do
    proc ← ChooseProcessToGrow(wheel)
    hasImproved ← proc.Grow(train)
    if not hasImproved then
        if proc.split.score $\geqslant$ bestScore then
            if proc.split.score $>$ bestScore then
                bestScore ← proc.split.score
                bestSplits ← []
            end if
            bestSplits.Add(proc.split)
        end if
        proc.Reinitialize();
    end if
end for
split ← bestSplits.OneRandomElement()
return split
```

We defined three stopping criteria for the random restart hill-climbing algorithm. If at some point at least one of the following applies, all random restart hill-climbing processes stop, and the best aggregate found over all processes is considered as the optimal solution.

- A maximum number of steps over all processes has been defined. It depends on two task-related parameters. Firstly, the algorithm considers the aggregation processes globally, by changing the considered one for hill-climbing advancement at every turn. The more aggregation processes we have, the longer the hill-climbing should last to give every process a chance. Secondly, for each aggregation process, we evaluate a certain number of neighbors by modifying conditions on all secondary attributes. For each numerical attribute, the number of possible conditions is related to the number of secondary objects available in the training dataset, so this is our second parameter. Thus, we define the maximum number of global turns of hill-climbing as the square root of the product of these two task-related parameters, i.e.

$$MAX\_ITERATIONS = \sqrt{|AggregationProcess| \cdot |S|}$$

- When a hill-climbing process has been restarted 3 times without improving the best aggregate it found, it is discarded and will not be advanced further. Thus, if no hill-climbing process remains, the algorithm stops.

- Elaborating on this idea, if a certain amount of hill-climbing processes finish and restart without improving the best aggregate found over all processes, the algorithm may stop. We define this maximum number of restarts with no improvement as

$$MAX\_RESTARTS = 3 \cdot |AggregationProcess|$$

On the urban block dataset, we consider the aggregation functions *count*, *maximum*, *minimum* and *average*. With 3 numerical attributes in the secondary table, we have 10 possible aggregation processes, i.e. the *count* and the combinations of the three other functions with the three numerical attributes. They are all used to create a complex aggregate with empty selection conjunction. The splits obtained from these simple aggregates are evaluated, and the best metric score achieved for each aggregate gives a first weight for the associated hill-climbing process in the wheel selection. Table 4.1 shows possible weights of this wheel selection. We see that currently, the average area is the most promising aggregation process, since the best split it found had a quality of 0.35. Thus, it has the highest probability of being chosen for further refinement, with 28.23%.

Table 4.1 – Wheel selection of the Rrhcca algorithm

Function	Attribute	Best Score	Probability
Count		0.09	7.26%
Minimum	Area	0.05	4.03%
Minimum	Elongation	0.1	8.06%
Minimum	Convexity	0.12	9.68%
Maximum	Area	0.03	2.42%
Maximum	Elongation	0.17	13.71%
Maximum	Convexity	0.13	10.48%
Average	Area	0.35	28.23%
Average	Elongation	0.11	8.87%
Average	Convexity	0.09	7.26%
	Total	1.24	1

Let us consider the "average area" process is chosen for refinement, and the current selection conjunction is *elongation* $\in [0.7; 0.9[$. Possible refinements of this conjunction are:

- $elongation \in [0.7; 0.9[\wedge convexity \in [0.5; +\infty[$: Addition of a condition on convexity, with a randomly chosen interval.
- $elongation \in [0.7; 0.9[\wedge area \in [-\infty; 220[$: Addition of a condition on area, with a randomly chosen interval.
- $true$: Deletion of the condition on elongation.
- $elongation \in [0.65; 0.9[$: Decrease of the lower bound on elongation.
- $elongation \in [0.75; 0.9[$: Increase of the lower bound on elongation.
- $elongation \in [0.7; 0.85[$: Decrease of the upper bound on elongation.
- $elongation \in [0.7; 0.95[$: Increase of the upper bound on elongation.

If a split on complex aggregates built using one of these refinements improves on the split built using the original selection conjunction, the search will continue from this refinement, when the aggregation process will next be chosen by the wheel selection.

If none of the refinements results in an improvement, two kinds of refinement will change at the next step: the addition of a condition on an attribute not already present will be tried again, with other random intervals; the modification of the intervals in existing conditions will also change, since the range parameter will be halved. Thus, the modification of the interval belonging conditions on elongation may become:

- $elongation \in [0.68; 0.9[$
- $elongation \in [0.72; 0.9[$
- $elongation \in [0.7; 0.88[$
- $elongation \in [0.7; 0.92[$

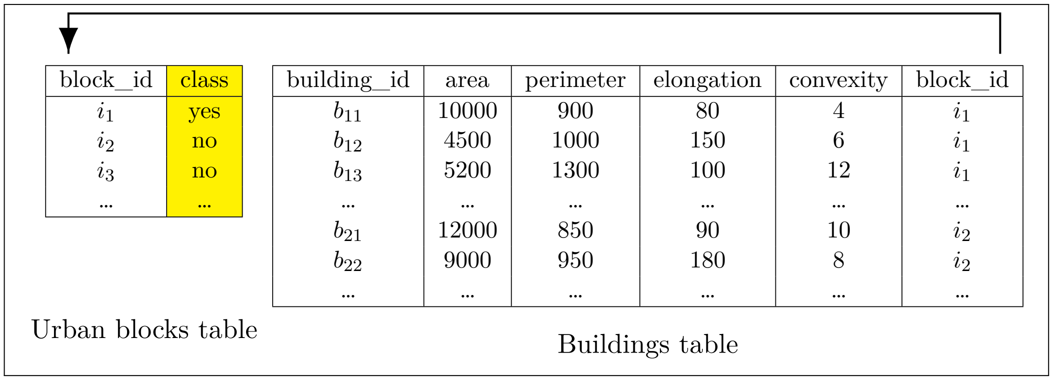
If the range parameter is too small to induce a modification of the intervals, then the search stops and will restart from a new conjunction when the aggregation process is chosen again.

4.1.2 Experiments and Results

In this section, we first assess the reliability of our algorithm on an artificially generated dataset. We also perform experimental comparison of our Rrhcca algorithm with existential decision tree learner Tilde and propositionalization algorithm Relaggs in combination with an attribute-value decision tree learner, on real-world datasets. Even though Tilde has been extended to complex aggregation, the decision tree algorithm needs to consider all complex aggregate features allowed by the language bias. Due to memory problems, we were not able to run Tilde decision tree with complex aggregates. Thus, in this section, we compare with the original version of Tilde, based on existential quantification.

Experiments on Artificial Data

Our artificial dataset consists in a binary classification task, with a main table containing a class attribute. This example is inspired by urban block classification: the main objects are blocks and secondary objects are buildings that compose the blocks. The schema is given in Figure 4.1. There are 200 instances of urban blocks in the dataset. The buildings (secondary table) have 4 numerical attributes generated using a random draw from an exponential distribution with means 10000 for area, 1000 for perimeter, 100 for elongation and 10 for convexity. We use the exponential distribution because (Jelali, Braud, and Lachiche 2012; Puissant et al. 2011) observed that the values of attributes of buildings follow this distribution in real-world cases. The number of secondary objects associated to a main object has been determined using a random draw from a geometric distribution, i.e. a discrete exponential distribution, with mean 12.



block_id	class
i_1	yes
i_2	no
i_3	no
...	...

Urban blocks table

building_id	area	perimeter	elongation	convexity	block_id
b_{11}	10000	900	80	4	i_1
b_{12}	4500	1000	150	6	i_1
b_{13}	5200	1300	100	12	i_1
...	...	...	...	...	...
b_{21}	12000	850	90	10	i_2
b_{22}	9000	950	180	8	i_2
...	...	...	...	...	...

Buildings table

Figure 4.1 – Schema of the synthetic urban block dataset

The class attribute of the main table is *yes* if *avg(convexity, buildings, area* $\geqslant$ *8000 and elongation* $<$ *150)* $\geqslant$ *7*, and *no* otherwise, i.e. if the inequality is not satisfied, or if the aggregate is not defined because no such building exists in the block.

We first measure the accuracy in 10-fold cross-validation of the 3 algorithms on this artificial dataset. The results shown in Table 4.2 are averaged over 5 runs. We observe that our algorithm learns a model for the dataset very well, the accuracy being close to 100%, while the two other algorithms are less effective on this learning task. This is due to the complexity of the aggregation condition, which neither Relaggs nor Tilde can consider.

Table 4.2 – Accuracy in 10-fold cross-validation on the artificial dataset

Algorithm	Accuracy
Relaggs + J48	66.3%
Tilde	71%
Rrhcca	97.4%

Then we observe, for every aggregation process, the behavior of the corresponding hill-climbing processes in terms of evolution of the split quality. The results are shown on Figure 4.2. Each curve represents the evolution of one hill-climbing process: it shows the evolution of the metric score of the best split found with the corresponding aggregation process during the process so far with respect to the number of iterations elapsed in the hill-climbing process. We focus on two attributes and two aggregation functions only, including the target aggregation process *average convexity*, which should produce the final aggregate choice.

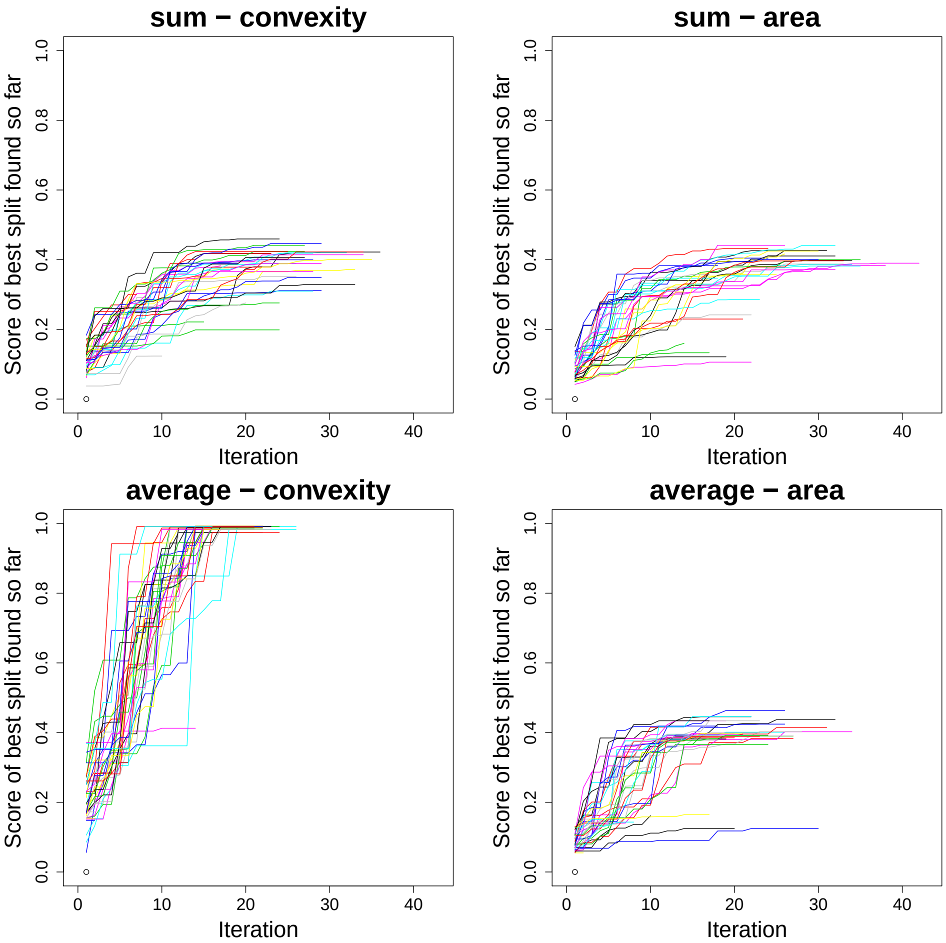


Figure 4.2 – Evolution of information gain with respect to the number of iterations elapsed in the hill-climbing process for 4 different aggregation processes.

We observe that the target branch, i.e. average convexity, does outperform the others in terms of score at the end of one hill-climbing process. We make the same observation for the branches we do not show. This assesses the ability of our algorithm to find the appropriate aggregation condition for one *(Function, Feature)* aggregation process. Moreover, we observe that, in a given branch, a majority of hill-climbing processes reach the same plateau at the end of the process.

Then, we look at the gain in terms of metric score that a modification operation on the aggregation condition achieves, i.e. we want to observe the most useful kind of neighbors. For this, we focus on hill-climbing processes for the target aggregation process. The plots in Figure 4.3 show the evolution of the difference in split metric between the modified aggregate and the original one for different hill-climbing processes. The operations shown consist in adding and removing a relevant attribute for the aggregation condition (area) and an irrelevant one (perimeter).

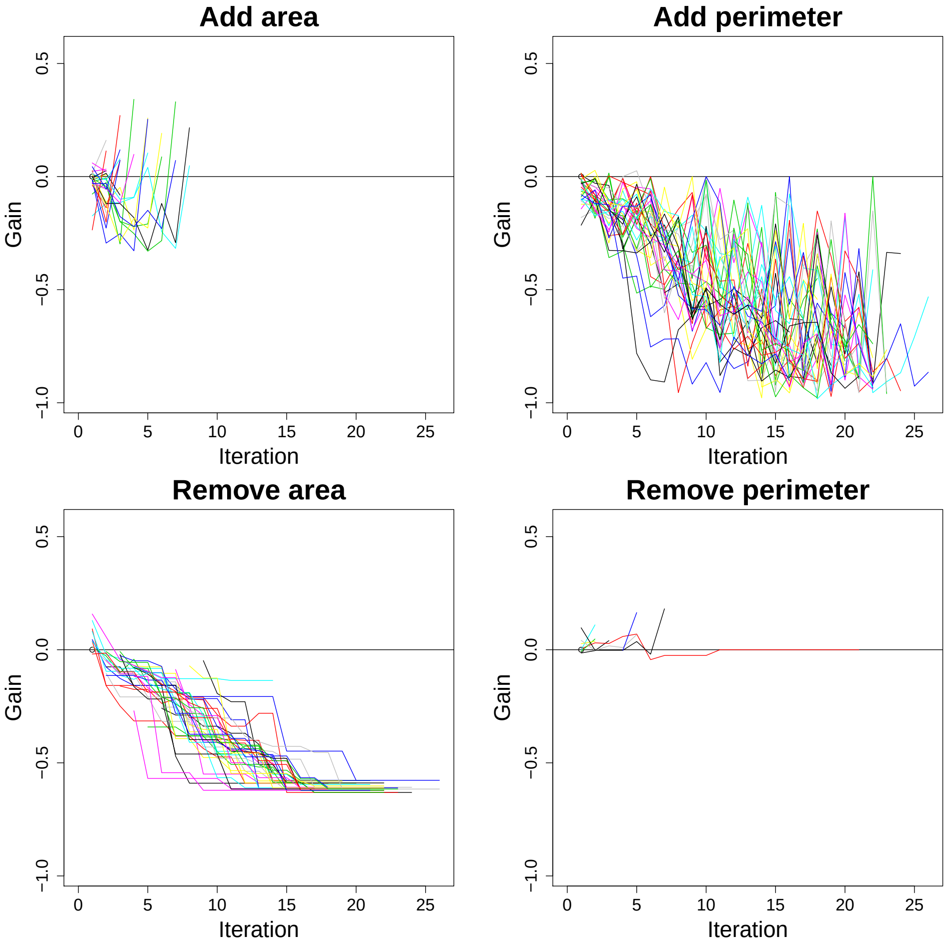


Figure 4.3 – Gain of addition/removal of conditions on area/perimeter with respect to the number of iterations elapsed in the hill-climbing process.

For both addition and deletion, we observe clear differences of behavior. For addition, we observe that once the gain of adding a condition on the area becomes positive, i.e. the condition becomes useful, the modification is chosen by the hill-climbing process and the change is not reverted later in the process since the curve stops. This means the addition of a condition on the area is never considered again, because one is already present in the aggregation condition. On the other hand, the addition of a condition on the perimeter, which should not be present in the final choice of the algorithm, almost always yields a negative gain, especially when the process is close to its end and converging to the right conjunction of conditions. Curves do not stop because the addition of a condition of perimeter is never retained as the best neighbor, thus this addition is tested at every iteration.

For deletion, we observe that removing the condition on the area almost always yields a negative gain, especially when the process is converging. This shows why the addition of a condition on the area is never considered again, since the removal of the area is considered a bad refinement choice. On the other hand, the removal of the condition on the perimeter yields a positive gain at the beginning of the process, it is then chosen by the hill-climbing and never considered again because we observed that adding a condition on the perimeter is a bad choice. High variations in gain are due to the difference between random conditions on the attribute when it is added.

To conclude, these experiments show that the addition and deletion operations are useful and compensate for a faulty initialization of the aggregation condition, by adding conditions on the relevant attributes and removing conditions on the irrelevant ones. Although we do not show them here, the plots for elongation are similar to the plots for area, since it is a relevant attribute for the aggregation condition. Likewise, the plots for convexity are similar to the plots for perimeter.

Experiments on Real-World Data

We compare our Rrhcca algorithm to Relaggs in combination with J48 decision tree learner, and Tilde on the following real-world datasets:

- Diterpenes (Dzeroski et al. 1998) is a molecule structure prediction task.
- Elephant, Fox, and Tiger (Andrews, Tsochantaridis, and Hofmann 2002) are multi-instance image recognition tasks.
- Financial (Berka 2000) is a classification task aiming at distinguishing good bank loans from bad ones.
- Urban blocks (Geo6) (Puissant et al. 2011) is a geographical classification task.
- Japanese vowels, available on the UCI repository (Lichman 2013), is related to recognition of Japanese vowels utterances from cepstrum analysis.
- Musk1 and Musk2 (Dietterich, Lathrop, and Lozano-Pérez 1997) are molecule classification tasks.

- Mutagenesis (Srinivasan et al. 1996) is a molecule classification task.
- Opt-digits, also available on the UCI repository, deals with optical recognition of handwritten digits.
- Splice junction, also available on the UCI repository, is a DNA boundary recognition task.
- The five Stulong[1] datasets are medical prediction tasks. The aim is to predict presence of a risk factor for atherosclerosis in a patient. There are five risk factors, including obesity, smoking habits, or high cholesterol, yielding five datasets.

A quantitative description of the datasets is given in Table 4.3. For each dataset, we report the number of instances, i.e. main objects, in the dataset, the number of associated secondary objects, and the number of values of the class attribute.

Table 4.3 – Description of the datasets used in the experimental comparison.

Dataset	Instances	Secondary Objects	Classes
Diterpenes	1,503	30,060	23
Elephant	200	1,388	2
Financial	682	54,859	2
Fox	200	1,320	2
Geo6	591	7,692	6
Japanese vowels	270 + 370	4,274 + 5,687	9
Musk1	92	476	2
Musk2	102	6,598	2
Mutagenesis	188	4,893	2
Opt-digits	3,823 + 1,797	244,672 + 115,008	10
Splice Junction	3,178	191,400	3
Stulong-chol	1,263	11,611	2
Stulong-ht	1,247	11,540	2
Stulong-kour	1,274	11,687	2
Stulong-obiz	1,274	11,685	2
Stulong-rar	1,268	11,666	2
Tiger	200	1,234	2

[1] The study (STULONG) was realized at the 2nd Department of Medicine, 1st Faculty of Medicine of Charles University and Charles University Hospital, U nemocnice 2, Prague 2 (head. Prof. M. Aschermann, MD, SDr, FESC), under the supervision of Prof. F. Boudík, MD, ScD, with collaboration of M. Tomečková, MD, PhD and Ass. Prof. J. Bultas, MD, PhD. The data were transferred to the electronic form by the European Centre of Medical Informatics, Statistics and Epidemiology of Charles University and Academy of Sciences (head. Prof. RNDr. J. Zvárová, DrSc). The data resource is on the web pages `http://euromise.vse.cz/challenge2004`. At present time the data analysis is supported by the grant of the Ministry of Education CR Nr LN 00B 107.

The accuracy results of the three approaches over each dataset are reported in Table 4.4. The "Japanese Vowels" and "Opt Digits" datasets come along with a test set, thus test set accuracy is reported for these. For the other datasets, mean and standard deviation of the accuracy over a 10-fold cross-validation are reported.

We performed statistical rank-based tests to search for significant differences between the three approaches. A Friedman test with 95% confidence rejected the equality hypothesis of the three approaches. Average ranks of the three methods are reported in the final line of Table 4.4. This shows a dominance of Rrhcca over the two other methods, with a lower average rank.

Table 4.4 – Average accuracy over 10-fold cross-validation or test set accuracy of the three different decision-tree-based methods on real-world datasets.

Dataset	Relaggs	Rrhcca	Tilde
diterpenes	85.7±4.77	91.35±2.57	39.45±3.65
elephant	86.5±6.26	88.5±7.47	89.5±7.62
financial	90.77±4.1	89.88±3.14	83.13±3.59
fox	86.5±9.44	84±6.15	86±4.59
geo6	76.82±4.11	74.45±3.81	59.9±8.36
jpvowels	80.27	79.19	83.51
musk1	81.89±13.7	80.33±8.86	78.33±14.58
musk2	68.64±18.62	80.55±8.84	70.82±22.99
mutagenesis	88.25±8.63	88.83±8.06	87.16±8.2
optdigits	19.53	84.36	33.33
stulong-chol	80.29±3.17	80.91±3.86	79.02±2.75
stulong-ht	77.78±2.58	76.5±3.47	77.14±2.3
stulong-kour	61.62±5.04	63.67±4.01	56.44±3.84
stulong-obiz	88.85±1.82	81.39±3.85	77.87±3.96
stulong-rar	72.32±2.84	70.35±3.31	71.22±3.35
tiger	90.5±4.97	92±4.83	93±7.89
Average rank	1.863	1.831	2.306

Then, we performed a Nemenyi post-hoc test to identify which pairs of algorithms were significantly different. We show the resulting significance graph in Figure 4.4. The graph is to be interpreted as follows: an edge from a first method to a second indicates the first method outperforms significantly the second. This relationship is transitive: if a method A outperforms a method B, and B outperforms C, then A outperforms C. Thus, if no directed path relates two algorithms, then there is no significant difference between them. We indicate in red the "best" methods, i.e. the algorithms that are never significantly outperformed in terms of predictive performance, while we indicate the "worst" methods in blue, i.e. the methods that do not outperform significantly any other method. The Figure shows that Relaggs and Rrhcca outperform significantly the existential Tilde, while no significant differences exist between Rrhcca and Relaggs.

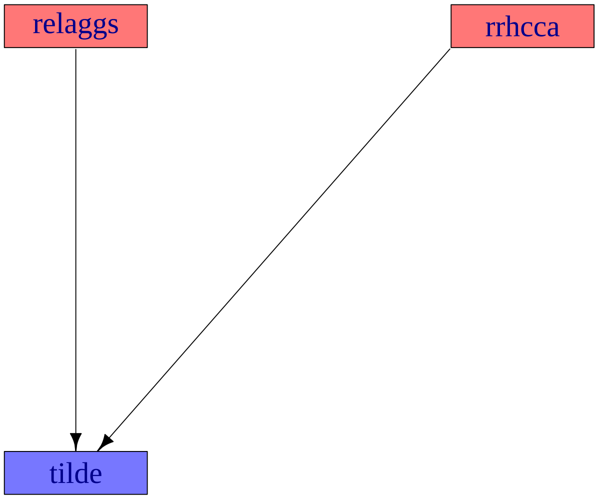


Figure 4.4 – Significance graph of the compared decision tree approaches.

We explain this lack of significant improvement by overfitting of our method. Indeed, Relaggs considers simple aggregate features, i.e. without a selection conjunction of conditions. Our algorithm also considers these features before introducing complex conditions. Our models do introduce complex aggregates because they lead to an improvement on training data. However, our results show this improvement on training data does not lead to a significant improvement on test data. Thus, our objective will be to reduce this overfitting effect.

4.2 Complex Aggregates within Random Forests

The overfitting effect induced by the use of complex aggregate features is a drawback of their great expressivity. On one hand, complex aggregate features can express very specific and precise properties. On the other hand, introduced complex aggregates may be too specific, especially in a decision tree model already prone to overfitting. Indeed, the definition of a subset of secondary objects is very strict. For instance, we consider the complex aggregate feature "*average area of buildings with convexity between 0.65 and 0.85, and elongation between 0.3 and 0.5*", learned as optimal for a split on the training set. For a new urban block to classify, a relevant building to calculate the average area may have convexity 0.851 and elongation 0.4, and thus will not be selected for aggregation, even though it is "close" to the definition of the subset to aggregate, because its elongation fits but not its convexity.

In this section, we describe the implementation of Complex Aggregates within RAndom Forests (Caraf). Compared to Rrhcca, the decision tree model is replaced by a Random Forest model combining several decision trees, known to be less prone to overfitting than a single decision tree. Since we also observed that the specificity of complex aggregate features induces overfitting, we decide to relax the search for an optimal aggregate, by replacing the random restart hill-climbing approach by a faster, but more likely to miss the optimum, random hill-climbing algorithm.

4.2.1 Random Forests

Random Forests are an ensemble method for machine learning, which relies on an ensemble of decision trees instead of a single one, to make up for the tendency of decision trees to overfit training data.

The algorithm, introduced in (Breiman 2001), relies on two main ideas to create diversity among decision trees:

- Bootstrapping: Each tree is trained using a subsample of the original training data, drawn randomly with replacement. The classic implementation creates for each tree a sample of examples with the same size as the original training set, but where repetitions may occur.

- Feature sampling: For each internal node of each tree, the split is chosen using a random subsample of the possible features. If there are F available features, a random sample of features of size $\sqrt{F}$ is commonly used.

The pseudo-code of the learning algorithm is given in Algorithm 4.5. The original implementation is based on the CART decision tree learner and thus uses Gini index as a metric for split evaluation. The bootstrapping of training examples corresponds to lines 5 to 8 of the algorithm, while feature sampling corresponds to lines 9 to 15.

Once the model is learned, prediction on an unseen example is achieved as follows:

- For classification tasks, each tree classifies the example, and outputs a prediction. The final prediction of the random forest for the example is the mode of the predictions of the trees, i.e. the majority class through a voting process.

- For regression tasks, the average of individual numerical predictions of the trees is the final prediction of the random forest.

This whole process, from classification to deployment, is sketched in Figure 4.5.

Random Forests have been previously used for relational purposes. Forf (Van Assche et al. 2006) is an extension of the relational decision tree learner Tilde to random forests with complex aggregates. However, the feature sampling is performed split-wise: all possible splits for the internal node at hand are enumerated and sampled uniformly, which may not create enough diversity in the samples of splits. In the next subsection, we propose an alternative.

Another relational Random Forest algorithm is described in (Anderson and Pfahringer 2009). It uses random rules based on the existential quantifier.

Algorithm 4.5 BuildRandomForest

```
Input: train: set of training examples, feats: set of possible split features, target: the target attribute, n: number of trees in the forest
Output: forest: a random forest

forest ← InitEmptyForest()
for k = 1 to n do
  trainForTree ← InitEmptyInstances()
  for i = 1 to train.Size() do
    trainForTree.Add(train.OneRandomElement())
  end for
  featsCopy ← feats.Copy()
  featsForTree ← InitEmptyFeatures()
  for j = 1 to $\sqrt{feats.Size()}$ do
    f ← featsCopy.OneRandomElement()
    featsCopy.Remove(f)
    featsForTree.Add(f)
  end for
  tree ← BuildDecisionTree(trainForTree, featsForTree, target)
  forest.Add(tree)
end for
return forest
```

4.2.2 CARAF: An Implementation of Complex Aggregates within RAndom Forests

The Caraf software implements our algorithms for learning complex-aggregate-based relational decision trees and Random Forests, along with the stochastic hill-climbing algorithms to search for complex aggregates. The Random Forest implementation follows Breiman's recommendations: the bootstrapping of training examples is performed in the same way, by drawing with replacement n examples from the training set of size n.

The feature sampling part has been adapted to the structure of the complex aggregate search space. In Forf, all possible splits over all complex aggregates allowed by the language bias are enumerated, and a subsample is retained by a uniform draw from the enumeration. However, our assumption is that two neighbor complex aggregate features, i.e. with same aggregation process and close selection parts, will be close in value for many examples, and thus can be considered the same feature. Therefore, complex aggregate uniform random draws of Forf may be very similar, even though the features drawn are not exactly equal from one draw to another.

As opposed to this approach, we decide to sample complex aggregates according to the structure of the complex aggregate search space. In other words, in every draw, we sample the aggregation processes first, and for each aggregation process the secondary attributes to use in basic conditions in the selection part. Let us remind a result

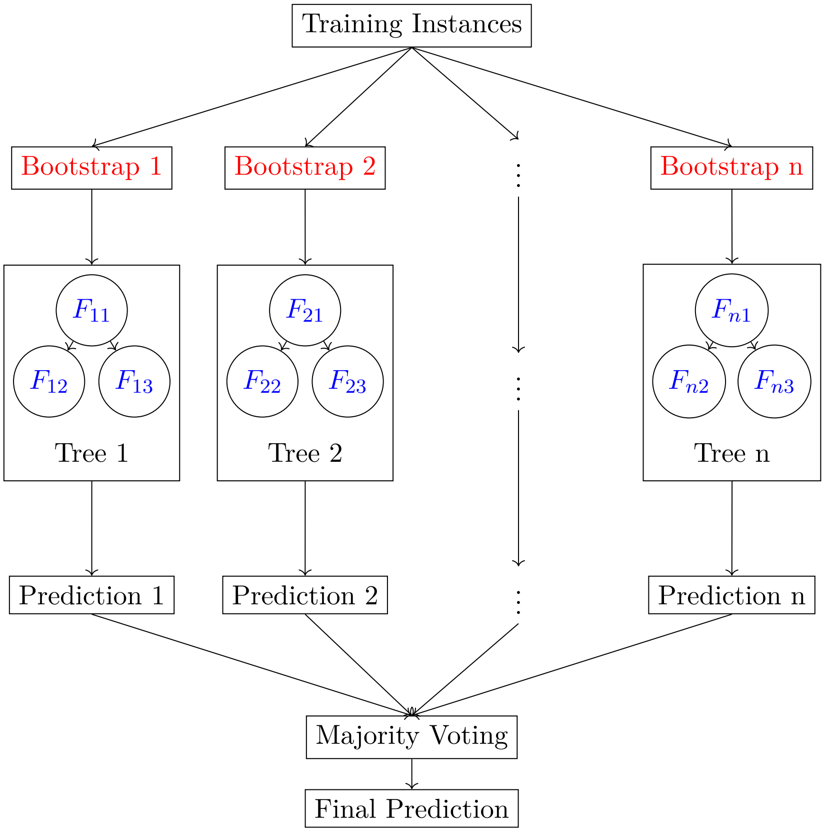


Figure 4.5 – Whole process, from training to classification, of a Random Forest.

from previous sections, with numerical attributes, the complex aggregate space size is approximately:

$$|ComplexAggregates| = |AggregationProcess| \cdot \left(\frac{|S|^2}{2}\right)^{a(S)}$$

A usual threshold for sampling features is to use approximately the square root of the original number of features. To achieve that with complex aggregates, we can keep, each time a split is to be found, a number of aggregation processes which is the square root of the original number of the aggregation processes, and half of the secondary attributes for use in the selection conjunction of conditions. According to our approximation, this gives the desired subsampling. Indeed, the square root of the number of complex aggregates can be written as:

$$\sqrt{|ComplexAggregates|} = \sqrt{|\mathbf{AggregationProcess}|} \cdot \left(\frac{|S|^2}{2}\right)^{\frac{\mathbf{a(S)}}{\mathbf{2}}}$$

More precisely, we round down the number of aggregation processes to keep, i.e. we retain $\lfloor\sqrt{|AggregationProcess|}\rfloor$ to search for a complex aggregate at each node of a tree. We round up the number of secondary attributes we keep for conditions, i.e. $\lceil a(S)/2 \rceil$.

Let us consider again the urban block dataset example, with the same aggregation processes as in the previous section. Table 4.5 shows an example of complex aggregate subsampling on this dataset. Out of the 10 aggregation processes available, the square root will be considered at each node, i.e. 3, as shown in Table 4.5a. For each aggregation process, half of the 3 secondary attributes will be kept for use in the selection conjunction of conditions, i.e. 2 per aggregation process, as shown in Table 4.5b.

Table 4.5 – Subsampling of complex aggregates.

(a) Subsampling of aggregation processes.

Function	Attribute	Chosen
Count		x
Minimum	Area	
Minimum	Elongation	
Minimum	Convexity	
Maximum	Area	
Maximum	Elongation	x
Maximum	Convexity	
Average	Area	
Average	Elongation	
Average	Convexity	x

(b) Subsampling of secondary attributes.

Attribute	Chosen
Area	x
Elongation	
Convexity	x

The Rrhcca algorithm presented in the previous section is quite time-consuming, and thus difficult to apply on high-dimensional tasks. In this section, we present two faster hill-climbing algorithms, more randomized than Rrhcca.

The first algorithm is a simpler version of the Rrhcca algorithm. Its pseudo-code is shown in Algorithm 4.6. Like Rrhcca, the aim is to look for an appropriate conjunction of basic conditions for a fixed aggregation process. Nevertheless, instead of considering all neighbors of an aggregate at each step of hill-climbing, the *Random* algorithm will consider only one, randomly chosen, neighbor, for split evaluation. If the chosen neighbor improves over the original aggregate, the search resumes from this neighbor. The random hill-climbing process has two possible stopping criteria: firstly, when a maximum number of hill-climbing steps have been performed, the search stops and the current aggregate is considered optimal. This maximum number of iterations allowed has been arbitrarily fixed at $MAX\_ITERATIONS = \sqrt{a(S) \cdot |S|}$. Secondly, when a certain number of neighbors of a given aggregate have been considered without improvement, the hill-climbing process stops. This number has been arbitrarily fixed to 20% of the maximum number of hill-climbing steps, i.e. $0.2 \cdot MAX\_ITERATIONS$. In other words, if 20% of the maximum number of iterations have passed without finding an improving neighbor for the current aggregate, the search stops and the current aggregate is considered as optimal. The pseudo-code of this hill-climbing routine is given in Algorithm 4.7.

This hill-climbing search is then performed once for each aggregation process available, starting from an empty conjunction of conditions, without a restart.

Algorithm 4.6 Random Hill-Climbing Algorithm

```
Input: functions: list of aggregation functions, features: list of attributes of the
    secondary table, train: labeled training set
Output: split: best complex aggregate found through hill-climbing
─────────────────────────────────────────────
procs ← InitializeProcesses(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
for all proc ∈ procs do
    proc.GrowRandom(train)
    if proc.split.score ⩾ bestScore then
        if proc.split.score > bestScore then
            bestScore ← proc.split.score
            bestSplits ← []
        end if
        bestSplits.Add(proc.split)
    end if
end for
split ← bestSplits.OneRandomElement()
return split
```

Following the idea of the *Random* hill-climbing algorithm, we propose to invert the loops of hill-climbing and aggregation process, materialized in the *Global* hill-climbing algorithm. Concretely, only one hill-climbing search is performed, which aims at finding the best conjunction of conditions for all aggregation processes available. For a given conjunction of conditions, all aggregation processes are used to form aggregates and splitting conditions, and the conjunction is evaluated according to the best score achieved over all aggregation processes. The maximum number of iterations allowed in the random hill-climbing process is the same as for the *Random* algorithm introduced above. The pseudo-code is given in Algorithm 4.8.

4.2.3 Experimental Results

In this section, we perform an experimental comparison of Caraf using the 3 different hill-climbing approaches with Relaggs used in combination with the Random Forest implementation in Weka (Hall et al. 2009), and with the relational Random Forest learner Forf, also based on complex aggregates. Due to memory limitations, Forf is limited to a single basic condition as the selection condition. All learned forest models consist of 100 trees.

Algorithm 4.7 Process.GrowRandom: Hill-Climbing Algorithm for One Aggregation Process

```
Input: train: labeled training set
iterWithoutImprovement ← 0
for i = 1 to MAX_ITERATIONS and iterWithoutImprovement < 0.2*MAX_ITERATIONS do
    allNeighbors ← EnumerateNeighbors(this.aggregate.condition)
    neighbor ← allNeighbors.OneRandomElement()
    aggregateToTry ← CreateAggregate(this.aggregate.function, this.aggregate.feature, neighbor)
    spl ← EvaluateAggregate(aggregateToTry, train)
    hasImproved ← UpdateBestSplit(spl)
    if hasImproved then
        iterWithoutImprovement ← 0
    else
        iterWithoutImprovement++
    end if
end for
```

Table 4.6 reports the accuracy results of our Rrhcca algorithm, used in combination with a decision tree model and a Random Forest model respectively. The results show that the Random Forest model leads to an improvement in predictive performance over the decision tree model for all datasets. The last column shows the accuracy percentage won by the Random Forest over the decision tree. The difference is particularly striking for the urban block dataset, the average accuracy over a 10-fold cross-validation increasing by 13.2% from 74.45% to 87.65%.

The accuracy results for all methods are reported in Table 4.7. Like in Section 4.1, it is test set accuracy when available, and mean and standard deviation of accuracy over 10-fold cross-validation when there is no defined test set. The average rank shows that our algorithms outperform both Relaggs and Forf, with an average rank below 3 for our methods and above for the two other competitors.

A Friedman test with 95% confidence rejected the hypothesis of equality of the five methods. The significance graph induced by the Nemenyi post-hoc test, given in Figure 4.6, shows that our three methods outperform significantly Forf, while only the use of the *Random* hill-climbing algorithm allows Caraf to outperform significantly Relaggs. Thus, our recommendation is to use the *Random* hill-climbing algorithm rather than *Global* or Rrhcca in combination with a Random Forest model.

Algorithm 4.8 Global Hill-Climbing Algorithm

```
Input: functions: list of aggregation functions, features: list of attributes of the secondary table, train: labeled training set
Output: split: best complex aggregate found through hill-climbing

aggregationProcesses ← InitializeProcesses(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
conjunction ← InitEmptyConjunction()
iterWithoutImprovement ← 0
for i = 1 to MAX_ITERATIONS and iterWithoutImprovement < 0.2*MAX_ITERATIONS do
    allNeighbors ← EnumerateNeighbors(conjunction)
    neighbor ← allNeighbors.oneRandomElement()
    hasImproved ← false
    for all aggProc ∈ aggregationProcesses do
        aggregateToTry ← CreateAggregate(aggProc.function, aggProc.feature, neighbor)
        spl ← EvaluateAggregate(aggregateToTry, train)
        if spl.score ⩾ bestScore then
            if spl.score > bestScore then
                bestScore ← spl.score
                bestSplits ← []
                hasImproved ← true
            end if
            bestSplits.Add(spl)
        end if
    end for
    if hasImproved then
        iterWithoutImprovement ← 0
    else
        iterWithoutImprovement++
    end if
end for
split ← bestSplits.OneRandomElement()
return split
```

4.3 Further Extensions of Complex Aggregates with Random Forests

In this section, we describe extensions of the work on complex aggregates in relational data mining presented in this part. Firstly, a use of complex-aggregate-based random

Table 4.6 – Accuracy difference between Rrhcca used with a decision tree model and a Random Forest model.

Dataset	Rrhcca-Tree	Rrhcca-RF	Difference
diterpenes	91.35±2.57	95.81±1.6	+ 4.46
elephant	88.5±7.47	96±3.16	+ 7.5
financial	89.88±3.14	92.96±2.76	+ 3.08
fox	84±6.15	86.5±7.47	+ 2.5
geo6	74.45±3.81	87.65±2.65	+ 13.2
jpvowels	79.19	97.57	+ 18.38
musk1	80.33±8.86	83.33±15.04	+ 3
musk2	80.55±8.84	84.27±15.53	+ 3.73
mutagenesis	88.83±8.06	92.54±5.69	+ 3.71
optdigits	84.36	95.83	+ 11.46
stulong-chol	80.91±3.86	84.72±3.26	+ 3.8
stulong-ht	76.5±3.47	84.84±2.65	+ 8.34
stulong-kour	63.67±4.01	70.96±1.73	+ 7.3
stulong-obiz	81.39±3.85	87.36±2.58	+ 5.97
stulong-rar	70.35±3.31	81.78±2.94	+ 11.44
tiger	92±4.83	95.5±7.25	+ 3.5

Table 4.7 – Average accuracy over 10-fold cross-validation or test set accuracy of the five different Random-Forest-based methods on real-world datasets.

Dataset	Forf	Global	Random	Relaggs	Rrhcca
diterpenes	92.95±1.82	95.54±1.54	95.48±1.62	90.29±2.98	95.81±1.6
elephant	94±6.99	95.5±4.38	95.5±4.97	94.5±3.69	96±3.16
financial	92.96±2.93	93.11±2.78	93.11±2.78	92.96±3.17	92.96±2.76
fox	88±7.89	86.5±7.09	89.5±3.69	86.5±7.47	86.5±7.47
geo6	79.03±4.32	87.48±2.54	88.83±2.56	84.1±4.35	87.65±2.65
jpvowels	96.22	97.3	97.03	95.41	97.57
musk1	86.67±11.48	84.44±11.94	85.67±11.81	85.67±11.81	83.33±15.04
musk2	74.64±15.86	81.45±16.21	82.45±15.73	76.55±15.8	84.27±15.53
mutagenesis	88.33±6.43	91.49±7.11	91.49±5.1	89.91±8.01	92.54±5.69
optdigits	77.52	95.88	96.16	20.76	95.83
stulong-chol	83.68±3.38	83.61±4.63	83.77±2.94	84.08±2.94	84.72±3.26
stulong-ht	84.2±2.52	85.48±2.88	85.16±3.2	85.32±3.16	84.84±2.65
stulong-kour	66.57±3.99	70.5±2.89	72.46±3.45	71.36±2.6	70.96±1.73
stulong-obiz	84.3±2.26	89.24±2.63	88.38±2.5	89.17±2.51	87.36±2.58
stulong-rar	81.78±2.94	81.78±2.94	81.78±2.94	81.47±3.08	81.78±2.94
tiger	94±8.1	95.5±6.43	93.5±6.69	95.5±6.85	95.5±7.25
Average rank	3.687	2.789	2.577	3.218	2.729

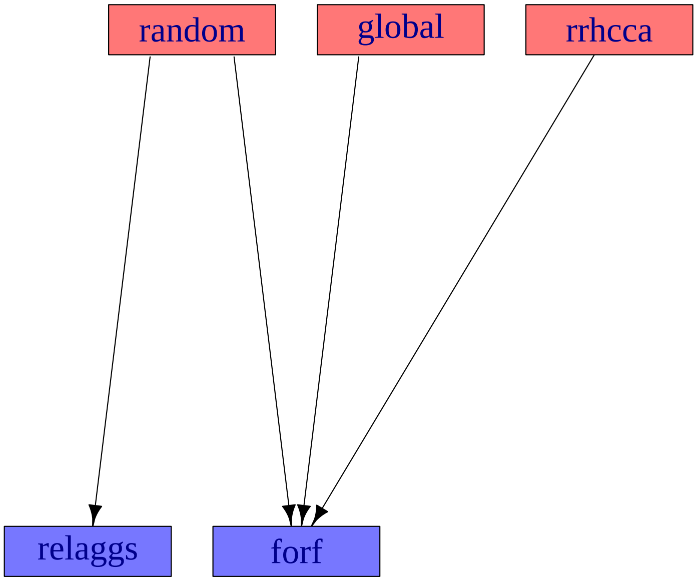


Figure 4.6 – Significance graph of the compared Random Forest approaches.

forests to perform feature selection is described. Finally, a generalization of complex aggregates, inspired by fuzzy logic, is presented.

4.3.1 Aggregation Processes Selection with Random Forests

Random Forests can be used to perform feature selection, as introduced by Breiman in (Breiman 2001). The aim would be to first check which families of complex aggregates are the most promising, to learn a model afterwards using only these useful families.

First, we will define, for a given tree in a random forest, its out-of-bag error. Every tree is trained using a subsample of the original training set, i.e. for each tree, there is a fraction of the training set that has not been actually used to build the tree. The out-of-bag error for the tree is the error made by the tree on this set of unseen examples, called the out-of-bag examples. Any error metric can be used. For classification tasks, error rate will be most likely used, while for regression tasks root mean squared error could be used.

Our aim is to perform feature selection, i.e. to assess the importance of an input feature for prediction of the output attribute. This is achieved using permutation tests. For a given tree, we first measure the out-of-bag error. The second step is to permute among the out-of-bag examples the value for the input feature we want to measure the importance. This gives a new out-of-bag examples set, for which we compute an after-permutation out-of-bag error. The importance of the feature at the tree-level is the increase in error between the after-permutation out-of-bag set and the original out-of-bag set. The final feature importance is then obtained by averaging tree-level feature importances over the whole forest.

In a relational context where complex aggregates are being used, this method needs adaptation. Indeed, the size of the complex aggregates search space implies that a given complex aggregate is rarely used twice in the same model. However, the structure of the complex aggregates allows us to define families of complex aggregates, and to measure importance of the families rather than specific complex aggregates.

Families of complex aggregates can be defined according to two elements:

- Aggregation processes: Complex aggregates sharing a common aggregation process will belong to the same family.
- Attributes in selection conjunctions: Complex aggregates whose selection conjunctions of conditions have a condition on a common attribute will belong to the same family.

These two elements can be combined to define more specific attributes, e.g. complex aggregates with the same aggregation process whose conjunctions of conditions contain a condition on the same given attribute.

For instance, on the urban block dataset, we can define families of complex aggregates at the aggregation process level, obtaining as many families as aggregation processes, 10 in this example. Thus, the following aggregates will fall into the same family, since they are all based on the same aggregation process, the average area of buildings:

- $average(area, buildings, true)$
- $average(area, buildings, elongation \geqslant 0.7)$
- $average(area, buildings, convexity < 0.5)$

If we define families based on one common attribute in the conjunction of conditions, we have as many families as attributes in the secondary table, 3 in this example. Thus, following aggregates will fall into the same family, since their conjunctions of conditions all have a condition on elongation of buildings:

- $average(area, buildings, elongation \geqslant 0.7)$
- $maximum(convexity, buildings, elongation < 0.6)$
- $count(buildings, elongation < 0.8 \wedge area \geqslant 100)$

Both family definitions can be combined to create families based on the aggregation process and a common attribute in conjunction of conditions, 30 in this example. For instance, following aggregates will belong to the same family, sharing both the aggregation process of average area of buildings and a condition on elongation of buildings:

- $average(area, buildings, elongation \geqslant 0.7)$
- $average(area, buildings, elongation < 0.9 \wedge convexity \geqslant 0.7)$
- $average(area, buildings, elongation \geqslant 0.5 \wedge area < 100)$

The permutation of values of complex aggregates has then to be performed. Since we are not permuting the values of a single feature, but of a whole family, we have to keep some coherence: each training example has one value for each aggregate in the family, and they should not be separated by the permutation. An example that obtains the value of a second example for a first aggregate, should not obtain the value of a third example for a second aggregate, but rather the value of the second example. In other words, for a given family of aggregates, only one permutation of examples has to be found, since a set of aggregate values for a given example should be conserved through permutation. We achieve this by permuting groups of secondary objects, i.e. the set of secondary objects related to one example will be assigned to another example. By doing this, all aggregate values are transferred from one example to another.

As we explained above, the family importance for each tree is computed by substracting the out-of-bag error achieved after permuting groups of secondary objects among out-of-bag examples, to the out-of-bag error obtained on the original out-of-bag set of examples. These tree-level importance values can then be averaged over the forest to obtain the final complex aggregate family importance value.

As an example, the importances of blocks main features and buildings aggregation processes are reported in Table 4.8. Importances were obtained using a forest of 100 trees built using the *Random* hill-climbing heuristic to search for optimal complex aggregate features.

Table 4.8 – Importance of main features and aggregation processes in urban blocks.

Feature	Score
Area	0.039
Elongation	0.003
Convexity	0.005
Density	0.157
Count	0.027
Minimum Area	0.062
Minimum Elongation	0.034
Minimum Convexity	0.028
Maximum Area	0.111
Maximum Elongation	0.054
Maximum Convexity	0.038
Average Area	0.177
Average Elongation	0.061
Average Convexity	0.039

We observe that the 3 most important features for urban blocks classification are the average area of buildings, the density of blocks, and the maximum area of buildings.

4.3.2 Fuzzification of Complex Aggregates

As already noticed, the strength of complex aggregates is their expressivity, but this comes with high specificity. In particular, the selection conjunction of conditions on numeric attributes uses intervals, which limits the selection of secondary objects to an hyperrectangle shape. To overcome this limitation, and ”despecialize” aggregates to make them more general, we introduce a membership degree of a secondary object to be selected for aggregation. This is inspired by works in fuzzy logic (Zadeh 1965). The fuzzification can be performed at two steps. Firstly, it can be used in the selection conditions, to assign a weight to secondary objects used for aggregation. To achieve this, some aggregation functions have to be modified to take these weights into account. Secondly, fuzzy conditions can be introduced at the level of the tree splits.

Fuzzification of Selection Conditions

To define a membership degree of secondary entities to the aggregated set, we need to move from basic conditions, i.e. interval conditions for numeric features and values set membership for categorical features, to fuzzy conditions. In other words, we want to move from 0-1, boolean, membership, to a continuous one, defined as follows:

categorical values set fuzzy membership: Given *Attr* a categorical attribute of the secondary table, this fuzzy condition has the shape $Attr \in \{(v_1, m_1), \cdots, (v_k, m_k)\}$ where $\forall 1 \leqslant i \leqslant k, v_i \in domain(Attr) \wedge m_i \in]0; 1]$. Concretely, a secondary object with value V for attribute *Attr* will have a membership degree to the aggregation set equal to the membership of V to the right hand side of the condition: if (V, m) belongs to $\{(v_1, m_1), \cdots, (v_k, m_k)\}$, the secondary object will be selected with degree m, 0 otherwise, i.e. it will not be selected for aggregation.

fuzzy interval membership: Given *Attr* a numerical attribute of the secondary table, this fuzzy condition has the shape $Attr \in [v_1; [v_2; v_3]; v_4]$ where $v_1 \leqslant v_2 \leqslant v_3 \leqslant v_4$. In the ”core” of the fuzzy interval, i.e. between v_2 and v_3, the membership degree is 1. Out of the extreme bounds, i.e. below v_1 and above v_4, it is 0. In the ”fuzzy” areas, i.e. the $[v_1; v_2]$ and $[v_3; v_4]$ intervals, it increases or decreases linearly to join the core interval and the outside. The evolution of degree membership of the secondary entity with respect to the feature value is shown in Figure 4.7.

fuzzy conjunction of conditions: The selection part of the aggregate becomes a conjunction of the kinds of fuzzy conditions defined above. The membership degrees of all basic fuzzy conditions are combined into a membership degree given by the whole conjunction using a fuzzy conjunction operator. We choose to use the minimum function to do so. The degree membership given by the conjunction is the minimum of the membership degrees given by the individual fuzzy conditions.

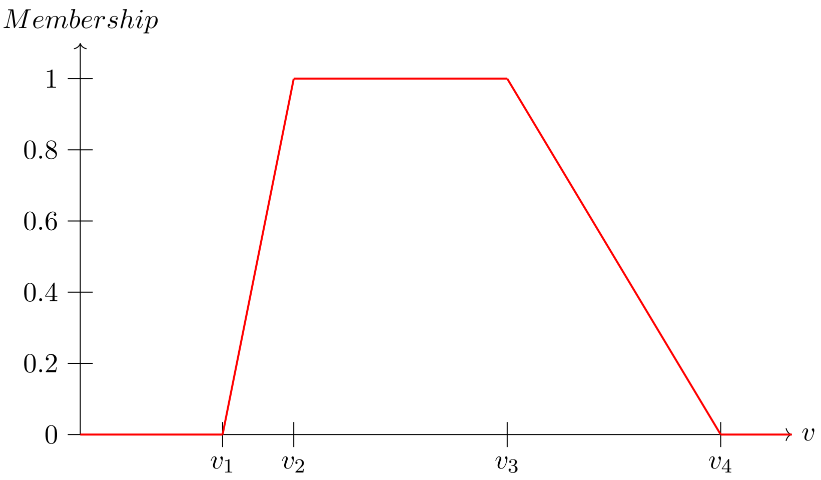


Figure 4.7 – Degree membership of secondary object with feature value v for fuzzy interval $[v_1; [v_2; v_3]; v_4]$.

Fuzzification of Aggregation Functions

We now have to aggregate secondary entities by taking into account the weights given by the fuzzy selection step. The fuzzy set to aggregate is $SecSet = \{(s_1, m_1), \cdots, (s_N, m_N)\}$, where m_i is the membership degree of secondary entity s_i to $SecSet$. If the aggregation process is based on an attribute *Attr*, we will denote by $v_{Attr}(s_i)$ the value of *Attr* for object s_i.

Some functions do not change, i.e. minimum and maximum of *Attr* do not take membership degrees into account.

The fuzzy version of the *count* function is the sum of membership degrees.

$$FuzzyCount(SecSet) = \sum_{i=1}^{N} m_i$$

The fuzzy sum of *Attr* is a weighted sum, similarly the fuzzy average is a weighted average, and the fuzzy standard deviation can be extended the same way:

$$FuzzySum_{Attr}(SecSet) = \sum_{i=1}^{N} m_i \cdot v_{Attr}(s_i)$$

$$FuzzyAvg_{Attr}(SecSet) = \frac{FuzzySum_{Attr}(SecSet)}{FuzzyCount(SecSet)} = \frac{\sum\limits_{i=1}^{N} m_i \cdot v_{Attr}(s_i)}{\sum\limits_{i=1}^{N} m_i}$$

$$FuzzyStdDev_{Attr}(SecSet) = \sqrt{\frac{\sum\limits_{i=1}^{N} m_i \cdot (v_{Attr}(s_i) - FuzzyAvg_{Attr}(SecSet))^2}{FuzzyCount(SecSet)}}$$

Finally, fuzzy quantiles can be expressed. The k$^{\text{th}}$ fuzzy n-quantile of *Attr* requires ordering *SecondarySet* by ascending value of *Attr*, i.e. ascending $v_{Attr}(s_i)$. Once this is done, we have:

$$FuzzyQuantile_{k,n,Attr}(SecSet) = \max_{1 \leqslant i \leqslant N} \left(v_{Attr}(s_i) \;\middle|\; \frac{\sum\limits_{j=1}^{i} m_j}{\sum\limits_{j=1}^{N} m_j} \leqslant \frac{k}{n} \right)$$

In other words, the k$^{\text{th}}$ fuzzy n-quantile of *Attr* is the value of *Attr* for the last secondary entity in the ordered *SecSet* such that the fuzzy proportion of elements below the entity in the ordered set, i.e. the fuzzy count of elements below the entity in the ordered set divided by total fuzzy count of the set, is below k/n. For instance, the fuzzy median corresponds to the greatest entity for which the sum of weights of the inferior elements is below half of the total weights in the set.

Fuzzification of Splitting Conditions

Decision trees can be fuzzified using the same process as for selection conditions. Splitting conditions of internal nodes are used to assign to examples a degree of membership to each of the child branches. This can be seen as a degree of confidence that the example verifies the splitting condition, and therefore should descend in the left branch. The complementary to 1 would then be the degree of confidence for the right branch. An example with weight w which receives a membership degree of m from a splitting condition will be included in the examples set to build the left branch with weight $m \cdot w$, and in the examples set for the right branch with weight $(1 - m) \cdot w$. Usually, at the root of the tree, all examples have weight 1.

Fuzzy splitting conditions on feature *Feat* will have the following shapes:

- If *Feat* is categorical, splits will have the shape $Feat \in \{(v_1, m_1), \cdots, (v_k, m_k)\}$ with $\forall 1 \leqslant i \leqslant k, v_i \in domain(Feat) \wedge m_i \in]0;1]$, like in selection conditions of complex aggregates.

- If $Feat$ is numerical, splits will have the shape $Feat > v_1 > v_2$. This fuzzy “greater than” operator is similar to the fuzzy interval used in selection conjunctions. If the value of $Feat$ for the example at hand is greater than v_1, the degree of membership to the left branch of the example will be 1, and therefore 0 to the right branch. If the value of $Feat$ is lower than v_2, then the degree of membership to the left branch is 0, and 1 to the right branch. Between v_1 and v_2, the degree of membership to the left branch evolves linearly between 0 and 1; if it is p, then for the right branch it is $1 - p$. This evolution is shown on Figure 4.8.

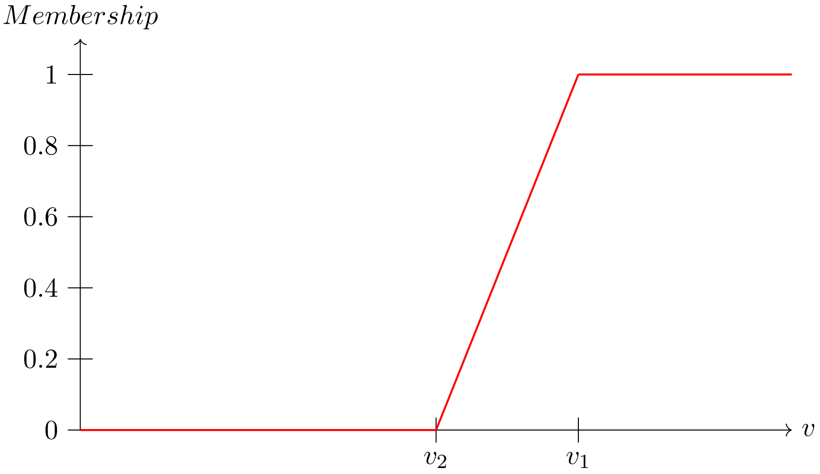


Figure 4.8 – Membership degree of an example with feature value v for fuzzy ”greater than” operator $> v_1 > v_2$.

Split comparison metrics introduced in Chapter 2 are modified to include example weights. For classification tasks, this means that impurity measures such as Gini index or Entropy are based on weight-wise class proportions, and not count-wise anymore. Concretely, p_i, which was the proportion of class i examples in the examples set, is now the ratio of weights of examples of class i on the total weight of the set. Where an example counted as 1 in the original proportions, it now counts as its weight. Similarly, the gain uses sum of weights of the examples in the sets, instead of the cardinality of the sets.

Finally, classifying an example using a fuzzy decision tree implies it may be duplicated to descend through several children of the same internal node, in case the membership degree for left and right branches are both above zero. This means the example will end up in several leaves of the same tree, which may not have the same label. Then, predictions are recombined using a majority vote: the example arrives at a given leaf with a weight w, the score for the label of the leaf is then increased by w. Once this is done for each leaf the example reached, we have scores for corresponding labels, and the one with highest score is the final prediction of the tree for the example.

4.4 Conclusion

In this chapter, we presented two stochastic heuristics to include complex aggregate features in prediction models. We addressed a twofold goal: on one hand, exploring the complex aggregate search space in an efficient, non-exhaustive way; on the other hand, avoiding the overfitting effect induced by the precision and expressivity of complex aggregate features.

The Rrhcca algorithm, based on random restart hill-climbing, fulfills the first goal by optimizing the selection part of the aggregate for a given aggregation process. According to experimental results, our method compares favorably with Relaggs simple aggregation algorithm and relational decision tree learner Tilde. However, the lack of statistical significance in the results compared to simple aggregation led us to reconsider our approach with respect to overfitting.

This led to the Caraf system, based on Random Forest models, adapted to the relational setting, rather than single decision trees, and simplified random hill-climbing algorithms to explore the complex aggregate search space. Experimental results on real-world datasets showed significant improvements over both Relaggs simple aggregates in combination with attribute-value Random Forests, and the Forf extension of Tilde to complex aggregation and Random Forests.

This extension to Random Forests can be used for feature selection processes, by identifying the most relevant aggregation processes for the classification task at hand. An extension to “fuzzy” aggregates can be considered, to palliate the excessive precision of complex aggregate features. An extension to nested relationships, as described in Section 3.3, will be necessary to handle more complex data. The inclusion of complex aggregate features in other kinds of models, such as logistic or linear regression, may be interesting. Indeed, complex aggregates are mostly numerical, which makes regression models obvious candidates for combination with complex aggregates. Finally, a potential domain of application of the relational setting and complex aggregates: multi-dimensional data and in particular spatio-temporal data, will be presented in Chapter 7.

Part II

Adaptation to Context Change with Reframing

Chapter 5

Pairwise Naive Bayes Classifiers and Output Reframing

This chapter presents an algorithm for multi-class cost-sensitive classification purposes. The resulting model relies on thresholds on the output of multiple binary models that can be modified to adapt to different contexts, which constitutes an example of output reframing.

As defined in Section 2.1, a multi-class classification task is a supervised learning task where the output attribute Y is categorical, i.e. its domain is a finite, unordered, set of values, and this set of values contains strictly more than two elements. This opposes multi-class classification, where there are at least 3 classes, to binary classification, which involves only two possible class values.

Cost-sensitive classification introduces in the learning process a new element, called the cost matrix, which represents the penalty associated to every kind of classification error, i.e. how serious is each type of error. Thus, cost-sensitive classification does not rely on the usual error rate as a loss function, but on the total misclassification cost over the test dataset. This setting is relevant for many classification tasks. For instance, in a medical binary classification task, where we learn a model to classify a patient as sane or sick, classifying a sick patient as sane is worse than classifying a sane patient as sick.

We propose to tackle these two combined tasks by addressing the multi-class aspect with a binarization approach, i.e. reducing the original multi-class problem to several binary classification tasks. We then deal with the cost-sensitive aspect in every binary sub-problem. To achieve this, we rely on soft classifiers, which output a score for each class value, rather than a “hard” class prediction. This scoring approach combines well with binarization, since in a binary problem, a scorer outputs only one, probability-related score. This scoring model is versatile, since its output has to be interpreted to give a final “hard” prediction. The output score can be reframed. To do so, we rely on a threshold for each binary scorer, to turn the score into a class prediction. These thresholds, optimized according to the costs associated to the task, are embedded in the reframed model and can be tuned to adapt to the context, without modifying binary classifiers. In this way, our approach allows for output reframing.

This chapter is organized as follows: in Section 5.1, we detail background on multi-class classification and associated binarization approaches, and on cost-sensitive learning with a ROC analysis interpretation. In Section 5.2, we review related work and state-of-the-art approaches we will compare to. In Section 5.3, we present our method to perform multi-class cost-sensitive classification. In Section 5.4, we perform an experimental comparison between our method and the state-of-the-art approaches. Finally, in Section 5.5, we present the use of our method in a reframing context.

5.1 Background on Multi-Class and Cost-Sensitive Classification Tasks

The algorithm presented in this chapter tackles a double issue: on the one hand, multi-class classification, on the other hand, cost-sensitive learning. Several approaches to multi-class classification exist: most attribute-value learning algorithms for classification, such as decision trees, were originally designed for the binary case, and were later extended to handle more than two classes. One approach consists in splitting the multi-class task into several binary tasks. This idea is called binarization and we will focus on it.

Similarly, most learning algorithms are designed to optimize accuracy performance, with no possibility to take costs into account. Nevertheless, classifiers that rely on scores can achieve cost-sensitive learning. This can be achieved by reweighting the class-wise scores according to costs.

In this section, we provide background elements on these two aspects.

5.1.1 Binarization Approaches for Multi-Class Classification

Multi-class classification is a particular family of classification tasks, i.e. supervised learning where the domain of the output attribute is a finite, unordered, set of values. Let us remind that we denote by Y the output attribute. In a classification task, we denote by C the number of possible values for the output attribute. Since, we are in a multi-class setting, we have $C \geqslant 3$ and we denote by $\{y_1, y_2, \cdots, y_C\}$ the domain of the output attribute Y, i.e. the set of its possible values.

The binarization approach consists in splitting the main multi-class task into several binary tasks: instead of learning a single multi-class model, several binary models are built for the sub-tasks. The multi-class model consists in the set of these binary models. To achieve a global multi-class prediction, each model outputs a prediction, either a “hard” prediction or a score for one predefined class among the two at hand, and these predictions are recombined into a final prediction for the multi-class task.

A general framework for binarization techniques is Error-Correcting Output Codes (ECOC), presented in (Dietterich and Bakiri 1995). The aim is to encode each class in a binary string. Table 5.1 shows this binarization setting for a 3-class dataset, the class encodings being given in rows. All strings, composed of zeros and ones, have the same length. The bits with the same index over all strings, i.e. columns of the Table, define a binary classification task: classes that have a zero at a given index in their encoding will constitute the negative class, while classes with a one in their encoding at this index will constitute the positive class. A model is be trained for each binary sub-task. Then, classification of a new example is achieved as follows: each binary model makes a prediction, either a "0" or a "1". The concatenation of these predictions gives a binary string, a multi-class prediction, which is matched to the closest string representation of a class to give the final prediction.

Table 5.1 – Exhaustive binarization with ECOC for a 3-class task.

Task Class	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
y_1	0	0	0	0	1	1	1	1
y_2	0	0	1	1	0	0	1	1
y_3	0	1	0	1	0	1	0	1

To find the encoding, the basic method is the exhaustive one: all possible binary sub-tasks achievable from the original training set are generated. Each column defines a binary sub-task. At first sight, there are 2^C binary problems, each class having the possibility to be either negative or positive. However, the first and last columns do not define binary tasks, since all classes are either negative (first column) or positive (last column). Moreover, the rest of the columns define complementary pairs: for instance, p_2 and p_7 define the same task, since all classes that constitute the positive class in p_2 define the negative class in p_7 and vice versa. Thus, only half of the remaining columns are useful, and each class can be encoded using a string of length $2^{C-1} - 1$, highlighted in yellow in Table 5.1.

The binary encoding of a class corresponds to the associated row, i.e. class y_1 is encoded by "000", y_2 by "011", and y_3 by "101". The columns define three binary sub-problems:

- p_2: y_3 (positive) VS $y_1 \cup y_2$ (negative)
- p_3: y_2 (positive) VS $y_1 \cup y_3$ (negative)
- p_4: $y_2 \cup y_3$ (positive) VS y_1 (negative)

Each binary model outputs a hard prediction, either for the positive class or the negative class. A correspondence is made with a one or a zero, and a binary string is built based on the predictions. This predicted binary string is compared to the class encodings, and the predicted class is the one whose encoding is the closest to the predicted binary string. The distance between two binary strings is usually measured with Hamming distance, which is the number of differing bits between the two strings. If we consider two strings a and b of length n, where s_i corresponds to the i$^{\text{th}}$ bit in string s, Hamming distance is defined as:

$$Hamming(a, b) = \sum_{i=1}^{n} a_i \oplus b_i$$

with $\oplus$ the exclusive OR operator, which is 0 when the elements are the same, and 1 when they are different.

Given a prediction string p of length $n = 2^{C-1} - 1$, and denoting by $enc(y_k)$ the encoding of class y_k, the predicted class $\hat{Y}$ is:

$$\hat{Y} = y_m \text{ with } m = \underset{1 \leqslant k \leqslant C}{argmin} \left(Hamming(enc(y_k), p) \right)$$

A particular case of the ECOC setting is the one-versus-all binarization setting. This consists in defining, for a C-class problem, C binary sub-problems in which one class is the positive class, while all the others constitute the negative class. Table 5.2 shows the ECOC representation of this setting for a 4-class task.

Table 5.2 – ECOC representation of the one-versus-all setting for a 4-class task.

Task Class	p_1	p_2	p_3	p_4
y_1	1	0	0	0
y_2	0	1	0	0
y_3	0	0	1	0
y_4	0	0	0	1

The construction of the ECOC setting implies that every binary sub-task considers all the classes, either on the positive side or on the negative side. An extension of ECOC to ternary codes allows to leave classes aside. For each sub-task, classes belonging to the positive class are denoted with “1”, classes belonging to the negative class are denoted by “-1”, and classes left aside are denoted by “0”. A particular case of this setting is the one-versus-one, or **pairwise**, binarization setting, discussed in (Fürnkranz 2002). For a C-class problem, it defines one binary sub-problem per pair of classes, one being the positive, the other being the negative. Table 5.3 shows the ternary ECOC representation of the one-versus-one setting, still on a 4-class task.

Table 5.3 – Ternary ECOC representation of the one-versus-one setting for a 4-class task.

Task Class	p_1	p_2	p_3	p_4	p_5	p_6
y_1	1	1	1	0	0	0
y_2	-1	0	0	1	1	0
y_3	0	-1	0	-1	0	1
y_4	0	0	-1	0	-1	-1

To obtain a final, multi-class, prediction, a classic recombination method is to consider each individual, binary prediction as a vote for the corresponding class, and to use as final prediction the class which received a majority of votes. Let us denote by M_{y_i,y_j} the binary model learned with class y_i as positive and class y_j as negative, with $1 \leqslant i < j \leqslant C$. Let us consider a test example t in our 4-class example task. One binary model predicts one of the two classes it is in charge of, and these predictions, or votes, are as follows:

- $M_{y_1,y_2}(t) = y_1$
- $M_{y_1,y_3}(t) = y_3$
- $M_{y_1,y_4}(t) = y_1$
- $M_{y_2,y_3}(t) = y_3$
- $M_{y_2,y_4}(t) = y_2$
- $M_{y_3,y_4}(t) = y_3$

Class y_1 is predicted twice, class y_2 is predicted once, class y_3 is predicted three times, and class y_4 is never predicted. Therefore, the majority vote process leads to the final prediction of class y_3 for example t.

The number of models learned in the pairwise setting is $C \cdot (C-1)/2$, which is quadratic with respect to the number of classes, unlike the one-versus-all setting where the number of models grows linearly with the number of classes. However, in the pairwise setting, only subsets of the original training set are used to train the models, since the examples that do not belong to one of the two involved classes are discarded, while in the one-versus-all setting, all training examples are used in every sub-problem. This makes up for the higher number of models to learn, and this higher number allows to introduce more parameters, which will prove useful to adapt to the cost-sensitive setting.

5.1.2 Cost-Sensitive Learning: a ROC Analysis Point of View

In some learning tasks, classification errors are not equal, and the performance is not measured by the error rate, which considers equally all types of errors. These tasks are referred to as *cost-sensitive learning*, for which (Elkan 2001) gives an introduction. For a C-class classification problem, we associate a $C \times C$ matrix, called the cost-matrix. The element (i, j) of the cost matrix, denoted by $Cost(i, j)$ is the cost of classifying in class y_j an example of actual class y_i. Depending on the context, we may use the class label instead of the index to refer to an element of the cost matrix, i.e. $Cost(y_i, y_j) = Cost(i, j)$. Usually, diagonal elements of the cost matrix are zeros, i.e. a good prediction (classifying into class y_i an example of actual class y_i) is not penalized. In the classic learning framework, where performance is based on accuracy, the cost matrix has ones out of the diagonal, indicating that all errors have the same cost, as shown on Table 5.4.

We reuse the medical diagnosis task, where classifying a patient as sick when he is actually sane is different from classifying a patient as sane when he is actually sick. This

Table 5.4 – Cost matrix for C classes in the classic learning framework.

Predicted Actual	y_1	y_2	$\cdots$	y_C
y_1	0	1	$\cdots$	1
y_2	1	0	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
y_C	1	1	$\cdots$	0

difference can be modeled by a cost matrix, such as the one shown on table 5.5. If the cost of misclassifying a sane patient as sick is 1, then the cost of misclassifying a sick patient as sane has a bigger order of magnitude, 10 for instance.

Table 5.5 – Possible cost matrix for the binary medical task.

Predicted Actual	sane	sick
sane	0	1
sick	10	0

To perform prediction in a cost-sensitive context, a way is to consider a scoring classifier M which outputs class probabilities. Usually, this is not the case, scoring classifiers output class scores, which are not necessarily probabilities. The operation transforming scores into probabilities is called calibration. Let us consider we can perform such an operation and that we have access to posterior probabilities for each class, i.e. predicted probabilities, we omit the calibration operation and consider the model M outputs a vector of posterior probabilities. For an example t, the score given by model M for class y_i is denoted by $M(t, y_i)$, while the final "hard" class prediction of the model is denoted by $M(t)$. Probability for predicting class y_i is then $p_i(t) = M(t, y_i)$. These probabilities do not take into account the cost-sensitive setting, but they allow to compute an expected cost for each class, i.e. the average misclassification cost expected if a given class is predicted. Formally:

$$ExpectedCost(t, y_j) = \sum_{i=1}^{C} p_i(t) \cdot Cost(i, j)$$

The element $p_i(t) \cdot Cost(i, j)$ of the sum is interpreted as the cost induced by the prediction of y_j when the example is of actual class y_i, weighted by the probability that the example is actually of class y_i. Summing over all classes, we obtain the expected cost, the risk induced by the prediction of class y_j. The predicted class should then be the one minimizing this risk, i.e.

$$M(t) = y_k \text{ such that } k = \underset{1 \leqslant j \leqslant C}{argmin}\,(ExpectedCost(t, y_j))$$

Another way to achieve cost-sensitive prediction is to find a weight for each class, denoted by w_i for class y_i, and to predict the class maximizing the weighted score, i.e.

$$M(t) = y_k \text{ such that } k = \underset{1 \leqslant j \leqslant C}{argmin}\,(w_j \cdot p_j(t))$$

In binary classification, this approach boils down to find an optimal threshold: if we consider a binary model, outputting two probabilities, one for each class, p_1 and p_2. Let us consider the associated weights w_1 and w_2. The prediction process works as follows:

$$M(t) = \begin{cases} y_1 & \text{if } w_1 \cdot M(t, y_1) \geqslant w_2 \cdot M(t, y_2) \\ y_2 & \text{if } w_1 \cdot M(t, y_1) < w_2 \cdot M(t, y_2) \end{cases}$$

Since we deal with probabilities, we have $p_2 = 1 - p_1$. Therefore, the prediction function can be rewritten as:

$$M(t) = \begin{cases} y_1 & \text{if } \frac{M(t,y_1)}{1-M(t,y_1)} \geqslant \frac{w_2}{w_1} \\ y_2 & \text{otherwise} \end{cases}$$

Denoting $W = w_2/w_1$, we only have one score to consider, the fraction of the two original probabilities, and so only one threshold, the ratio W of the two original class weights.

In this binary classification context, this decision process based on scores and the choice of an optimal threshold can be graphically represented by the Receiver Operating Characteristic, abbreviated in ROC curve. Let us consider a deployment dataset, with 18 examples, 10 positive and 8 negative. A scoring model gives, for each example, the scores shown in Table 5.6, for instance according to the method presented above.

Table 5.6 – Examples of the sample dataset and scores associated to them.

Class	n	n	n	p	n	p	n	p	n
Score	0.05	0.1	0.15	0.18	0.2	0.21	0.22	0.25	0.3

Class	n	p	p	p	n	p	p	p	p
Score	0.33	0.35	0.4	0.5	0.65	0.7	0.75	0.8	0.9

ROC curve construction is based on the variation of the decision threshold, which induces a variation of the hard predictions for the examples, illustrated by an evolution in the confusion matrix, and of the performance metrics. In this binary problem, the confusion matrix has size 2 × 2 as shown in Table 5.7. The notations of the matrix are:

True Positives - TP: Actual positives well classified as positives.

False Negatives - FN: Actual positives misclassified as negatives.

False Positives - FP: Actual negatives misclassified as positives.

True Negatives - TN: Actual negatives well classified as negatives.

Table 5.7 – General confusion matrix for binary classification.

Actual \ Predicted	p	n
p	TP	FN
n	FP	TN

From these notations, we define:

True Positive Rate - TPR: Ratio of well classified positives over all positives, i.e. $TPR = TP/(TP + FN)$

False Positive Rate - FPR: Ratio of misclassified negatives over all negatives, i.e. $FPR = FP/(FP + TN)$

The ROC curve is then the graphical representation of the evolution of the true positive rate TPR with respect to the false positive rate FPR when the decision threshold between negatives and positives varies, as shown in green on Figure 5.1. Concretely, we first consider the threshold to be lower than the lowest score in our dataset. All examples are then classified as positives, since all scores are above the threshold. This is illustrated by the confusion matrix from Table 5.8a: all examples are counted in the first column, counting examples predicted positive. True positive rate and false positive rate both equal 1, so the first point of the ROC curve is $(1, 1)$.

Then, we increase the threshold to 0.07, so that the prediction for the example with the lowest score changes. It is now predicted as a negative and is counted in the second column, as a true negative since it is an actually negative example. As shown in the confusion matrix in Table 5.8b, TPR is still 1, but FPR has decreased to $7/8 = 0.875$. This gives the second point of the curve at $(0.875, 1)$.

We keep increasing the threshold this way, shifting progressively all predictions from positive to negative. For instance, threshold 0.34 corresponds to the point $(0.125, 0.7)$, as can be induced from Table 5.8c.

The final point is reached when all examples are classified as negative, i.e. when they are all counted in the second column, as shown in Table 5.8d. True positive rate and false positive rate are then both 0, and the curve ends at $(0, 0)$.

In this example, the aim is to find the optimal decision threshold according to accuracy. The ROC curve allows to achieve this graphically. We denote by $\pi = (TP + FN)/(TP + FN + FP + TN)$ the proportion of positive examples in the dataset, also called prior positive probability. Then, the following holds:

Table 5.8 – Confusion matrices on our sample dataset for different threshold values.

(a) Threshold = 0
TPR = 1
FPR = 1

P A	p	n
p	10	0
n	8	0

(b) Threshold = 0.07
TPR = 1
FPR = 0.875

P A	p	n
p	10	0
n	7	1

(c) Threshold = 0.34
TPR = 0.7
FPR = 0.125

P A	p	n
p	7	3
n	1	7

(d) Threshold = 1
TPR = 0
FPR = 0

P A	p	n
p	0	10
n	0	8

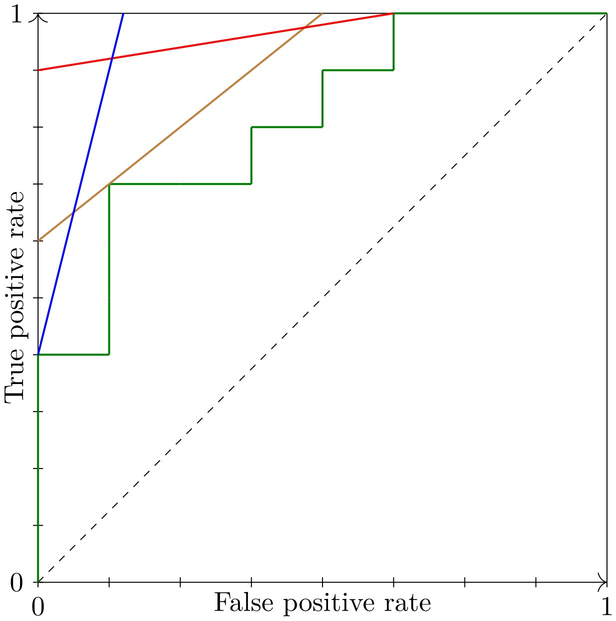


Figure 5.1 – ROC curve for the sample dataset.

$$\begin{aligned} Acc &= \frac{TP+TN}{TP+FN+FP+TN} \\ &= \frac{TP}{TP+FN} \cdot \frac{TP+FN}{TP+FN+FP+TN} + \frac{TN}{FP+TN} \cdot \frac{FP+TN}{TP+FN+FP+TN} \\ &= \pi \cdot TPR + (1-\pi) \cdot (1-FPR) \end{aligned}$$

Accuracy is a linear combination of TPR and FPR, so in the ROC space, points with same accuracy are a line. An iso-accuracy line is then defined by the equation:

$$TPR = \frac{1-\pi}{\pi} \cdot FPR + \frac{Acc}{\pi} - \frac{1-\pi}{\pi}$$

This implies all iso-accuracy lines are parallel, since they all have the same slope $(1-\pi)/\pi$. The best threshold, accuracy-wise, is the one corresponding to the point placed on the ”tangent” iso-accuracy line, i.e. the line crossing the ROC curve in only one point, which corresponds to the highest accuracy achieved by the model on the examples set. On Figure 5.1, it is shown in brown, corresponds to an accuracy $14/18 \approx 0.78$, and to the point associated to the confusion matrix shown in Table 5.8c. The optimal accuracy-wise threshold lies between 0.33 and 0.35.

The ROC curve also allows to find an optimal threshold with respect to cost minimization. We introduce a 2×2 cost matrix. This matrix has two non-zero elements out of the diagonal, the cost of a false positive, and the cost of a false negative. We fix the latter to 1, while the former is a variable c, as shown in Table 5.9.

Table 5.9 – Possible cost matrix for the binary medical task.

Actual \ Predicted	p	n
p	0	1
n	c	0

Three possible cases arise:

$c = 1$: Both types of error cost the same, this is the accuracy-based case discussed above.

$c > 1$: Misclassifying a negative as positive has a higher cost. The threshold may be set to classify more negatives accurately.

$c < 1$: Misclassifying a positive has a higher cost. The threshold may be set to classify more positives accurately.

Average cost of misclassification on the dataset can be expressed as follows:

$$\begin{aligned} AvgCost &= \frac{FN + c \cdot FP}{TP + FN + FP + TN} \\ &= \frac{FN}{TP+FN} \cdot \frac{TP+FN}{TP+FN+FP+TN} + c \cdot \frac{FP}{FP+TN} \cdot \frac{FP+TN}{TP+FN+FP+TN} \\ &= \pi \cdot (1 - TPR) + c \cdot (1-\pi) \cdot FPR \end{aligned}$$

Like in the accuracy-based task, iso-cost curves are lines in the ROC space, defined by:

$$TPR = c \cdot \frac{1-\pi}{\pi} \cdot FPR - \frac{AvgCost}{\pi} + 1$$

For a given c, iso-cost lines are parallel, and the optimal threshold can be chosen in the same way as for accuracy. On Figure 5.1, the red line corresponds to $c = 0.2$, more positives are to be classified correctly, thus the optimal threshold is lower than for accuracy and lies between 0.15 and 0.18. The blue line corresponds to $c = 5$, more negatives are to be classified correctly, thus the optimal threshold is higher than for accuracy and lies between 0.65 and 0.7.

In this chapter, the aim will be to extend the idea of a cost-sensitive decision threshold in binary classification to the multi-class setting, thanks to the pairwise binarization method.

5.2 Related Work

Some existing approaches focus on modifying the balance between classes in the training set by sampling the instances or assigning weights to the instances, e.g. (Brefeld, Geibel, and Wysotzki 2003). Few approaches try to address multi-class cost-sensitive tasks by binarization techniques and decision threshold setting. Here are state-of-the-art works that deal with this topic.

(Lachiche and Flach 2003) presents a method, denoted by “LF” in the experimental comparison, based on threshold optimization which is adapted to cost-sensitive problems. This one-versus-all approach looks for one threshold per class, one class after another. An ordering of the classes is defined, according to the decreasing order of their proportion in the training set: the first class in the ordering will be the one most represented in the set, while the last will be the least represented.

Then, the threshold for the first class is set to 1, and the threshold of a given class will be optimized according to the ones, already found, of the previous classes in the ordering. Formally, for a C-class problem, we denote by $o : [\![1; C]\!] \longrightarrow [\![1; C]\!]$ the inverse ordering function mapping the indexes of classes in the ordering to the original class indexes, e.g. $y_{o(1)}$ corresponds to the first class in the ordering. The algorithm looks for a set of weights $\{w_i | 1 \leqslant i \leqslant C\}$ with $w_{o(1)} = 1$.

A multi-class scoring model is learned on the training set. For each example, it outputs a vector of C scores, one for each class, we denote by $s_i(t)$ the score associated to class y_i for example t. The aim is to find the weights so that the weighted scores $w_i \cdot s_i(t)$ lead to the best predictive performance for example t, given that the hard prediction will be the class for which the weighted score is maximal, as defined in the previous section.

Each weight is optimized in a binary context: for class of index $j \geqslant 2$ in the ordering, the aim is to optimize the weight $w_{o(j)}$ associated to class $y_{o(j)}$, in a context where $y_{o(j)}$ is the positive class, and the negative class is composed of all previous classes in the ordering, i.e. $\bigcup\limits_{1 \leqslant i < j} y_{o(i)}$. Possible weights are defined according to the training examples from the involved classes, each weight is associated to an example, corresponding to the modified scores which causes its predicted class to switch between positive and negative. To optimize $w_{o(j)}$, the score for the positive class associated to example t is taken as $w_{o(j)} \cdot s_{o(j)}(t)$, the weighted score for class $y_{o(j)}$. For the negative class, the score is taken as the maximum weighted score for the classes composing the negative class, their associated weights having already been set. The switch point is reached when both scores are equal, i.e.

$$w_{o(j)} \cdot s_{o(j)}(t) = \underset{1 \leqslant i < j}{max} w_{o(i)} \cdot s_{o(i)}(t)$$

$$\text{Candidate weight: } \frac{\underset{1 \leqslant i < j}{max} w_{o(i)} \cdot s_{o(i)}(t)}{s_{o(j)}(t)}$$

A candidate weight is defined for each example t in the training set restricted to examples of the involved classes, and evaluated on this subset according to the goal loss, error rate or cost on the training set. The process finishes when weights have been set for all classes.

(Bourke et al. 2008) presents an approach similar to the previous one, denoted by "MC" in the experimental comparison, but resulting in a binary decision tree. There is exactly one leaf per class, and each node of the decision tree makes a split between the classes at hand. First, a multi-class scoring model is learned using the whole training set, it outputs a vector of scores in the same way as in the previous approach. At the root node, the set of all classes is split into two subsets, with approximately the same size in terms of examples in the training set. These subsets define two "meta-classes", a binary context in which the aim is to find an optimal threshold to separate the examples according to the meta-classes and a cost matrix, i.e. examples with a given class should fall in the subset of examples associated with the meta-class their original class belongs to. Formally, if we denote by $S = \{y_1, y_2, \cdots, y_C\}$ the set of classes, the meta-classes S_1 and S_2 are defined such that $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$, i.e. all classes belong to exactly one meta-class.

The score of an example for each meta-class is computed as the sum of the scores obtained from the model for the classes associated to the meta-class. For an example t, the scores for the meta-classes are defined as:

$$s_{S_1}(t) = \sum_{i|y_i \in S_1} s_i(t)$$
$$s_{S_2}(t) = \sum_{i|y_i \in S_2} s_i(t)$$

The score associated to an example is the ratio of these two scores, which defines the switch point for this example, and thus the associated threshold. The optimal threshold value is then optimized according to the binary classification task defined by the meta-classes. In a cost-sensitive setting, a misclassification cost between the two meta-classes has to be defined. It is taken as the average misclassification cost of all pairs of classes that do not belong to the same meta-class, i.e.

$$\begin{aligned} MetaCost(S_1, S_2) &= \sum_{i|y_i \in S_1} \sum_{j|y_j \in S_2} Cost(i, j) \\ MetaCost(S_2, S_1) &= \sum_{i|y_i \in S_1} \sum_{j|y_j \in S_2} Cost(j, i) \end{aligned}$$

Training examples are classified according to the candidate decision threshold, which defines a 2×2 confusion matrix with respect to the meta-classes as shown in Table 5.10.

Table 5.10 – Confusion matrix for meta-classes.

Predicted / Actual	S_1	S_2
S_1	N_{11}	N_{12}
S_2	N_{21}	N_{22}

The optimal threshold value is the one leading to the minimal meta-cost:

$$MetaCost = N_{12} \cdot MetaCost(S_1, S_2) + N_{21} \cdot MetaCost(S_2, S_1)$$

This process is repeated in the children branches, i.e. on the subset of classes which defined the meta-classes in the parent node, until all classes have been separated, defining the leaves of the tree. A new test example is classified by the tree according to the scores predicted by the multi-class scorer, which defines the scores for the meta-classes that will be used for comparison with the threshold. Depending on the outcome of this comparison, the test example will be predicted as belonging to one meta-class or the other, and will continue in one of the children branches, until it reaches a leaf giving its final, hard, class prediction.

Finally, (Landgrebe and Duin 2007) presents a pairwise method for cost-sensitive ROC optimization, denoted by "PG" in the experimental comparison. This pairwise method relies on one weight per class. First, ROC curves are computed for all pairs of classes. To reduce the dimensionality of the problem, some pairs of classes are discarded. The discarded pairs of classes are the ones that present a low degree of interaction: in terms of ROC analysis, this corresponds to classes that can be easily separated with a decision threshold, i.e. the area under the associated ROC curve is high. For the remaining pairs, an optimal decision threshold value is computed with respect to the pairwise binary task defined by the pair, using the subset of examples belonging to one of the two classes. This decision threshold is converted into two weights, one per class, as sketched previously. One of the two weights is set to 1, which determines the other, and they are then normalized so that their sum equals 1, so that weights for all pairs are on the same scale.

As in the other approaches, a multi-class scoring model is built on the training set. Then, all possible combinations of pairs covering all classes are generated. In each combination, a set of class weights is defined from the pairwise weights. A combination is then evaluated by computing the total misclassification cost it induces over the training set. This is done by using as predicted class for each example the class for which the weighted score, defined previously for each class as the product of the class weight and the class score given by the scoring model, is maximal. The weights associated to the best combination are considered the optimal weights and will the be used for prediction on test examples.

These methods have in common the concept of using one weight per class, while we want to introduce one threshold per pair of classes, which will result in a more expressive model since it will rely on more parameters. Therefore, we expect it to perform better on more complex problems such as cost-sensitive tasks.

5.3 Pairwise Classification with Threshold Optimization

In this section, we present our approach for multi-class cost-sensitive classification. It is based on an association between pairwise binarization using a scoring classifier as the base binary model, and threshold optimization to obtain predictions from these scores. As a base model, we will consider the use of a Naive Bayes classifier, that we presented in Section 2.1. Since we use a pairwise approach, our method will introduce one Naive Bayes classifier per pair of classes, i.e. for a C-class problem, $\frac{C\cdot(C-1)}{2}$ classifiers will be learned.

For each pairwise binary Naive Bayes classifier, we will refer to the two classes by the terms “positive” and “negative” class. There is a binary classifier for each pair of classes, one class being the positive class, the other being the negative. In the context of the classifier handling classes y_i and y_j, with $1 \leqslant i < j \leqslant C$, the former will be the positive class and the latter will be the negative one. We denote the class scores it returns for an example t by $s_{ij,p}(t)$ and $s_{ij,n}(t)$ for the positive and negative class respectively. Moreover, we consider the scores $s_{ij,p}(t)$ and $s_{ij,n}(t)$ are normalized, i.e. $s_{ij,p}(t) + s_{ij,n}(t) = 1$, which is the case in the WEKA implementation of Naive Bayes we use. Therefore, we will focus on only one of the two scores, since the value of one score is determined by the value of the other. We keep the score for the positive class, i.e. for class y_i and simplify the score notation as $s_{ij}(t) = s_{ij,p}(t)$. By construction, the higher $s_{ij}(t)$, the more likely the example is of class y_i.

For each binary classifier, the aim is to find a threshold th_{ij} on score s_{ij} such that each binary model achieves a hard class prediction. The classifier handling classes y_i and y_j using decision threshold th_{ij} is denoted by $M_{ij,th_{ij}}$, and the binary prediction is made as follows:

$$M_{ij,th_{ij}}(t) = \begin{cases} y_i & \text{if } s_{ij}(t) > th_{ij} \\ y_j & \text{if } s_{ij}(t) \leqslant th_{ij} \end{cases}$$

Under this assumption, since score s_{ij} lies between 0 and 1, a basic decision threshold is $th_{ij} = 0.5$. This would be optimal if the scores were probabilities, which they are not necessarily. Thus, an optimal decision threshold for each binary classifier has to be found.

At the global level, i.e. the final multi-class prediction process based on the individual predictions of all binary classifiers, a voting method is used. Each individual prediction of the binary classifiers is considered as a vote for one of the two classes it handles, and the predicted class is the one obtaining the majority of the votes. Formally:

$$M(t) = y_k \text{ such that } \underset{1\leqslant k\leqslant C}{argmax}\left(\sum_{i=1}^{C-1}\sum_{j=i+1}^{C} I(M_{ij,th_{ij}}(t) == y_k)\right)$$

$$\text{where } I(true) = 1 \text{ and } I(false) = 0$$

As an example, let us consider a 3-class classification task , with classes labeled “1”, “2” and “3”, indexed by their natural ordering. According to our pairwise binarization approach, three binary classifiers are introduced. For each classifier, the score returned is the confidence score for the class with the lowest label. Figure 5.2 shows the prediction process for a new test example. First, the classifier handling classes 1 and 2 returns a score for the example of 0.5, while the decision threshold is 0.2. Since the score is above the decision threshold, the classifier votes in favor of the example being predicted as from class 1. The same process is applied for the classifier handling classes 1 and 3: the example score 0.9 is above the decision threshold 0.7, thus this classifier votes in favor of class 1. Finally, the classifier handling classes 2 and 3 returns a score of 0.2, below the decision threshold 0.4, so the vote is in favor of class 3. The results of the voting process indicate that class 1 received 2 votes, classes 3 received 1, and class 2 received none. Thus, the final prediction for the test example is class 1.

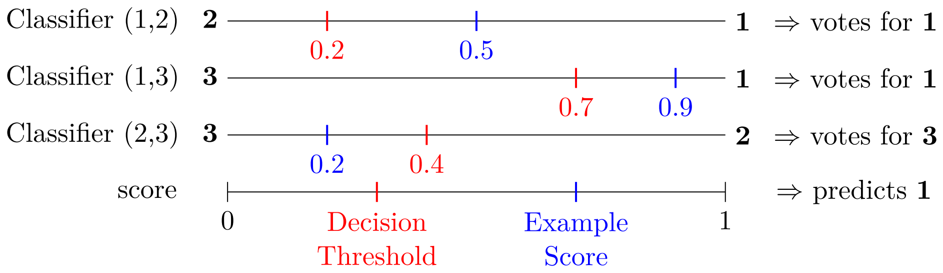


Figure 5.2 – Illustration of the pairwise binarization and of the threshold-based voting process.

The threshold optimization problem can be written as follows. Let us denote by $Th = (th_{12}, th_{13}, \cdots, th_{1C}, th_{23}, \cdots, th_{2C}, \cdots, th_{C-1,C})$ the set of thresholds for all binary classifiers. The optimal set of thresholds Th_{opt} minimizes the misclassification cost over the examples set D, i.e. with $v_Y(t)$ the actual class value of example t:

$$Th_{opt} = \underset{Th}{argmin} \left(\sum_{t \in D} Cost(v_Y(t), M(t)) \right)$$

For one binary classifier, the set of possible thresholds is the set of the scores of all examples. Indeed, as explained previously, a change of cost over the set of examples occurs when the predicted class of an example switches, i.e. when the threshold value passes from below the score of the example to above. Thus, in a dataset with n examples, if we consider a threshold value increasingly sweeping over the threshold range, at the beginning n examples are predicted positive, and 0 at the end of the sweep. Thus, there can be up to $n+1$ candidate thresholds, if we consider all examples for candidate thresholds in every binary classifier. Therefore, with $n+1$ possible thresholds per classifier, for $\frac{C \cdot (C-1)}{2}$ classifiers, the count of possible threshold sets $|Th|$ is

$$|Th| = (n+1)^{\frac{C \cdot (C-1)}{2}}$$

With a high number of classes or instances, the exhaustive exploration of all combinations is impossible to achieve in an acceptable duration. This is the reason why we will reduce the search space: instead of searching for an optimal set of thresholds, we will look for an optimal threshold at the level of the binary classifier, i.e. set all thresholds independently. In other words, we look for a set of optimal thresholds rather than an optimal set of thresholds. With this approach, the size of the search space is no longer exponential with respect to the number of classes, but quadratic:

$$|Th_{reduced}| = (n+1) \cdot \frac{C \cdot (C-1)}{2}$$

To find an optimal threshold at the binary classifier level, we propose two approaches: the first one only uses examples from the classes handled by the given binary classifier to optimize the associated threshold. The second approach takes advantage of all examples in order to optimize every individual threshold.

Our first approach, denoted by “Pairwise” in the experimental comparison, aims at finding an optimal threshold for each binary classifier. Let us focus on one binary classifier $M_{ij,th}$, handling classes y_i and y_j. Let us consider an example set D used to optimize the thresholds, it can be the original training set used to train all the binary classifiers, or a different deployment dataset, if the goal is to adapt the thresholds to a new context.

In this approach, the thresholds are optimized using the subset of examples $D_{ij} \subseteq D$ containing the examples from D with classes y_i and y_j, i.e. $D_{ij} = \{d \in D | v_Y(d) \in \{y_i, y_j\}\}$. The set of possible thresholds, denoted by Th_{ij}, is the set of scores returned by the scoring model for the examples in D_{ij}, and the optimal threshold is taken as the one minimizing the misclassification cost over the dataset D_{ij}:

$$Th_{ij} = \{s_{ij}(d) | d \in D_{ij}\}$$
$$th_{ij,opt} = \underset{th \in Th_{ij}}{argmin} \left(\sum_{d \in D_{ij}} Cost(v_Y(d), M_{ij,th}(d)) \right)$$

If the minimum is not unique, i.e. several threshold values achieve the same minimal cost, the median of the candidate values is taken as the optimal threshold. The pseudo-code for this optimization procedure is shown in Algorithm 5.1. Rather than computing the total cost over the whole examples set for each threshold value, which would result in a quadratic complexity with respect to the number of examples, we first sort the examples according to the associated score as returned by the scoring model, as stated in line 6. Then, the loop over the examples, from line 10 to 20, sweeps over the candidate thresholds in increasing order. The initial cost corresponds to the cost of all examples being classified as y_i. Indeed, for a threshold lower than all example scores, all examples will be predicted as y_j, therefore the initial cost can be computed as shown on line 7. Then, when considering the next candidate threshold, the associated example will switch from being predicted as y_i to being predicted as y_j. Thus, the total cost is modified according to the difference of cost for this example between being predicted as y_i and being predicted as y_j, as shown in line 12. The most expensive operation, computationally speaking, in this procedure, is the sorting operation, with complexity $O(n \cdot log(n))$ with respect to the number of examples, so there is a gain over the naive approach, whose complexity is quadratic.

In an accuracy-driven task, this would be enough to set the threshold. Indeed, for the binary classifier handling classes y_i and y_j, misclassifying an example of another class y_k into y_i and y_j is the same from an error-rate point of view. This is not true in a cost-sensitive task, since the cost of misclassifying an example of class y_k into class y_i, the element $Cost(y_k, y_i)$ of the cost matrix, is not necessarily the same as the cost of misclassifying an example of class y_k into class y_j, the element $Cost(y_k, y_j)$ of the cost matrix. Thus, we propose a second threshold optimization approach, denoted by “PairwiseAll” in the experimental comparison, which uses all examples for each threshold optimization instead of using only the examples from the two classes that are to be separated by the threshold. In other words, instead of using subset D_{ij} to find candidate threshold values and minimize the total cost over a set of examples, we use the whole examples set D. Formally, the set of possible thresholds and the cost minimization problem are now expressed as follows:

$$Th_{ij} = \{s_{ij}(d) | d \in D\}$$
$$th_{ij,opt} = \underset{th \in Th_{ij}}{argmin} \left(\sum_{d \in D} Cost(v_Y(d), M_{ij,th}(d)) \right)$$

Algorithm 5.1 Find optimal threshold for classifier handling classes y_i and y_j using only examples from classes y_i and y_j.

```
Input: $D$: examples set, $M_{ij}$: scoring model - handling classes $y_i$ and $y_j$, $Cost$: cost matrix.
Output: $th_{ij}$: optimal decision threshold for model $M_{ij}$.

$D_{ij} \leftarrow \{d \in D | v_Y(d) \in \{y_i, y_j\}\}$
$n_{ij} \leftarrow$ size($D_{ij}$)
$exThresh = \big((d_1, s_{ij}(d_1)), \cdots, (d_{n_{ij}}, s_{ij}(d_{n_{ij}}))\big) \leftarrow$ set of couples binding examples from $D_{ij}$ to their associated score as returned by $M_{ij}$
Sort $exThresh$ by score, $exThresh \leftarrow \Big((d_{o(1)}, s_{ij}(d_{o(1)})), \cdots, (d_{o(n_{ij})}, s_{ij}(d_{o(n_{ij})}))\Big)$
cost $\leftarrow \sum_{d \in D_{ij}} Cost(v_Y(d), y_i)$
minCost $\leftarrow$ cost
bestThresholds $\leftarrow$ [0]
for k=1 to $n_{ij}$ do
  threshold $\leftarrow s_{ij}(d_{o(k)})$
  cost $\leftarrow$ cost - $Cost(v_Y(d_{o(k)}), y_i)$ + $Cost(v_Y(d_{o(k)}), y_j)$
  if cost $\leqslant$ minCost then
    if cost < minCost then
      bestThresholds $\leftarrow$ []
      minCost $\leftarrow$ cost
    end if
    bestThresholds.Add(threshold)
  end if
end for
return $th_{ij} \leftarrow$ Median of bestThresholds
```

The pseudo-code for this approach is shown in Algorithm 5.2. It is very similar to Algorithm 5.1, the only difference being in the examples set used to optimize the threshold: the whole set D is used in the second approach.

The two approaches do not necessarily return the same optimal threshold. Let us consider again our 3-class classification example, described in Table 5.11. We focus on optimizing the threshold for the classifier handling classes “1” and “2”, in a cost-sensitive context. The examples, with their actual class and score returned by the classifier, are shown in Table 5.11a. The associated cost matrix is shown in Table 5.11b.

Figure 5.3 shows the evolution of the total misclassification cost with respect to the threshold value. For the Pairwise approach, it is computed using only examples from classes 1 and 2, while for the PairwiseAll approach, all examples are used. This explains why the total cost is higher for this approach, since examples from class 3 cannot be well classified by the classifier handling classes 1 and 2. The Pairwise approach gives an optimal threshold of 0.15, which is lower than 0.5 and could be expected since misclassifying an example of class 1 into class 2 costs more than misclassifying an example

Algorithm 5.2 Find optimal threshold for classifier handling classes y_i and y_j using examples from all classes.

```
Input: $D$: examples set, $M_{ij}$: scoring model - handling classes $y_i$ and $y_j$, $Cost$: cost matrix.
Output: $th_{ij}$: optimal decision threshold for model $M_{ij}$.

$n \leftarrow$ size($D$)
$exThresh = ((d_1, s_{ij}(d_1)), \cdots, (d_n, s_{ij}(d_n))) \leftarrow$ set of couples binding examples from $D$ to their associated score as returned by $M_{ij}$
Sort $exThresh$ by score, $exThresh \leftarrow \big((d_{o(1)}, s_{ij}(d_{o(1)})), \cdots, (d_{o(n)}, s_{ij}(d_{o(n)}))\big)$
cost $\leftarrow \sum_{d \in D} Cost(v_Y(d), y_i)$
minCost $\leftarrow$ cost
bestThresholds $\leftarrow$ [0]
for k=1 to $n$ do
    threshold $\leftarrow s_{ij}(d_{o(k)})$
    cost $\leftarrow$ cost - $Cost(v_Y(d_{o(k)}), y_i) + Cost(v_Y(d_{o(k)}), y_j)$
    if cost $\leqslant$ minCost then
        if cost $<$ minCost then
            bestThresholds $\leftarrow$ []
            minCost $\leftarrow$ cost
        end if
        bestThresholds.Add(threshold)
    end if
end for
return $th_{ij} \leftarrow$ Median of bestThresholds
```

Table 5.11 – 3-class cost-sensitive example dataset.

Class	2	2	3	2	1	3	2
Score	0	0.05	0.1	0.15	0.2	0.35	0.4
Class	3	1	1	2	3	1	1
Score	0.5	0.65	0.7	0.8	0.9	0.95	1

(a) Examples from the 3-class cost-sensitive sample dataset and associated scores returned by the scoring model handling classes 1 and 2.

Predicted Actual	1	2	3
1	0	3	3
2	1	0	4
3	4	2	0

(b) Cost matrix associated to the 3-class classification task.

of class 2 into class 1. Thus the optimization will prefer a threshold classifying as well as possible the examples from class 1. This can be observed in the figure: according to the blue curve, all examples from class 1 are well-classified by the threshold, while two examples from class 2 are misclassified. The PairwiseAll approach gives an optimal threshold of 0.5, which is higher than the one obtained with the Pairwise approach. This is explained by the influence of the examples from class 3, now taken into account. Indeed, misclassifying an example from class 3 into class 1 costs more than misclassifying it into class 2. Thus, this difference in cost will be considered by the optimization process, and the optimal threshold value increases so that the examples from class 3 are classified into class 2 rather than class 1.

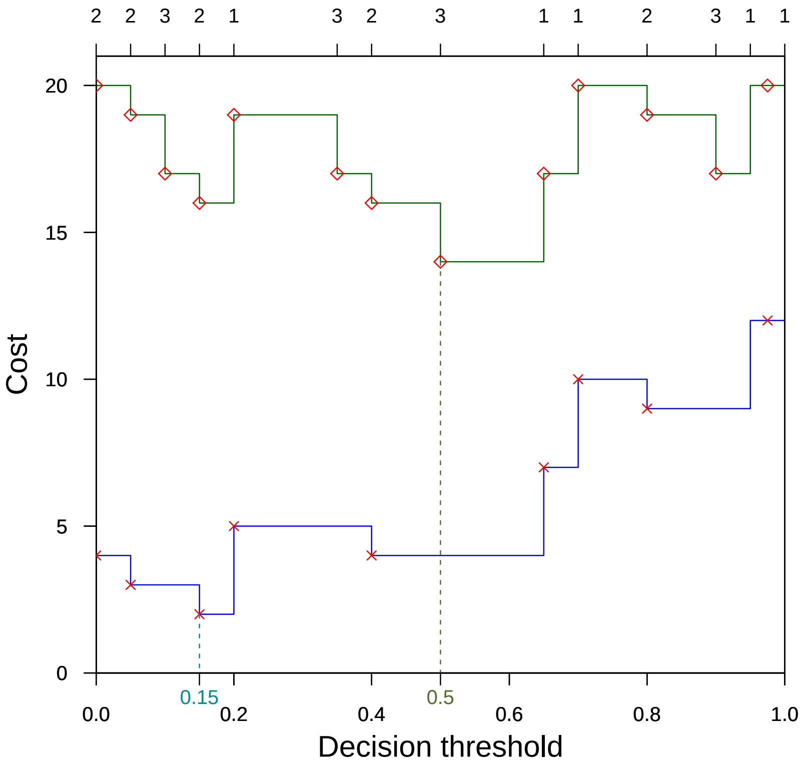


Figure 5.3 – Total cost with respect to the threshold value in the threshold optimization process for the classifier handling classes 1 and 2, for Pairwise approach (in blue) and PairwiseAll approach (in green).

5.4 Experimental Results

To measure the efficiency of our method, we ran tests on several common multi-class UCI datasets (Lichman 2013). For every dataset, we generated 30 random cost matrices, and for each of them, we performed a 10-fold cross-validation. The costs in the cost matrices are 0 on the diagonal, and a random power of 10 between 1 and 1000 elsewhere to create unbalanced costs, with all symmetric costs being different, i.e. $Cost(i, j) \neq Cost(j, i)$. As the performance metric, we consider the average misclassification cost over the test set of examples. We also used Weka (Hall et al. 2009) to discretize numerical attributes, rather than relying on a kernel to handle them.

Our tests were made using Naive Bayes from Weka as the basic classifier for all state-of-the-art methods. We compared out two approaches "Pairwise" and "PairwiseAll", and the "LF", Meta-Cost (labeled "MC"), and PairsG (labeled "PG") described in Section 5.2. We also included in our experimental comparison the cost-sensitive extensions of Naive Bayes and decision tree learner REPTree, included in Weka. This cost-sensitive extension involves the cost matrix in the prediction process, by computing the expected cost for each class as defined in Section 5.1 using the multi-class scores returned by the base model.

The predictive performance results as measured by the average cost are presented in Table 5.12. For each dataset, we show the average cost achieved by every approach, along with the standard deviation around this average. Finally, the last row of the table indicates the average rank of each algorithm over all experiments. We observe that, on this aspect, the PairwiseAll approach is the most performant, and that the difference with the Pairwise approach including only examples of the two classes handled by the classifier to optimize the threshold of this classifier is quite remarkable.

Similarly, Table 5.13 shows the runtime of all algorithms on every dataset, along with the average rank. Our approaches have more parameters to learn so it is not surprising that they are slower than cost-sensitive Naive Bayes, which learns only one model and use costs only for prediction. The same goes for cost-sensitive decision trees, "LF" and Meta-Cost which train only one model, the two latter having less parameters to optimize than our approaches. On the other hand, the coverings of classes the "PG" algorithm relies on makes it very time-consuming, even when reducing the number of considered combinations, thus it is the slowest considered approach.

To assess the statistical significance of these results, we performed a Friedman test with 95% confidence. This test rejected the null hypothesis for both predictive performance and runtime experiments. Thus, we performed a Nemenyi posthoc test to identify the pairwise differences between methods. These results are presented as significance graphs in Figure 5.4. Significance for predictive performance according to average cost is shown in Figure 5.4a, showing the PairwiseAll approach significantly outperforms all the others, while the basic Pairwise approach was outperformed by three methods including the PairwiseAll approach. This shows that considering all examples to optimize every threshold is meaningful.

Figure 5.4b shows the significance graph for runtime performance. As suggested by

Table 5.12 – Average cost per instance of algorithms for each dataset.

Dataset	Classes	CSNB	CSTree	LF	MC	Pairwise	PairwiseAll	PG
audiology	24	80±59	76±58	56±47	84±57	93±63	42±36	137±117
bridges-material	3	31±53	36±63	29±52	46±67	54±74	32±53	49±70
bridges-rel-l	3	27±54	29±58	26±55	41±64	49±78	21±42	32±60
bridges-span	3	39±67	50±94	66±98	63±95	89±104	57±90	60±95
bridges-type	6	96±89	136±124	71±80	103±95	105±93	56±66	111±96
colic-outcome	3	69±58	45±63	14±20	17±22	35±51	15±23	68±65
colic-site	63	212±67	206±83	141±57	217±65	194±67	106±42	215±66
colic-type	8	41±35	8±14	9±15	9±15	18±29	6±11	29±31
ecoli	8	37±34	42±37	18±21	33±32	35±47	18±18	30±29
flags	8	104±77	130±127	42±48	105±78	86±65	33±37	115±85
glass	7	102±74	75±64	38±53	80±64	61±53	26±32	79±62
postop	3	43±73	40±77	38±71	47±69	55±80	35±61	63±100
segmentation	7	37±19	10±8	51±85	27±21	35±33	30±29	25±17
solar-flare-c	8	58±41	40±43	11±9	44±42	42±44	10±8	42±32
solar-flare-m	6	18±17	12±16	5±8	12±16	32±74	5±8	13±15
solar-flare-x	3	4±8	1±3	1±2	1±3	2±4	1±2	2±6
Average rank		4.75	3.95	3.31	4.08	4.27	3.14	4.51

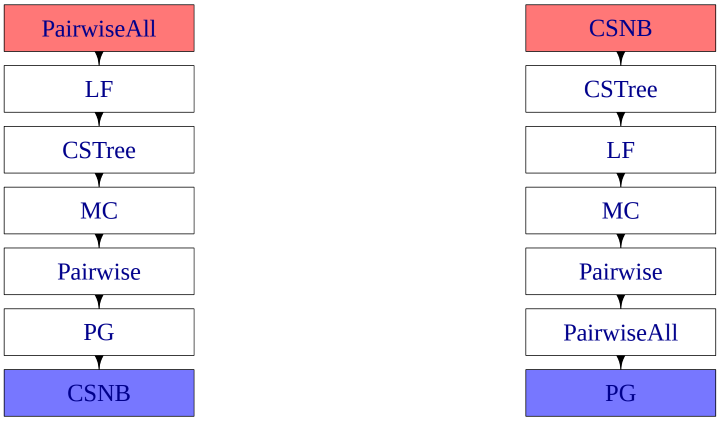
Table 5.13 – Average runtime of algorithms for each dataset.

Dataset	Classes	CSNB	CSTree	LF	MC	Pairwise	PairwiseAll	PG
audiology	24	5±4	8±25	48±17	196±38	342±82	1196±167	548±370
bridges-material	3	0±0	1±1	1±1	1±0	1±0	1±2	3±6
bridges-rel-l	3	0±0	0±1	0±1	1±1	1±1	1±1	3±4
bridges-span	3	0±0	0±0	0±1	0±1	1±1	1±1	3±1
bridges-type	6	0±0	0±1	1±0	2±1	3±1	6±4	16±6
colic-outcome	3	2±1	60±16	12±4	21±8	22±8	28±12	83±33
colic-site	63	23±2	67±14	228±10	1125±31	4356±332	16769±419	7608±3848
colic-type	8	4±1	60±24	29±2	78±4	102±6	245±11	627±374
ecoli	8	3±1	7±2	27±2	71±5	85±6	210±11	861±159
flags	8	1±1	5±5	10±2	26±3	32±3	75±3	469±128
glass	7	2±1	6±2	19±2	50±4	55±3	127±6	1494±87
postop	3	0±0	0±0	0±0	1±0	1±0	1±0	3±1
segmentation	7	170±9	598±106	1396±69	3644±89	4219±216	9443±276	124853±7753
solar-flare-c	8	1±1	4±1	13±2	28±3	39±3	90±5	566±137
solar-flare-m	6	1±0	3±1	9±3	20±2	25±4	49±23	155±14
solar-flare-x	3	1±0	1±0	5±1	8±1	8±1	10±2	32±2
Average rank		1.41	2.5	2.89	3.86	4.76	5.76	6.81

the previous average rank results, our methods are outperformed on time consumption, only doing better than PairsG.

5.5 Output Reframing with Threshold Adaptation

In this section, we present an extension of our pairwise multi-class cost-sensitive algorithms for reframing. The context change we consider is a change of cost matrix between two contexts. For instance, on the Postop dataset, where the aim is to predict if the



(a) Significance graph for costs.

(b) Significance graph for runtime.

Figure 5.4 – Significance of the experimental comparison of the multi-class cost-sensitive learners.

patient should be kept in the hospital or not, such a change can happen between two different hospitals. If we consider a first hospital, with a certain cost of keeping the patient in the hospital when he is actually sane and could be sent back home, this cost could be higher in another hospital, for instance if the hospital has less beds to receive patients and cannot afford to take patients when it is not necessary. This is illustrated in Table 5.14, where the costs of keeping the patients in the hospital, i.e. predicting classes GHF (keeping the patient in the General Hospital) and ICU (sending the patient to the Intensive Care Unit), are higher in hospital 2.

Table 5.14 – Example of cost matrix context change between two hospitals.

P A	GHF	ICU	home
GHF	0	2	10
ICU	5	0	20
home	2	5	0

(a) Cost matrix associated to hospital 1.

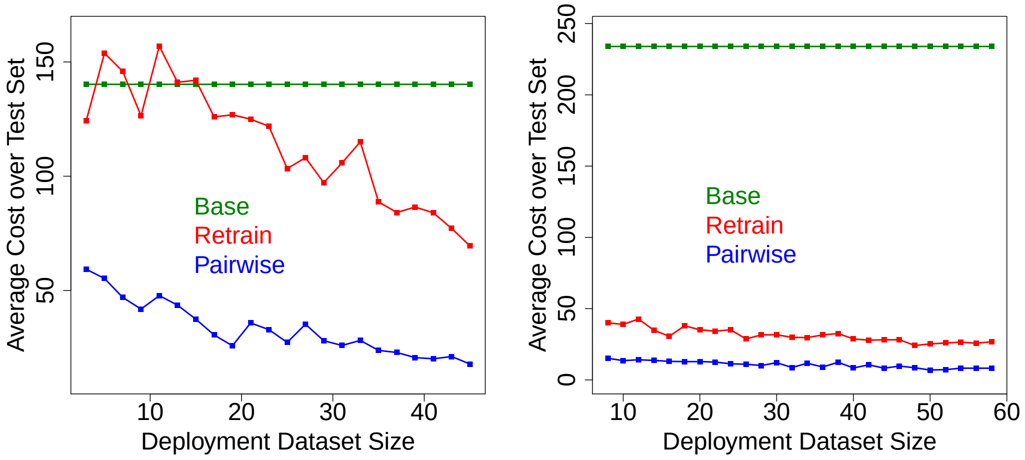
P A	GHF	ICU	home
GHF	0	**5**	10
ICU	**10**	0	20
home	**5**	**10**	0

(b) Cost matrix associated to hospital 2.

Output reframing with our pairwise approach consists in learning the pairwise binary models using a training example set in the original context, and then using a few labeled examples and the cost matrix from the deployment context to optimize the thresholds for this deployment context. We compare this approach with the base model learned using the training set from the original context. Therefore, this base model uses the cost matrix from the original context to make predictions in the deployment context. We also compare with the retraining approach, where the scoring model is learned using the few labeled data from the deployment context, and makes predictions using the cost matrix from the deployment context. All models are Naive Bayes classifiers as implemented in Weka.

We compare the approaches using the average test set cost with respect to the number of labeled instances from the deployment context used for reframing. For each dataset, two cost matrices were randomly generated as described in the previous section, denoted by $Cost_{train}$ for the original training context, and $Cost_{deploy}$ for the deployment context. Corresponding elements between the two matrices are different, i.e. $Cost_{train}(i, j) \neq Cost_{deploy}(i, j)$. We consider the same datasets as in the previous section. To create one dataset per context, the original dataset is separated into two halves: one standing for the original context, the other for the deployment context. The deployment part is again separated into two halves: one fixed test set, and one for reframing purposes. For every considered number of reframing examples, we make 30 random draws of the appropriate number of examples from the reframing examples set.

Figure 5.5 shows comparison results on two datasets. We observe that performance of both retraining and our pairwise threshold optimization approach improves with the number of deployment examples available to retrain or reframe. However, our pairwise approach, taking advantage of the knowledge learned in the original context, performs better than retraining.



(a) Colic-outcome dataset.

(b) Solar Flare-c dataset.

Figure 5.5 – Average cost of different reframing approaches.

5.6 Conclusion

In this chapter, we presented a pairwise approach for multi-class cost-sensitive classification, introducing more parameters than usual state-of-the-art models to deal with this complex dual task. The pairwise thresholding approach becomes powerful with the use of examples of all classes to optimize the cost-sensitive threshold relative to two classes, and allows our method to outperform state-of-the-art algorithms.

Our method is able to tackle some kinds of context changes, such as a difference of costs, by re-optimizing the thresholds according to the deployment context. This constitutes an example of output reframing, where thresholds on the model output are modified in a systematic way to adapt the new context.

For other kinds of context changes, such as distribution shifts, we will consider other families of reframing methods, focusing on input data reframing and “hard” model output reframing.

Chapter 6

Input and Output Reframing of Numerical Features

In this chapter, we propose methods to perform input and output reframing of numerical features. This is joint work with C.F. Ahmed in the context of his postdoctoral work on the Reframe project. As presented in Section 2.3, input reframing consists in, given a model trained in a first context, to transform the input attributes of data from another context so that the first model is relevant for prediction on this new data. On the other hand, output reframing consists in modifying the final prediction of the model so that it is consistent for the deployment context.

Our proposed methods focus on reframing numerical features. As a simple motivating example, let us consider again a temperature-based model to predict high pullover sales. It is a binary classification task (high sales or not, *true* or *false*), with respect to a single, numerical, input attribute. In a first city, the sales are high when the temperature is lower than 5°C, and they are not otherwise, as shown on the target model from Figure 6.1a. In a second city, in which "cold" has a slightly different meaning, high sales of pullovers occur when the temperature is below 10°C, as shown on Figure 6.1b.

Temp < 5
true
false
True
False
Temp < 10
true
false
True
False

(a) Target model for city 1.

(b) Target model for city 2.

Figure 6.1 – Target models, decision-tree shaped, for pullover sales in the two context cities considered.

A model is trained to perform such prediction in the context of City 1. It fits the given target model for City 1. If we use it to predict sales on data from City 2, it will not be as accurate, since it will fail when temperature falls between 5 and 10°C. A possible transformation would be to substract 5 to temperatures in data from City 2, and the original model would be accurate again. This is the intuitive idea we propose: to learn a transformation of numerical input attributes of the deployment context so that their values ”look like” values of the input in the original training context.

At the other extremity of the supervised learning process, the prediction of the model can also be reframed. Since we focus on reframing numerical features, we place ourselves in the context of regression tasks. On the pullover example, let us consider now the regression task which consists in predicting directly the quantity of pullover sold over a day. In the first city, pullover sales are, on average, more important than in the second city. Figure 6.2 shows the observed distribution over a year of these daily sales values in City 1 (in red) and City 2 (in green). In City 1, there are, on average over a year, 100 pullover sold per day, while in City 2, there are 50.

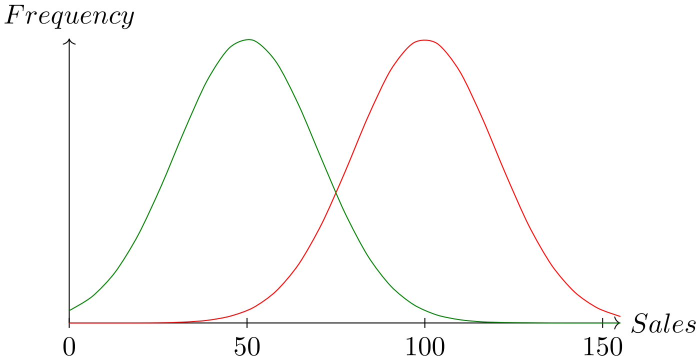


Figure 6.2 – Observed distribution over a year of daily pullover sales in the context of City 1 (in red) and City 2 (in green).

Most regression models are unable to output predictions deviating from the range of the labels in the training set they were built with. Thus, if we train a model in City 1, in which pullover sales are higher than in City 2, it will be unable to make correct predictions on data from City 2, in which less pullover are sold. In City 2, the model will overestimate sales. Again, a possible transformation would be to substract 50 to the original prediction of the model, so that the output, originally closer to values from City 1, looks like an output from City 2. This is the same intuitive idea as the one behind input reframing, but applied to the output.

Such a transformation is useful when, for instance, few labeled examples from the deployment context are available, i.e. there is not enough labeled data to train a new model suitable for the deployment context. Our objective is twofold: firstly, we want to learn a useful transformation, i.e. the performance of the model trained in the original context on data from the deployment context has to be better after reframing the data. In other words, the transformation has to make an improvement in prediction performance. Secondly, the transformation also has to result in a performance improvement over a model trained on the few labeled examples from the deployment context. These two simple approaches, the original base model and retraining on deployment data, will constitute our two comparison baselines.

The rest of the chapter is organized as follows. In Section 6.1, we present work related to input reframing and similar tasks. In Section 6.2, we present our approaches to perform data reframing using affine transformation of numerical features. In Section 6.3, we compare our reframing algorithms to the baselines and the state-of-the-art algorithm GP-RFD. In Section 6.4, we provide a real-world example of dataset shift in a regression task, in which both input and output shifts are present. In Section 6.5, we extend our reframing algorithms to the relational setting through the use of complex aggregates. Finally, in Section 6.6, we conclude on our work on reframing.

6.1 Related Work

Input reframing is one way to address covariate shift. As presented in 2.3, covariate shift corresponds to the change of input distribution while the conditional distribution of the output given the input remains the same. Some methods developed to deal with this kind of tasks include:

- Importance Weighted Cross-Validation (IWCV) (Sugiyama, Krauledat, and Müller 2007) is a variant of cross-validation designed to overcome the bias problem that occurs with classic cross-validation in presence of covariate shift. It proposes a new, unbiased estimate of the empirical risk based on weighting the individual validation errors with an importance factor measuring the distribution shift between train and test set, materialized by a density ratio between the two sets.

- (Bickel, Brückner, and Scheffer 2009) presents a method to learn a discriminative model under covariate shift without explictly modeling train and test distributions, but rather by generating a model to estimate the likelihood for an example to belong to train or test set, and use this likelihood as example weight to train the discriminative model.

- Kernel Mean Matching (KMM) (Gretton et al. 2009), as the name may indicate, reweights the training data so that means of weighted training and original test input distributions coincide in a high dimensional feature space.

However, none of these approaches are reusing a model, since they can adapt to only one testing context, and thus need retraining to fit other deployment contexts.

Other research areas proposed methods to deal with tasks where training and test data follow different distributions:

- Theory revision (Ourston and Mooney 1994) consists in an explicit modification of a given model to improve its performance. The associated so-called example setting is a rule-based model, which is made more specific or more general depending on the examples not-covered by the current theory. This family of methods is closer to structural reframing than input reframing which does not aim at a modification of the model, and thus does not depend on the nature of the model.

- Transfer learning (Pan and Yang 2010) is a reuse of knowledge acquired from learning a model in a first context, to learn a new model for a second context. It does not reuse the model from the first context itself, but rather trains a new one for the second context, biasing the learning phase with knowledge acquired in the first context. Thus, it cannot be considered reframing, since reframing does not retrain a model for the second context.

- Domain adaptation (Daumé and Marcu 2006) adopts a statistical point of view on transfer learning. It considers changes in distributions between two or more contexts, assuming one context is the target, the one for which we want a prediction model. The idea is to learn several models, capturing context-specific or general information, and to combine them to obtain an accurate target-specific model. It is actually the opposite of reframing, in which a model is trained in a single source context and then adapted to fit other target contexts, while domain adaptation consists in learning models from possibly more than one context, in order to deploy in a single target context.

A structural reframing approach has been proposed to adapt decision tree models (Al-Otaibi et al. 2015). These versatile decision trees, rather than using “hard” numerical thresholds in node split conditions, retain how data was optimally split in the original context, in terms of data distribution in children branches. Thresholds in the deployment context are then chosen to keep the data distributions equal to those of the training context.

The approach closest to ours is the Genetic Programming-based feature extraction for the Repairing of Fractures between Data (GP-RFD) introduced in (Moreno-Torres, Llorà, et al. 2013). The approach aims at tackling all dataset shift tasks. Considering two datasets, a training one and a test one differing in data distributions, a model is built on the training dataset. Then, the algorithm looks for an optimal transformation of the test dataset through usual genetic operators (selection, mutation, crossover), giving a transformed test dataset on which the model learned previously can be applied. However, this approach is highly time-consuming and requires a lot of deployment data to find the transformation. Our approaches will overcome these limitations.

6.2 Reframing of Numerical Input Attributes

In attribute-value supervised learning, the aim is to learn a model, a function M mapping the input features vector to the output feature value: $\hat{Y} = M(\mathbf{X})$. In the reframing paradigm, we consider the existence of a model M_1, trained in a first context using a training dataset D_1. In an input reframing task from the first context to a second, deployment context, we have a deployment dataset D_2 that cannot be used to train a model M_2 suitable for the new context. This may happen because D_2 is not enough, or not at all, labeled, i.e. only the values of the input features are accessible. Another possibility is that the size of D_2 is small, too small to train a model. This may happen when data has to be labeled by human experts. Depending on the classification task, labeling the data may be time-consuming or expensive.

We denote by $X = (X_1, X_2, \cdots, X_a)$ the vector of input attributes, i.e. an element $X_i \in X$ is an attribute, and by Y the output attribute. The respective domains of attributes are denoted by $\mathbb{X} = \mathbb{X}_1 \times \mathbb{X}_2 \times \cdots \times \mathbb{X}_a$ where $\mathbb{X}_i = domain(X_i), 1 \leqslant i \leqslant a$, and $\mathbb{Y} = domain(Y)$. Input reframing is about reusing model $M_1 : \mathbb{X} \longrightarrow \mathbb{Y}$ for prediction in context 2, by learning a transformation $f : \mathbb{X} \longrightarrow \mathbb{X}$ of the input feature vector, so that the model defined by $M_1 \circ f$ is accurate in context 2. Formally:

$$\text{Context 1: } \hat{Y} = M_1(X)$$
$$\text{Context 2: } \hat{Y} = M_1(f(X))$$

Figure 6.3 summarizes this process: the model is built using the training set, and used along with the deployment set with few labeled data from the second context to find an input transformation. This input reframing transformation is applied to test examples before the application of the model, which outputs predictions for the test set examples.

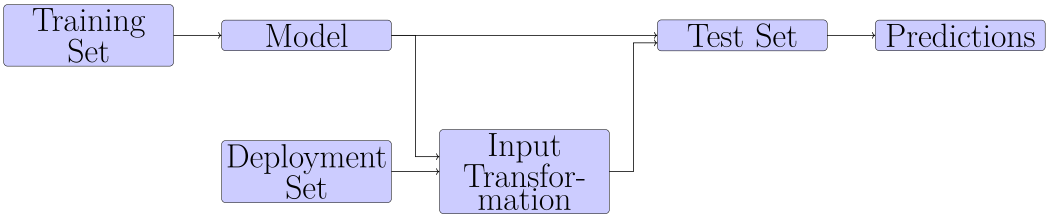


Figure 6.3 – Illustration of the input reframing process.

Our aim is to find such transformations for numerical attributes. We will achieve this through simple affine transformations and stochastic optimization.

6.2.1 Affine Transformation of Numerical Input Features

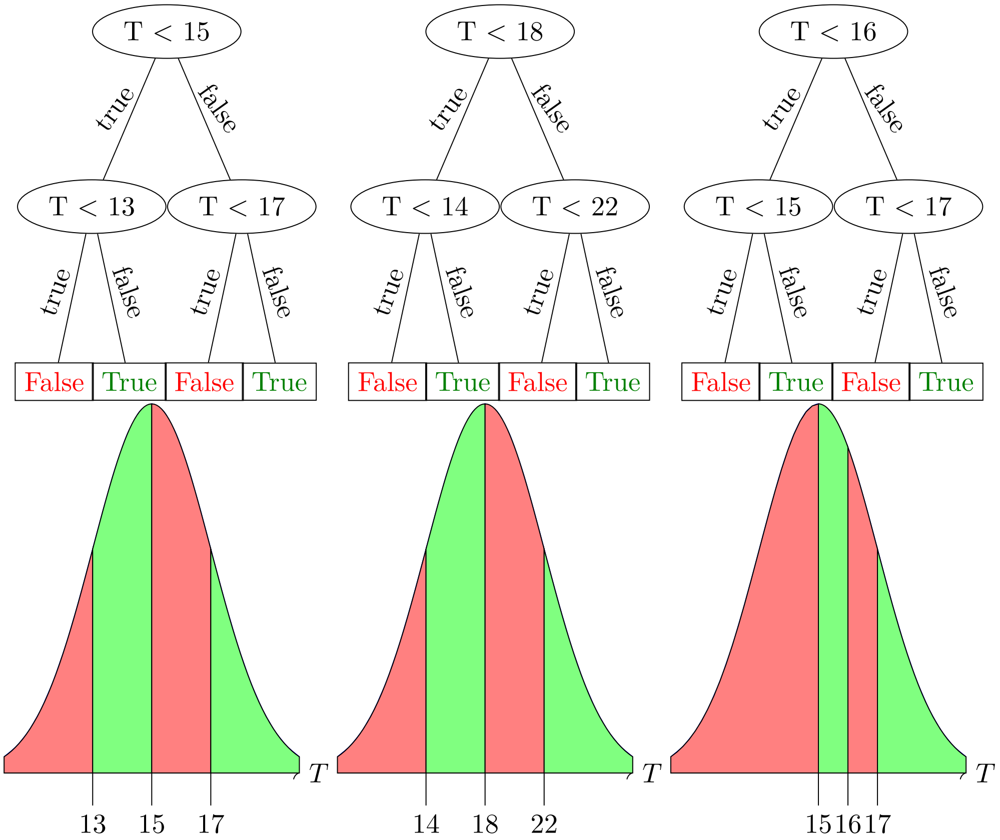
We will first focus on affine transformations of numerical features. Under this assumption, f follows:

$$f(X) = (f_1(X_1), f_2(X_2), \cdots, f_a(X_a))$$
$$\text{where } \forall 1 \leqslant i \leqslant a, \begin{cases} f_i(X_i) = X_i & \text{if } X_i \text{ is categorical} \\ f_i(X_i) = \alpha_i X_i + \beta_i & \text{if } X_i \text{ is numerical} \end{cases}$$

In other words, each numerical input feature gives two transformation parameters, the slope α_i and the intercept β_i of the corresponding affine function. The challenge is then to find a set of such parameters that optimizes the performance of model $M_1 \circ f$ on dataset D_2.

Example 6.1. Affine reframing in a one-dimensional setting.

We consider a fictive binary classification task, based on a single input attribute. The target model involves two levels and three splits in a decision tree. The single input attribute, Temperature as denoted by T, is normally distributed. We assume three different cities, standing for three different contexts, where City 1 is the training context. Temperature distribution, as well as target models, are shown on Figure 6.4.



(a) City 1.
$T \sim \mathcal{N}(15, 2^2)$

(b) City 2.
$T \sim \mathcal{N}(18, 4^2)$

(c) City 3.
$T \sim \mathcal{N}(15, 2^2)$

Figure 6.4 – Target models and input attribute distribution in three context cities.

In the three contexts, target models consist of a partition of the temperature space, i.e. $\mathbb{R}$, in four intervals. One of the two classes is assigned to each interval, and the same schema is used in the three contexts: if we consider the natural ascending ordering of the four intervals, the first one corresponds to class label *False*, the second to *True*, the third to *False*, and the last one to *True* again. As mentioned previously, temperature follows a normal distribution in the three contexts, denoted by $\mathcal{N}(\mu, \sigma^2)$ with μ the mean, or center, of the distribution, and σ its standard deviation.

The first context, i.e. City 1, is characterized as on Figure 6.4a: the cutpoints for the intervals are located at 13, 15 and 17, while temperature is normally distributed with mean 15 and standard deviation 2.

In the second context, City 2, described on Figure 6.4b, the temperature distribution differs from City 1. It is now centered on 18 with standard deviation 4. We observe that, with respect to the temperature distribution, the target model has not changed. Indeed, the cutpoints in City 1 were located at $\mu - \sigma$, μ and $\mu + \sigma$, using notations from the temperature normal distribution, and this still holds in City 2. The difference is in the values of mean and standard deviation of the distribution. The optimal reframing affine function f, if we have a model trained on City 1 data that we want to reuse on data from City 2, is $f(T) = 0.5 \cdot T + 6$. Then, the cutpoints of the target models from City 2 shift to the cutpoints of City 1, and the original model is usable.

In the third context, City 3, described on Figure 6.4c, the temperature distribution is the same as in City 1. However, the target model cutpoints are different, being located at 15, 16 and 17. A ”perfect” affine transformation is still achievable though: $f(T) = 2 \cdot T - 17$ shifts the cutpoints correctly and makes the original model usable.

These two possible context changes show the interest of the use of an affine reframing function to deal with distribution shifts. All reframing tasks will not have a straightforward unique solution, but the use of affine transformation, quite simple and introducing a limited amount of parameters, will be found to perform well on a lot of reframing tasks. ■

6.2.2 Stochastic Algorithms for Reframing Numerical Input Attributes

We introduce two hill-climbing algorithms to achieve reframing of numerical input attributes through affine transformations. Both are based on stochastic optimization, this choice being motivated by the absence of convexity properties of our problem. According to (Bergstra and Bengio 2012), this family of techniques is well-suited for this kind of tasks. First, we describe the search space of the affine parameters and how we sample it in stochastic processes. Then, we detail the two stochastic hill-climbing algorithms we introduce.

Sampling of Reframing Functions Parameters

Since we use stochastic algorithms, we need to randomly sample values for slopes and intercepts. Since they are numerical parameters, they both lie in $\mathbb{R}$. However, we can limit this search space. First, for every attribute, we consider only increasing functions, i.e. with a strictly positive slope. This is based on the assumption that, if the model from the training context considers at some point low values of the attribute with respect to the range of the attribute in the training context, it should also use low values in the deployment context, with respect to the range in the deployment context. Let us denote, for a given numerical attribute Z, $[l_{Z1}; u_{Z1}]$ its range in the training context, and $[l_{Z2}; u_{Z2}]$ its range in the deployment context. An illustration is given in Figure 6.5, for the temperature attribute of our synthetic reframing task from City 1 to City 2. We consider that temperature $T1$ in City 1 lies between 11 and 19, while it lies between 10 and 26 in City 2.

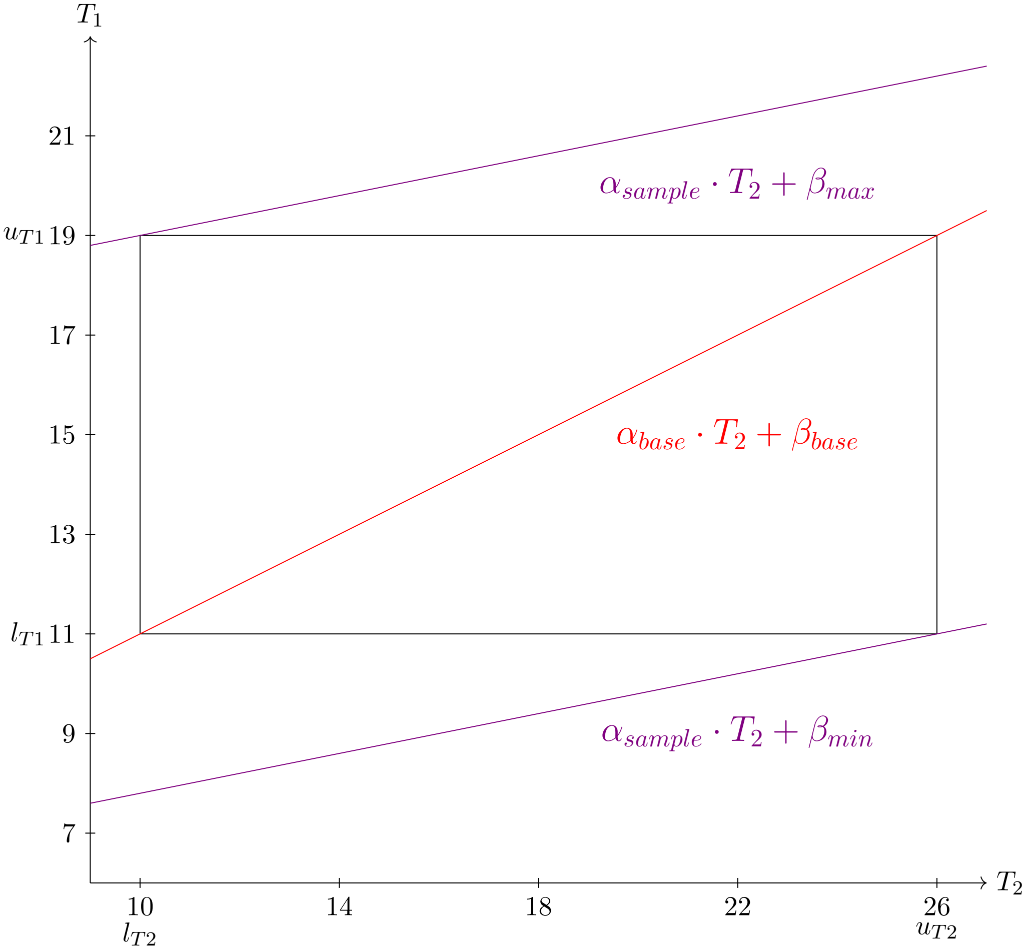


Figure 6.5 – Affine mapping of deployment context values(x-axis) to training context values (y-axis) of an input attribute.

A basic reframing function would be the one mapping the deployment context to the training context, since we want the input values from the deployment context to look like values from the training context. This function, denoted by $f_{Z,id}$, also gives an initial parameter set $(\alpha_{Z,base}, \beta_{Z,base})$. Formally,

$$\begin{aligned}
f_{Z,id}(x) &= \alpha_{Z,base} \cdot x + \beta_{Z,base} \\
&= \frac{u_{Z1} - l_{Z1}}{u_{Z2} - l_{Z2}} \cdot (x - l_{Z2}) + l_{Z1} \\
\alpha_{Z,base} &= \frac{u_{Z1} - l_{Z1}}{u_{Z2} - l_{Z2}} \\
\beta_{Z,base} &= l_{Z1} - \alpha_{Z,base} \cdot l_{Z2}
\end{aligned}$$

For the temperature-based task, such a function maps the interval [10; 26] to the interval [11; 19]. As mentioned above, it is $T_2 \longmapsto 0.5 \cdot T_2 + 6$, represented in red on Figure 6.5.

From now on, we will consider optimization of parameters for a given attribute, thus we will drop the Z reference to the attribute in our notations. In further pseudo-code, this initialization of parameters to their basic value is instantiated by the procedure *InitParams*.

The slope of the basic function is used as a baseline to optimize the slope parameter of the reframing function, which should be the "center" of the random sampling. We want to sample equally functions that increase faster than the baseline and function that increase slower. Thus, we perform a uniform sampling of the logarithm of the slope, i.e. we draw a real number from a distribution centered on zero, from a procedure denoted by *SampleAroundZero*. Then, we consider a factor obtained by taking 10 to the power of the number drawn previously, and multiply this factor with our baseline to obtain a candidate value for the slope parameter.

$$\begin{aligned}
k &= SampleAroundZero() \\
\alpha_{sample} &= 10^k \cdot \alpha_{base}
\end{aligned}$$

As a sampling distribution, we arbitrarily use a normal distribution with mean 0 and standard deviation 0.5, this standard deviation actually controls the deviation of the slope around the baseline.

From this candidate, denoted by α_{sample}, we induce a range for the intercept parameter. The function output should at least partially lie in the range of the attribute in the training context. From this, we define two bounds, the minimal intercept β_{min} is achieved when the function reaches the minimum value of the training context at the maximum value of the deployment context, i.e. the function output lies in the training range only at the end of the deployment range, formally $f_Z(u_{Z2}) = \alpha_{sample} \cdot u_{Z2} + \beta_{min} = l_{Z1}$. On the other hand, the maximal intercept is achieved when the function reaches the maximum value of the training context at the minimum value of the deployment context, i.e. $f_Z(l_{Z2}) = \alpha_{sample} \cdot l_{Z2} + \beta_{max} = u_{Z1}$. The bounds are then:

$$\beta_{min} = l_{Z1} - \alpha_{sample} \cdot u_{Z2}$$
$$\beta_{max} = u_{Z1} - \alpha_{sample} \cdot l_{Z2}$$

An illustration is provided in Figure 6.5: considering $\alpha_{sample} = 0.6 \cdot \alpha_{base}$, the two lines in magenta represent the functions obtained by using either β_{min} as intercept (lower line), or β_{max} (upper line).

We sample the intercept from a distribution centered on the average of the bounds. More precisely, we use the following normal distribution:

$$\beta_{sample} \sim \mathcal{N}\left(\frac{\beta_{max} + \beta_{min}}{2}, \frac{\beta_{max} - \beta_{min}}{4}\right)$$

This sampling of parameters is instantiated in pseudo-code by the procedure *SampleParams*.

Reframing with Stochastic Hill-Climbing

The first reframing algorithm we introduce, called Reframing with Stochastic Hill-Climbing (RSHC), is detailed in Algorithm 6.1. It is a random restart hill-climbing algorithm, similar to the one we developed for complex aggregates generation in Chapter 4.

The initial set of parameters, as defined above, is used as a baseline and as the first starting point for random restart, as shown in line 6. Then, the random restart loop, from line 10 to line 28, is initiated. The number of random restarts is arbitrarily set to 50. The hill-climbing loop, from line 13 to line 22, continues as long as at least one modification of a parameter over all reframing functions leads to an improvement. From line 15 to 20, tests of modification of the parameters are performed for each numerical attribute, as a hill-climbing on slope and intercept parameters, one after the other. When the hill-climbing optimization over all parameters stops, a random restart is performed, i.e. a new sample of parameters is drawn and the global hill-climbing process is performed again, until 50 random restarts have been tested.

These hill-climbing algorithms are detailed in Algorithms 6.2 and 6.3. The algorithms are very similar and follow the same scheme. The difference is how slopes and intercepts evolve: slopes are modified geometrically, by multiplying them with a factor, while intercepts are modified arithmetically, by adding a step proportional to the range of the considered attribute. The performance baseline is obtained by evaluating the current set of parameters, as shown on line 5.

Then, the algorithm looks for a hill-climbing direction, to which it will stick. First, the parameter to optimize is decreased, and the set of parameters is evaluated, as shown from line 8 to 12. The same evaluation is perfomed after increasing the parameter, from line 13 to 17. The performance of the parameters, and the possible update of the parameters, are performed using the *TestPerformance* function detailed in Algorithm 6.4. If at least one of the two moves has led to an improvement, the direction that led

Algorithm 6.1 Reframing with Stochastic Hill-Climbing Algorithm (RSHC)

```
 1: Input: cls: base classifier, training: labeled training dataset from original context, deploy: labeled little-size dataset from deployment context, attrs: set of input attributes.
 2: Output: bestAlphas (= $(\alpha_1, \cdots, \alpha_a)$): slopes of the affine reframing function, bestBetas (= $(\beta_1, \cdots, \beta_a)$): intercepts of the affine reframing function.
 3: model ← cls.Train(training, attrs)
 4: numAttrs ← attrs.NumericalSubset()
 5: alphas ← [], betas ← []
 6: InitParams(alphas, betas, training, deploy)
 7: bestAlphas ← alphas, bestBetas ← betas
 8: adjustAlpha ← 0.05, adjustBeta ← 0.05
 9: count ← 0
10: for count = 1 to 50 do
11:   loopImprovement ← false
12:   iterImprovement ← true
13:   while iterImprovement do
14:     iterImprovement ← false
15:     for i=1 to numAttrs.Size do
16:       localImprovement ← SlopeHillClimbing(model, training, deploy, alphas, betas, i, adjustAlpha)
17:       iterImprovement ← iterImprovement or localImprovement
18:       localImprovement ← InterceptHillClimbing(model, training, deploy, alphas, betas, i, adjustBeta, $numAttrs[i].Max - numAttrs[i].Min$)
19:       iterImprovement ← iterImprovement or localImprovement
20:     end for
21:     loopImprovement ← loopImprovement or iterImprovement
22:   end while
23:   if loopImprovement then
24:     count ← 0
25:     bestAlphas ← alphas, bestBetas ← betas
26:   end if
27:   SampleParams(alphas, betas, training, deploy)
28: end for
29: return (bestAlphas, bestBetas)
```

to the best improvement is memorized, and the hill-climbing will be performed only in this direction, i.e. either by only decreasing the parameter, or by only increasing it.

This hill-climbing loop, from line 19 to line 31, introduces a speed parameter, controlling how fast the parameter moves. As long as an hill-climbing step leads to an improvement, the speed parameter is doubled, so that the modification of the parameter is stronger and stronger, this corresponds to line 21 to 29. When the improvement stops, the speed is reinitialized, and the acceleration process starts over. If the acceleration process fails to improve at first step, the speed parameter remains equal to 1 and the hill-climbing stops, as shown in line 21.

Reframing with Randomized Search

The second algorithm we introduce, called Reframing with Randomized Search (RRS), is detailed in Algorithm 6.5. It is a simple random search algorithm: it randomly tries a fixed amount of sets of parameters, and considers the best it tested as the optimum. At every iteration of the loop, it generates a random set of parameters in the way described earlier for the RSHC algorithm. The first candidate parameter set is the basic one, defined above and instantiated in the *InitParams* procedure, it gives the initial best performance. Then, random samples are generated according to the method described above and implemented in the *SampleParams* procedure, and evaluated against the best parameter set found so far. We set the number of samples evaluated to 1000.

This algorithm has the advantage of being easy to use, as well as being faster than RSHC. This will be highlighted in the upcoming experimental results.

6.3 Experimental Results

We performed experiments to assess the performance of our reframing algorithms. Our approaches were implemented in Java using the WEKA API (Hall et al. 2009). Firstly, we compare our approaches to our baselines, i.e. direct reuse of base model and retraining, on synthetic data. Secondly, we compare our approaches to the GP-RFD (Moreno-Torres, Llorà, et al. 2013) algorithm on real-world classification tasks. Finally, we will show an example of use of our reframing algorithms on a real-world regression task.

6.3.1 Performance of Reframing on Synthetic Data

We compare the performance of the RHSC and RRS algorithms to the performance of their simplest baselines, i.e. direct application of the model learned in the original training context (denoted by Base), and training of a new model using the few labeled deployment data from the second context (denoted by Retrain). As shown in (Moreno-Torres 2013), the base learner family giving best predictive performances when coupled with GP-RFD is the decision tree family. Therefore, for these two first sets of experiments, aiming at measuring performance on classification tasks, we use the J48 decision tree learner from WEKA, which is an implementation of C4.5 (Quinlan 1993).

We consider again the synthetic binary classification task example based on temperature from Section 6.2 and the three contexts introduced in Figure 6.4. According to the same data distributions and target models, we generated 5 datasets:

- For city 1: one training set containing 1000 examples.

Algorithm 6.2 Slope Hill-Climbing Function

```
Input: model: model to reframe, training: labeled training dataset from original context, deploy: labeled little-size dataset from deployment context, alphas: set of slope parameters for affine reframing transformation, betas: set of intercept parameters for affine reframing transformation, i: index of numerical attribute to optimize slope for, adjust: evolution factor of the slope between two hill-climbing moves.
Output: localImprovement: boolean indicating if the hill-climbing process led to an improvement of the model performance.
localImprovement ← false
deployTransOrig ← deploy, deployTransOrig.AffineReframe(alphas, betas)
bestPerf ← model.Test(deployTransOrig)
direction ← 0
alphasOrig ← alphas
alphasDown ← alphasOrig, alphasDown[i] ← alphasDown[i]·$10^{-adjust}$)
perf ← TestPerformance(model, deploy, alphasDown, betas, alphas, betas)
if perf > bestPerf then
    direction ← -1, bestPerf ← perf
end if
alphasUp ← alphasOrig, alphasUp[i] ← alphasUp[i]·$10^{+adjust}$)
perf ← TestPerformance(model, deploy, alphasUp, betas, alphas, betas)
if perf > bestPerf then
    direction ← 1, bestPerf ← perf
end if
localImprovement ← (direction $\neq$ 0), localKeepGoing ← localImprovement
while localKeepGoing do
    speed ← 1, accelerate ← true
    while accelerate do
        alphasNext ← alphas, alphasNext[i] ← alphasNext[i]·$10^{direction \cdot adjust \cdot speed}$)
        perf ← TestPerformance(model, deploy, alphasNext, betas, alphas, betas)
        if perf > bestPerf then
            speed ←2 · speed, bestPerf ← perf
        else
            accelerate ← false
        end if
    end while
    localKeepGoing ← (speed > 1)
end while
return localImprovement
```

Algorithm 6.3 Intercept Hill-Climbing Function

```
Input: model: model to reframe, training: labeled training dataset from original context, deploy: labeled little-size dataset from deployment context, alphas: set of slope parameters for affine reframing transformation, betas: set of intercept parameters for affine reframing transformation, i: index of numerical attribute to optimize intercept for, adjust: evolution factor of the intercept between two hill-climbing moves, range: range of the numerical attribute.
Output: localImprovement: boolean indicating if the hill-climbing process led to an improvement of the model performance.
localImprovement ← false
deployTransOrig ← deploy, deployTransOrig.AffineReframe(alphas, betas)
bestPerf ← model.Test(deployTransOrig)
dir ← 0
betasOrig ← betas
betasDown ← betasOrig, betasDown[i] ← betasDown[i]$-adjust \cdot range$
perf ← TestPerformance(model, deploy, alphas, betasDown, alphas, betas)
if perf > bestPerf then
    dir ← -1, bestPerf ← perf
end if
betasUp ← betasOrig, betasUp[i] ← betasUp[i]$+adjust \cdot range$
perf ← TestPerformance(model, deploy, alphas, betasUp, alphas, betas)
if perf > bestPerf then
    dir ← 1, bestPerf ← perf
end if
localImprovement ← (dir $\neq$ 0), localKeepGoing ← localImprovement
while localKeepGoing do
    speed ← 1, accelerate ← true
    while accelerate do
        betasNext ← betas, betasNext[i] ← betasNext[i]$+dir \cdot adjust \cdot speed \cdot range$
        perf ← TestPerformance(model, deploy, alphas, betasNext, alphas, betas)
        if perf > bestPerf then
            speed ←2 · speed, bestPerf ← perf
        else
            accelerate ← false
        end if
    end while
    localKeepGoing ← (speed > 1)
end while
return localImprovement
```

- For both city 2 and 3: one deployment dataset containing 200 examples, and one

Algorithm 6.4 TestPerformance

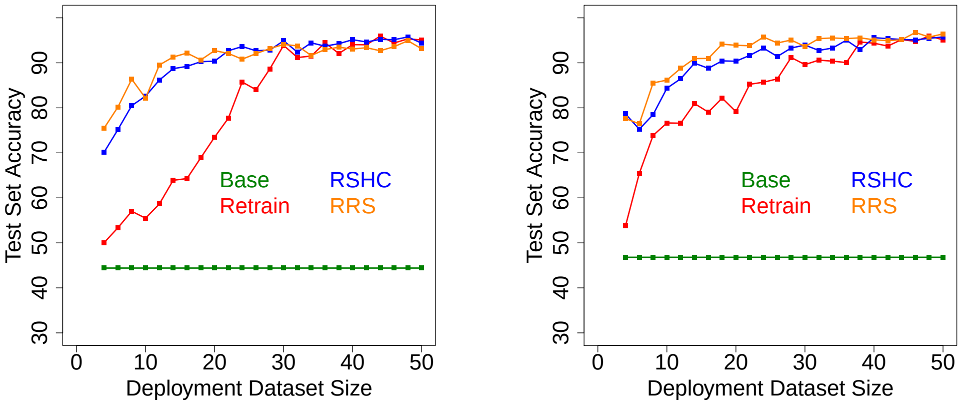
```
Input: model: model to reframe, deploy: labeled little-size dataset from deployment context, alphasCand: candidate set of slope parameters for affine reframing transformation, betasCand: candidate set of intercept parameters for affine reframing transformation, alphas: current set of slope parameters for affine reframing transformation, betas: current set of intercept parameters for affine reframing transformation, bestPerf: best performance achieved so far.
Output: localImprovement: whether the candidate parameters improved over the current parameters.
--------------------------------------------------------------------------------
deployTrans ← deploy
deployTrans.AffineReframe(alphasCand, betasCand)
perfCand ← model.Test(deployTrans)
localImprovement ← false
if perfCand.IsBetterThan(bestPerf) then
    bestPerf ← perfCand
    alphas ← alphasCand
    betas ← betasCand
    localImprovement ← true
end if
return localImprovement
```

Algorithm 6.5 Reframing with Randomized Search (RRS)

```
Input: cls: base classifier, training: labeled training dataset from original context, deploy: labeled little-size dataset from deployment context.
Output: bestAlphas (= $(\alpha_1, \cdots, \alpha_a)$): slopes of the affine reframing function, bestBetas (= $(\beta_1, \cdots, \beta_a)$): intercepts of the affine reframing function.
--------------------------------------------------------------------------------
model ← cls.Train(training)
bestPerf ← model.Test(deploy)
alphas ← [], betas ← []
InitParams(alphas, betas, training, deploy)
bestAlphas ← alphas, bestBetas ← betas
for count = 1 to 1000 do
    SampleParams(alphas, betas, training, deploy)
    TestPerformance(model, deploy, alphas, betas, bestAlphas, bestBetas, bestPerf)
end for
return (bestAlphas, bestBetas)
```

test dataset containing 1000 examples.

We want to measure the difference in predictive performance when the number of available deployment examples for either retraining or reframing varies. We consider a number n of deployment examples, which varies between 4 and 50 by step of 2. For each value of n, we draw 30 random samples of examples, each of size n, from the deployment dataset of size 200 of the considered deployment city. Each random sample is used to train a model (Retrain approach), or learn a reframing transformation associated to the model learned on the dataset from city 1 (RSHC and RRS approaches). The Base model is trained using only the dataset from city 1. Thus, its predictive performance does not depend on the deployment dataset from the target context, and its performance will be constant in all experiments. In the end, all models are evaluated by measuring accuracy on the test dataset of size 1000 from the considered deployment city. For each value of n, we report on Figure 6.6 the average test set accuracy of the 30 corresponding retrained/reframed models.



(a) City 2.

(b) City 3.

Figure 6.6 – Test set accuracy of learners with respect to the number of examples in the deployment dataset (average over 30 deployment datasets of the corresponding size).

The first observation is that, even with low amounts of labeled deployment data, reframing approaches make a significant improvement over both the base model and retraining a new model. Indeed, the test set accuracy in the context of city 2 of the base model learned in the original city 1 context is 44.4%, while with a deployment dataset containing only 4 examples, training a model using these 4 examples leads to a test set accuracy of 50.11% when RSHC and RRS find an input transformation leading to test set accuracy performances of 70.57% and 75.14% respectively. These results show that our algorithms fulfill their first goal of finding a useful reframing transformation of data improving over using no transformation at all.

The second observation is that the fewer labeled examples there are to retrain/reframe, the greater the difference in performance between retraining and reframing, to the advantage of the latter. When there are few labeled examples, using these examples to learn a transformation proves to be more efficient in terms of accuracy performance than using the examples to train a new model.

6.3.2 Comparison to GP-RFD on Real-World Data

We now evaluate our reframing algorithms with respect to GP-RFD, on both prediction accuracy and runtime performance. To do so, we follow the same experimental methodology as in (Moreno-Torres 2013). We take several binary classification tasks available on the UCI Machine Learning Repository (Lichman 2013). For each dataset, 120 random partitionings are generated. Each partitioning divides the dataset in 3: a deployment part with 8 examples, a testing part using half of the remaining instances, and a training part using the other half.

Then, we perform domain shift on the deployment and testing datasets. Domain shift consists in applying an affine transformation function to a numerical input attribute of the data. In the original methodology, the shift was only applied to one, randomly chosen, attribute. However, in a real-life setting, multiple attributes may be involved in the shift. Moreover, the attribute to shift is randomly chosen, thus a non-influential attribute may be shifted, in which case there is no need for adaptation. To overcome these limitations, we determined three influential numerical attributes in each dataset by learning a decision tree using the whole dataset. The three retained attributes are the ones used in splits closest to the root of the tree. Then, we define 4 levels of shift, labeled as Low, Medium, High and Extreme, depending on how they differ from the identity function, i.e. no shift at all, as defined in Table 6.1. The values of the slope for the shift function are taken between 0.1 and 10, by drawing from a uniform distribution the decimal logarithm of the slope. For a shift in the Low category, the slope will remain close to 1, while in the Extreme category it will be closer to 0.1 or 10. The value for the intercept is then drawn from a uniform distribution in an interval bounded by slope-related values.

Table 6.1 – Definition of levels of shift.

Level	Range of $\log_{10}(\alpha)$	Range of β
Low	$]-0.25;0.25[$	$]-\alpha\cdot\frac{max-min}{2};\alpha\cdot\frac{max-min}{2}[$
Medium	$]-0.5;-0.25]\cup[0.25;0.5[$	
High	$]-0.75;-0.5]\cup[0.5;0.75[$	
Extreme	$]-1;-0.75]\cup[0.75;1[$	

For each task, 30 partitionings are assigned to each category of shift. For each partitioning, an affine shift function is randomly determined in the associated category, and applied to the deployment and testing parts. Then, as in the previous evaluation, we compare the base model built using the training part, the retraining approach learning a model using the deployment part, and reframing-based approaches, i.e. RSHC, RRS and GP-RFD, which learn a model using the training part and look for a transformation using the deployment part. We take the respective duration of these processes as measures of runtime performance of the respective approaches. Prediction accuracy is measured on the testing part of the partitionings.

Table 6.2 shows the average accuracy percentage of the different methods for each dataset and shift category, along with the standard deviation of the accuracy over the 30 associated experiments. Table 6.3 shows the runtime of the algorithms. We observe that in all experiments, our RSHC and RRS algorithms outperform GP-RFD, both in terms of accuracy and runtime. The accuracy levels also show that our methods are more robust than GP-RFD, since the standard deviations of the accuracy are lower than for GP-RFD. The runtime performance is also a high advantage of our algorithms: when our algorithms learn a transformation in a tenth of second, GP-RFD takes several minutes, to achieve a lower prediction performance.

Table 6.2 – Accuracy results for different datasets and levels of shift.

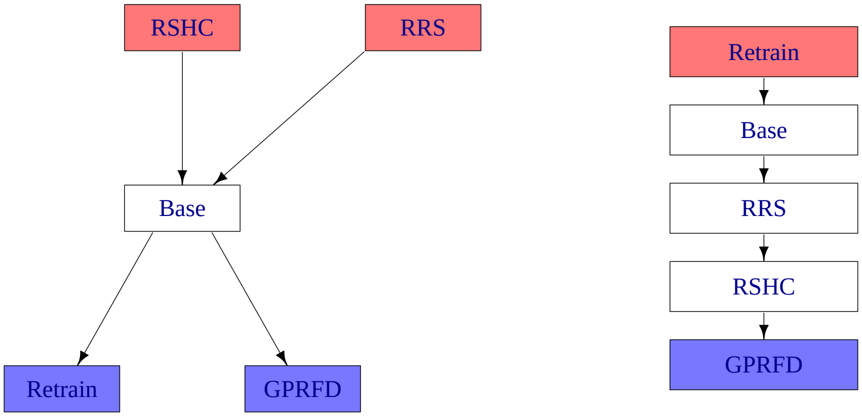
Dataset	Shift	Base	Retrain	RSHC	RRS	GPRFD
appendicitis	Low	79.73±13.67	79.32±7.71	80.88±6.6	80.34±6.03	67.89±20.29
	Medium	78.57±8.12	78.23±6.26	78.78±9.14	79.12±7.22	71.5±16.3
	High	78.5±13.61	78.64±7.52	78.64±9.23	81.22±8.83	74.01±15.83
	Extreme	73.88±20.75	76.87±10.66	80.34±10.08	79.59±7.84	74.69±14.25
breast-w	Low	81.07±21.09	82.43±8.85	91.07±5.78	90.63±7.03	84.62±11.64
	Medium	75±25.15	84.2±9.31	90.41±7.58	90.25±5.17	76.67±17.65
	High	78.96±16.33	79.99±14.99	92.14±4.15	88.79±10.73	83.25±15.88
	Extreme	70.51±21.89	81.8±10.41	90.88±3.43	90.85±4.72	81.87±14.56
bupa	Low	54.27±6.16	53.75±6.36	57.6±6.66	56.96±5.62	53.87±6.35
	Medium	54.76±5.54	50.67±6.57	56.35±6.38	58.35±6.05	52.6±6.36
	High	52.72±7.11	53.04±6.95	56.21±5.17	55.69±5.63	53.73±6.67
	Extreme	53.19±5.87	54.05±6.45	56.73±5.21	57.32±5.01	54.54±5.42
heart	Low	71.76±6.94	61.98±11.49	72.65±4.17	74.22±3.39	59.59±12.22
	Medium	73±6.88	62.6±11.42	71.93±7.73	72.72±7.62	64.05±11.15
	High	68.02±10.52	61.83±8.84	72.8±6.38	72.93±5.85	61.93±9.83
	Extreme	69.06±10.05	63.16±8.85	70.87±7.48	71.6±6.46	57.02±10.01
pima	Low	59.11±13.5	63.48±5.27	67.66±6.38	67.18±5.66	60.24±7.69
	Medium	58.62±14.43	60.12±9.16	68.01±5.76	68.22±4.9	60.89±7
	High	55.08±13.56	61.32±6.51	65.04±9.55	67.14±7.89	61.39±6.37
	Extreme	54.47±14.43	60.18±9.02	66.91±8	66±8.48	59.15±7.23

Table 6.3 – Time performance (in milliseconds) for different datasets and levels of shift.

Dataset	Base	Retrain	RSHC	RRS	GPRFD
appendicitis	0.36±0.48	0.04±0.2	49.84±10.09	22.83±3.76	9,413.24±1,464.22
breast-w	1.82±0.51	0.03±0.18	81.09±19.95	28.34±3.48	17,469.21±1,783.36
bupa	1.35±0.48	0.04±0.2	64.1±18.41	24.32±5.62	16,369.69±2,289.93
heart	1.32±0.93	0.09±0.5	39.86±37.99	24.68±13.15	22,925.67±3,406.43
pima	4.52±0.95	0.06±0.24	99.36±22.77	30±2.89	30,328.04±3,337.01

We also performed statistical tests to assess the significance of the results. As advised in (Demsar 2006), we performed pairwise comparisons of methods using a Friedman rank sum test with a level of confidence of 95%. The test always rejected the null hypothesis, i.e. at least two methods were significantly different. Therefore, we performed the Nemenyi posthoc test to determine the pairs of significantly different methods. The results of this test are illustrated in Figure 6.7, which graphs are to be read as follows. A first method is statistically significantly better than a second one if and only if there is a directed path in the graph from the first to the second. Therefore, a method with no edge going in is outperformed by no other method, and thus can be considered the best, they are indicated in red on the graphs. On the other hand, a method with no edge going out does not outperform any other method, they can be considered the worst and are indicated in blue.

Figure 6.7a shows the significance graph for prediction accuracy, and indicates that RSHC and RRS significantly outperform both the baselines and GP-RFD, while the two of them do not differ. Figure 6.7b shows the significance graph for runtime performance. As expected, the baselines which only train a model are faster than the reframing-based methods that learn a transformation in addition to the model. Nevertheless, we observe that both RRS and RSHC are significantly faster than GP-RFD, with an advantage to RRS over RSHC.



(a) Prediction accuracy.

(b) Runtime performance.

Figure 6.7 – Significance graphs on the real benchmarks.

6.4 Output Reframing for Regression

In this section, we apply our techniques for numerical attribute reframing to output attributes. When the output attribute is numerical, the task at hand is called a regression task. In this context, output reframing is useful when the distribution of the value to predict varies from one context to another. Indeed, a regression model trained in a first context can only output predictions that are consistent with the output values it was trained with, in other words its predictions will be in the range of the output values from training. If in a second context the output distribution differs from the first context in a way that the output values are not in the same range anymore, then the original model cannot be accurate. The idea behind “hard” output reframing is then to modify the prediction of the classifier trained in the first context so that the transformed predictions lie in the range of the output values from the second context.

6.4.1 Extension of Affine Transformation Optimization to Output Reframing

Formally, output reframing is about reusing model $M_1 : \mathbb{X} \mapsto \mathbb{Y}$ trained in context 1 for prediction in context 2, by learning a transformation $g : \mathbb{Y} \longrightarrow \mathbb{Y}$ of the output attribute, so that the model defined by $g \circ M_1$ is accurate in context 2. Formally:

$$\text{Context 1: } \hat{Y} = M_1(X)$$
$$\text{Context 2: } \hat{Y} = g(M_1(X))$$

Figure 6.8 summarizes this process: the model is built using the training set, and used along with the deployment set from the second context to find an output transformation. This reframing transformation is applied to test examples after the application of the model, whose prediction is modified by the transformation to give the actual prediction.

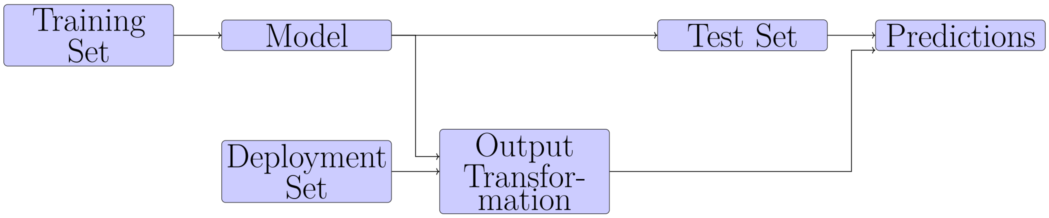


Figure 6.8 – Illustration of the output reframing process.

Our stochastic algorithms RSHC and RRS can be used to search for an affine tranformation of the numerical output Y. The difference with input reframing lies in the range to consider to find the parameters. In input reframing, we were looking for a transformation such that values from context 2 looked like values from context 1, in order to apply the model on values from its original training context. For output reframing, it is exactly the opposite process: the model outputs values in the range of context 1, and we want these values to look like values from context 2. Thus, the basic function has to be reverted to map values from context 1 to context 2:

$$\begin{aligned}
f_{Y,id}(y) &= \alpha_{Y,base} \cdot y + \beta_{Y,base} \\
&= \frac{u_{Y2} - l_{Y2}}{u_{Y1} - l_{Y1}} \cdot (y - l_{Y1}) + l_{Y2} \\
\alpha_{Y,base} &= \frac{u_{Y2} - l_{Y2}}{u_{Y1} - l_{Y1}} \\
\beta_{Y,base} &= l_{Y2} - \alpha_{Y,base} \cdot l_{Y1}
\end{aligned}$$

The sampling of the slope parameter is then performed the same way as for inputs, using this new base slope. The sampling of the intercept is then performed by inverting the roles of contexts 1 and 2 with respect to input reframing.

$$\begin{aligned}
\beta_{min} &= l_{Y2} - \alpha_{sample} \cdot u_{Y1} \\
\beta_{max} &= u_{Y2} - \alpha_{sample} \cdot l_{Y1}
\end{aligned}$$

Input and output reframing are not mutually exclusive: both types of shifts may occur at the same time, in which case there is a need for both types of reframing. Formally, if we denote by $f : \mathbb{X} \longrightarrow \mathbb{X}$ the input attribute vector reframing transformation, and $g : \mathbb{Y} \longrightarrow \mathbb{Y}$ the output attribute reframing transformation, we want the application $g \circ M_1 \circ f$ to be accurate in context 2, i.e.

$$\begin{aligned}
&\text{Context 1: } \hat{Y} = M_1(X) \\
&\text{Context 2: } \hat{Y} = g(M_1(f(X)))
\end{aligned}$$

6.4.2 Application to the Bike Sharing Dataset

We will consider as an example a real dataset that presents such shifts. The Bike Sharing dataset (Fanaee-T and Gama 2014) contains two years of daily records of bikes rented in Washington, D.C. The aim is to predict the count of bikes rented during one day with respect to weather conditions of the day and calendar information related to the day: the day of the week, whether it is a working day, or a holiday.

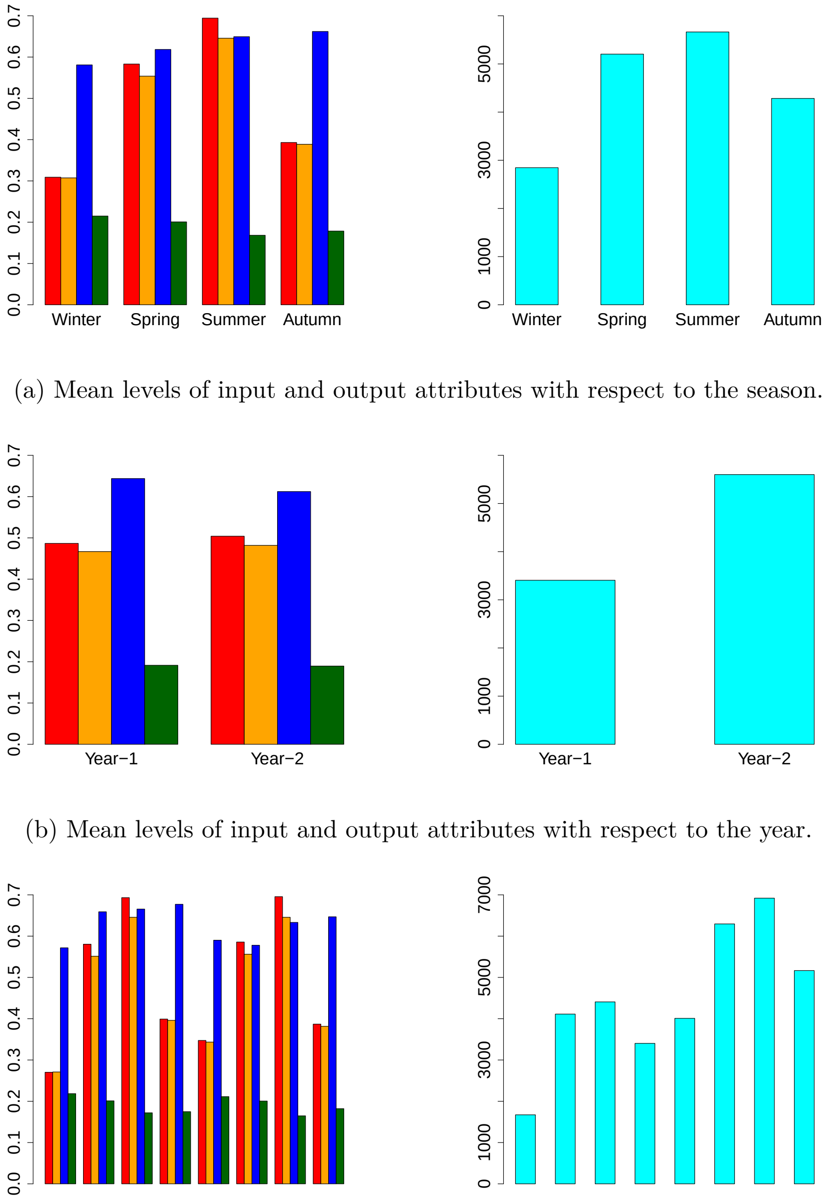
Two families of contexts can be defined on this dataset, both presenting shifts. These families are not mutually exclusive and can be combined to obtain a third family.

Firstly, contexts can be defined according to seasons. We refer to seasons as groups of 3 months each, which roughly correspond to the actual definition of seasons: "Winter" covers months of January, February and March, "Spring" covers April, May and June, "Summer" covers July, August and September, and "Autumn" covers October, November and December. Each season defines a context. The reframing task consists in learning a model to predict daily bike rental with data from one season, and to reuse this model on data from another season. Thus, we can imagine as many reframing tasks as combinations of a training context and a deployment context. Since we want both contexts to be different, there are 12 possibilities. Figure 6.9a shows histograms representing mean levels of the four input attributes (on the left), and of the output attribute (on the right), for each season. Since data cover a two-year period, seasonal values are averages over the two occurrences of each season. The distributions of the input attributes vary depending on the season, especially the temperature attributes, which seems trivial. Therefore, there is a need for input reframing in this family of contexts. From the output distribution, i.e. the average daily counts of rented bikes per season, it appears that these mean levels differ from a season to another, there are less bikes rented on average during winter than in summer, which also seems trivial. Therefore, there may also be a need for output reframing in this family of contexts.

The second family of contexts is defined according to the two years available in the dataset. Figure 6.9b shows the mean levels of the distributions of input and output attributes for the two years. We observe that the average count of bikes rented in the second year is much greater than the average count of the first year, increasing from 3400 bikes per day to 5600. On the other hand, mean levels of weather-related input attributes do not present much variation from one year to another. There are two possible tasks in this family of contexts: from the first year to the second, and the opposite.

Finally, we can combine the contexts of the 4 seasons and the 2 years to obtain 8 contexts, one for each season of a given year. This family, based on seasons and years, presents both types of distribution shift, an input shift because of season change, and an output shift because of both season and year change, as shown on Figure 6.9c. The contexts are ordered chronologically, from the autumn of the first year to the summer of the second year. Reframing tasks can be defined according to pairs of contexts, which gives 56 tasks in this family of contexts. However, we restrict ourselves to pairs of contexts that correspond neither to the same season nor to the same year. Thus, a context from a given season and a given year will be matched with all the other seasons for the other year. We assume that pairs of contexts with same season or year are special cases of the two previous families of contexts.

We perform an experimental comparison of our input and output reframing methods on tasks from this family of contexts. The “Base” model and “Retraining” are still our baselines. We include three variants of our RSHC and RRS algorithms: one using only input reframing (with suffix “-I” in the results, one only using output reframing (suffix “-O”), and one using both input and output reframing (suffix “-IO”). As a base model, we use the REPTree regression tree implementation of the Weka software. For every pair of contexts, the full dataset of the origin context was used to train the base model. The reframing transformation to adapt to the target context was optimized over a tenth of the full dataset associated to this context, while the rest was used for testing. For instance, in the Seasons family of contexts, each season covers a period of three months and occurs twice, once per year. Thus, the dataset associated to each context contains approximately 180 examples, from which we keep roughly 18 to find the reframing transformation. For each pair of contexts, 30 experiments were performed, each with a



(a) Mean levels of input and output attributes with respect to the season.

(b) Mean levels of input and output attributes with respect to the year.

(c) Mean levels of input and output attributes with respect to the season and the year.

Figure 6.9 – Mean levels of input (on the left, temperature in red, feel-like temperature in orange, humidity in blue, and wind speed in green) and output (on the right) attributes in the three families of contexts.

different, randomly defined, reframing set of 18 instances. The loss function, used for both evaluating the quality of a reframing transformation and to evaluate algorithms on the test set, was root mean squared error.

Table 6.4 shows, for each pair of contexts in all families of contexts, the average root mean squared error over the 30 experiments of the different methods, the last row indicating the average over all pairs of contexts. Figure 6.10 shows the significance graph of all compared methods. It appears that the best methods in all families of tasks are our affine reframing algorithms, transforming only the output. They perform better than the same algorithms using input reframing in addition to output reframing, which themselves perform better than reframing only inputs, the base model, and retraining. This suggests that, for this family of contexts, the need for output reframing is greater than the need for input reframing.

As could be observed in Part I, complex aggregates in the relational setting often involve numerical features, secondary attributes used in selection conditions and aggregated feature as well as the output of the aggregation function itself, since most aggregation functions are numerical. Thus, an extension of our reframing algorithms for numerical features to the relational setting through complex aggregates is presented in the next section.

6.5 Reframing in the Relational Setting

In this section, we extend the affine reframing of numerical input attributes to the relational setting. As a motivating example, let us consider an artificial dataset inspired by the urban block example, whose schema is shown in Figure 6.11.

The task is a binary classification task consisting in predicting the class of a block between two values: “true” or “false”. There is no predictive attribute in the main table, i.e. at the block level, and two numerical attributes in the secondary table: the area and perimeter of the buildings.

In this relational context, two kinds of context changes may occur:

- As in the attribute-value setting, the distribution of the attributes may change from one context to another. In this urban block example, the two contexts are two cities, with the size of the buildings differing between the cities, which will affect the distribution of the area and perimeter.

- A context change specific to the relational setting is the difference in cardinality, e.g. if there are more buildings per block in the second city than in the first.

In Part I, we focused on the introduction of complex aggregate features to deal with relational supervised learning. Numerical inputs occur at two levels in the complex aggregates: most aggregation functions output a numerical value, e.g. the count or the average of a numerical secondary attribute, and in the secondary objects selection part, where conditions on secondary numerical attributes can be used. Thus, it is meaningful to reframe at these two levels. Indeed, a change in the distribution of the secondary

Table 6.4 – Average root mean squared error of the different methods on tasks from the the three families of contexts.

Origin	Target	Base	Retrain	RSHC-I	RRS-I	RSHC-O	RRS-O	RSHC-IO	RRS-IO
Winter	Spring	1433	1577	1527	1511	1621	1462	1513	1534
Winter	Summer	1634	1684	1692	1669	1571	1571	1667	1616
Winter	Autumn	1766	1893	1823	1819	1666	1664	1771	1795
Spring	Winter	2466	1564	2456	2436	1608	1601	1629	1618
Spring	Summer	1595	1638	1661	1658	1540	1540	1561	1553
Spring	Autumn	2094	1918	1992	1987	1817	1812	1842	1834
Summer	Winter	3949	1659	2189	2215	1664	1662	1737	1729
Summer	Spring	1748	1686	1548	1577	1548	1543	1548	1567
Summer	Autumn	2620	1930	1852	1854	1817	1809	1910	1873
Autumn	Winter	1368	1577	1315	1327	1255	1256	1359	1363
Autumn	Spring	1418	1678	1553	1546	1524	1523	1536	1537
Autumn	Summer	1533	1599	1658	1639	1572	1572	1673	1656
	Average	1969	1700	1772	1770	1600	1585	1646	1640

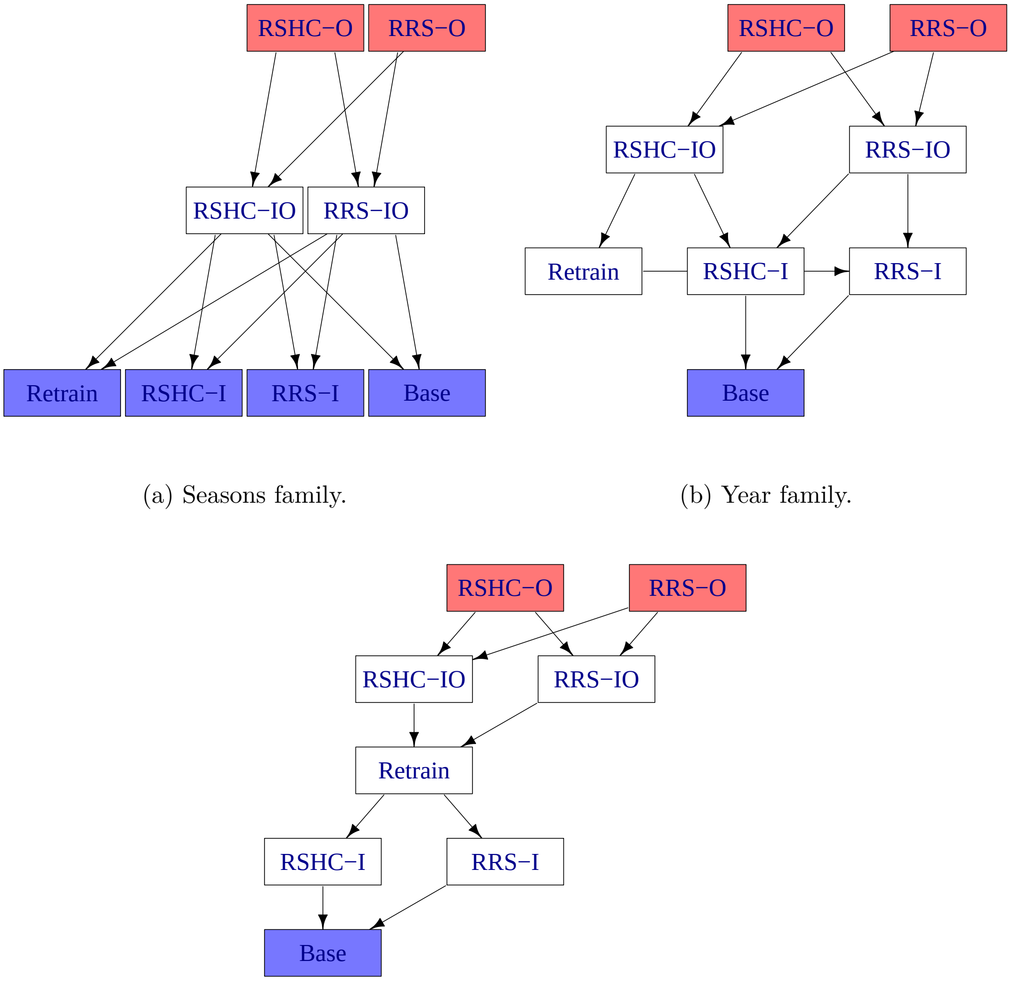
(a) Seasons family.

Origin	Target	Base	Retrain	RSHC-I	RRS-I	RSHC-O	RRS-O	RSHC-IO	RRS-IO
Year-1	Year-2	2294	1414	1953	1981	1182	1189	1249	1304
Year-2	Year-1	2171	1013	1077	1093	816	812	924	940
	Average	2232	1214	1515	1537	999	1000	1086	1122

(b) Years family.

Origin	Target	Base	Retrain	RSHC-I	RRS-I	RSHC-O	RRS-O	RSHC-IO	RRS-IO
Winter-1	Spring-2	3526	1235	3431	3430	1172	1165	1223	1147
Winter-1	Summer-2	3926	973	3963	3942	957	957	1120	975
Winter-1	Autumn-2	3361	2032	2840	2837	1503	1521	1766	1745
Spring-1	Winter-2	1661	1426	1126	1134	1497	1186	1183	1175
Spring-1	Summer-2	2386	976	2472	2463	1001	996	1140	1092
Spring-1	Autumn-2	2890	1995	1865	1873	1900	1841	1941	1829
Summer-1	Winter-2	1441	1433	1472	1464	1504	1492	1693	1594
Summer-1	Spring-2	1956	1242	1967	1985	973	980	1000	1031
Summer-1	Autumn-2	1938	2014	1992	2014	2042	2038	2138	2129
Autumn-1	Winter-2	1350	1395	1204	1206	1299	1278	1304	1321
Autumn-1	Spring-2	2166	1260	1954	1947	1111	1111	1092	1113
Autumn-1	Summer-2	2526	990	2602	2578	988	988	1051	1026
Winter-2	Spring-1	1675	1060	974	979	744	742	824	845
Winter-2	Summer-1	1701	885	1086	1088	881	881	973	943
Winter-2	Autumn-1	1608	1137	1091	1076	1106	1161	1098	1107
Spring-2	Winter-1	4477	627	4477	4477	644	640	643	638
Spring-2	Summer-1	1943	893	1943	1943	773	773	781	782
Spring-2	Autumn-1	2839	1132	2839	2839	1087	1089	1084	1074
Summer-2	Winter-1	5681	605	4537	4540	617	613	697	656
Summer-2	Spring-1	3066	1081	2268	2285	1082	1077	1183	1162
Summer-2	Autumn-1	3905	1149	2978	2963	1156	1150	1300	1244
Autumn-2	Winter-1	2265	587	1035	947	544	547	560	559
Autumn-2	Spring-1	2648	1085	1103	1154	1024	1008	913	883
Autumn-2	Summer-1	2518	865	1218	1210	860	859	938	927
	Average	2644	1170	2185	2182	1103	1087	1152	1125

(c) Seasons and Years family.

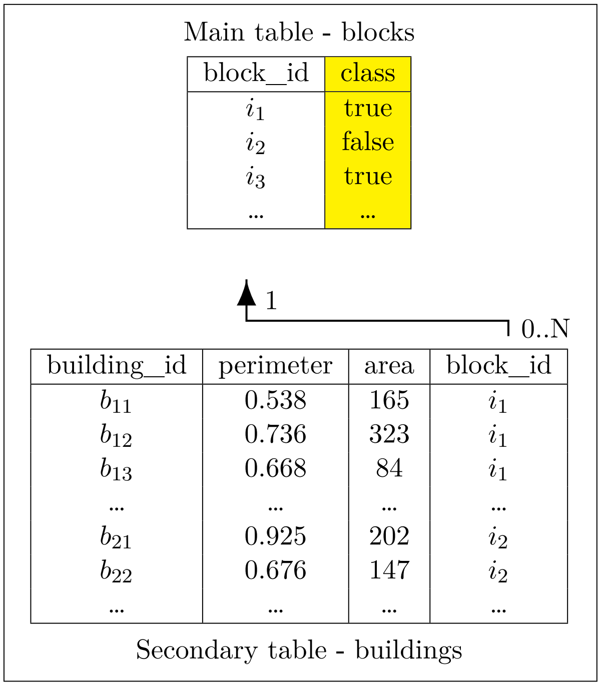


(a) Seasons family.

(b) Year family.

(c) Seasons and Years family.

Figure 6.10 – Significance graph of the different methods for the three families of contexts.



Main table - blocks

block_id	class
i_1	true
i_2	false
i_3	true
...	...

building_id	perimeter	area	block_id
b_{11}	0.538	165	i_1
b_{12}	0.736	323	i_1
b_{13}	0.668	84	i_1
...	...	...	...
b_{21}	0.925	202	i_2
b_{22}	0.676	147	i_2
...	...	...	...

Secondary table - buildings

Figure 6.11 – Schema of the simplified urban block dataset.

attributes will disturb the selection of secondary objects, since the values of the attribute in the deployment context may not match the values involved in the selection condition. Moreover, a change in cardinality will affect the output of the count function, e.g. if there are more buildings in the deployment city than in the training one, the condition learned on the count of buildings in the training context will not be accurate, since the overall number of buildings in the block is higher.

From the urban block example, we define 4 artificial context cities. The distributions of the secondary attributes, i.e. area and perimeter, and the cardinality, i.e. the average number of buildings per block, are shown in Table 6.5.

For instance, in the context of the first city, a dataset of 500 blocks has been generated. For each block, the corresponding number of buildings is chosen randomly following a geometric distribution of parameter 1/60, giving an average cardinality of 60 buildings per urban block. For each building generated, the area has been drawn randomly from a normal distribution with mean 200 and standard deviation 20. To obtain complex aggregate features, i.e. aggregates with a condition having an impact on the output value of the aggregate, we need a relationship between the two secondary attributes. Indeed, if area and perimeter are independent, the average value of the area of buildings does not depend on the buildings selected through a condition on the perimeter, thus there is no need for complex aggregation. This is the reason why for each building the perimeter is taken as the double of the area plus a random number drawn from a normal distribution with mean 40 and standard deviation 4.

Data for the second city has been generated following the same cardinality for the relationship between blocks and buildings, but the distributions of the secondary attributes changed so that the buildings are smaller, with a normal distribution of mean 100 and standard deviation 10 for the area, the perimeter being still the double of the area with the addition of a smaller normal random term of mean 20 and standard deviation 2. The conditions on complex aggregates have been chosen to adapt this change of distribution in area and perimeter: the thresholds on area and perimeter in selection conditions, and on their average value have been halved with respect to context 1.

In the third city, distributions of the secondary attributes are the same as in the first city, but the cardinality of the relationship between blocks and buildings changes, following a geometric distribution of parameter 1/30 so that there are on average half buildings per block compared to the first city.

Finally, the fourth city combines the changes of the second and third cities with respect to the first: the cardinality is the same as in the third city, and the distributions of the secondary attributes are the same as in the second city.

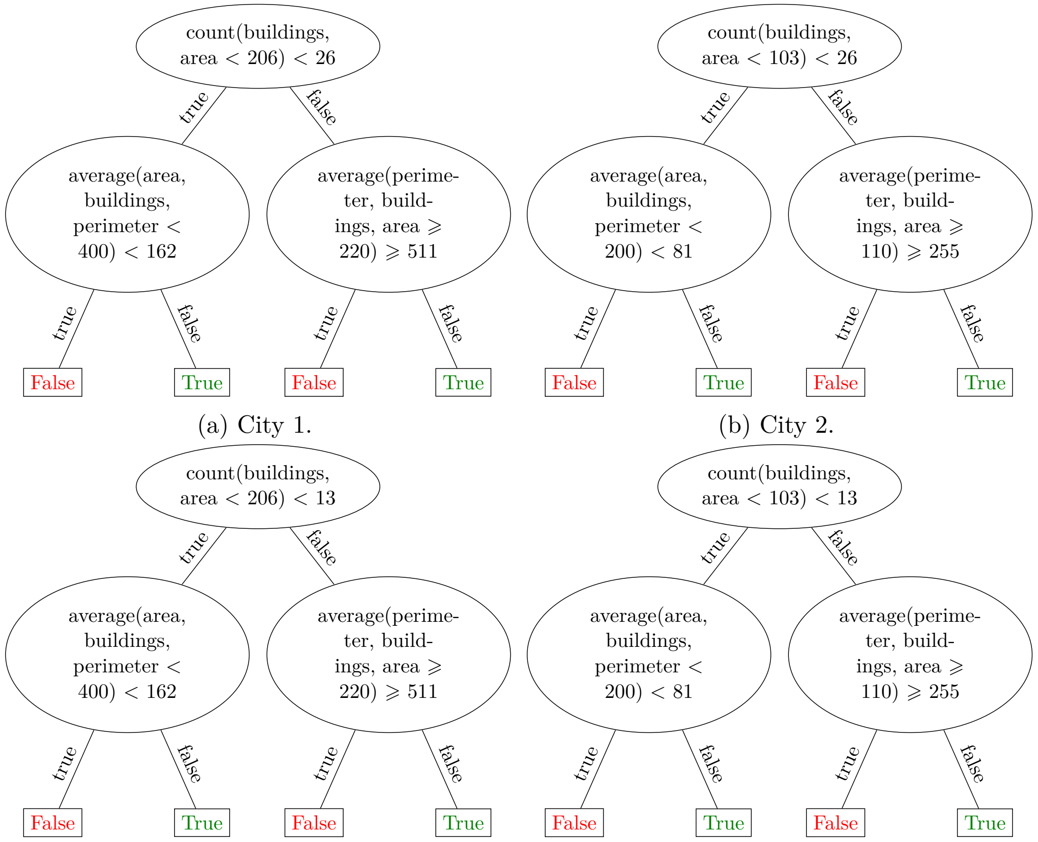
	City 1	City 2	City 3	City 4
Area	$\mathcal{N}(200, 20)$	$\mathcal{N}(100, 10)$	$\mathcal{N}(200, 20)$	$\mathcal{N}(100, 10)$
Perimeter	$2 \cdot area$ $+\mathcal{N}(40, 4)$	$2 \cdot area$ $+\mathcal{N}(20, 2)$	$2 \cdot area$ $+\mathcal{N}(40, 4)$	$2 \cdot area$ $+\mathcal{N}(20, 2)$
Cardinality	60	60	30	30

Table 6.5 – Description of the generation process of the artificial data in the four contexts.

The target models for the four context cities are shown in Figure 6.12. They all consist in two-level decision trees. The first split on a count-based aggregate is chosen so that data are split in a balanced way, i.e. approximately half of the examples follow each child branch. The second level splits have been chosen so that 90% of the examples follow one child branch and 10% the other child branch. The resulting child nodes are labeled with different classes. For instance, in Figure 6.12a, half of the examples will follow the left branch. Among this half, 90% will be labeled as "True" and 10 % as "False". In the right branch, we do the opposite: 90% of the examples will be labeled as "False", and 10% as "True".

The reframing task is to learn a classification model on a training set of 500 examples from the first city, and to reframe it to be able to use it in the three other cities. To do so, we adapt our stochastic reframing algorithms RSHC and RRS to reframe numerical features at two levels: both the secondary numerical attributes and the output of the aggregates will be reframed through affine transformations. For instance, if we consider the base model learned in the first city as given in Figure 6.12a, the following features will be reframed:

- The secondary numerical features: area and perimeter.
- At the root node: the output of the count aggregation function.


count(buildings, area < 206) < 26
true
false
average(area, buildings, perimeter < 400) < 162
average(perimeter, buildings, area ⩾ 220) ⩾ 511
true
false
true
false
False
True
False
True
count(buildings, area < 103) < 26
true
false
average(area, buildings, perimeter < 200) < 81
average(perimeter, buildings, area ⩾ 110) ⩾ 255
true
false
true
false
False
True
False
True
count(buildings, area < 206) < 13
true
false
average(area, buildings, perimeter < 400) < 162
average(perimeter, buildings, area ⩾ 220) ⩾ 511
true
false
true
false
False
True
False
True
count(buildings, area < 103) < 13
true
false
average(area, buildings, perimeter < 200) < 81
average(perimeter, buildings, area ⩾ 110) ⩾ 255
true
false
true
false
False
True
False
True

(a) City 1.

(b) City 2.

(c) City 3.

(d) City 4.

Figure 6.12 – Target models for the four contexts.

- In the left child node of the root: the output of the average area aggregation function.

- In the right child node of the root: the output of the average perimeter aggregation function.

This gives a total of 5 affine functions to optimize with our stochastic reframing algorithms, following the same procedures as described in Algorithms 6.1 and 6.5. However, since our reframing algorithms rely on the range of the numerical features to optimize the slope and intercept parameters, a difficulty will occur when the complex aggregate to be reframed is defined for too few examples, making impossible to evaluate the range. In particular, this will occur in the deployment context, in which we use few labeled examples to reframe. In this case, the range will be evaluated following a different strategy. As an example, we consider the impossibility to evaluate the range of the complex aggregate in the left branch of the model given in Figure 6.12a, i.e.

$average(area, buildings, perimeter < 400)$.

- If the range of the complex aggregate cannot be evaluated, we evaluate the range of the corresponding simple aggregate and consider it as the range of the complex aggregate. In our example, if $average(area, buildings, perimeter < 400)$ is undefined for all examples in the reframing set, because no example block contains buildings with $perimeter < 400$, we use the range of $average(area, buildings)$, which is the corresponding simple aggregate, i.e. the aggregate obtained by removing the selection condition.
- If the range of the simple aggregate cannot be evaluated either, for instance because all example blocks contain no building, we consider the complex aggregate feature has the same range as in the training context.

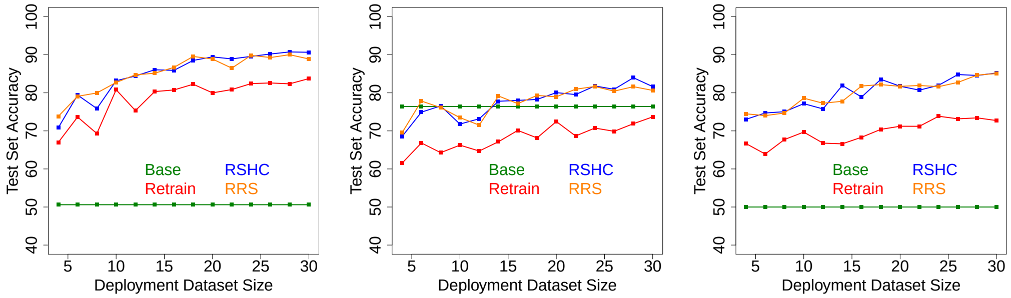
In the experimental comparison, we compare our RSHC and RRS algorithms to the base model learned in the original context, i.e. the first city, and the retraining approach using the few labeled data from the deployment context, i.e. the second, third or fourth city. For these three cities, two datasets have been generated: a test dataset containing 500 examples, and a reframing dataset containing 100 examples. We compare the accuracy performance of the different approaches on the test set with respect to the number of examples used to reframe/retrain the models. We vary this number of examples from 4 to 30. For each value, we perform 30 random draws of the appropriate number of examples from the 100 possible examples in the reframing dataset.

The average accuracy results over the 30 random draws for each reframing dataset size are shown in Figure 6.13. We observe that, for the three context changes, our stochastic reframing algorithms outperforms the retraining approach, especially when few labeled data is available, which supports the observations from Section 6.3. We notice that the base model is less sensitive to the change of cardinality than to the change of attribute distribution. Indeed, it still reaches a test set accuracy of 76.4% when trained in city 1 and deployed in city 3, when only the average number of buildings per block changes. In this case, our reframing approaches need more examples to adapt and outperform the base model. On the other hand, the performance of the base model when deployed in city 2, when only the attribute distribution changes, is around 50.6%, and reframing algorithms outperform it even with few examples to adapt.

6.6 Conclusion

In this chapter, we presented algorithms to perform input and output reframing of numerical features to adapt models to context changes. Our input and output reframing method consists in affine transformations of numerical features, whose parameters are chosen through stochastic optimization methods.

Our reframing algorithms outperform the retraining method when few labeled data is available to retrain/reframe a model. They also outperform the GP-RFD algorithms on real-life classification tasks in presence of dataset shift.



(a) City 2.

(b) City 3.

(c) City 4.

Figure 6.13 – Accuracy results of the four approaches with respect to the number of reframing/retraining examples in the three possible deployment contexts.

Numerical output reframing, i.e. in regression tasks, has been illustrated on the real-life bike sharing dataset, where natural input and output shifts occur.

Finally, the extension of reframing to the relational setting through the use of complex aggregates has been shown. This extension is natural since complex aggregate feature are usually numerical and can easily be reframed with our methods. This approach has been shown to be effective in presence of both distribution shift and cardinality change.

As a lead for future work, an objective is to propose similar algorithms to perform reframing of categorical features. Indeed, only numerical features have been considered in this Chapter. A large-scale application should also be considered to further assess the potential of our approaches. An example of such a real-world task, of which the bike sharing dataset is an example will be given in Chapter 7.

Chapter 7

Conclusions and Perspectives

In this chapter, we summarize the contributions and results of this thesis. We also present a potential domain of application for future work on both relational learning and reframing.

7.1 Contributions and Results

This thesis has dealt with machine learning tasks, namely learning predictive models from data. Specifically, we have tackled two aspects: firstly, we have focused on relational learning, i.e. predicting on one main kind of object, which is itself related to other secondary objects having their specific properties. This is opposed to attribute-value learning, where prediction is made on one kind of object with its individual set of properties. Thus, when attribute-value data are often represented in a single table with properties as columns and records of data as rows, relational data are represented across multiple tables linked by relationships. The second aspect we have tackled is reframing, i.e. reusing and adapting models through multiple contexts. A model is learned in one context, i.e. under specific conditions, and is adapted to perform well in a second context, where conditions such as data distribution may differ from the original context. This thesis has explored the difficulties related to these two aspects and has proposed learning algorithms adapted to these paradigms.

7.1.1 Relational Learning and Complex Aggregate Features

To perform relational learning, we have combined two common ideas used in this field: introduction of relevant secondary objects, as popularized by Inductive Logic Programming, and secondary objects aggregation, often used in propositionalization. These two ideas have been combined into complex aggregate features, i.e. aggregates of subsets of relevant secondary objects, that we presented in Chapter 3. The secondary objects to be aggregated are selected through conditions on their properties. The complex aggregates search space is too wide to be considered exhaustively. Thus, we have focused on the use of complex aggregates in a decision tree model, which is easy to read and allows to introduce a relevant complex aggregate for a given node of the tree, rather than generating the best complex aggregates for the whole set of examples. This introduction is made using non-exhaustive hill-climbing optimization algorithms, avoiding consideration of billions of features. However, the first algorithm we propose is still too slow for actual use on real-world data.

This is the reason why, in Chapter 4, we have proposed a stochastic optimization algorithm to explore the complex aggregates search space, based on random restart hill-climbing. This algorithm takes advantage of the structure of complex aggregates: the aggregation process on one hand, the selection condition on the other hand, to optimize the selection condition given the aggregation process. Nevertheless, this algorithm is suffering from overfitting, because of the introduction of too specific aggregates. Thus, it does not improve significantly over a model based on simple aggregates, such as Relaggs (Krogel and Wrobel 2003), with no selection condition. Moreover, the decision tree structure was also known to be sensitive to overfitting.

To overcome this double limitation, we have focused on the introduction of complex aggregates in a Random Forest model, i.e. a set of decision trees learned over different subsets of examples and features. This approach presents the advantage of introducing diversity in the model and thus to reduce overfitting. Based on its structure, we have proposed a straightforward subsampling of the complex aggregates search space, subsampling the number of aggregation processes on one hand, and the number of attributes to use in the selection condition on the other hand. In this context, we have introduced two simplified random hill-climbing algorithms, since the diversity induced by the Random Forest removes the necessity for very specific and accurate aggregates. These two solutions indeed have led to a performance increase, over both existential decision tree learner Tilde (Blockeel and De Raedt 1998) and its Random Forest, complex-aggregate-based extension Forf (Van Assche et al. 2006). These algorithms have been implemented as a Java software, called CARAF (Complex Aggregates within RAndom Forests). We have also presented the application of our Random Forest learner to complex aggregate feature selection.

Future improvement of Caraf may consider other stochastic optimization algorithms to perform the search for complex aggregates. In particular, genetic algorithms may be a good candidate. The use of non-tree-based models is also a lead for future work. Since most aggregates are numerical features, their inclusion in a regression model, such as logistic regression for classification or linear regression, may be interesting. The handling of nested relationships in data, introducing complex aggregates inside complex aggregates, will be necessary to handle complex data.

7.1.2 Adaptation to Context Change with Reframing

We have focused on two families of reframing methods. First, we have tackled a first kind of output reframing, which consists in modifying the interpretations of the "soft" predictions of the model to achieve possibly different "hard" predictions depending on the context. Then, we have dealt with input and "hard" output reframing, the former consists in transforming the inputs of the model, i.e. the predictive features, before applying the model, while the latter consists in transforming the output value of the model, i.e. the value it predicts, depending on the context.

In Chapter 5, we have introduced a multi-class cost-sensitive learning algorithm, based on pairwise binarization, i.e. the original multi-class task is divided into several binary classification tasks, one per pair of classes. To take into account misclassification costs, we have applied a thresholding approach on the scores returned by the individual binary models. These thresholds are optimized independently for each model, taking into account examples from all classes for each binary classifier, which considers only two classes. Experiments on real-world datasets have shown this gives a better predictive power to the approach, which outperforms state-of-the-art algorithms. This pairwise approach has been used to perform reframing and adapt to a change of misclassification costs between two contexts. This adaptation is achieved by using a few sample of labeled data from the target context to optimize the binary thresholds.

In Chapter 6, we have tackled input and output reframing of numeric features through the use of affine transformations. Parameters of the affine functions are optimized using stochastic optimization algorithms. For input reframing, the aim is to transform the values of the input attributes in the target context so that the transformed values look like values from the original training context, so that the original model is usable. These methods outperform a retraining approach with few labeled data available to reframe or retrain. They also perform better, in both accuracy performance and runtime, than the genetic programming-based algorithm GP-RFD (Moreno-Torres, Llorà, et al. 2013). Our algorithms are also able to deal with an output shift, i.e. a change of distribution in the numeric output attribute of a regression task. In this case, called output reframing, the role of the affine transformation is the opposite of its role in input reframing. The affine transformation is applied to the value predicted by the model, which originally looks like an output value from the training context, for it to look like a value from the target context. We have proposed an extension of these algorithms to achieve reframing in the relational setting, through the use of complex aggregates, where both numeric features of the secondary objects and outputs of the mostly numeric aggregation functions can be reframed.

Future work in reframing will consist in proposing similar methods to reframe categorical features, since we have only considered numerical inputs or outputs so far. A large-scale, real-world application task should also be identified to further assess the strength of our reframing approaches. This could be found in a spatio-temporal domain, which is of interest to apply both our relational data mining techniques and reframing algorithms.

7.2 Future Work: Learning on Spatio-Temporal Data

A potential application for both relational data mining and reframing is multi-dimensional data. In particular, temporal and spatial data can be represented naturally in the relational setting. Inspired by the bike sharing example dataset (Fanaee-T and Gama 2014) described in Section 6.4, let us consider a regression task where the aim is to predict hourly bike rental in a particular station. The schema of such a dataset and a few records are given in Table 7.1. The "Id" column identifies the record. The notation i_{ab} of the elements of this column reads as "record for station a at hour b". The "Hour" column represents a timestamp, the number of hours that passed since the origin of the records, it is not the hour of the day. Columns "X" and "Y" correspond to spatial coordinates of the station in an orthonormal basis, with one meter as unit. These two columns identify a station, i.e. the first three rows are hourly counts for a first station located at coordinates $(100, 100)$, while rows 4 to 6 are records for a second station located at $(500, 400)$, and rows 7 to 9 correspond to a third station at $(-300, 200)$. The "BikesRented" column is the count of bikes rented during the given hour at the given station. It is the target column, the one to be predicted. More input attributes could be used to predict this count, i.e. there could be more columns in this database, we omit them to focus on the transformation to the relational setting.

Table 7.1 – Hourly bike rental records per station.

Id	Hour	X	Y	BikesRented
i_{10}	0	100	100	10
i_{11}	1	100	100	9
i_{12}	2	100	100	13
...	...	...	...	...
i_{20}	0	500	400	15
i_{21}	1	500	400	8
i_{22}	2	500	400	12
...	...	...	...	...
i_{30}	0	-300	200	20
i_{31}	1	-300	200	15
i_{32}	2	-300	200	25
...	...	...	...	...

To predict the hourly count of bikes rented at a particular station, it is relevant to consider the history of bikes rented at the station, e.g. the average count of bikes rented on the previous hours, since it indicates the temporal dynamics of bike rentals. Similarly, it is also relevant to consider the history of bikes rented in the nearby stations, e.g. the average count of bikes rented during the previous hour in the stations in a 500-meter radius, since it shows the spatial dynamics of bikes rented.

Both temporal and spatial dynamics can be represented in the relational setting. Then, the main table is the one given in Table 7.1, and an association table defines a one-to-many relationship between each record of the main table and the records prior to it. The schema and part of the records of this association table are shown in Table 7.2. For each row of the association column, the "IdMain" column represents the Id of the record in the main table that stands for the main object, i.e. the *one* side of the one-to-many relationship. The "IdSec" column is the Id of the record from the main table that stands for the secondary object, i.e. the record related to the main one through the *many* side of the one-to-many relationship. This representation allows for the introduction of dimensional distances between records, in this example temporal and spatial distances. Indeed the “HourDist” column represents the temporal distance in hours between the records *IdMain* and *IdSec*, i.e. the number of hours that passed between *IdSec* and *IdMain*. Similarly, the “XDist” and “YDist” columns represent the spatial distance in meters according to the corresponding coordinate between the two stations associated to the records, i.e. the difference in the “X” and “Y” column respectively between the two records.

Table 7.2 – Spatio-temporal association table for bike rentals.

IdMain	IdSec	HourDist	XDist	YDist
i_{11}	i_{10}	1	0	0
i_{11}	i_{20}	1	400	300
i_{11}	i_{30}	1	-400	100
...	...	...	...	...
i_{12}	i_{10}	2	0	0
i_{12}	i_{11}	1	0	0
i_{12}	i_{20}	2	400	300
i_{12}	i_{21}	1	400	300
i_{12}	i_{30}	2	-400	100
i_{12}	i_{31}	1	-400	100
...	...	...	...	...
i_{22}	i_{10}	2	-400	-300
i_{22}	i_{11}	1	-400	-300
i_{22}	i_{20}	2	0	0
i_{22}	i_{21}	1	0	0
i_{22}	i_{30}	2	-800	-200
i_{22}	i_{31}	1	-800	-200
...	...	...	...	...

For instance, the three first rows in Table 7.2 indicate the relationship between the record in the first station for hour 1 to the records in the first three stations at hour 0. The temporal distance is 1, since the secondary records correspond to the hour directly before the main record. The spatial distance with the record corresponding to the same station one hour before is 0 for both “X” and “Y” coordinates. The first station is located at coordinates $(100, 100)$, while the second station is at $(500, 400)$, the spatial algebraic distance from station 1 to station 2 is then 400 in the “X” coordinate, and 300 in the “Y” coordinate, which corresponds to the second row. Similarly, station 3 is at $(-300, 200)$, so the algebraic distance from station 1 is $(-400, 100)$, which corresponds to the third row.

In the six next rows, the main object is the record for hour 2 in station 1. Hence, there are twice more records related to this record, since it can be linked to the records of hour 1. The temporal distance with records from hour 0 is now 2, while the distance with records from hour 1 is 1. In the last six rows, the main record is associated to station 2. The algebraic distance from station 2 to station 1 is the opposite of the distance from station 1 to station 2, it is $(-400, -300)$ in these records, where it was $(400, 300)$ in the first records.

In this context, complex aggregates constitute interesting features, since, for a given record associated to a station and timestamp, the role of the selection condition is to select relevant records, i.e. records of relevant stations at relevant timestamps. For instance, the following complex aggregate features can be constructed:

- $average(BikesRented, XDist = 0 \wedge YDist = 0 \wedge HourDist \leqslant 3)$: for a given station and timestamp, the average number of bikes rented in the same station for the past 3 hours.
- $median(BikesRented, XDist \in [-300; 300] \wedge YDist \in [-200; 200] \wedge HourDist \leqslant 5)$: for a given station and timestamp, the median number of bikes rented for the past 5 hours in the nearby stations, in a rectangle centered on the station and of size 600 meters in the “X” direction, and 400 meters in the “Y” direction.

Finally, there is a potential application for reframing in this context. We consider an history of records for stations in a given neighborhood of a city, on which we train a first prediction model of the number of bikes rented. Then, the bike rental system is extended to another neighborhood, where the spatial organization is different, for instance the distance between stations is higher than in the first neighborhood. The demand for bike rental may also be higher in this neighborhood. The stations in the second neighborhood are new, thus there is few history of bikes rented, i.e. not enough data to train a prediction model. The model trained in the first neighborhood can be reframed to adapt to the second neighborhood, to obtain an accurate model. The different spatial organization constitutes a shift in input attributes, since the distances between stations are higher, so there is a need for input reframing. Moreover, if the demand is higher, the number of bikes rented, i.e. the target for prediction, will be higher than in the original neighborhood. This constitutes a shift in the output attribute that can be solved with output reframing.

The need for predictive models in spatio-temporal tasks is increasing, with the amount of such data that are collected through the localization function of the growing number of smartphones, and intelligent sensors. In this context, the contributions of this thesis, in both relational learning and reframing, are a suitable solution.

Bibliography

Al-Otaibi, Reem, Ricardo B. C. Prudêncio, Meelis Kull, and Peter A. Flach (2015). "Versatile Decision Trees for Learning Over Multiple Contexts". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part I*. Ed. by Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, Carlos Soares, João Gama, and Alípio Jorge. Vol. 9284. Lecture Notes in Computer Science. Springer, pp. 184–199. ISBN: 978-3-319-23527-1. DOI: `10.1007/978-3-319-23528-8_12`.

Amores, Jaume (2013). "Multiple instance classification: Review, taxonomy and comparative study". In: *Artif. Intell.* 201, pp. 81–105. DOI: `10.1016/j.artint.2013.06.003`.

Anderson, Grant and Bernhard Pfahringer (2009). "Relational Random Forests Based on Random Relational Rules". In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier, pp. 986–991. URL: `http://ijcai.org/papers09/Papers/IJCAI09-167.pdf`.

Andrews, Stuart, Ioannis Tsochantaridis, and Thomas Hofmann (2002). "Support vector machines for multiple-instance learning". In: *Advances in neural information processing systems* 15, pp. 561–568.

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13, pp. 281–305. URL: `http://dl.acm.org/citation.cfm?id=2188395`.

Berka, Petr (2000). "Guide to the financial data set". In: *PKDD2000 discovery challenge*.

Bickel, Steffen, Michael Brückner, and Tobias Scheffer (2009). "Discriminative Learning Under Covariate Shift". In: *Journal of Machine Learning Research* 10, pp. 2137–2155. DOI: `10.1145/1577069.1755858`.

Blockeel, Hendrik and Luc De Raedt (1998). "Top-Down Induction of First-Order Logical Decision Trees". In: *Artif. Intell.* 101.1-2, pp. 285–297.

Boullé, Marc (2014). "Towards Automatic Feature Construction for Supervised Classification". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part I*. Ed. by Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo. Vol. 8724. Lecture Notes in Computer Science. Springer, pp. 181–196. ISBN: 978-3-662-44847-2. DOI: `10.1007/978-3-662-44848-9_12`.

Bourke, Chris, Kun Deng, Stephen D. Scott, Robert E. Schapire, and N. V. Vinodchandran (2008). “On reoptimizing multi-class classifiers”. In: *Machine Learning* 71.2-3, pp. 219–242. DOI: 10.1007/s10994-008-5056-8.

Brefeld, Ulf, Peter Geibel, and Fritz Wysotzki (2003). “Support Vector Machines with Example Dependent Costs”. In: *ECML*. Ed. by Nada Lavrac, Dragan Gamberger, Ljupco Todorovski, and Hendrik Blockeel. Vol. 2837. Lecture Notes in Computer Science. Springer, pp. 23–34. ISBN: 3-540-20121-1.

Breiman, Leo (2001). “Random Forests”. In: *Machine Learning* 45.1, pp. 5–32. DOI: 10.1023/A:1010933404324.

Breiman, Leo, J. H. Friedman, R. A. Olshen, and C. J. Stone (1984). *Classification and Regression Trees*. Wadsworth. ISBN: 0-534-98053-8.

Daumé, Hal, III and Daniel Marcu (2006). “Domain Adaptation for Statistical Classifiers”. In: *J. Artif. Intell. Res. (JAIR)* 26, pp. 101–126. DOI: 10.1613/jair.1872.

Demsar, Janez (2006). “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *Journal of Machine Learning Research* 7, pp. 1–30. URL: http://www.jmlr.org/papers/v7/demsar06a.html.

Dietterich, Thomas G. and Ghulum Bakiri (1995). “Solving Multiclass Learning Problems via Error-Correcting Output Codes”. In: *J. Artif. Intell. Res. (JAIR)* 2, pp. 263–286. DOI: 10.1613/jair.105.

Dietterich, Thomas G., Richard H. Lathrop, and Tomás Lozano-Pérez (1997). “Solving the Multiple Instance Problem with Axis-Parallel Rectangles”. In: *Artif. Intell.* 89.1-2, pp. 31–71. DOI: 10.1016/S0004-3702(96)00034-3.

Dinh, Quang-Thang, Christel Vrain, and Matthieu Exbrayat (2012). “A Link-Based Method for Propositionalization”. In: *Late Breaking Papers of the 22nd International Conference on Inductive Logic Programming, Dubrovnik, Croatia, September 17-19, 2012*. Ed. by Fabrizio Riguzzi and Filip Zelezný. Vol. 975. CEUR Workshop Proceedings. CEUR-WS.org, pp. 10–25. URL: http://ceur-ws.org/Vol-975/paper-01.pdf.

Dzeroski, Saso, Steffen Schulze-Kremer, Karsten R. Heidtke, Karsten Siems, Dietrich Wettschereck, and Hendrik Blockeel (1998). “Diterpene Structure Elucidation from 13CNMR Spectra with Inductive Logic Programming”. In: *Applied Artificial Intelligence* 12.5, pp. 363–383. DOI: 10.1080/088395198117686.

Elkan, Charles (2001). “The Foundations of Cost-Sensitive Learning”. In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*. Ed. by Bernhard Nebel. Morgan Kaufmann, pp. 973–978. ISBN: 1-55860-777-3.

Fanaee-T, Hadi and João Gama (2014). “Event labeling combining ensemble detectors and background knowledge”. In: *Progress in AI* 2.2-3, pp. 113–127. DOI: 10.1007/s13748-013-0040-3.

Fürnkranz, Johannes (2002). “Round Robin Classification”. In: *Journal of Machine Learning Research* 2, pp. 721–747. URL: http://www.jmlr.org/papers/v2/fuernkranz02a.html.

Getoor, Lise (2001). "Multi-relational data mining using probabilistic relational models: research summary". In: *Proceedings of the First Workshop in Multi-relational Data Mining*.

Gretton, Arthur, Alex Smola, Jiayuan Huang, Marcel Schmittfull, Karsten Borgwardt, and Bernhard Schölkopf (2009). "Covariate Shift by Kernel Mean Matching". In: Quionero-Candela, Joaquin, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence. *Dataset Shift in Machine Learning*. The MIT Press. ISBN: 9780262170055.

Hall, Mark A., Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten (2009). "The WEKA data mining software: an update". In: *SIGKDD Explorations* 11.1, pp. 10–18. DOI: `10.1145/1656274.1656278`.

Jelali, Soufiane El, Agnès Braud, and Nicolas Lachiche (2012). "Propositionalisation of Continuous Attributes beyond Simple Aggregation". In: *ILP*. Ed. by Fabrizio Riguzzi and Filip Zelezný. Vol. 7842. Lecture Notes in Computer Science. Springer, pp. 32–44. ISBN: 978-3-642-38811-8.

Knobbe, Arno J., Marc de Haas, and Arno Siebes (2001). "Propositionalisation and Aggregates". In: *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD 2001, Freiburg, Germany, September 3-5, 2001, Proceedings*. Ed. by Luc De Raedt and Arno Siebes. Vol. 2168. Lecture Notes in Computer Science. Springer, pp. 277–288. ISBN: 3-540-42534-9. DOI: `10.1007/3-540-44794-6_23`.

Krogel, M.-A. and S. Wrobel (2003). "Facets of Aggregation Approaches to Propositionalization". In: *Work-in-Progress Track at the Thirteenth International Conference on Inductive Logic Programming (ILP)*. Ed. by T. Horvath and A. Yamamoto.

Kuzelka, Ondrej and Filip Zelezný (2009). "Block-wise construction of acyclic relational features with monotone irreducibility and relevancy properties". In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, pp. 569–576. ISBN: 978-1-60558-516-1. DOI: `10.1145/1553374.1553448`.

Lachiche, Nicolas and Peter A. Flach (2003). "Improving Accuracy and Cost of Two-class and Multi-class Probabilistic Classifiers Using ROC Curves". In: *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*. Ed. by Tom Fawcett and Nina Mishra. AAAI Press, pp. 416–423. ISBN: 1-57735-189-4. URL: `http://www.aaai.org/Library/ICML/2003/icml03-056.php`.

Landgrebe, Thomas and Robert P. W. Duin (2007). "Approximating the multiclass ROC by pairwise analysis". In: *Pattern Recognition Letters* 28.13, pp. 1747–1758. DOI: `10.1016/j.patrec.2007.05.001`.

Lavrac, Nada and Saso Dzeroski (1994). *Inductive logic programming - techniques and applications*. Ellis Horwood series in artificial intelligence. Ellis Horwood. ISBN: 978-0-13-457870-5.

Lichman, M. (2013). *UCI Machine Learning Repository*. URL: `http://archive.ics.uci.edu/ml`.

Mitchell, Tom M. (1997). *Machine learning.* McGraw Hill series in computer science. McGraw-Hill. ISBN: 978-0-07-042807-2.

Moreno-Torres, José G. (2013). "Dataset shift in classification: Terminology, benchmarks and methods". PhD thesis. Editorial de la Universidad de Granada.

Moreno-Torres, José G., Xavier Llorà, David E. Goldberg, and Rohit Bhargava (2013). "Repairing fractures between data using genetic programming-based feature extraction: A case study in cancer diagnosis". In: *Inf. Sci.* 222, pp. 805–823. DOI: 10.1016/j.ins.2010.09.018.

Moreno-Torres, José G., Troy Raeder, Rocío Alaíz-Rodríguez, Nitesh V. Chawla, and Francisco Herrera (2012). "A unifying view on dataset shift in classification". In: *Pattern Recognition* 45.1, pp. 521–530. DOI: 10.1016/j.patcog.2011.06.019.

Muggleton, Stephen (1993). "Inverting Entailment and Progol". In: *Machine Intelligence 14, Proceedings of the Fourteenth Machine Intelligence Workshop, held at Hitachi Advanced Research Laboratories, Tokyo, Japan, November 1993.* Ed. by Koichi Furukawa, Donald Michie, and Stephen Muggleton. Oxford University Press, pp. 135–190.

Ontañón, Santiago and Enric Plaza (2015). "Refinement-based disintegration: An approach to re-representation in relational learning". In: *AI Commun.* 28.1, pp. 35–46. DOI: 10.3233/AIC-140621.

Ourston, Dirk and Raymond J. Mooney (1994). "Theory Refinement Combining Analytical and Empirical Methods". In: *Artif. Intell.* 66.2, pp. 273–309. DOI: 10.1016/0004-3702(94)90028-0.

Pan, Sinno Jialin and Qiang Yang (2010). "A Survey on Transfer Learning". In: *IEEE Trans. Knowl. Data Eng.* 22.10, pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

Perovsek, Matic, Anze Vavpetic, Janez Kranjc, Bojan Cestnik, and Nada Lavrac (2015). "Wordification: Propositionalization by unfolding relational data into bags of words". In: *Expert Syst. Appl.* 42.17-18, pp. 6442–6456. DOI: 10.1016/j.eswa.2015.04.017.

Puissant, Anne, Nicolas Lachiche, Grzegorz Skupinski, Agnès Braud, Julien Perret, and Annabelle Mas (2011). "Classification et évolution des tissus urbains à partir de données vectorielles". In: *Revue Internationale de Géomatique* 21.4, pp. 513–532.

Quinlan, J. Ross (1986). "Induction of Decision Trees". In: *Machine Learning* 1.1, pp. 81–106. DOI: 10.1023/A:1022643204877.

— (1990). "Learning Logical Definitions from Relations". In: *Machine Learning* 5, pp. 239–266. DOI: 10.1007/BF00117105.

— (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann. ISBN: 1-55860-238-0.

Srinivasan, Ashwin, Stephen Muggleton, Michael J. E. Sternberg, and Ross D. King (1996). "Theories for Mutagenicity: A Study in First-Order and Feature-Based Induction". In: *Artif. Intell.* 85.1-2, pp. 277–299. DOI: 10.1016/0004-3702(95)00122-0.

Sugiyama, Masashi, Matthias Krauledat, and Klaus-Robert Müller (2007). "Covariate Shift Adaptation by Importance Weighted Cross Validation". In: *Journal of Machine Learning Research* 8, pp. 985–1005. URL: http://dl.acm.org/citation.cfm?id=1390324.

Van Assche, Anneleen, Celine Vens, Hendrik Blockeel, and Saso Dzeroski (2006). “First order random forests: Learning relational classifiers with complex aggregates”. In: *Machine Learning* 64.1-3, pp. 149–182.

Vens, Celine (2007). “Complex aggregates in relational learning”. Blockeel, Hendrik (supervisor). PhD thesis. Informatics Section, Department of Computer Science, Faculty of Engineering Science. URL: `https://lirias.kuleuven.be/handle/1979/839`.

Vens, Celine, Jan Ramon, and Hendrik Blockeel (2006). “Refining Aggregate Conditions in Relational Learning”. In: *PKDD*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4213. Lecture Notes in Computer Science. Springer, pp. 383–394. ISBN: 3-540-45374-1.

Zadeh, Lotfi A. (1965). “Fuzzy Sets”. In: *Information and Control* 8.3, pp. 338–353. DOI: `10.1016/S0019-9958(65)90241-X`.

Associated Publications

This work has led to several publications in international conferences. They are listed below.

- On relational supervised learning aspects:
  - Preliminary considerations on complex aggregates:
    Clément Charnay, Nicolas Lachiche, and Agnès Braud (2013a). “Incremental Construction of Complex Aggregates: Counting over a Secondary Table”. In: *Late Breaking Papers of the 23rd International Conference on Inductive Logic Programming, Rio de Janeiro, Brazil, August 28th - to - 30th, 2013.* Ed. by Gerson Zaverucha, Vítor Santos Costa, and Aline Marins Paes. Vol. 1187. CEUR Workshop Proceedings. CEUR-WS.org, pp. 1–6. URL: `http://ceur-ws.org/Vol-1187/paper-02.pdf`
  - Random restart hill-climbing algorithm to search for complex aggregates in a decision tree model:
    Clément Charnay, Nicolas Lachiche, and Agnès Braud (2014). “Construction of Complex Aggregates with Random Restart Hill-Climbing”. In: *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers.* Ed. by Jesse Davis and Jan Ramon. Vol. 9046. Lecture Notes in Computer Science. Springer, pp. 49–61. ISBN: 978-3-319-23707-7. DOI: `10.1007/978-3-319-23708-4_4`
  - Random hill-climbing algorithms for complex aggregates introduction in a Random Forest Model:
    Clément Charnay, Nicolas Lachiche, and Agnès Braud. “CARAF: Complex Aggregates within Random Forests”. In: *25th International Conference on Inductive Logic Programming (ILP’15)*
  - Propositionalization approach using quantiles:
    Chowdhury Farhan Ahmed, Nicolas Lachiche, Clément Charnay, Soufiane El Jelali, and Agnès Braud (2015). “Flexible propositionalization of continuous attributes in relational data mining”. In: *Expert Systems with Applications* 42.21, pp. 7698–7709
- On reframing aspects:

  - Input and output reframing of numeric features through affine transformations:
    Chowdhury Farhan Ahmed, Clément Charnay, Nicolas Lachiche, and Agnès Braud (2014a). “Reframing Continuous Input Attributes”. In: *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, pp. 31–38. ISBN: 978-1-4799-6572-4. DOI: 10.1109/ICTAI.2014.16. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6979774
  - Multi-class cost-sensitive thresholds for Naive Bayesian scorers in one-versus-one binarisation:
    Clément Charnay, Nicolas Lachiche, and Agnès Braud (2013c). “Pairwise Optimization of Bayesian Classifiers for Multi-class Cost-Sensitive Learning”. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*. IEEE Computer Society, pp. 499–505. ISBN: 978-1-4799-2971-9. DOI: 10.1109/ICTAI.2013.80. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6734837
    Clément Charnay, Nicolas Lachiche, and Agnès Braud (2013b). “Optimisation par paires des classifieurs bayésiens dans le cadre d’un apprentissage sensible au coût”. In: *Conférence francophone sur l’Apprentissage (CAp)*. URL: http://icube-publis.unistra.fr/papr/2013/7-CLB13
  - Input reframing of numeric complex aggregates:
    Chowdhury Farhan Ahmed, Clément Charnay, Nicolas Lachiche, and Agnès Braud (2014b). “Reframing on Relational Data”. In: *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*. Ed. by Jesse Davis and Jan Ramon. Vol. 9046. Lecture Notes in Computer Science. Springer, pp. 1–15. ISBN: 978-3-319-23707-7. DOI: 10.1007/978-3-319-23708-4_1
  - Real-life example of dataset shift:
    Chowdhury Farhan Ahmed, Nicolas Lachiche, Clément Charnay, and Agnès Braud (2014). “Dataset Shift in a Real-Life Dataset”. In: *1st International Workshop on Learning over Multiple Contexts @ ECML (LMCE’14)*

- On relational representation of spatio-temporal data:
  Clément Charnay, Nicolas Lachiche, and Agnès Braud (2015). “Approche relationnelle de l’apprentissage de séquences”. In: *15èmes Journées Francophones Extraction et Gestion des Connaissances, EGC 2015, 27-30 Janvier 2015, Luxembourg*. Ed. by Benoît Otjacques, Jérôme Darmont, and Thomas Tamisier. Vol. E-28. Revue des Nouvelles Technologies de l’Information. Hermann-Éditions, pp. 481–482. URL: http://editions-rnti.fr/?inprocid=1002118

Amélioration de l'apprentissage supervisé par l'utilisation d'agrégats complexes et la prise en compte du contexte

1 Introduction

Cette thèse se place dans le cadre général de l'intelligence artificielle, domaine qui vise à créer des machines, ou programmes informatiques, « intelligents ». Les récentes avancées du domaine lui ont permis de recevoir une attention croissante de la part du grand public. Les applications sont vastes, de la robotique à la détection de fraude bancaire, ainsi que les systèmes de recommandation analysant les goûts des utilisateurs pour proposer des offres personnalisées. L'aide au diagnostic médical, visant à prédire une maladie chez un patient en fonction de ses symptômes, est une application de l'intelligence artificielle appartenant au sous-domaine qui nous intéresse plus précisément dans cette thèse : l'apprentissage automatique.

L'apprentissage automatique consiste à apprendre à un programme à effectuer une certaine tâche à partir de données, servant d'exemples pour l'apprentissage, dont la qualité est évaluée à l'aide d'une mesure de performance. Formellement, Mitchell donne la définition suivante (traduite de l'anglais) :

"Un programme apprend à partir d'une expérience E pour une certaine catégorie de tâches T suivant une mesure de performance P, si sa performance sur les tâches T, mesurée par P, augmente avec l'expérience E." (MITCHELL 1997)

Par exemple, un programme destiné à jouer aux échecs apprend lorsque sa capacité à gagner des parties augmente au fur et à mesure qu'il dispute des parties. Les tâches d'apprentissage automatique se divisent en deux grandes catégories :

L'apprentissage supervisé consiste à entraîner un modèle de prédiction : à partir de données dites étiquetées, représentées par une caractéristique cible, que le modèle doit apprendre à prédire, et des caractéristiques descriptives que le modèle va utiliser pour effectuer ses prédictions. Reprenant l'exemple d'aide au diagnostic, le programme d'apprentissage dispose d'une base de patients, avec les valeurs des caractéristiques vitales telles que la température interne ou la pression artérielle, ainsi que l'éventuelle maladie dont ils souffrent. Cette dernière information sera la caractéristique cible de la prédiction, qui va elle-même se baser sur les caractéristiques vitales.

L'apprentissage non-supervisé consiste à découvrir des catégories dans des données non étiquetées, i.e. les données rassemblent des caractéristiques, sans cible à prédire. La découverte de catégories se base alors sur les ressemblances, les proximités qui peuvent exister entre les données.

Nous nous intéresserons à l'apprentissage supervisé, i.e. à l'apprentissage de modèles de prédiction. Un modèle d'apprentissage est vu comme une fonction, reliant les caractéristiques descriptives servant à la prédiction en entrée, à la caractéristique cible de la prédiction en sortie. La tâche d'apprentissage est une tâche de classification lorsque la cible prend ses valeurs dans un ensemble fini et non-ordonné, i.e. des classes. On parle de tâche de régression lorsque la cible est numérique et ordonnée.

L'apprentissage du modèle s'effectue à partir de données, i.e. d'exemples associant des couples entrées/sortie. Nous prenons comme exemple le jeu de données post-opératoires, dont un sous-ensemble est donné en Tableau F.1. La tâche consiste à prédire, pour un patient sortant d'une opération chirurgicale, sa destination post-opératoire, en fonction de ses constantes vitales. La caractéristique à prédire, i.e. sa destination, correspond à la dernière colonne de la table, surlignée en jaune, qui peut prendre trois valeurs différentes : « h » s'il est gardé en observation à l'hôpital, « dom » s'il est renvoyé chez lui, et « si » s'il est envoyé en soins intensifs. Il s'agit donc d'un problème de classification, puisque la caractéristique cible prend ses valeurs dans un ensemble de 3 catégories. Les huit premières colonnes constituent les caractéristiques descriptives utilisées pour la prédiction, i.e. les constantes vitales. On y retrouve dans les trois premières colonnes les températures interne et externe du patient et sa pression artérielle. Les trois colonnes suivantes représentent la stabilité de ces trois facteurs. Enfin, on retrouve la saturation en oxygène et la sensation de confort du patient.

Tableau F.1 – Sous-ensemble du jeu de données post-opératoires.

INT	EXT	PA	INT-STBL	EXT-STBL	PA-STBL	O2	CONF	Dest
moyen	bas	moyen	stable	stable	stable	excellent	15	h
moyen	élévé	élévé	stable	stable	stable	excellent	10	dom
moyen	bas	moyen	stable	stable	instable	bon	10	si
moyen	moyen	moyen	moy-stable	stable	instable	excellent	15	h
moyen	moyen	moyen	instable	stable	instable	bon	10	dom
bas	moyen	élévé	instable	moy-stable	moy-stable	bon	10	si
moyen	élévé	bas	instable	stable	stable	bon	10	h
moyen	bas	moyen	stable	stable	instable	excellent	10	dom
...	...	...	...	...	...	...	...	...

Parmi les grandes familles de modèles de prédiction, nous utiliserons plus précisément des arbres de décision. Ces modèles présentent l'avantage d'être faciles à interpréter pour un utilisateur extérieur, grâce à leur représentation graphique. Un exemple d'arbre appris sur le jeu de données post-opératoires est donné en Figure F.1. L'apprentissage d'un arbre de décision s'effectue de la façon suivante : à partir des exemples d'entraînement à disposition, l'algorithme cherche une condition sur une caractéristique d'entrée qui sépare les données de façon à créer des partitions *pures* en termes de caractéristique cible. En d'autres termes, la séparation cherche à créer des partitions où les exemples d'une seule classe sont très majoritaires voire seuls, créant ainsi une partition pure. Cet algorithme d'apprentissage d'arbres de décision est le plus courant, popularisé par les systèmes CART (Breiman et al. 1984) et C4.5 (Quinlan 1993).

L'arbre de décision donné en exemple utilise comme première condition, à la racine de l'arbre, une séparation sur la valeur de la stabilité de la température interne du patient. Les exemples sont donc séparés suivant les trois valeurs possibles de cette caractéristique, et le processus de séparation recommence dans chaque branche, sur chacun des sous-ensembles d'exemples, jusqu'à ce que des sous-ensembles purs soit créés. Ces nœuds terminaux des arbres sont appelés des feuilles et sont étiquetés avec la classe représentée dans la partition pure correspondante. Ainsi, les patients ayant une température interne moyennement stable sont tous envoyés à l'hôpital, cela correspond à la branche gauche de la racine de l'arbre donné en exemple, qui est donc une feuille étiquetée « h », puisque tous les exemples d'entraînement correspondants sont de cette classe.

La prédiction de la destination d'un nouveau patient sortant d'opération s'effectue en suivant le chemin correspondant dans l'arbre : la stabilité de la température interne du patient est évaluée, et l'une des branches-filles est suivie en fonction du résultat. Si elle est moyennement stable, la branche de gauche est suivie, le nœud suivant est une feuille et l'arbre prédit que le patient doit rester en observation à l'hôpital. Si la température interne est stable, la branche du milieu est suivie, et mène à une séparation sur la pression artérielle. Suivant la valeur de cette caractéristique, la branche-fille correspondante est suivie, jusqu'à ce que ce qu'une feuille de l'arbre soit atteinte, qui donnera la prédiction de la destination du patient.

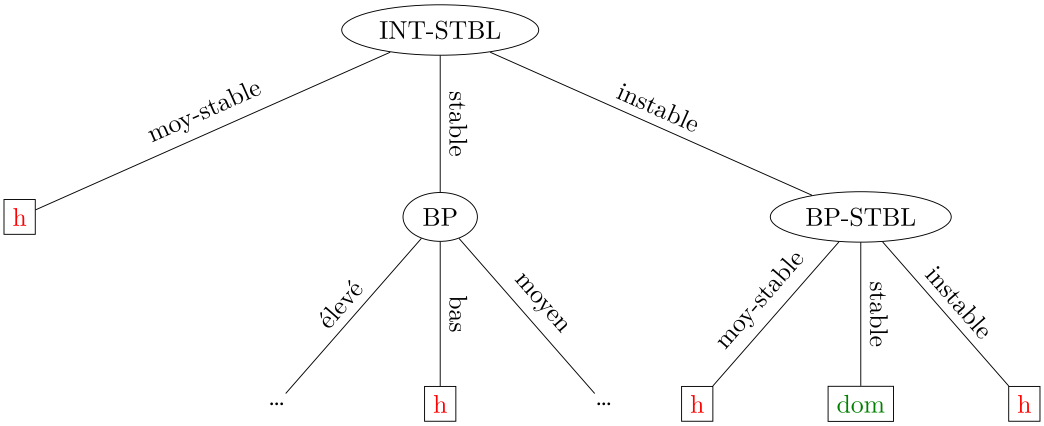


Figure F.1 – Début de l'arbre de décision appris sur le jeu de données post-opératoires.

Le jeu de données post-opératoires possède une représentation attribut-valeur : il n'est composé que d'une table, représentant une entité, un patient, avec ses caractéristiques propres. Ce type de données est le plus courant, et les grandes familles de modèles ont été développées pour apprendre à partir de cette représentation de données. Nous nous intéressons à une représentation de données réparties sur plusieurs tables, dite relationnelle. Cette représentation intervient lorsque plusieurs entités, liées par des relations, interviennent dans la tâche d'apprentissage. La tâche consiste à prédire la valeur d'une caractéristique de l'une des entités, à partir de ses caractéristiques propres mais également de celles des entités liées. Par exemple, considérons un problème de prédiction du potentiel de mutation d'une molécule, elle-même composée d'atomes. L'apprentissage se basera sur les propriétés physico-chimiques de la molécule, mais également sur celles des atomes qui la composent. La tâche fait donc intervenir deux entités, les molécules et les atomes, liées par une relation de composition.

Notre première tâche d'adaptation consistera donc à étudier une nouvelle extension des arbres de décision à une représentation relationnelle, basée sur des propriétés appelées agrégats complexes. Le nombre d'agrégats complexes constructibles pour une tâche donnée, i.e. le nombre de caractéristiques utilisables pour la prédiction, est trop élevé pour que tous soient considérés. Notre objectif est donc de mettre au point des heuristiques pour identifier des agrégats complexes pertinents pour séparer les données, sans les considérer exhaustivement.

Notre deuxième tâche d'adaptation concernera la réutilisation de modèles de prédiction entre différents contextes. Par exemple, nous apprenons un modèle de prédiction des ventes de glaces à Marseille, en se basant sur la température. Nous voulons maintenant appliquer ce modèle pour prédire les ventes de glace à Lille, plutôt que d'apprendre un nouveau modèle spécifique à Lille, par manque de données d'entraînement. Le climat de Lille étant différent de celui de Marseille au niveau des températures, il est certain que le modèle appris à Marseille ne sera pas fiable tel quel à Lille. Notre objectif sera donc de proposer des méthodes d'adaptation pour prendre en compte le changement de contexte, ici le changement de ville, et faire en sorte que le modèle d'origine soit utilisable dans le nouveau contexte, soit en le modifiant légèrement suivant une procédure peu coûteuse, soit en adaptant les données.

2 Apprentissage relationnel

L'apprentissage relationnel est un sous-domaine de l'apprentissage automatique où les données ne sont pas représentées sous le format attribut-valeur classique. Dans ce dernier format, dont l'exemple du jeu de données post-opératoires a été donné en Tableau F.1, chaque ligne d'une unique table représente un exemple d'apprentissage, et dont les colonnes représentent les caractéristiques de ces exemples, dont la cible de la prédiction. Dans le domaine relationnel, les données sont représentées par plusieurs tables représentant les différentes entités intervenant dans la tâche de prédiction. Une table, appelée table principale, correspond à l'entité principale, i.e. celle qui possède la caractéristique cible de la prédiction. Les autres tables, appelées tables secondaires, correspondent aux entités liées à l'entité principale par des relations.

Nous donnons un exemple dans le domaine d'application de la géographie : le jeu de données des ilots urbains représenté en Figure F.2. La tâche de prédiction porte ici sur le rôle d'un ilot urbain, i.e. un ensemble de bâtiments. Le but est de prédire si un ilot, l'objet principal, correspond à de l'habitat collectif, par exemple des immeubles, de l'habitat individuel, avec des maisons en lotissement, un mélange de ces deux catégories, une zone industrielle, ... Six catégories ont été définies, il s'agit donc d'une tâche de classification. La prédiction s'effectue à partir des caractéristiques géométriques de l'ilot, mais également à partir de celles des bâtiments qui le composent. Ainsi la table des ilots que nous montrons possède 6 colonnes : la première identifie un ilot et ne sert pas à la prédiction, les quatre suivantes correspondent aux caractéristiques géométriques de l'ilot, qui vont servir à la prédiction de la valeur de la dernière colonne, surlignée en jaune, qui correspond à la catégorie de l'ilot. Le sous-ensemble que nous montrons contient trois ilots, identifiés comme i_1, i_2 et i_3.

La table des bâtiments, les objets secondaires, est composée de 5 colonnes : la première identifie le bâtiment et n'est donc pas utile non plus à la prédiction, les trois suivantes correspondent aux caractéristiques géométriques du bâtiment. Enfin, la dernière colonne relie le bâtiment à l'ilot dont il fait partie : ainsi les bâtiments b_{11}, b_{12} et b_{13} appartiennent à l'ilot i_1, tandis que les bâtiments b_{21} et b_{22} appartiennent à l'ilot i_2.

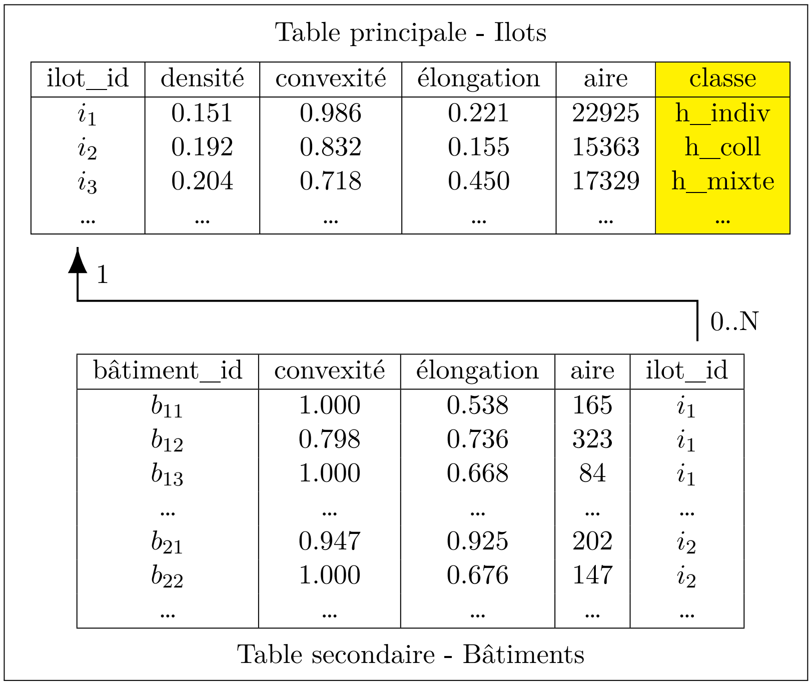


Table principale - Ilots

ilot_id	densité	convexité	élongation	aire	classe
i_1	0.151	0.986	0.221	22925	h_indiv
i_2	0.192	0.832	0.155	15363	h_coll
i_3	0.204	0.718	0.450	17329	h_mixte
...	...	...	...	...	...

bâtiment_id	convexité	élongation	aire	ilot_id
b_{11}	1.000	0.538	165	i_1
b_{12}	0.798	0.736	323	i_1
b_{13}	1.000	0.668	84	i_1
...	...	...	...	...
b_{21}	0.947	0.925	202	i_2
b_{22}	1.000	0.676	147	i_2
...	...	...	...	...

Table secondaire - Bâtiments

Figure F.2 – Schéma du jeu de données des ilots urbains.

Il s'agit maintenant de tirer parti de cette relation de composition, et d'utiliser les propriétés des bâtiments composant l'ilot pour effectuer la prédiction de la classe de l'ilot.

La principale difficulté réside dans la manière de gérer cette relation de composition : diverses méthodes de construction de propriétés de l'ilot à partir de celles des bâtiments peuvent être utilisées. Nous nous focaliserons plus précisément sur l'utilisation d'agrégats complexes.

2.1 Méthodes d'apprentissage relationnel et agrégats complexes

En apprentissage relationnel, il s'agit d'utiliser les propriétés des entités secondaires pour l'apprentissage sur l'entité principale. Il existe dans l'état de l'art deux grandes familles de méthodes.

La première famille est liée à la Programmation Logique Inductive (PLI) (Lavrac et Dzeroski 1994). Dans cette catégorie, les méthodes d'apprentissage reposent essentiellement sur l'introduction par le quantificateur existentiel d'objets secondaires liés à l'objet principal vérifiant certaines conditions. Ceci permet l'introduction de règles logiques telles que « *Si l'ilot contient au moins un bâtiment d'aire supérieure à 500 m^2, alors il s'agit d'un ilot d'habitat collectif* ». L'algorithme auquel nous nous intéresserons plus particulièrement dans cette famille est Tilde (Blockeel et De Raedt 1998). Il s'agit d'une extension de l'algorithme d'apprentissage d'arbres de décision C4.5 aux données relationnelles, basé sur l'introduction d'objets secondaires pertinents par le quantificateur existentiel, qui donne lieu à des propriétés telles que celle exposée plus haut. La puissance du formalisme de la PLI permet un raisonnement avancé. Néanmoins, la principale limite de ce type d'approches réside dans le fait qu'elles ne prennent pas en compte la cardinalité des relations, i.e. elles ne peuvent pas introduire de règles telles que « *Si l'ilot contient au moins **deux** bâtiments d'aire supérieure à 500 m^2, alors il s'agit d'un ilot d'habitat collectif* », ou plus généralement « *Si l'ilot contient au moins **n** bâtiments d'aire supérieure à 500 m^2, alors il s'agit d'un ilot d'habitat collectif* », avec n un nombre à déterminer. Notre premier objectif, par l'introduction d'agrégats complexes, sera de dépasser cette limite.

La deuxième grande famille d'algorithmes d'apprentissage relationnel est la propositionalisation, les méthodes de cette famille transforment la représentation relationnelle des données en une représentation attribut-valeur. Autrement dit, elles ramènent le schéma relationnel multi-tables à une seule table, correspondant à l'entité principale, mais enrichie en propriétés construites à partir des entités secondaires. Cette transformation présente l'avantage de permettre d'utiliser des algorithmes d'apprentissage attribut-valeur classiques, qui sont bien plus nombreux que les algorithmes d'apprentissage purement relationnels. Dans cette famille, certaines méthodes sont basées sur la construction de propriétés dites d'agrégation. L'agrégation consiste à appliquer une fonction à un ensemble, pour le résumer à une seule valeur. Ceci permet de créer des propriétés telles que « *le nombre de bâtiments de l'ilot* », ou « *la moyenne de l'aire des bâtiments de l'ilot* ». L'algorithme RELAGGS (Krogel et Wrobel 2003) est basé sur la construction de tels agrégats. Contrairement aux méthodes basées sur la PLI, les algorithmes basés sur l'agrégation tiennent compte de la cardinalité et des propriétés d'ensemble des objets secondaires liés à un objet principal. Néanmoins, elles perdent la possibilité de se focaliser sur les objets secondaires pertinents, qui était une idée de base des méthodes basées sur la PLI.

Nous nous intéressons donc à la construction d'agrégats complexes, qui combinent les idées des deux grandes familles de méthodes d'apprentissage relationnel : l'introduction d'objets secondaires pertinents d'une part, et l'utilisation des dynamiques d'ensemble permise par l'agrégation d'autre part. Les agrégats complexes consistent à agréger un sous-ensemble des objets secondaires liés à un objet principal, défini par une condition. Ceci permet de créer des propriétés telles que « *le nombre de bâtiments de l'ilot ayant une aire supérieure à 500 m^2* », ou « *la moyenne de l'aire des bâtiments de l'ilot ayant une élongation inférieure à 0.7* ». On peut ainsi capturer les dynamiques d'ensemble d'objets secondaires pertinents. Cependant, la sélection d'objets secondaires pertinents, par des conditions sur leurs propriétés, donne un nombre extrêmement élevé d'agrégats complexes à considérer pour une tâche donnée. Ainsi, le jeu de données des ilots urbains contient environ 7 500 bâtiments. Si l'on considère que les valeurs numériques de chaque propriété sont différentes pour chaque bâtiment, on peut créer au moins 15 000 conditions possibles sur chacune des trois propriétés. La condition de sélection pouvant être composée de conditions pour plusieurs attributs, on obtient environ 3 000 milliards de conditions de sélection, qu'il faut multiplier par le nombre de processus d'agrégation, i.e. d'associations d'une fonction et d'une éventuelle propriété à agréger, utilisables. Si l'on considère la fonction de comptage, et les fonctions numériques usuelles que sont la moyenne, le minimum et le maximum pour chacune des trois propriétés numériques des bâtiments, on obtient 10 processus d'agrégation, et plus de 33 000 milliards d'agrégats complexes possibles. Il paraît évident qu'il est impossible de tous les considérer. Notre objectif sera donc de proposer des heuristiques d'exploration non exhaustive de l'espace de recherche des agrégats complexes, afin d'identifier les plus pertinents sans tous les tester.

Nous introduirons les agrégats complexes dans le cadre d'un modèle d'arbre de décision. À chaque nœud de l'arbre, l'heuristique cherchera une condition sur un agrégat complexe pertinente pour séparer les données. Un exemple d'un tel arbre de décision est donné en Figure F.3.

La première heuristique proposée pour générer les agrégats complexes est un algorithme de hill-climbing. Dans ce cadre, partant d'un agrégat complexe, il s'agit de lui appliquer une « légère modification » et d'observer si l'agrégat complexe ainsi créé permet de mieux séparer les données. À partir d'un agrégat, plusieurs modifications sont essayées, qui forment le voisinage de l'agrégat. Si un ou plusieurs voisins permettent de mieux séparer les données que l'agrégat d'origine, la recherche reprend à partir du meilleur voisin, dont le voisinage est alors testé. Si aucun voisin ne sépare mieux les données, l'agrégat d'origine est considéré comme le meilleur possible et est utilisé dans l'arbre de décision.

La notion de voisinage est définie de la façon suivante, à partir d'un agrégat complexe dont la condition de sélection est une conjonction de n conditions simples sur des propriétés des objets secondaires, le voisinage rassemble :

- à processus d'agrégation constant, la conjonction peut devenir

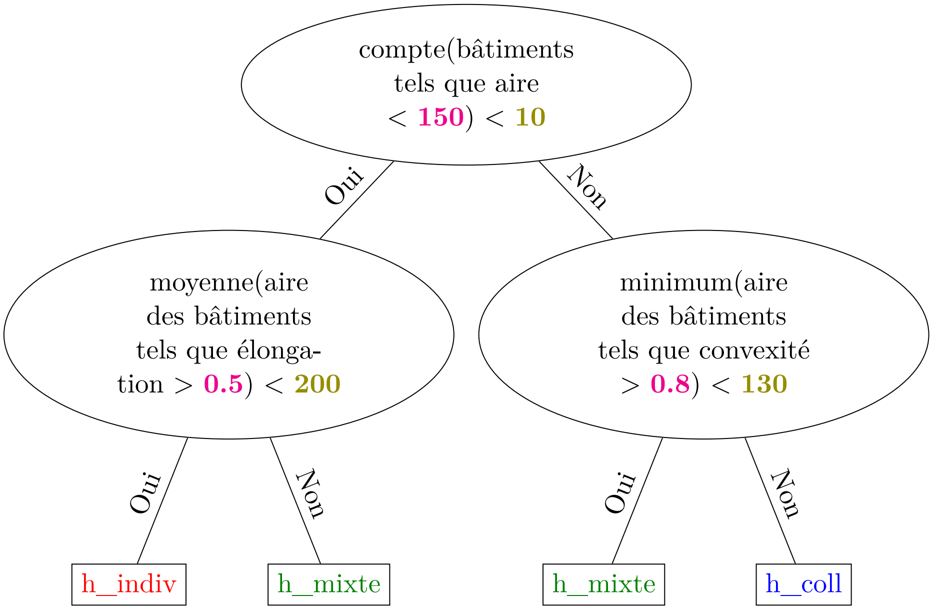

compte(bâtiments tels que aire < 150) < 10
Oui
Non
moyenne(aire des bâtiments tels que élongation > 0.5) < 200
minimum(aire des bâtiments tels que convexité > 0.8) < 130
Oui
Non
Oui
Non
h_indiv
h_mixte
h_mixte
h_coll

FIGURE F.3 – Exemple d'arbre de décision utilisant des agrégats complexes.

  - une conjonction de $n+1$ conditions qui rajoute une condition sur une propriété non déjà présente dans la conjonction actuelle,
  - une conjonction de n conditions qui modifie l'une des n conditions basiques : sur une propriété numérique, cela prend la forme d'un changement de seuil de comparaison.
  - une conjonction de $n-1$ conditions qui enlève une condition basique de la conjonction.

- à conjonction de conditions constante, le processus d'agrégation peut être modifié parmi toutes les possibilités.

Le point de départ de l'algorithme est un agrégat avec la fonction *comptage* comme processus d'agrégation et une conjonction vide, qui sélectionne donc tous les objets secondaires.

Néanmoins, cette méthode est encore trop exhaustive pour les problèmes à haute dimension, i.e. avec beaucoup de propriétés à agréger et sur lesquelles mettre des conditions. Le prochain objectif est donc de réduire l'espace de recherche tout en gardant une bonne performance prédictive.

2.2 Hill-climbing stochastique et forêts d'arbres décisionnels

La deuxième heuristique que nous proposons est donc basée sur un hill-climbing stochastique, i.e. incluant une part d'aléatoire. La notion de voisinage est toujours présente :

à partir d'un agrégat de départ, l'algorithme recherche de meilleurs voisins et recommence jusqu'à ne plus en trouver. Cependant, si aucun voisin ne permet d'améliorer l'agrégat d'origine, la recherche recommence du début, à partir d'un nouvel agrégat généré aléatoirement. Ce processus de *random restart* hill-climbing ne s'effectue que sur la conjonction de conditions, le processus d'agrégation étant fixé. Il y a donc autant de processus de hill-climbing que de processus d'agrégation possibles. À chaque itération de l'algorithme, la main est donnée à un processus tiré au hasard, les meilleurs processus ayant une probabilité plus importante d'être sélectionnés. Le processus choisi avance d'un pas de hill-climbing, i.e. il teste le voisinage de son agrégat courant, le met à jour en cas d'amélioration, ou réinitialise aléatoirement la condition dans le cas contraire.

La notion de voisinage a également été réduite : comme seule la conjonction de conditions peut être modifiée, l'ajout d'une condition sur un attribut est limité à une condition choisie au hasard par attribut, alors que toutes les possibilités étaient testées auparavant. De la même manière, la modification d'une condition existante est limitée à un certain nombre de voisins. Dans le cas d'une condition sur une propriété numérique, qui consiste en l'appartenance à un intervalle, seules de légères augmentation ou baisse des deux bornes de l'intervalle sont testées, soit quatre modifications possibles. L'algorithme de hill-climbing lié à un processus d'agrégation, i.e. ne faisant évoluer que la condition de sélection, est illustré par la Figure F.4. L'ensemble de l'heuristique a été baptisée RRHCCA, pour Random Restart Hill-Climbing of Complex Aggregates.

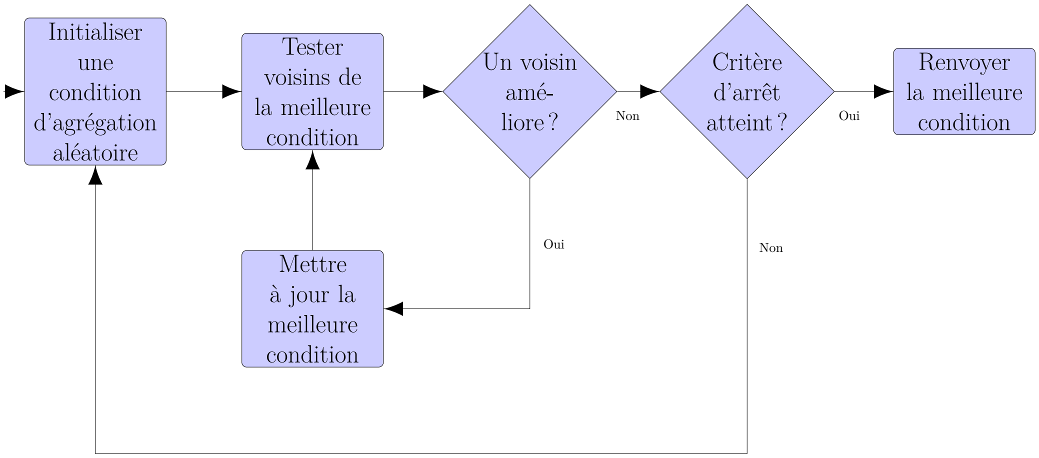

Initialiser une condition d'agrégation aléatoire
Tester voisins de la meilleure condition
Un voisin amé- liore ?
Non
Critère d'arrêt atteint ?
Oui
Renvoyer la meilleure condition
Oui
Non
Mettre à jour la meilleure condition

Figure F.4 – Fonctionnement de l'algorithme de hill-climbing pour un processus d'agrégation donné au sein de RRHCCA.

La nouvelle approche a été validée sur des données artificielles, sur lesquelles les arbres de décision relationnels appris avec Tilde ou ceux entraînés après application de l'algorithme de propositionalisation RELAGGS n'apprennent pas aussi bien en terme de performance prédictive. Néanmoins, les résultats n'étaient significativement meilleurs que sur peu de jeux de données réels. En effet, le caractère précis, spécifique des agrégats complexes, entraîne un problème de sur-apprentissage, ou incapacité à généraliser. Plus concrètement, les données d'entraînement qui servent à apprendre le modèle sont trop bien apprises, au point que le modèle est moins efficace sur de nouvelles données non vues à l'entraînement.

Pour résoudre ce problème, une autre famille de modèles, les *Random forests*, a été adaptée aux agrégats complexes. Les Random forests (ou forêts d'arbres décisionnels) (Breiman 2001) consistent en un ensemble d'arbres de décision. Chaque arbre est appris différemment pour introduire de la diversité dans le modèle. L'idée principale, appelée *bagging* en anglais, est que la combinaison de plusieurs modèles « faibles » est plus efficace qu'un seul modèle « fort ». Le schéma en Figure F.5 résume le processus d'apprentissage et d'utilisation d'une forêt de n arbres. La diversité entre les arbres est introduite à deux niveaux :

- Chaque arbre est entraîné sur un jeu de données différent : la technique du bootstrap consiste, à partir d'un jeu de données d'entraînement de E exemples, à tirer E exemples au hasard de ce jeu avec remise, i.e. un même exemple peut être tiré plusieurs fois. Chaque arbre est appris sur un jeu de données différent tiré au hasard suivant cette technique.

- À chaque nœud d'un arbre, la meilleure séparation des données est choisie sur un sous-ensemble des caractéristiques possibles : si l'on dispose de k caractéristiques d'entrées pour la prédiction, l'apprentissage de chaque nœud n'en considèrera qu'une partie tirée au hasard. En général, on utilise $\sqrt{k}$ caractéristiques à chaque nœud.

La prédiction sur un nouvel exemple s'effectue en combinant les prédictions de chaque arbre pour cet exemple. Dans une tâche de classification, la recombinaison est un vote majoritaire : chaque arbre prédit pour l'exemple l'une des classes possibles, et la forêt prédit la classe la plus souvent prédite par les arbres.

Dans le cadre d'une utilisation des forêts avec des agrégats complexes, le bootstrap des exemples pour entraîner chaque arbre peut être réutilisé comme décrit dans la méthode d'origine. Le tirage au sort des caractéristiques à utiliser à chaque nœud d'un arbre nécessite une adaptation aux agrégats complexes, dont le nombre astronomique empêche l'énumération, et rend difficile un sous-échantillonnage complètement aléatoire. Une extension de Tilde aux agrégats complexes et aux forêts d'arbres décisionnels, baptisée FORF (Van Assche et al. 2006), propose une méthode pour effectuer ce sous-échantillonnage qui, sans énumérer l'espace des agrégats complexes, donne un tirage uniformément réparti sur l'espace. Nous avons choisi d'adopter une démarche inverse, en nous basant sur la structure de l'espace des agrégats complexes. En effet, un agrégat complexe est décomposable en deux parties : le processus d'agrégation, et la condition de sélection. Le premier peut être vu comme définissant une famille d'agrégats : modifier le processus d'agrégation sans modifier la condition de sélection change de manière drastique le potentiel prédictif de l'agrégat. À l'inverse, une modification de la condition de sélection sans modifier le processus d'agrégation aura moins d'impact sur le potentiel prédictif de l'agrégat. Respectant l'idée d'origine du sous-échantillonnage des caractéristiques, à savoir introduire de la diversité entre les tirages, pour p processus d'agrégation

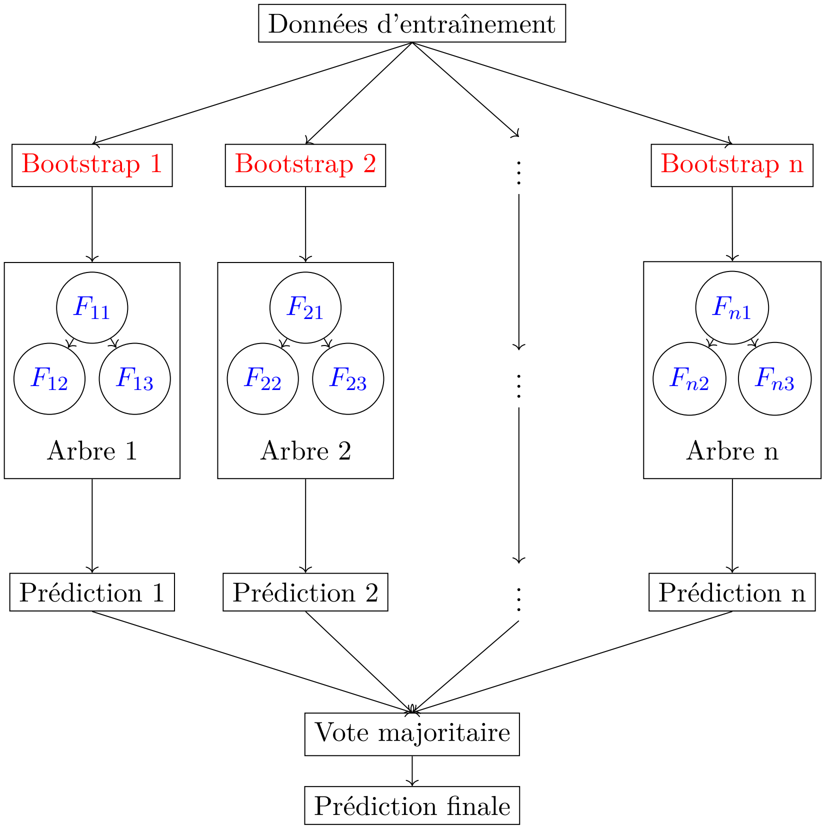


FIGURE F.5 – Processus d'entraînement d'une forêt d'arbres décisionnels.

disponibles, et k propriétés des objets secondaires, nous gardons $\sqrt{p}$ processus d'agrégation et $k/2$ propriétés pour la condition de sélection à chaque nœud. Contrairement à un sous-échantillonnage uniforme, qui garderait des agrégats de chaque processus d'agrégation à chaque tirage et de fait peu de diversité entre les tirages, notre approche ne garde à chaque tirage qu'un sous-ensemble des caractéristiques réellement différentes, tous les processus d'agrégation n'étant pas représentés.

Nous illustrons ce processus de sous-échantillonnage sur l'exemple du jeu de données des ilots urbains en Tableau F.2. Le Tableau F.2a montre l'exemple du sous-échantillonnage des processus d'agrégation : sur les 10 disponibles, on en garde donc 3 pour trouver la séparation optimale à un nœud donné d'un arbre donné. Le tirage des propriétés secondaires à retenir pour la condition de sélection est différent pour chaque processus d'agrégation retenu. Ainsi, la Table F.2b montre un sous-échantillonnage possible de ces propriétés secondaires, en conservant la moitié.

L'algorithme de hill-climbing a également été simplifié pour être plus rapide : pour chaque processus d'agrégation, un hill-climbing est effectué pour optimiser la condition de sélection, sans notion de réinitialisation aléatoire contrairement à RRHCCA. De plus, Tableau F.2 – Sous-échantillonnage des agrégats complexes.

(a) Sous-échantillonnage des processus d'agrégation

Fonction	Propriété	Choisi
Compte		x
Minimum	Aire	
Minimum	Élongation	
Minimum	Convexité	
Maximum	Aire	
Maximum	Élongation	x
Maximum	Convexité	
Moyenne	Aire	
Moyenne	Élongation	
Moyenne	Convexité	x

(b) Sous-échantillonnage des propriétés secondaires.

Propriété	Choisie
Aire	x
Élongation	
Convexité	x

chaque pas de l'hill-climbing ne teste qu'un seul voisin de l'agrégat courant parmi ceux définis dans RRHCCA, au lieu de tous les considérer. Si ce voisin est meilleur que l'agrégat courant, la recherche reprend à partir de lui, sinon la recherche continue depuis l'agrégat courant, qui est considéré comme optimal quand un certain nombre de voisins ont été testés sans succès.

Une approche de sélection de propriétés à l'aide de forêts d'arbres décisionnels a été envisagée. Elle reprend les idées introduites en ce sens par les forêts d'arbres décisionnels originales, en les adaptant au cas des agrégats complexes. L'objectif est d'évaluer l'importance de chaque propriété à l'aide de la forêt. Néanmoins, le nombre d'agrégats complexes disponibles est très élevé, et il est donc improbable qu'un agrégat donné soit présent plusieurs fois dans la forêt. Nous évaluons donc l'importance de familles d'agrégats complexes, une famille étant définie par un processus d'agrégation commun, une condition sur une propriété commune dans la condition de sélection, ou une combinaison des deux. L'approche classique évalue l'importance d'une propriété de la façon suivante : chaque arbre de la forêt a été appris sur un sous-ensemble du jeu de données d'entraînement d'origine, il y a donc un autre sous-ensemble d'exemples d'entraînement que l'arbre n'a pas considéré. Cet ensemble d'exemples est utilisé pour calculer la performance prédictive de l'arbre. Ensuite, les valeurs de la propriété dont on veut évaluer l'importance sont permutées aléatoirement entre les exemples de cet ensemble, et la performance prédictive de l'arbre sur ce jeu de données modifiées est évaluée. Si la performance de l'arbre sur le jeu permuté est plus faible que la performance sur le jeu original, la propriété est importante. On quantifie cette importance au niveau d'un arbre par la différence entre les deux performances. L'importance finale de la propriété est la moyenne des importances au niveau des arbres de la forêt.

3 Adaptation aux changements de contexte et reframing

Le reframing consiste à créer des modèles de prédiction, dits versatiles, réutilisables dans différents contextes, pour éviter d'entraîner un nouveau modèle dans chaque contexte. Ce besoin est motivé par le coût qu'implique l'entraînement complet d'un nouveau modèle par rapport à la réutilisation de connaissances apprises dans un autre contexte, avec une étape d'adaptation au nouveau contexte. Ce processus est illustré en Figure F.6.

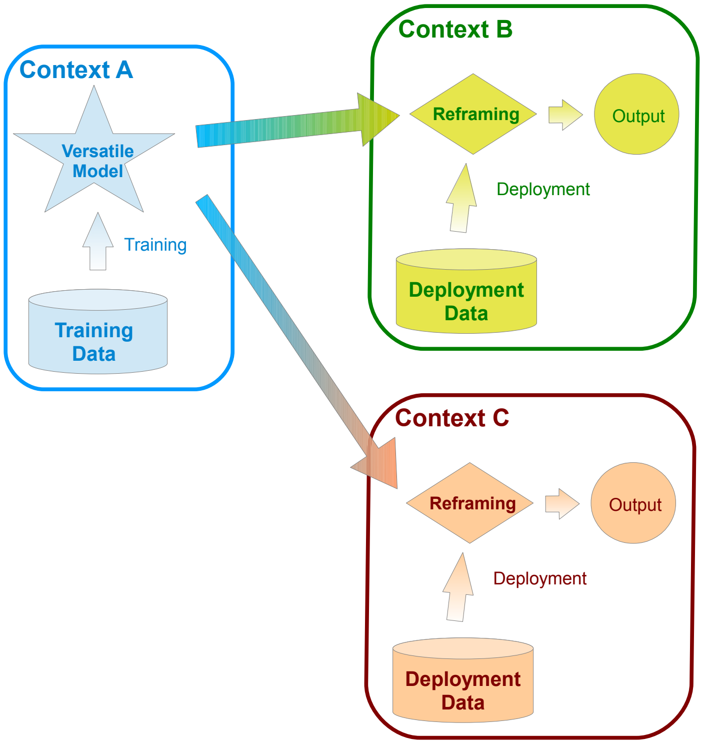


FIGURE F.6 – Schéma global du processus de reframing[1].

Un premier type de changement de contexte auquel nous nous intéressons est un changement de coûts d'erreurs de classification, dans une tâche de classification sensible au coût (ELKAN 2001). En effet, dans certaines tâches de prédiction, il arrive que les différents types d'erreur n'aient pas le même impact. Pour reprendre l'exemple du jeu de données post-opératoires, envoyer un patient en soins intensifs n'a pas le même coût pour l'hôpital que de le renvoyer chez lui. De plus, renvoyer un patient chez lui alors qu'il aurait dû rester à l'hôpital constitue une erreur bien plus grave que de le garder à l'hôpital alors qu'il aurait pu être renvoyé chez lui. Ces différences sont matérialisées par une matrice de coûts, dont un exemple est donné en Tableau F.3a. Dans cette matrice,

[1] Je remercie le projet REFRAME, financé par CHIST-ERA, pour cette figure.

l'élément en ligne i et colonne j, correspond au coût de prédire la classe en colonne j alors que l'exemple aurait dû être prédit comme étant de la classe en ligne i. Les éléments diagonaux, associés aux prédictions correctes, sont nuls. Comme expliqué plus haut, les coûts en troisième colonne, correspondant aux coûts induits par le renvoi du patient à son domicile alors qu'il aurait dû rester à l'hôpital voire en soins intensifs, sont plus élevés que les coûts en troisième ligne, correspondant aux coûts associés au maintien du patient à l'hôpital alors qu'il aurait pu rentrer chez lui.

On considère maintenant un deuxième hôpital, avec une moins grande capacité d'accueil, impliquant un coût plus élevé du maintien d'un patient à l'hôpital. Ceci entraîne un changement de coûts par rapport au premier hôpital, matérialisé par la matrice en Tableau F.3b, où les coûts de maintien à l'hôpital, en première et deuxième colonnes, sont plus élevés que dans le premier hôpital. Nous avons développé un algorithme d'apprentissage pour des tâches multi-classes, où la propriété cible peut prendre trois valeurs ou plus, par opposition à une tâche de classification binaire où la propriété cible n'a que deux valeurs possibles. Cet algorithme a été conçu pour répondre à la double problématique de la multiplicité des classes et de la sensibilité au coût, et sa structure permet une adaptation à un changement de matrice de coûts.

Tableau F.3 – Exemple de changement de contexte, matérialisé par un changement de coûts de misclassification, entre deux hôpitaux.

P A	h	si	dom
h	0	2	10
si	5	0	20
dom	2	5	0

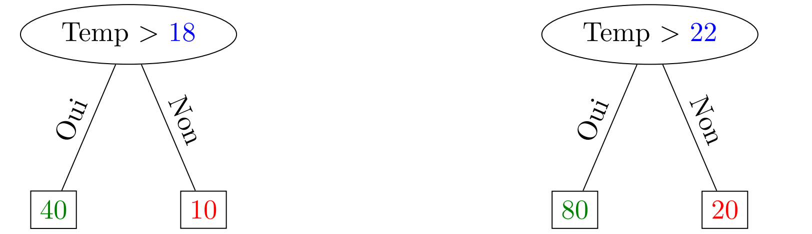
(a) Matrice de coûts associée au premier hôpital.

P A	h	si	dom
h	0	**5**	10
si	**10**	0	20
dom	**5**	**10**	0

(b) Matrice de coûts associée au deuxième hôpital.

Le deuxième type de changement de contexte auquel nous nous intéressons concerne le changement de distribution de propriétés des données, en entrée et/ou en sortie. Nous avons donné l'exemple d'un modèle de prédiction des ventes de glaces en fonction de la température appris à Lille, que nous voulons adapter pour l'utiliser à Marseille. Le climat marseillais étant différent du climat lillois, notamment en termes de température, on a un exemple de changement de contexte où la distribution des propriétés en entrée, i.e. la température, change du contexte lillois au contexte marseillais. Les Figures F.7a et F.7b montrent les modèles de prédiction de la vente ou non de glaces en fonction de la température, respectivement pour Lille et pour Marseille. Des glaces sont vendues à Lille à partir de 18°C, tandis qu'à Marseille, où la température moyenne est plus élevée, elles ne commencent qu'à 22°C. L'utilisation du premier modèle lillois à Marseille ne sera pas efficace pour des températures entre 18°C et 22°C. L'idée est donc de « recaler » les températures des données marseillaises pour qu'elles ressemblent aux températures lilloises et que le modèle soit applicable, par exemple en enlevant systématiquement 4°C à la température d'entrée du modèle lors de l'application à Marseille.

Ce recalage peut également intervenir en sortie, sur la propriété cible de la prédiction. En effet, on observe que la quantité de glaces vendues à Marseille est en moyenne plus élevée qu'à Lille. L'application du modèle lillois aura donc tendance à sous-évaluer les ventes de glaces. On peut donc envisager de modifier la prédiction du modèle lillois pour que les quantités de glaces prédites en ayant appris le modèle suivant les normes lilloises soient proches des quantités vendues à Marseille, par exemple en multipliant systématiquement par deux les prédictions du modèle.



(a) Modèle cible à Lille.

(b) Modèle cible à Marseille.

FIGURE F.7 – Modèles de prédiction de ventes de glaces, à Lille et à Marseille.

Nous avons conçu des algorithmes de reframing recalant les propriétés numériques suivant ces idées, en utilisant des fonctions affines pour transformer les valeurs de ces propriétés et ainsi s'adapter aux changements de distribution.

3.1 Apprentissage multi-classes sensible au coût et reframing des sorties

Nous nous plaçons dans le cadre d'un apprentissage multi-classes sensible au coût, i.e. sur des tâches de classification où la propriété cible prend 3 valeurs ou plus et possédant une matrice de coûts définissant les importances des différentes erreurs de classification. Nous proposons une méthode d'apprentissage basée sur la binarisation du problème multi-classes, i.e. la tâche d'origine est décomposée en plusieurs tâches de classification binaire, ne faisant intervenir que deux classes. Considérons un problème à C classes, dénotées $(1, 2, \cdots, C)$, l'utilisation d'entiers ne définit ici aucun ordonnancement, la notation entière des classes ne désigne donc que des catégories. Nous nous intéressons à une décomposition par paires (FÜRNKRANZ 2002) : du problème d'origine à C classes, nous considérons $C(C-1)/2$ problèmes binaires, chacun se focalisant sur deux des C classes d'origine.

Pour chaque problème binaire défini par une paire de classes, on considère un modèle de prédiction renvoyant, au lieu d'une pure prédiction de classe pour un exemple, un score par classe indiquant la confiance que le modèle a en l'appartenance de l'exemple à la classe. Dans le cas d'une tâche de classification binaire, deux scores sont donc renvoyés, un pour chacune des deux classes considérées dans le problème. Considérant que les scores sont normalisés, appartenant à l'intervalle $[0; 1]$, et que leur somme vaut 1, ce qui est le cas dans les implémentations que nous utilisons, seul l'un des deux scores est nécessaire. Dans le cas du modèle appris pour les classes i et j, avec $1 \leqslant i < j \leqslant C$, on retient le score de la classe d'index le moins élevé, à savoir la classe i.

Pour chaque modèle de prédiction renvoyant un score, on veut maintenant définir un seuil sur ce score, qui va permettre pour chaque exemple la prédiction par le modèle de l'une des deux classes, i ou j, en fonction du score prédit par le modèle pour l'exemple. Comme défini plus haut, le score considéré est la confiance du modèle en l'appartenance de l'exemple à la classe i, la classe prédite sera donc i si le score est supérieur au seuil, et j si le score est inférieur au seuil. L'objectif de nos algorithmes sera d'optimiser ces seuils dans un contexte de sensibilité au coût.

Enfin, la prédiction multi-classes s'effectue par un processus de vote majoritaire : chaque modèle binaire a prédit pour l'exemple l'une des deux classes dont il a la charge, la classe prédite au final pour l'exemple sera la classe la plus prédite par les $C(C-1)/2$ modèles binaires.

La Figure F.8 présente un exemple de ce processus pour un problème à 3 classes. La binarisation par paires induit l'apprentissage de trois modèles binaires, un pour chaque paire. On apprend ensuite les seuils sur les scores pour chaque modèle, ils sont indiqués en rouge sur la Figure. Enfin, on veut classer un nouvel exemple, les scores renvoyés pour l'exemple par chaque modèle sont indiqués en bleu. Le modèle en charge des classes 1 et 2 renvoie un score de 0.5 avec un seuil de décision de 0.2, le score étant au-dessus du seuil, le modèle prédit la classe 1. De même, le modèle en charge des classes 1 et 3 renvoie un score de 0.9 pour un seuil de 0.7, il prédit donc la classe 1. Enfin, le modèle en charge des classes 2 et 3 donne un score de 0.2 pour un seuil de 0.4, il prédit donc la classe 3. Au total, la classe 1 est donc prédite deux fois, la classe 3 une fois, et la classe 2 jamais. La prédiction finale du modèle multi-classes pour l'exemple est donc la classe 1.

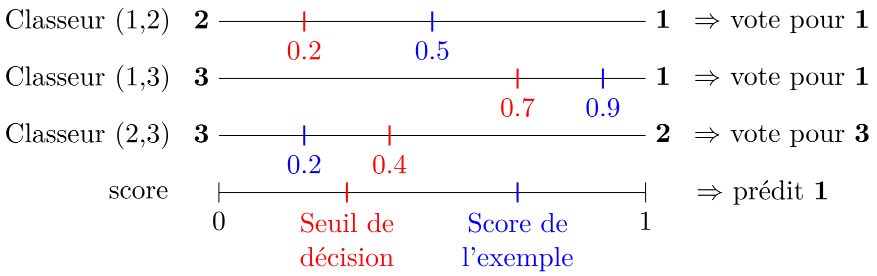

Classeur (1,2) **2** **1** ⇒ vote pour **1**
0.2
0.5
Classeur (1,3) **3** **1** ⇒ vote pour **1**
0.7
0.9
Classeur (2,3) **3** **2** ⇒ vote pour **3**
0.2
0.4
score ⇒ prédit **1**
0
Seuil de décision
Score de l'exemple
1

FIGURE F.8 – Illustration de la binarisation par paires et de la prédiction majoritaire à l'aide de seuils.

L'algorithme que nous proposons pour l'optimisation des seuils s'effectue modèle par modèle : il ne s'agit pas de trouver la combinaison de seuils optimale sur l'ensemble des modèles, qui est impossible à mettre en œuvre en termes de complexité algorithmique. Nous chercherons donc le meilleur seuil par modèle, l'un après l'autre, en tenant compte de la matrice de coûts. Notre algorithme est exhaustif : à chaque exemple d'entraînement correspond un score renvoyé par le modèle en charge des classes i et j, et tous sont testés comme valeurs pour le seuil de ce modèle. La valeur retenue sera celle qui permet de minimiser le coût total sur le jeu d'entraînement induit par les prédictions du modèle avec cette valeur de seuil.

Nous avons considéré deux approches pour l'optimisation du seuil du modèle en charge des classes i et j : l'une considère uniquement l'utilisation des exemples de classe i ou j, tandis que l'autre utilise l'ensemble des données d'entraînement, i.e. les exemples de toutes les classes. Prenons l'exemple du jeu de données à 3 classes donné en Tableau F.4, il s'agit d'optimiser le seuil associé au modèle en charge des classes 1 et 2. Les scores renvoyés par ce modèle pour les exemples du jeu d'entraînement, ainsi que la classe de ces exemples, sont donnés en Tableau F.4a, tandis que la matrice de coûts associée à la tâche de classification est donnée en Tableau F.4b.

Tableau F.4 – Jeu de données à 3 classes sensible au coût.

Classe	2	2	3	2	1	3	2
Score	0	0.05	0.1	0.15	0.2	0.35	0.4
Classe	3	1	1	2	3	1	1
Score	0.5	0.65	0.7	0.8	0.9	0.95	1

(a) Exemples d'entraînement du jeu de données, avec les scores associés renvoyés par le modèle en charge des classes 1 et 2.

P A	1	2	3
1	0	3	3
2	1	0	4
3	4	2	0

(b) Matrice de coûts associée au problème à 3 classes.

La différence entre nos deux approches tient en l'utilisation ou non des exemples de classe 3 pour l'optimisation de ce seuil. Cette utilisation a du sens, puisque la prédiction d'un exemple de classe 3 en classe 1 ou en classe 2 n'a pas le même coût : 4 dans le premier cas, 2 dans l'autre. De plus, le seuil optimal n'est pas le même suivant l'approche retenue, comme le montre la Figure F.9. Le seuil retenu sans l'utilisation des exemples de classe 3 est de 0.15, ce qui reflète la tendance du modèle à prédire plus facilement la classe 1 que la classe 2. En effet, le coût associé à la prédiction de la classe 2 quand la classe 1 aurait dû être prédite est de 3, plus élevé que le coût lié à la prédiction de la classe 1 quand la classe 2 aurait dû être prédite, qui est de 1. Il est donc moins risqué de prédire la classe 1. Dans l'approche utilisant les exemples de toutes les classes, le seuil retenu est 0.5, plus élevé que celui donné par l'autre approche. Ceci s'explique par la prise en compte des coûts de prédiction des exemples de la classe 3 en classe 1 ou en classe 2. Pour ces exemples, il est plus risqué de prédire la classe 1, puisque le coût de misclassification d'un exemple de classe 3 en classe 1 est de 4, quand le coût de misclassification d'un exemple de classe 3 en classe 2 est de 2. Cela rend la prédiction de la classe 2 plutôt que de la classe 1 moins risquée qu'avec la première approche, le seuil est donc plus élevé.

Nous utilisons comme modèle binaire renvoyant des scores un classeur bayésien naïf. Les comparaisons expérimentales ont été effectuées sur des jeux de données réels disponibles sur les dépôts de l'UCI (Lichman 2013), en générant des matrices de coûts aléatoires avec diagonale nulle, et un coût de 1, 10, 100 ou 1 000 en dehors. Nous com-

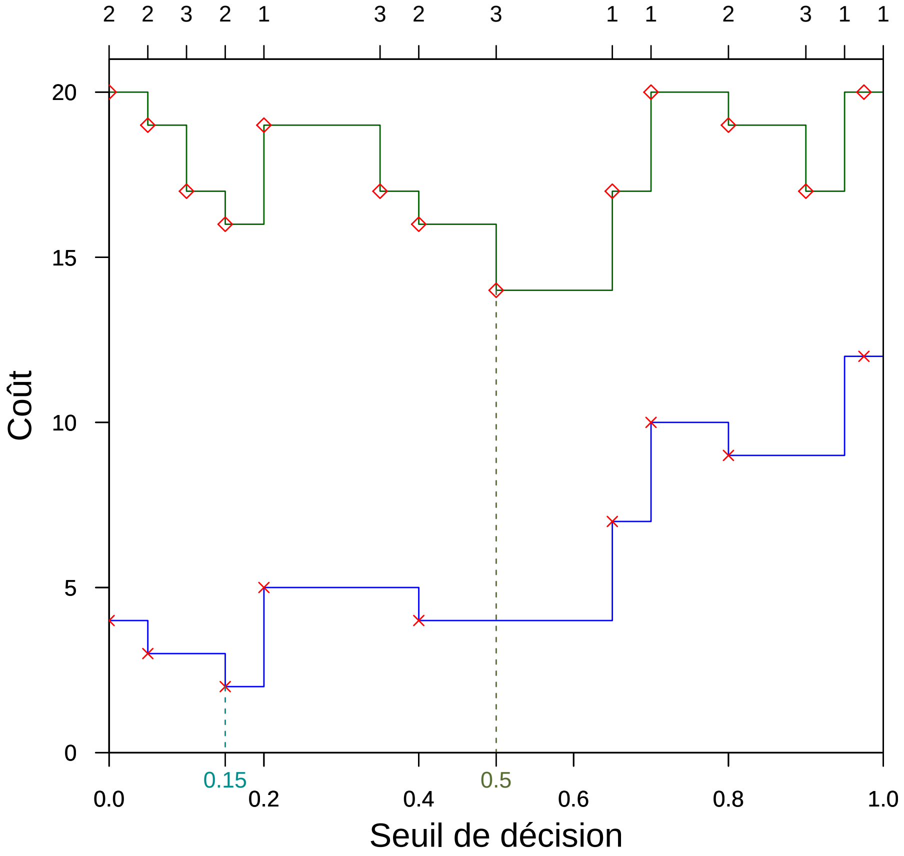


FIGURE F.9 – Seuils optimaux pour les deux approches, avec utilisation des exemples de classes 3 (en vert) et sans (en bleu).

parons avec des méthodes de l'état de l'art basées plus spécifiquement sur des approches de binarisation faisant intervenir des seuils, mais aucune ne considère un seuil par paire de classes. Nous comparons également avec des versions sensibles au coût de classeurs bayésiens naïfs et d'arbres de décision, qui sont naturellement applicables sur des tâches multi-classes. Tous les modèles sont implémentés, ou ont été implémentés le cas échéant, en Java dans le logiciel WEKA (HALL et al. 2009).

Les résultats expérimentaux montrent une supériorité, en termes de performance prédictive, de notre approche utilisant les exemples de toutes les classes pour optimiser les seuils. Cette supériorité est statistiquement significative par rapport à toutes les autres approches considérées dans la comparaison, la significativité ayant été obtenue par un test de Friedman avec une confiance de 95% ayant rejeté l'égalité de performance de toutes les approches, puis d'un test a posteriori de Nemenyi.

Enfin, nous avons étudié les possibilités de notre approche en termes de reframing des sorties. Il s'agit ici tout d'abord d'entraîner les modèles binaires individuels renvoyant des scores dans un premier contexte, avec une certaine matrice de coûts associée. Ensuite, les seuils de décision sur ces scores peuvent être appris dans le contexte de déploiement où le modèle va être utilisé, avec une deuxième matrice de coûts liée à ce contexte. Cette approche est très performante quand peu de données sont disponibles pour le contexte de déploiement, et que l'entraînement complet de nouveaux modèles est peu pertinent. Nous avons comparé cette approche, i.e. l'apprentissage des modèles binaires individuels dans le premier contexte et l'apprentissage des seuils dans le contexte de déploiement, avec une approche dite de « base », i.e. les deux apprentissages sont effectués dans le premier contexte, et une approche dite de « réentraînement », i.e. les deux apprentissages sont effectués dans le contexte de déploiement avec peu de données. Les comparaisons ont été effectuées sur un jeu de test en utilisant les coûts associés au contexte de déploiement. On évalue le coût moyen sur le jeu de test en fonction du nombre d'exemples utilisés dans le contexte de déploiement pour apprendre les seuils, dans le cas de notre approche, ou le modèle entier, dans le cadre d'un réentraînement. Nous observons que notre approche de reframing des sorties est plus performante que le réentraînement quand peu de données sont disponibles. Elle est également plus performante que la méthode dite de « base », ou le devient avec l'augmentation du nombre d'exemples utilisés dans le contexte de déploiement, ce paramètre n'intervenant pas dans la performance de l'approche de « base », puisqu'elle n'utilise pas ces exemples.

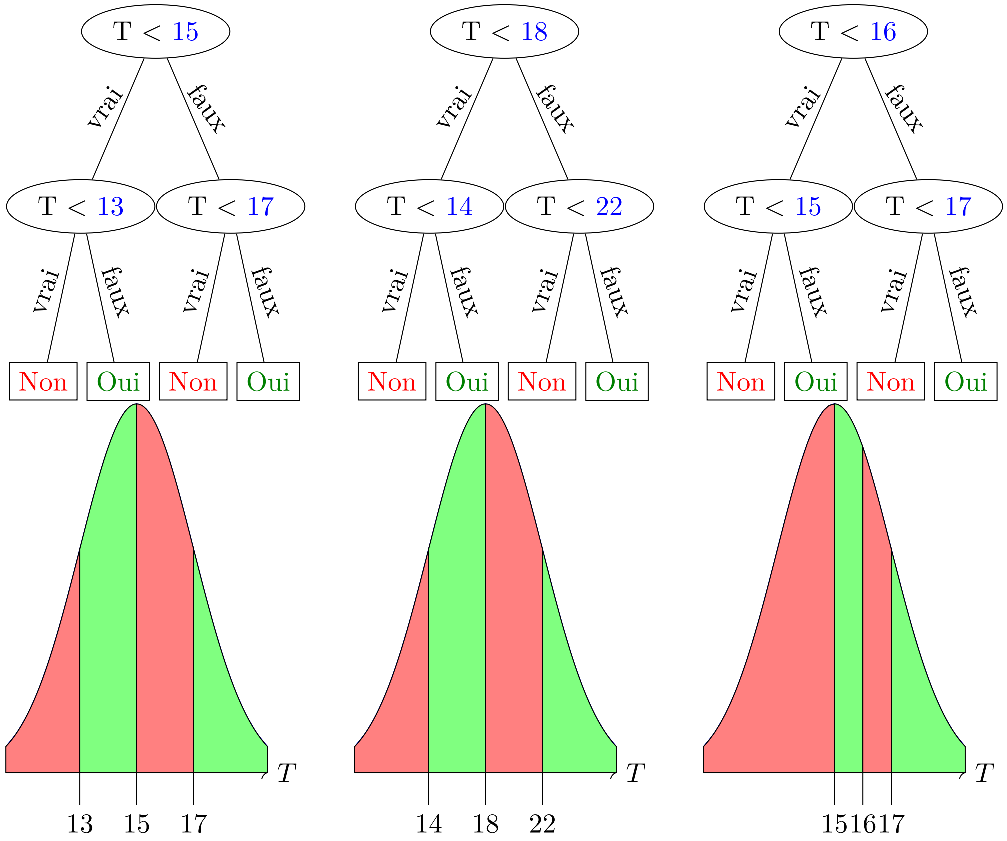
3.2 Reframing des propriétés numériques par transformation affine

Nous nous sommes intéressés à une autre famille de changements de contexte, caractérisée par le changement de distribution d'une propriété numérique entre le contexte d'entraînement et le contexte de déploiement. Nous proposons ainsi une solution au problème du *dataset shift*, dont une description et une typologie sont données dans (Moreno-Torres, Raeder et al. 2012).

Nous nous intéressons particulièrement à deux typologies : le *covariate shift*, qui constitue un changement de distribution des propriétés d'entrée tandis que la relation entre les entrées et la propriété de sortie, i.e. le modèle implicite, reste la même par rapport au changement de distribution. La Figure F.10 donne un exemple de ce problème : nous considérons une tâche de classification binaire, les valeurs possibles de la classe étant « Oui » et « Non », en fonction d'une propriété numérique, dénotée T. Nous considérons 3 contextes, le modèle implicite pour chaque contexte est donné en haut, tandis que la distribution de la propriété T, superposée avec la prédiction induite pour la classe, en rouge pour « Non » et en vert pour « Oui », est donnée en bas. Dans le premier contexte, décrit en Figure F.10a, T suit une distribution normale de moyenne 15 et d'écart-type 2, les seuils de séparation des classes du modèle implicite se situent à 13, 15 et 17, soit à la moyenne et à un écart-type de chaque côté de la moyenne. Dans le deuxième contexte, décrit en Figure F.10b, la distribution de T change par rapport au premier contexte, suivant maintenant une loi normale de moyenne 18 et d'écart-type 4.

Les seuils du modèle cible ont également changé, passant à 14, 18 et 22. Néanmoins, ils n'ont pas changé relativement à la distribution de T, étant toujours situés à la moyenne et à un écart-type de chaque côté de la moyenne. La différence entre les premier et deuxième contextes constitue donc un exemple de *covariate shift*.

La deuxième typologie à laquelle nous nous intéressons est le *concept shift*, qui constitue un changement de relation entre les entrées et la propriété cible, tandis que la distribution des entrées reste la même. Par exemple, dans le troisième contexte, décrit en Figure F.10c, la distribution de T est la même que dans le premier contexte. Par contre, le modèle implicite a changé, les seuils étant maintenant 15, 16 et 17. La différence entre les premier et troisième contextes constitue donc un exemple de *concept shift*.



(a) Contexte 1.
$T \sim \mathcal{N}(15, 2^2)$

(b) Contexte 2.
$T \sim \mathcal{N}(18, 4^2)$

(c) Contexte 3.
$T \sim \mathcal{N}(15, 2^2)$

Figure F.10 – Modèle cible et distribution de la propriété d'entrée dans les trois villes contextes.

Dans ce cadre, nous proposons une solution à ces problématiques reposant sur un reframing des données plutôt que du modèle. Concrètement, il s'agit d'entraîner un modèle sur des exemples d'entraînement d'un premier contexte, et lors du passage à un autre contexte de déploiement, de transformer les valeurs des propriétés des exemples de ce nouveau contexte, dans le but que le modèle appris dans le premier contexte soit efficace en termes de performance prédictive.

Plus précisément, nous nous concentrons sur la transformation des propriétés numériques par des transformations affines. Il s'agit d'apprendre une fonction de reframing par propriété d'entrée, de la forme $x \longmapsto \alpha x + \beta$, où x désigne la valeur de la propriété numérique. Sur l'exemple de la Figure F.10, le modèle du premier contexte peut être utilisé dans le deuxième contexte à condition que les valeurs de l'entrée T dans le deuxième contexte soit transformées par la fonction $x \longmapsto 0.5x + 6$, ce qui aligne les seuils utilisés dans le modèle cible du deuxième contexte avec ceux du premier contexte. De même, le modèle du premier contexte peut être utilisé dans le troisième contexte, en utilisant la fonction $x \longmapsto 2x - 17$ pour transformer l'attribut T des exemples du troisième contexte.

Le problème de reframing est d'optimiser les paramètres des fonctions affines pour chaque propriété numérique, i.e. le coefficient directeur α et l'ordonnée à l'origine β, pour obtenir une transformation utile du point de vue de la performance prédictive. Cette optimisation s'effectue sur un jeu de données du contexte de déploiement contenant peu d'exemples. Nous proposons deux algorithmes basés sur l'optimisation stochastique pour trouver les valeurs des paramètres. Chacun d'entre eux utilise des initialisations aléatoires de l'ensemble des paramètres des fonctions affines pour chaque propriété. Ces initialisations aléatoires sont néanmoins guidées par les intervalles de valeurs que prend la propriété correspondante. Ainsi, si une propriété prend ses valeurs dans l'intervalle $[l_1; u_1]$ dans le contexte d'entraînement et dans l'intervalle $[l_2; u_2]$ dans le contexte de déploiement, la fonction de transformation la plus « naturelle » est celle qui fait correspondre le deuxième intervalle au premier, à savoir :

$$x \longmapsto \frac{u_1 - l_1}{u_2 - l_2}(x - l_2) + l_1$$

Les initialisations aléatoires d'un coefficient directeur s'effectuent donc « autour » de la valeur du coefficient directeur de cette fonction. Il en va de même pour l'initialisation aléatoire d'une ordonnée à l'origine, qui tient néanmoins compte du coefficient directeur choisi.

Le premier algorithme que nous proposons utilise la même idée que l'algorithme RRHCCA introduit dans le cadre des agrégats complexes, à savoir un hill-climbing avec réinitialisations aléatoires. Concrètement, d'un tirage au sort de paramètres, l'algorithme effectue un hill-climbing pour chaque paramètre. La qualité d'un ensemble de paramètres correspond à la performance prédictive sur le petit jeu de données du contexte de déploiement, transformé à l'aide des fonctions définies par l'ensemble de paramètres. Les paramètres sont optimisés un par un, recommençant le parcours de l'ensemble de paramètres tant qu'un cycle complet n'a pas permis d'améliorer la performance, i.e. tous les paramètres ont fait l'objet d'une tentative d'optimisation infructueuse à la suite.

Le deuxième algorithme que nous proposons est plus simple, il n'est basé que sur des initialisations aléatoires. Concrètement, un certain nombre d'ensembles de paramètres sont générés aléatoirement et évalués en termes de performance prédictive. Le meilleur ensemble de paramètres testé est retenu comme optimum.

Ces algorithmes ont été évalués sur des données synthétiques générées comme présenté en Figure F.10, par rapport à une approche de réentraînement sur le petit jeu de données et la réutilisation directe du premier modèle, en fonction du nombre d'exemples composant le « petit » jeu de données de déploiement. Nos algorithmes sont plus performants que ces deux approches en termes de performance prédictive, et ce particulièrement avec un nombre d'exemples faible. Nous avons également effectué une comparaison sur des jeux de données réels, sur lesquels des transformations artificielles ont été appliquées aux attributs numériques. Encore une fois, nos algorithmes sont plus efficaces que les deux autres approches, ainsi que l'approche GP-RFD (Moreno-Torres, Llorà et al. 2013) basée sur une optimisation de fonctions de transformation par programmation génétique. Cette amélioration de la performance est statistiquement significative, suivant les mêmes critères que précédemment.

Nous avons également étudié le comportement de nos algorithmes sur un jeu de données réel, consistant en une tâche de régression où l'objectif est de prédire le nombre de bicyclettes louées sur une journée à Washington D.C., en fonction des conditions météorologiques (Fanaee-T et Gama 2014). Le jeu de données contient les enregistrements quotidiens sur deux années consécutives. Étant donné que nous sommes en présence d'une tâche de régression, la propriété cible est numérique. Nous pouvons donc appliquer nos algorithmes de reframing pour transformer la sortie du modèle avec une fonction affine. Cette idée trouve son utilité sur ce jeu de données, car le nombre de bicyclettes louées en moyenne est plus élevé la deuxième année. Un modèle appris la première année et déployé la seconde sous-estimerait donc le nombre de bicyclettes louées. On peut donc transformer la prédiction du modèle, qui ressemblera à une valeur du contexte d'origine, pour qu'elle soit proche d'une valeur du contexte de déploiement. Le jeu de données présente également un exemple de changement de distribution en entrée suivant la saison : en effet, les conditions météorologiques telles que la température varient suivant la saison.

Enfin, nous avons appliqué ces travaux au reframing dans un cadre d'apprentissage relationnel. Nous nous basons sur les agrégats complexes, qui sont des propriétés essentiellement numériques, où la transformation peut intervenir à deux niveaux. La sortie d'un agrégat complexe peut être transformée, ainsi que les propriétés numériques de la table secondaire, qui interviennent dans les conditions de sélection de l'agrégat. On peut donc modifier la définition des objets secondaires pertinents sélectionnés pour l'agrégation, ainsi que comme précédemment la propriété elle-même, ici l'agrégat complexe. Avec peu de données du contexte de déploiement, on observe encore que nos algorithmes adaptent le modèle initial de manière à ce qu'il soit plus efficace avec la transformation, mais également que le modèle appris avec le peu de données du contexte de déploiement.

4 Conclusion et travaux futurs

Cette thèse a étudié l'adaptation de modèles de prédiction dans les cadres de l'apprentissage relationnel et de la réutilisabilité des modèles entre différents contextes. Des algorithmes d'apprentissage et d'adaptation ont été proposés pour répondre à ces problématiques.

Dans le cadre de l'apprentissage relationnel, nous avons proposé de nouvelles extensions des arbres de décision et des forêts d'arbres décisionnels intégrant les propriétés relationnelles appelées agrégats complexes. Nous avons développé des heuristiques inspirées du domaine de l'optimisation stochastique pour construire les agrégats complexes pertinents, parmi les milliards de possibilités. Les modèles d'apprentissage ainsi créés ont été évalués expérimentalement avec succès.

Des algorithmes d'adaptation de modèles aux changements de contexte ont également été mis au point. Dans le cadre de problèmes multi-classes sensibles au coût, notre approche permet l'apprentissage d'un modèle dont la structure peut être adaptée. Dans le cadre de problèmes faisant intervenir des propriétés numériques, nos algorithmes d'apprentissage de transformations affines permettent l'adaptation des données pour rendre un modèle interopérable entre plusieurs contextes avec des distributions différentes.

Un piste de travail est l'application de ces travaux à l'apprentissage sur des données multi-dimensionnelles, en particulier spatio-temporelles. En effet, ces dernières peuvent se représenter de manière relationnelle. Par exemple, sur le jeu de données de location de bicyclettes, on peut mettre en relation l'enregistrement correspondant à un jour donné avec les enregistrements des jours précédents. L'utilisation d'agrégats complexes permet alors de créer des propriétés telles que « *la moyenne du nombre de bicyclettes louées sur les 5 jours précédents* », qui peuvent se révéler pertinentes pour la prédiction du nombre de bicyclettes louées le jour même. De même, si l'on considère la prédiction du nombre de bicyclettes louées par station de location, on peut mettre en relation les enregistrements d'un station avec ceux des autres stations, en considérant la distance spatiale qui les sépare, on peut alors créer des propriétés telles que « *la moyenne du nombre de bicyclettes louées sur les 3 jours précédents dans les stations dans un rayon de 200 m* ».

De plus, ce type de problème peut présenter des changements de contexte : comme mentionné précédemment, le nombre de bicyclettes louées évolue avec le temps et la popularité du service. De même, les tendances de location de bicyclettes au sein d'une ville peuvent varier d'un quartier à l'autre. Nos méthodes d'adaptation trouvent donc leur utilité dans le cadre de changements de contexte liés au temps et/ou à l'espace.

Bibliographie

Blockeel, Hendrik et Luc De Raedt (1998). "Top-Down Induction of First-Order Logical Decision Trees". In : *Artif. Intell.* 101.1-2, p. 285–297.

BREIMAN, Leo (2001). "Random Forests". In : *Machine Learning* 45.1, p. 5–32. DOI : 10.1023/A:1010933404324.

BREIMAN, Leo, J. H. FRIEDMAN, R. A. OLSHEN et C. J. STONE (1984). *Classification and Regression Trees*. Wadsworth. ISBN : 0-534-98053-8.

ELKAN, Charles (2001). "The Foundations of Cost-Sensitive Learning". In : *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*. Sous la dir. de Bernhard NEBEL. Morgan Kaufmann, p. 973–978. ISBN : 1-55860-777-3.

FANAEE-T, Hadi et João GAMA (2014). "Event labeling combining ensemble detectors and background knowledge". In : *Progress in AI* 2.2-3, p. 113–127. DOI : 10.1007/s13748-013-0040-3.

FÜRNKRANZ, Johannes (2002). "Round Robin Classification". In : *Journal of Machine Learning Research* 2, p. 721–747. URL : http://www.jmlr.org/papers/v2/fuernkranz02a.html.

HALL, Mark A., Eibe FRANK, Geoffrey HOLMES, Bernhard PFAHRINGER, Peter REUTEMANN et Ian H. WITTEN (2009). "The WEKA data mining software : an update". In : *SIGKDD Explorations* 11.1, p. 10–18. DOI : 10.1145/1656274.1656278.

KROGEL, M.-A. et S. WROBEL (2003). "Facets of Aggregation Approaches to Propositionalization". In : *Work-in-Progress Track at the Thirteenth International Conference on Inductive Logic Programming (ILP)*. Sous la dir. de T. HORVATH et A. YAMAMOTO.

LAVRAC, Nada et Saso DZEROSKI (1994). *Inductive logic programming - techniques and applications*. Ellis Horwood series in artificial intelligence. Ellis Horwood. ISBN : 978-0-13-457870-5.

LICHMAN, M. (2013). *UCI Machine Learning Repository*. URL : http://archive.ics.uci.edu/ml.

MITCHELL, Tom M. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill. ISBN : 978-0-07-042807-2.

MORENO-TORRES, José G., Xavier LLORÀ, David E. GOLDBERG et Rohit BHARGAVA (2013). "Repairing fractures between data using genetic programming-based feature extraction : A case study in cancer diagnosis". In : *Inf. Sci.* 222, p. 805–823. DOI : 10.1016/j.ins.2010.09.018.

MORENO-TORRES, José G., Troy RAEDER, Rocío ALAÍZ-RODRÍGUEZ, Nitesh V. CHAWLA et Francisco HERRERA (2012). "A unifying view on dataset shift in classification". In : *Pattern Recognition* 45.1, p. 521–530. DOI : 10.1016/j.patcog.2011.06.019.

QUINLAN, J. Ross (1993). *C4.5 : Programs for Machine Learning*. Morgan Kaufmann. ISBN : 1-55860-238-0.

VAN ASSCHE, Anneleen, Celine VENS, Hendrik BLOCKEEL et Saso DZEROSKI (2006). "First order random forests : Learning relational classifiers with complex aggregates". In : *Machine Learning* 64.1-3, p. 149–182.

Clément CHARNAY

Enhancing Supervised Learning with Complex Aggregate Features and Context Sensitivity

Résumé

Dans cette thèse, nous étudions l'adaptation de modèles en apprentissage supervisé. Nous adaptons des algorithmes d'apprentissage existants à une représentation relationnelle. Puis, nous adaptons des modèles de prédiction aux changements de contexte.

En représentation relationnelle, les données sont modélisées par plusieurs entités liées par des relations. Nous tirons parti de ces relations avec des agrégats complexes. Nous proposons des heuristiques d'optimisation stochastique pour inclure des agrégats complexes dans des arbres de décisions relationnels et des forêts, et les évaluons sur des jeux de données réelles.

Nous adaptons des modèles de prédiction à deux types de changements de contexte. Nous proposons une optimisation de seuils sur des modèles à scores pour s'adapter à un changement de coûts. Puis, nous utilisons des transformations affines pour adapter les attributs numériques à un changement de distribution. Enfin, nous étendons ces transformations aux agrégats complexes.

Mots-clés : Fouille de données relationnelles, Reframing, Agrégation complexe, Optimisation stochastique, Classification sensible au coût, Adaptation de modèles, Apprentissage automatique, Intelligence artificielle

Abstract

In this thesis, we study model adaptation in supervised learning. Firstly, we adapt existing learning algorithms to the relational representation of data. Secondly, we adapt learned prediction models to context change.

In the relational setting, data is modeled by multiples entities linked with relationships. We handle these relationships using complex aggregate features. We propose stochastic optimization heuristics to include complex aggregates in relational decision trees and Random Forests, and assess their predictive performance on real-world datasets.

We adapt prediction models to two kinds of context change. Firstly, we propose an algorithm to tune thresholds on pairwise scoring models to adapt to a change of misclassification costs. Secondly, we reframe numerical attributes with affine transformations to adapt to a change of attribute distribution between a learning and a deployment context. Finally, we extend these transformations to complex aggregates.

Keywords : Relational Data Mining, Reframing, Complex Aggregation, Stochastic Optimization, Cost-Sensitive Classification, Model Adaptation, Machine Learning, Artificial Intelligence