Università degli Studi di Ferrara

Università degli Studi di Ferrara

Corso di Laurea in Informatica

A Laravel package for systematic construction of symbolic intelligent systems

Relatore:

Prof. Guido Sciavicco

Correlatori:

Giovanni Pagliarini

Prof. Giacomo Piva

Laureando:

Alberto Paparella

Anno Accademico 2020 – 2021

Contents

Riassunto

Pitòn è un pacchetto sviluppato per il framework Laravel che offre strumenti di machine learning interpretabile per la risoluzione di problemi organizzati in struttura gerarchica a partire dai dati presenti in un database MySQL.
Scopo delle seguenti pagine è descrivere la struttura e le caratteristiche di questi problemi gerarchici e l'approccio del pacchetto alla risoluzione di questi per mezzo della creazione di modelli basati su regole (e quindi facilmente interpretabili dall'utente umano), di cui ci impegneremo a descriverne l'utilità.
Verranno inoltre discusse le tecnologie utilizzate e presentati degli esempi applicativi del pacchetto, allo scopo di dimostrarne, fra le altre cose, la generalità, e quindi la vastità dei campi e delle modalità in cui può essere utilizzato: dal suggerimento automatico di inserzioni pubblicitarie in un social network, al suggerimento delle terapie per i medici sulla base dei pazienti serviti in passato, ai problemi di classificazione, come per esempio quello di classificare specie e famiglie di fiori.
Varie sono le caratteristiche che rendono unico e innovativo questo pacchetto:

- la scelta di svilupparlo utilizzando come linguaggio principale PHP, sicuramente meno efficiente di altri per quanto riguarda l'ambito del machine learning, ma che rimane tutt'oggi il linguaggio più usato in ambito web;
- la possibilità di poter specificare direttamente al pacchetto come leggere da un database MySQL i dati da cui estrarre i modelli basati su regole;
- la capacità di poter derivare e gestire gerarchie di problemi.

Chapter 1

Introduction

Pitòn is a Laravel package which offers machine learning utilities for solving hierarchically-arranged problems, where the available data is stored in a MySQL database, through the creation of rule-based models. Therefore, the intent of the first chapters will be to discuss what is a hierarchical structured problem, what are rule-based models, what are rules, and what algorithms are used to extract them.
We will then discuss the technologies used by the package, starting from the Laravel framework, that is, the environment in which the package has been developed, and introducing the machine learning Python packages Scikit-Learn and Wittgenstein.
Chapter 5 will finally present how the package works, including its structure, the structure of the database used to store the models, which we decided to separate from the application database, and how to use it at its best.
Then we will present a possible application of the package to solve a real problem, that is, a social network automatic advertisement system which decides what ads to show the user based on the researches of other users with similar characteristics, also going through a possible package configuration.
Last but not least, we will discuss the environment which already implements the package, that is, the University of Ferrara's Research Center for the Study

of Menopause and Osteoporosis platform, serving as a clinical decision support system and therefore allowing physicians to receive suggestions for prevention and treatment for osteoporosis, given their previous choices for similar patients.

Chapter 2

Acknowledgements

I would like to express my deep gratitude to Professor Guido Sciavicco, for orchestrating this thesis and for inspiring my interest in the development of explainable AI, and Dr. Giovanni Pagliarini, for his patient guidance and his valuable and constructive suggestions during the development of this work. You are both a great inspiration for me.
I would also like to extend my thanks to Professor Giacomo Piva, for giving me the opportunity to work on this project, and Dr. Andrea Bercé, as well as all the staff of the CMO project.
I would also like to offer my special thanks to Dr. Eduard Ionel Stan, for also inspiring me in taking the path of research.
Finally, I wish to thank my parents and my girlfriend, for their support and encouragement throughout these years, and my dear roommate Filippo.

Chapter 3

Preliminaries

The purpose of this chapter is to make an introduction of the mathematical concepts the package is built on: hierarchical problems, rule-based models, rules and learners. We will introduce some examples for a better understanding, and some important properties for some of these objects that are used by the package implementation for a better efficiency.

3.1 Hierarchical problems

The primary purpose of the package is the solution of *classification problems* arranged in hierarchies, easily representable as trees, using binary classification for each problem. The package refers to a hierarchy of sub-problems as **problem**.
Let's look at an example: Figure 3.1 represents an example of a hierarchy of problems associated with the classification of flowers belonging to the Iridaceae family. We can empathise how:

- at the first level of the hierarchy the classification is on the flower genre, and the domain of the classification is: {Crocosmia, Ferraria, Iris, Moraea, Tigridia} [1];

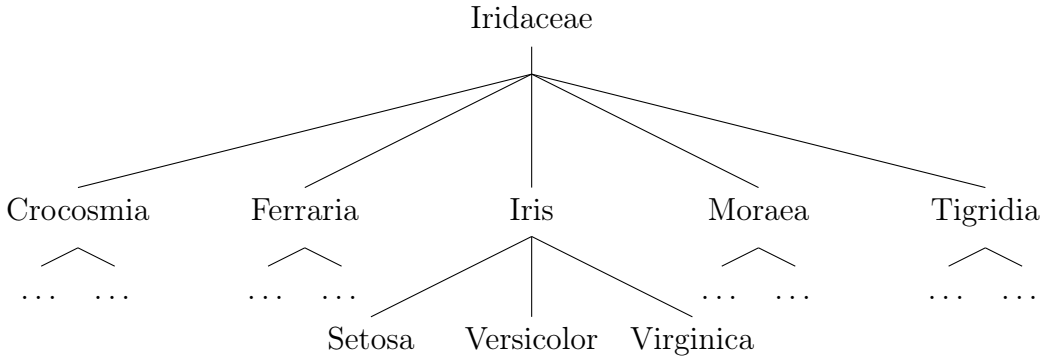

Figure 3.1: Hierarchy of problems for Iridaceae classification

- at the second level of the hierarchy the classification is based on the flower species, and for example for the Iris genre the species domain will be {Setosa, Versicolor, Virginica} [2].

3.2 Learners

To solve each binary sub-problem, we make use of machine learning algorithms, and in the package we refer to them as **learners**.
Machine learning offers many learning algorithms, and these are divided into many subareas [3]: supervised learning, unsupervised learning, reinforcement learning, semi-supervised learning, inductive learning, and so on.
Specifically, the package can be placed in the *supervised learning* area, making use of *labeled data*. This kind of learning characterises a class of problems involving the use of an algorithm to learn a mapping between input examples and the target variable. The result of this process is a model *fit* on training data, comprised of inputs and outputs, and used to make predictions on test sets, where only the inputs are provided and the outputs from the model are compared to the withheld target variables and used to estimate the accuracy of the model.
We distinguish between two main types of supervised learning: **regression**, which involves predicting a numerical label, and **classification**, which involves predicting a class label. During the development of the package, we decided to focus on the last one; however, the whole package can easily be extended to support regression problems as well.

For medical and research needs, we focused on interpretable models, in particular on rule-based models. These algorithms read from a given dataset and extract information about the classification on one class, giving as a result a **Rule-Based Model**, that is, an ordered set of rules, where rules are sets of conditions that characterise a class. Given a new instance similar to the ones in the dataset, using the model it is possible to predict the label associated with the new instance.

3.3 Rules

A rule is a formula of the type:

$$\rho : p_1 \wedge ... \wedge p_n \rightarrow \mathcal{C}$$

where p_1, ..., p_n are logical propositions over the attributes of the problem, and $\mathcal{C}$ is a class. Each p_i is called a *decision* or *antecedent*, and its format depends on the particular type of attribute on which is taken.
In fact, each antecedent is a tuple of:

$$p_i : (attribute, operator, value)$$

Specifically, we distinguish between continuous and discrete antecedents based on the associated attribute type:

- continuous antecedents are based on continuous or numerical attributes, and support conditional operators such as $>$, $\geq$, $<$, $\leq$;
- discrete antecedents are based on discrete or categorical attributes, that are attributes associated to a domain of possible values; they support conditional operators such as $=$ and $\neq$.

$\mathcal{C}$ is also called a *consequent*, and usually corresponds to a value of the domain of a particular categorical attribute, denoted as *class attribute*.
In other words, given the attributes of a problem, it is possible to choose a categorical attribute among them, and classify each instance based on its domain. We can refer to this attribute as *class attribute*, to its domain as $\{\mathcal{C}_1, ..., \mathcal{C}_m\}$ and to each value of the domain as a *class*.
Among all classification problems, important for us will be *binary classification*

problems, that are, problems where the class attribute has exactly two values in its domain, which can therefore be translated as {true, false}. We refer to these attributes as *binary categorical attributes*. In fact, every classification problem with more than two classes (*multiclass problem*) can be translated to a set of binary classification problems, one for each attribute of the original domain.
In designing the package, we chose to translate every multiclass problem into a set of binary classification problems, both for simplicity and, in some cases, to meet machine learning algorithms' constraints (i.e. Wittgenstein learners).
It is also important for our discussion to specify the following rule, needed to satisfy other machine learning algorithms' constraints (that are, Scikit-Learn learners, that require all attributes to be numerical): given a binary categorical attribute of domain *{value₁, value₂}*, it can be transformed into a numerical attribute of domain *{0, 1}*. This new attribute will assume, for each instance, value of *0* if the instance had *value₁* as a value, *1* otherwise. It is then possible to transform the associated antecedent into a categorical antecedent associated to the original categorical attribute of domain *{value₁, value₂}*. This new antecedent will assume value *value₁* if the numerical antecedent had *value <= 0.5*, *value₂* otherwise.
Doing so, we can translate all categorical attributes into numerical attributes before the learner is called, and then translate their relative antecedents into categorical antecedents associated to the original categorical attribute, so that we can use it at prediction time.

3.4 Rule-based models

A rule-based model is an ordered set of rules:

$$\Gamma = \begin{cases} p_1^1 \wedge ... \wedge p_n^1 \rightarrow \mathcal{C}_1 \quad else \\ p_1^2 \wedge ... \wedge p_n^2 \rightarrow \mathcal{C}_2 \quad else \\ ... \\ p_1^n \wedge ... \wedge p_n^n \rightarrow \mathcal{C}_n \end{cases}$$

In our discussion we will also refer to a model as a *classifier*. This is because given a new instance having at least the same attributes as the one found in the rules, the model can *classify* it, it can predict its class attribute value. Furthermore, it can

give information about the activated rule, motivating why it chose a specific class, and even some information about the reliability of the prediction if the model has been tested on a test dataset.
For our discussion, we will focus on *binary* rule-based model, that is, a model which classifies on a binary categorical class attribute. In other words, a binary rule-based classifier classifies the given instances on a single class, and given a new instance it will predict if it belongs or not to that class.
These models can be seen as follows:

$$\Gamma = \begin{cases} p_1 \vee ... \vee p_n \rightarrow \mathcal{C} & else \\ \neg\mathcal{C} \end{cases}$$

or, with an extended form:

$$\Gamma = \begin{cases} p_1^1 \wedge ... \wedge p_n^1 \rightarrow \mathcal{C} & else \\ p_1^2 \wedge ... \wedge p_n^2 \rightarrow \mathcal{C} & else \\ ... \\ p_1^n \wedge ... \wedge p_n^n \rightarrow \mathcal{C} & else \\ \neg\mathcal{C} \end{cases}$$

where the rule $p_1 \vee ... \vee p_n \rightarrow \mathcal{C}$ is equal to $(p_1^1 \wedge ... \wedge p_n^1) \vee ... \vee (p_1^n \wedge ... \wedge p_n^n) \rightarrow \mathcal{C}$.
In other words, binary rule-based models work as follows:

1. try the first rule, if it activates predict $\mathcal{C}$, if not try the following rule, and so on;

2. if all rules fail, then predict $\neg\mathcal{C}$, such as the instance is not of class $\mathcal{C}$.

The first $p_1^1 \wedge ... \wedge p_m^1$ antecedents of each rule (excluding the first rule), with $m < n$, correspond to the negation of all previous rules. So for example, the third rule for a binary rule based model can be seen as $p_1^3 \wedge ... \wedge p_n^3 \equiv \neg(p_1^1 \wedge ... \wedge p_n^1) \wedge \neg(p_1^2 \wedge ... \wedge p_n^2) \wedge p_{m+1}^3 \wedge ... \wedge p_n^3$.
Note that this brings another feature that we took for granted so far: only one rule can activate for each model. In fact, the final rule which predicts $\neg\mathcal{C}$ is not an empty rule! It is rather the conjunction of the negations of all previous rules.
These properties will come in handy for the package, as we will simplify the rules

by storing, for each one, only its new antecedents, taking advantage of the in-order exploration of the rules, knowing that if we are exploring the *i+1* rule, the first *i* rules have already been discharged.

3.5 Rule-extraction

Within the package, two are the main algorithms supported at the time of writing: the RIPPERk [4] algorithm and the CART [5] algorithm. The RIPPERk algorithm is implemented in three, slightly different ways, one fully developed in PHP, called PRIP, and two developed in Python and offered by the Wittgenstein Python package, IREP and RIPPERk. Meanwhile, an optimised version of the CART algorithm is offered by the Scikit-Learn package. In this section we will briefly discuss how these algorithms work and how they perform rule-extraction.

3.5.1 RIPPERk

The RIPPERk algorithm, originally proposed by Cohen, is a propositional rule learning algorithm that performs well on large, noisy datasets, and scales nearly linearly with the number of training examples. It is used in order to solve binary classification problems.
As a separate and conquer algorithm, it builds a rule set in a greedy fashion, one rule at a time. After a rule is found, all examples covered by the rule, both positive and negative, are deleted. This process repeats until some stopping condition is satisfied. After the initial rule set is acquired, it is then optimised.
In order to build a rule, RIPPERk uses the following strategy:

1. It randomly partitions all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.

2. It *grows* a rule by greedily adding conditions until the rule reaches an accuracy of 100%, so that the rule does not cover any negative example, using only the growing set, testing every possible value for each attribute and choosing the condition with the greatest information gain.

3. To prevent growing set *overfitting*, the algorithm immediately prunes the rule so as to maximize its performance on the pruning data. The pruning

considers deleting any final sequence of conditions from the rule and chooses the deletion that maximizes some function w. This process is repeated until no deletion improves the value of w.

4. Lastly, all the positive and negative examples covered by the rule are removed, and the algorithm repeats from the step 1.

5. RIPPERk stops adding rules when there are no more positive examples left or when a rule has an unacceptably large error rate, or when the last rule added is too *complicated*, such as the description length is more than 64 bits larger than the smallest description encountered so far.

The ruleset R produced by the learning algorithm is then taken as a starting point for a subsequent optimization process. This process re-examines all the rules in the same order they have been learned, and for each rule constructs two alternative rules, one called *replacement rule* r_p and the other referred to as *revision rule* r_v. The replacement rule is created by growing and the pruning a rule r_i from the ground up. The revision rule is created in a similar fashion, except that the revision is grown by greedily adding conditions to r_i rather that the empty rule.
To decide which version between r_v and r_p to retain, the **Minimum Description Length** criterion is used. Lastly, the rules are added to cover any remaining positive examples using the building stage.
The optimization stage can be reiterated again k times. The ruleset is then simplified by examining each rule in turn (starting with the last added rule) and deleting rules so as to reduce total description length.

3.5.2 CART

The Classification And Regression Trees algorithm, often abbreviated into CART, is a classification algorithm for building decision trees. It is based on Gini's impurity index splitting criterion, which is defined for a generic node t as:

$$i(t) = \sum_{i,j} C(i|j)p(i|t)p(j|t),$$

where $C(i|j)$ represents the cost of misclassifying a class j case as a class i case and $p(i|t)$ and $p(j|t)$ are the probabilities to be into a i or into a j case, respectively.

In the CART algorithm, $C(i|j) = 1$ if $i \neq j$, and $C(i|j) = 0$ if $i = j$.
The algorithm builds a binary tree splitting each node into two child nodes repeatedly using the following steps:

1. For each feature (i.e. each attribute) with K different values, there exist K-1 possible splits; choose for each feature the split that maximizes the splitting criterion.

2. Among the best splits from step 1 choose the one which maximizes the splitting criterion.

3. Split the node using the best node split from step 2 and repeat from step 1 until a certain stopping criterion is satisfied.

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied. The pruning algorithm is based on a number of folds N and consists of the following steps:

1. Split randomly training data into N folds.

2. Select a pruning level for the tree (level 0 equals to the full decision tree).

3. Use $N - 1$ folds to create $N - 1$ new pruned trees and estimate the error on the Nth fold.

4. Repeat from step 2 until all pruning levels are used.

5. Find the smallest error and use the pruning level assigned to it.

6. Until the pruning level is reached, remove all the leafs in the lowest tree level and assign the decision class to their parent node. The decision value is equal to the class with the higher number of cases covered by the node.

Chapter 4

Technologies

Initially developed as a PHP vanilla project, Pitòn found its best fit as a Laravel [6] package. More in detail, it has been developed as a Laravel 6.* package, which at the time of writing is a reasonably stable version and is easily supported by the following versions. The PHP distribution in use is version 7.1.
To offer a wider choice of learning algorithms, the package makes use of two Python packages for machine learning: Scikit-Learn [8] and Wittgenstein [12]. Their use is, of course, optional to the user, having the package its own learner, but is surely advised, offering a wider choice of models for the same sub-problem and therefore the possibility to switch between them in case of higher accuracies for specific cases. The meaning of this chapter is to briefly introduce these technologies, going through their configuration and overviewing their fundamental components, especially the ones needed by the package.

4.1 Laravel

For the Laravel framework, the focus will be on the structure and the configuration of a project [7] rather than how to use the framework and its tools for web developing, such as models, controllers, migrations, Blade templates, and so on.

The purpose of this choice is to give an introduction of the files and commands needed or used by the package, rather than explaining how to create a web application, which is not the scope of this document.

4.1.1 What is Laravel

Laravel is a web application framework with an expressive, elegant syntax. Its purpose is to ease common tasks used in most web projects, providing a structure and starting point for creating modern, full-stack web applications, as well as powerful features such as thorough dependency injection, an expressive database abstraction layer, queues and scheduled jobs, and unit and integration testing.
Laravel is often referred to as a *progressive* framework. In fact, it gives senior developers robust tools for dependency injection, unit testing, queues, real-time events, and more. Moreover, it is fine-tuned for building professional web applications and ready to handle enterprise work loads.
Laravel is also incredibly scalable, thanks to the scaling-friendly nature of PHP and Laravel's built-in support for fast, distributed cache systems like Redis. In fact, Laravel applications have been easily scaled to handle hundreds of millions of request per month. Should not this be enough, platforms like Laravel Vapor allows to run Laravel applications at nearly limitless scale on AWS's latest serverless technology. Furthermore, Laravel combines a large number of packages in the PHP ecosystem to offer a robust and developer friendly framework, and developers from all around the world can contribute to the framework.

4.1.2 Creating a Laravel project

There are a variety of options for developing and running a Laravel project, one of these being Sail, a built-in solution for running Laravel projects using Docker.
Docker is a tool for running applications and services in small, light-weight *containers* which do not interfere with local computers' installed software or configuration. This means the developer does not have to worry about configuring or setting up complicated development tools such as web servers and databases on his personal computer.
Laravel Sail is a light-weight command-line interface for interacting with Laravel's default Docker configuration. It provides a great starting point for building a Lar-

avel application using PHP, MySQL, and Redis without requiring prior Docker experience. Everything about Sail can be customized using the docker-compose.yml file included with Laravel.
Alternatively, it is possible to create a new Laravel project by using Composer via the following command:

```
composer create-project laravel/laravel example-app
```

After the application has been created, it is possible to start Laravel's local development server using the Artisan CLI's server command:

```
php artisan serve
```

4.1.3 Project configuration

All the configuration files for the Laravel framework are stored in the project's `config` directory. Laravel needs almost no additional configuration out of the box. However, it is good practice to review the `config/app.php` file and its documentation, which contains several options such as timezone and locale that are often changed according to the application.
Since many of Laravel's configuration option values may vary depending on whether an application is running on a local computer or on a production web server, many important configuration values are defined using the `.env` file that exists at the root of the application. This `.env` file should not be committed to the application's source control, since each developer/server using the application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to the source control repository, since any sensitive credentials would get exposed.
Laravel should always be served out of the root of the web directory configured for the web server. The developer should also not attempt to serve a Laravel application out of a subdirectory of the web directory. Attempting to do so could expose sensitive files that exist within the application.

4.1.4 Laravel popular use cases

There are a variety of ways to use Laravel. Laravel may serve as a full stack framework, using Laravel to route requests to the application and render its frontend via Blade templates or using a single-page application hybrid technology like

Inertia.js. This is the most common way to use the Laravel framework. It makes vast use of routing, views, and the Eloquent ORM, and community packages like Livewire and Inertia.js, which allows to use Laravel as a full-stack framework while enjoying many of the UI benefits provided by single-page JavaScript applications. Alternatively, Laravel may also serve as an API backend to a JavaScript single-page application or mobile application, such as a Next.js application. In this context, Laravel can also be used to provide authentication and data storage/retrieval for the application, while also taking advantage of Laravel's powerful services such as queues, emails, notifications, and more. It makes vast use of routing, Laravel Sanctum, and the Eloquent ORM.

4.2 Scikit-Learn

Scikit-Learn is a Python package containing a wide set of tools for machine learning. However, at the moment of writing, the Pitòn package uses only one of these tools, that is its DecisionTreeClassifier, a classifier which implements an optimised version of the CART algorithm.
Scikit-Learn does not support rule-based models. However, it offers another symbolic model, referred to as *decision trees*. Rule-based models and decision trees are equivalent; therefore it is possible to use an algorithm to create a decision tree and then translate it into a rule-based model.
As for Laravel, after a brief overview of the package, the focus will be more on what the Pitòn package actually uses rather than the usage of the Scikit-Learn package and all of its features. However, it is fair to specify that the Pitòn package is developed on the purpose to support, with little changes, more classifiers of the Scikit-Learn package in the future.

4.2.1 What is Scikit-Learn

Scikit-Learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.
It is, in fact, a Python module built on top of SciPy and is distributed under the 3-Clause SBD license.

The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed [9]. It is currently maintained by a team of volunteers, it is accessible to everybody, and it is reusable in various contexts.

4.2.2 How to use Scikit-Learn

Scikit-Learn provides dozens of built-in machine learning algorithms and models, called *estimators* [10]. Each estimator can be fitted to some data using its fit method. The fit method generally accepts 2 inputs:

- The samples' matrix `X`. The size of `X` is typically `(n_samples, n_features)`, which means that samples are represented as rows and features are represented as columns.
- The target values `y` which are real numbers for regression tasks, or integers for classification (or any other discrete set of values). For unsupervised learning tasks, `y` does not need to be specified. `y` is usually a one dimensional array where the ith entry corresponds to the target of the ith sample (row) of `X`.

Both `X` and `y` are usually expected to be numpy arrays or equivalent array-like data types, though some estimators work with other formats such as sparse matrices.

4.2.3 Classification using Scikit-Learn decision trees

Decision trees (DTs) [11] are a supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.
Decision trees are simple to understand and to interpret, in fact they can also be visualized. They require little data preparation, however it is important to notice that the Scikit-Learn module does not support missing values. The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree. They are able to handle both numerical and categorical data. However, Scikit-Learn implementation does not support categorical variables for now. Furthermore, they are a white box model: if a given situation is observable

in a model, the explanation for the condition is easily explained by boolean logic; by contrast, in a black box model (e.g., in an artificial network), results may be more difficult to interpret. It is also possible to validate a model using statistical tests: this makes it possible to account for the reliability of the model. Last but not least, they perform well even if their assumptions are somewhat violated by the true model from which the data were generated.
However, not all that glitters is gold. Decision tree learners can, in fact, create over-complex trees that do not generalise the data well. This is called *overfitting*. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem. Decision trees can also be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble. Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximation.
The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement. Last but not least, decision tree learners create biased trees if some classes dominate. It is therefore recommended balancing the dataset prior to fitting with the decision tree.
`DecisionTreeClassifier` is a class capable of performing multiclass classification on a dataset. As with other classifiers, `DecisionTreeClassifier` takes as input two arrays: an array `X`, sparse or dense, of shape `(n_samples, n_features)` holding the training samples, and an array `Y` of integer values, shape `(n_samples)`, holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X,Y)
```

After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict([[2., 2.]])
array([1])
```

In case that there are multiple classes with the same and highest probability, the classifier will predict the class with the lowest index amongst those classes.
As an alternative to outputting a specific class, the probability of each class can be predicted, which is the fraction of training samples of the class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

`DecisionTreeClassifier` is capable of both binary (where the labels are [-1, 1]) classification and multiclass (where the labels are [0, ..., K-1]) classification.
Using the Iris dataset, it is possible to construct a tree as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, y)
```

Once trained, it is possible to plot the tree with the `plot_tree` function:

```
>>> tree.plot_tree(clf)
```

Alternatively, the tree can also be exported in textual format with the function `export_text`.

4.2.4 A little digression on decision trees learning

ID3 (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. When a tree is grown, a pruning step is usually applied to improve the ability of the tree to generalise to unseen data.
C4.5 is the successor to ID3 and removed the restriction that features must be

categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. The accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it.
C5.0 is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate. CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node. Scikit-Learn uses an optimised version of the CART algorithm; however, Scikit-Learn implementation does not support categorical variables for now. Regarding complexity, the run time cost to construct a balanced binary tree is $\mathcal{O}(n_s n_f \log(n_s))$ and query time $\mathcal{O}(\log(n_s))$, with n_s samples and n_f features.
Although the tree construction algorithm attempts to generate balanced trees, they will not always be balanced. Assuming that the subtrees remain approximately balanced, the cost at each node consists of searching through $\mathcal{O}(n_f)$ to find the feature that offers the largest reduction in entropy. This has a cost of $\mathcal{O}(n_f n_s \log(n_s))$ at each node, leading to a total cost of over the entire trees (by summing the cost at each node) of $\mathcal{O}(n_f n_s^2 \log(n_s))$.

4.3 Wittgenstein

Like Scikit-Learn, Wittgenstein is another Python package which offers rule-based models and two classifiers to extract them. However, this time the Pitòn package offers support for all its algorithms implemented at the time of writing, still being developed to support other Wittgenstein classifiers in case they are added in the future.

4.3.1 What is Wittgenstein

Wittgenstein is a Python package which implements two interpretable coverage-based ruleset algorithms: IREP and RIPPERk, as well as additional features for model interpretation. Performance is similar to Scikit-Learn's DecisionTree CART implementation. Also, the syntax is very similar to Scikit-Learn's.

4.3.2 Classification using Wittgenstein rule-based models

Suppose we have already loaded and split our data, for example as follows:

```
>>> import pandas as pd
>>> df = pd.read_csv(dataset_filename)
>>> from sklearn.model_selection import train_test_split
>>> train, test = train_test_split(df, test_size=.33)
```

To train a RIPPER or IREP classifier, the `fit` method is used:

```
>>> import wittgenstein as lw
>>> ripper_clf = lw.RIPPER() # Or irep_clf = lw.IREP()
>>> ripper_clf.fit(df, class_feat='Poisonous/Edible',
    pos_class='p') # Or pass X and y data to .fit
```

It is then possible to access the underlying trained model with the `ruleset_` attribute, or output with `out_model()`. A ruleset is a disjunction of conjunctions. In other words, the model predicts positive class if any of the inner-nested condition-combinations are all true. IREP models tend to be higher bias, RIPPER's higher variance.

Chapter 5

Implementation

Now that we have introduced all the elements and the technologies the package is built on, it is finally time to see how all comes together.
We will first discuss how the package is structured, and how its parts work together to solve classification problems.
Then, we will take a look on the database the package uses. In fact, to maintain information about the created rule-based models, Pitòn will rely on a different MySQL database from the one containing the application information, to keep things clean.
Last but not least, we will finally discuss how the package can be configured and used. The next chapters will serve as an example.

5.1 Package structure

Being a Laravel package, Pitòn lies on the general structure provided by the framework, which is also very similar to the structure of Laravel's web applications.
The main folders we find on a first level are `config`, `database`, `src` and `tests`.
Figure 5.1 is a simple representation of the package structure.

- config
  - iris.php
  - prip.php
  - problem.php
  - sklearn_cart.php
  - wittgenstein_irep.php
  - wittgenstein_ripperk.php
- database
  - factories
    - ...
  - migrations
    - 2021_04_08_000000_create_piton_class_model_table.php
    - 2021_04_15_000000_create_piton_model_version_table.php
    - 2021_06_07_000000_create_piton_problems_table.php
    - 2021_06_07_000000_create_piton_rules_table.php
- src
  - Antecedents
    - Antecedent.php
    - ContinuousAntecedent.php
    - DiscreteAntecedent.php
  - Attributes
    - Attribute.php
    - ContinuousAttribute.php
    - DiscreteAttribute.php
  - Console
    - CreateExample.php
    - PredictByIdentifier.php

    - UpdateModels.php
    - UpdateModelsWithInterface.php
- DBFit
    - DBFit.php
- DiscriminativeModels
    - DiscriminativeModel.php
    - RuleBasedModel.php
- Examples
    - createIrisDataset.py
    - iris.csv
- Facades
    - Piton.php
    - Utils.php
- Instances
    - Instances.php
- Learners
    - PythonLearners
        - lib.py
        - sklearn_learner.py
        - wittgenstein_learner.py
    - Learner.php
    - PRip.php
    - SklearnLearner.php
    - WittgensteinLearner.php
- RuleStats
    - RuleStats.php
- Rules
    - ClassificationRule.php

        * RipperRule.php
        * Rule.php
    - ClassModel.php
    - ModelVersion.php
    - Piton.php
    - PitonBaseServiceProvider.php
    - Problem.php
    - Utils.php

- tests
    - ...

Figure 5.1: Package Structure

5.1.1 The config folder

The `config` folder, as the name suggests, contains the configuration files that the user can import in its project via the publishing command:

```
php artisan vendor:publish --tag=\<config-file-name\>
```

Pitòn offers a configuration file for each of its learners, so that the experienced user can change some properties of the execution, for example specifying the maximum depth for a decision tree, or the maximum number of RIPPERk iterations or maximum number of rules, and so on. However, these files do not need to be changed in order for the package to run, as they already offer a general standard configuration for most cases. They are extensively documented in file, so that the user can easily know what every property does and what values support. These files are `prip.php`, `sklearn_cart.php`, `wittgenstein_irep.php` and `wittgenstein_ripperk.php`. An example of publishing for one for these in a project is:

```
php artisan vendor:publish --tag=sklearn_cart
```

which imports the config file `sklearn_cart.php` in the `config` folder of the user's Laravel project. It is needed to import a learner configuration file before using

that algorithm, even if no changes are to be applied.
Together with these, `problem.php` will be the fundamental configuration file for the package to work. In this file, the user can specify how the package will read from the database, specifying among other things which tables to use and how to join them, which columns to use and even how to treat missing information on these columns (i.e. `NULL` values), which columns represent the classification attributes and on which level, and so on.
Differently from the learners' configuration files, we do not provide a general configuration for `problem.php`, being really dependent on the application. However, as the learners' configuration files, it is heavily documented. Furthermore, an example of configuration is represented by the file `iris.php`.

5.1.2 The database folder

The `database` folder contains two folders: `factories` and `migrations`. We will only discuss the `migrations` folder, as `factories` only contains files used for testing purpose, useful in the beginning of the development but at the moment of writing deprecated.
The `migrations` folder contains all the information about the structure of the database of the package. In fact, each file corresponds to a table in the Pitòn database:

- `2021_04_08_000000_create_piton_class_model_table.php` contains information about the *piton_class_model table*, which is used to store information about the models.

- `2021_04_15_000000_create_piton_model_version_table.php` contains information about the *piton_model_version table*, which is used to store information about the hierarchies and corresponds to one instance for each execution; each instance of *piton_model_version* is associated with many instances of *piton_model_class*.

- `2021_06_07_000000_create_piton_problems_table.php` contains information about the problems the package has associated a hierarchy so far; every instance can be associated with many instances of *piton_model_version*, as

we can have different executions on the same problem with new data, for example repeating the learning process at scheduled periods.

- `2021_06_07_000000_create_piton_rules_table.php` finally contains information about the rules of each model.

These tables will be discussed in the next section.

5.1.3 The src folder

Like the `app` folder for a Laravel project contains the hearth of an application, the `src` folder contains all the working bricks of the package.
The `Attributes` folder contains the logic behind attributes, split into two classes: one for continuous attributes and one for discrete attributes, both extending the abstract class `Attribute`. In fact, we offer the same operations for both classes, such as methods to create an attribute, to get information about its content, translate it into a json format to store it into the database, and so on, but they differ in structure (as seen in par. 3.3). For instance, a discrete attribute will have a domain, while a continuous attribute will not.
The `Antecedents` folder contains the logic behind antecedents, split in class with the same structure of the attributes. Antecedents and attributes are, as discussed in par. 3.3, very connected: an antecedent, in fact, will contain an attribute as an attribute of the class, which will be continuous for continuous antecedents and discrete for discrete attributes. Continuous attributes and discrete attributes support different operators.
The `Instances` folder contains the specification of the `Instances` class and all its methods, that is, the way of a package to define a dataset, an object which contains information about the attributes (even with their domain, in case they have one), the instances, and for each instance an id and a weight. It can be seen as a table containing all the problem data, with the first column being the class attribute and the last column being the weight, and moreover storing information about the table meta-data (such as information about the attributes). Normally, the package will read from a database and create an associated Instances object for each sub-problem. This is also the object passed to learners to create rule-based models. If the learner is an external learner written in Python, the Instances object is passed to the Python script storing it temporarily into the Pitòn database,

in a table which will be deleted as soon as the job is done.
The `Rules` and `RuleStats` folders contain all the information about the rules. The rules created by the learners are usually objects of the class `ClassificationRule`, while the rules created by `PRip` learner are stored as `RipperRule` objects.
The `DiscriminativeModels` folder contains the class `DiscriminativeModel` and it's subclass `RuleBasedModel`, which is the representation of a rule-based model. Each learner execution retrieves an object of `RuleBasedModel`, which contains an array of rules (objects of the `ClassificationRule` class or its extensions) and an array of associated attributes, so that it is possible to compare it to new instances to predict their classification.
The `Learners` folder contains all the implementation of the various machine learning algorithms the package supports and scripts to eventually call external learners. At the moment of writing, `PRip`, an implementation of the RIPPERk algorithm fully developed in PHP, is the only learner defined, while the other classes `SKlearnLearner` and `WittgensteinLearner` are used to support Scikit-Learn and wittengstein's learners. In fact, their methods are used to pass an instances object to the relative Python script through the Pitòn database, as well as all the eventual parameters for the learning process, and translate the information given by the script into a common format, so that the rule-based models created and stored by the package have all the same structure. Each PHP class which supports an external learner has its own associated Python script, which will simply do some simple dataset pre-processing to make it acceptable from the Python learner, and execute them as seen in par. 4.2.3 and 4.3.2 with the parameters passed from the PHP class.
The `DBFit` folder contains the `DBFit` class, that is, the main class of the package. In fact, it implements the methods to read from a database as specified into the problem configuration file, create the hierarchy of the problems, launching for each problem the learner to create, if possible, a rule-based model, store information into the database, and moreover the methods to make predictions.
The `Console` folder contains the definitions of all the package command line commands. The `UpdateModels` command accepts as parameters a user id, a problem name (the same name of the config file associated with the hierarchical problem, without the extension), and a learner name (and its specific algorithm in case it supports different implementations, such as Wittgenstein learner's RIPPERk and

IREP). `PredictByIdentifier` asks for an instance id and make prediction about its classification, using the last hierarchy of problems stored into the database (while the predictByIdentifier function of the DBFit class asks for a specific hierarchy id; in fact, in most web applications, the user would use this method, as it retrieves the information in json format easily representable on a web page). `CreateExample` is used to create a simple environment to try out the package, as it will create an easy dataset, with only one possible classification problem, into the user's database, using the `createIrisDataset.py` script and the information contained into the `iris.csv` file of the `Examples` folder.
Finally, the `Facades` folder contains the specification for the singleton classes, that are, classes containing only static methods, such as `Utils` and `Piton`.
`ClassModel.php`, `ModelVersion.php` and `Problem.php` are the Laravel models associated with the relative Pitòn tables into the Pitòn database, while `Utils.php` contains generally useful PHP methods that can be used in any class and `Piton.php` and `PitonBaseServiceProvider.php` contain the classes used to specify the package properties (such as commands, configuration files, etc.), as defaulted by Laravel.

5.1.4 The tests folder

Finally, `tests` contains some unit tests used at the beginning of the development. We used the Orchestra testing environment provided by Laravel for the development of packages.
As a general overview, the folder contains a `TestCase.php` file to overwrite some properties of the Orchestra TestCase class, for example to define a testing database to be used during the unit tests. We then find two folders: `Arff` and `Feature`. `Feature` contains various PHP files, each one of them used to group various unit tests on similar features. For instance, `SklearnLearnerTest.php` will test the creation of a model using decision trees. `Arff` contains some datasets stored in files with `.arff` extension. Doing so, we separated the tests concerning the interaction with the database and the tests concerning learners. In fact, for example, `SklearnLearnerTest.php` will not read from a database and then create a rule-based model, instead it will read information from a `.arff` file such as `Iris.arff`, which is ready to be used for learning.

5.2 Package database

The package stores information about the extracted rule-based models in its own database. As we will see in par. 5.3.2, it is important to declare this new connection into the `config/database.php` file and to declare how to access this database into the `.env` file as a second database. Note that the application database and the package database can be the same, however it is not advised.
This way, the package will read the data needed for the extraction process from the primary database (i.e. the application database) and store the extracted rule-based models, as well as other useful information, such as information about the hierarchy of problems and about the parameters of the execution, into a second database.
At prediction time, the package will read the instance to be classified from the main database based on a given id, while it will import the hierarchy of problems and their respective rule-based models from the second database.
Figure 5.2 represents the structure of the package database.
The *Problems* table stores information about the problems solved by the package so far and how the package read from the database to extract the rule-based models (i.e. on which tables, on which columns, which column provides the IDs for the instances, and so on).
The *Model version* table stores information about a single execution of the package learning process: for each execution, it stores the extracted hierarchy of problems, who launched the execution, what learner has been used, as well as parameters given to the execution.
Every instance of *Problems* can be associated with many instances of *Model version*, as it is possible to have more executions for the same problem, for example extracting a new hierarchy and updating the models when new data is provided.
Every instance of *Model version* is associated with only one problem.
The *Class model* table stores information about the extracted rule-based models, such as its level in the hierarchy, and eventually its father node, the class attribute used for the classification, the extracted rules and many test results. In fact, to *Test results** in the *Class model* table correspond many columns, each one representing a test value on the built model. We will now discuss only the most important among them.

We consider: *True Positive (TP)*, instances correctly predicted to be *true*; *False Positive (FP)*, instances predicted to be *true* but which where, in fact, *false*; *True Negative (TN)*, instances correctly predicted to be *false*; *False Negative (FN)*, instances predicted to be *false* but which where, in fact, true.
Then we can calculate:

- *Accuracy (ACC)*, which describes the accuracy of the model based on how many instances have been correctly classified on the total of tried instances:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

- *Sensitivity (TPR)*, which describes only the proportion of *true* values correctly classified:

$$TPR = \frac{TP}{TP + FN}$$

- *Specificity (TNR)*, which describes the proportion of *false* values correctly classified:

$$TNR = \frac{TN}{TN + FP}$$

- *Precision or Positive Predictive Value (PPV)*, which measures the proportion of instances correctly classified to be *true* in relation with all the instances classified to be *true*:

$$PPC = \frac{TP}{TP + FP}$$

- *Negative Predictive Value (NPV)*, which measures the proportion of instances correctly classified to be *false* in relation with all the instances classified to be *false*:

$$NPV = \frac{TN}{TN + FN}$$

Every instance of *Model version* is usually associated with many instances of *Class model*, as every execution of the package is supposed to extract a hierarchy of subproblems and each one of them will be associated with a rule-based model.
Finally, the *Rules* table contains information about the rules, such as its antecedents, its consequent, and various test results such as covered, support, confidence, lift and conviction. Each instance of *Class model* can be associated with many rules.

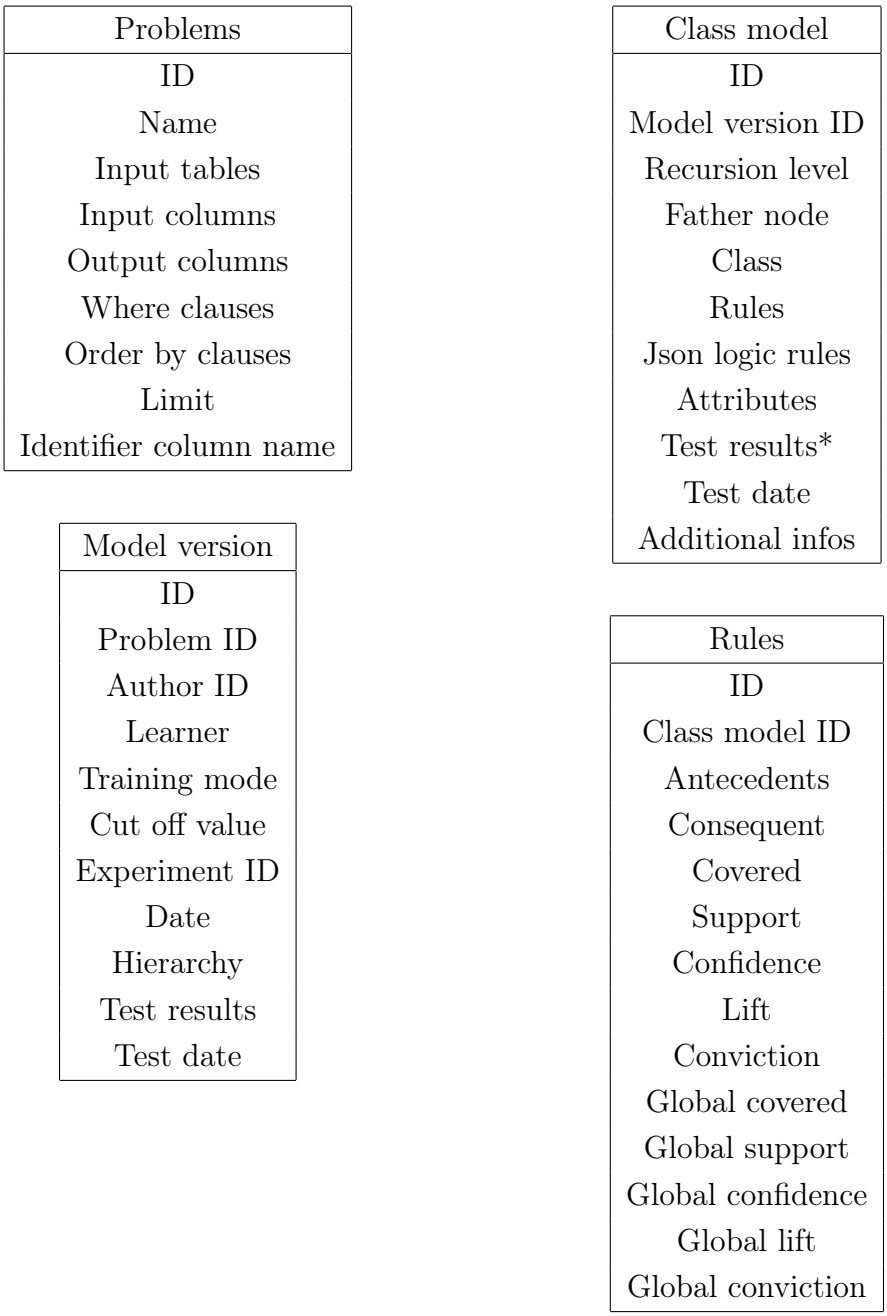

Figure 5.2: Database structure

5.3 How to use the package

We will now discuss how to use the package in a general Laravel application, going through its installation process, how to configure the Pitòn database in the project, how to use the package to extract rule-based models and to make predictions and which commands and files it needs in order to do so.

5.3.1 Install

The first thing to do to use the package is, of course, to install in within a Laravel application. In order to do so, once a project has been created and configured as seen in par. 4.1.2 and 4.1.3, the package must be declared into the `composer.json` file. This way, every time the `composer update` command is launched, the package will automatically be imported, or updated.
Specifically, one should add the GitHub project as a repository in `composer.json` as follows:

```
"repositories": [
        {
                "type": "vcs",
                "url": "https://github.com/aclai-lab/
                   piton"
        }
]
```

And then add the package in the require section:

```
"require": {
        "aclai/piton": "master"
}
```

And finally run in the terminal the `composer update` command.

5.3.2 Configuring the database

Once the package has been imported into a Laravel project, it is time to configure the database it will use to store the rule-based models.

First, one should add a new connection in the `connections` array inside the `config/database.php` file. This connection can be as follows:

```
'piton_connection' => [
            'driver' => env('DB_CONNECTION_PITON'),
            'host' => env('DB_HOST_PITON', '127.0.0.1'),
            'port' => env('DB_PORT_PITON', '3306'),
            'database' => env('DB_DATABASE_PITON', '
              forge'),
            'username' => env('DB_USERNAME_PITON', '
              forge'),
            'password' => env('DB_PASSWORD_PITON', ''),
            'unix_socket' => '',
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'prefix_indexes' => true,
            'strict' => true,
            'engine' => null,
        ],
```

Then, one should add the following to the `.env` file of the project, to specify how to access this database (in the same way as the main database connection is defined for every Laravel project):

```
DB_CONNECTION_PITON=mysql
DB_HOST_PITON=127.0.0.1
DB_PORT_PITON=3306
DB_DATABASE_PITON=<your_piton_database>
DB_USERNAME_PITON=<your_mysql_username>
DB_PASSWORD_PITON=<your_mysql_password>
```

Where `your_piton_database` corresponds to the name of the database chosen to host the rule-based models. After the database has been configured, it is finally possible to launch the `php artisan migrate` command from the command line to populate it with the tables seen in par. 5.2.

5.3.3 Training

Now that the package has been installed and the database has been configured, it is possible to use the package to train new models. To do so, it is suggested to publish the *problem-config* file via the following command:

```
php artisan vendor:publish --tag=problem-config
```

This will create a file in the `config` folder of the project named `problem.php` that will be used to tell the package how to read from the main MySQL database to create the rule-based models. This is done by filling the following property arrays:

- `trainingMode` specifies how to split data for training and test; if *FullTraining* is specified, the package both trains and tests onto all the data, while if an array of two floats between 0 and 1 is specified it will split the data between train and test according to these two weights.
- `cutOffValue` specifies the minimum percentage of any of the two classes of the classification that is needed for telling whether a dataset is too unbalanced; it is a value between 0 and 1.
- `defaultOptions` specifies additional options, such as the language of the text used for pre-processing.
- `inputTables` specifies the tables containing the attributes needed for training and how to join them; for the first table, only the name is needed, while the others can be specified with arrays of the form `[tableName, joinClauses, joinType]`, where `joinClauses` is a list of MySQL constraint strings.
- `whereClauses` specifies the SQL WHERE clauses for the concerning input tables, for each recursion level.
- `orderByClauses` specifies the SQL ORDER BY clauses; differently from `whereClauses`, they are fixed at all levels.
- `identifierColumnName` specifies which column will serve as an identifier for sql-based prediction and for a correct retrieval step of prediction results.
- `inputColumns` species the columns which will serve as attributes for the rule-extraction; each column is specified with an array of the form `[columnName,`

`treatment, attrName]` where *columnName* is the name of the column in the database, treatment an optional parameter to specify a specific type of treatment while transforming it into an attribute (i.e. if it must be divided in more attributes), and *attrName* is the name of the resulting attribute.

- `outputColumns` specifies the columns to be treated as classification attributes at each level; each column is specified with an array similar to the elements of `inputColumns`, with the difference that each column can be also derived from join operations and from tables that have not been specified in `inputTables`.

- `globalNodeArray` is used to tweak the order in which the problems are discovered and solved.

For a better understanding, par. 6.2 and par. 7.3 will offer an example on how to properly fill this file. It is advised, once the file has been filled, to rename it after the problem to be solved (i.e. `treatmentSuggestion.php`), as it will be a parameter of the command used to create rule-based models. This way, one can have multiple files of this type and solve multiple problems just changing this parameter when launching the command. Once this file has been configured, one needs to also publish the configuration file for the learner he would like to use. In fact, every learner has its own configuration file, used to specify its training parameters. However, this time a general configuration is also provided, so once a learner configuration file has been published the learner is ready to go. It is possible to publish them with one of the following commands:

```
php artisan vendor:publish --tag=prip-config
php artisan vendor:publish --tag=sklearn_cart-config
php artisan vendor:publish --tag=wittgenstein_irep-config
php artisan vendor:publish --tag=wittgenstein_ripperk-config
```

Now that all the configuration files have been set up, it is possible to run the `piton:update_models` command, which will create the rule-based models. This command accepts as parameters a *problem* name, that is the name previously given to the `problem.php` configuration file, an *author ID*, to specify who actually launched the command, the name of the learner to be used and, eventually, the specific algorithm to be used for the training (i.e. in case of a Wittgenstein learner, if it would use the IREP or the RIPPERk algorithm).

For example, let's suppose we want to solve the problem of the classification of flowers belonging to the iridaceae family discussed in par. 3.1 using the IREP algorithm implemented by the Wittgenstein package. The first thing to do is to publish the `problem.php` configuration file, fill it and renaming it, for example, as `iris.php`. Then, we must publish the configuration file required by the algorithm, that is `wittgenstein_irep-config.php`, and leave it as it is. Finally, we launch the following command:

```
php artisan piton:update_models iris 0 wittgenstein_learner IREP
```

5.3.4 Prediction

Once the rule-based models have been extracted, it is possible to use them to make predictions about new instances. This can be done in two ways.

- The first one is to use the `php artisan piton:predict_by:identifier` command. Once the command has been launched, it will ask for an instance id, and it will make a prediction about the classification of this instance based on the most recent rule-based models created for that problem.
  Suppose, for example, that we launched the models extraction for the iris problem on 1000 instances, and then added 50 more instances of flowers into the database and want to know the genre and family of the flower of id 1020. We would launch the command and give 1020 as instance id.

- However, the package is expected to be used more often into a web application rather than via the command line, so the most common way to ask for a prediction will be using the `predictByIdentifier()` function of the `DBFit` class. This function accepts as parameters an instance id, an array (which is used by the function implementation, but should always be an empty array), an instance of model version id, to specify which rule-based models to use (i.e. associated with which execution), and return the prediction in json format, so that it can easily be displayed on the page (even using AJAX).
  For example, consider the iris problem, if we want to use the rule-based models created within the last execution and predict on the instance of id 1020, we would specify the following in the code of our page.

```
$lastMV = ModelVersion::orderByDesc('id')->first();
$dbfit = new DBFit();
$prediction = $dbfit->predictByIdentifier(1020,
                                          [], $lastMV['id']);
```

Note that this can be useful especially when a new instance has just been inserted into the database, as it is possible to specify to use its id for the prediction and know its classification straight away.

Chapter 6

An example use case: automatic advertisement

The purpose of this chapter is to introduce an interesting and realistic possible application of the package, with the meaning of showing all of its key features.
We will first discuss the problem, how we have decided to solve it, the comparison between this solution and the structures we have seen in the third chapter, and finally how to actually apply the package, introducing a possible configuration file for reading from the database and create the hierarchy which will host the rule-based models to solve out the problem.

6.1 The problem explained

The problem consists of an automatic advertisement for an e-commerce of second-hand vehicles. That is, a service common among social networks which based on the user information (such as age, interests, activities, and so on), with the user's consent, improves the suggestion of advertisements. In our case, we specify on the advertisements of an affiliated site which sells second-hand vehicles, first and foremost deciding if the user would be interested in the purchase of a second-hand

vehicle in this precise moment, then eventually deciding which type of vehicle the user could be more interested in, and finally choosing the specific advertisement to be shown. So, a possible hierarchy which we can choose to structure the problem could be Figure 6.1 (we denote each single ad with the code 'ad###').

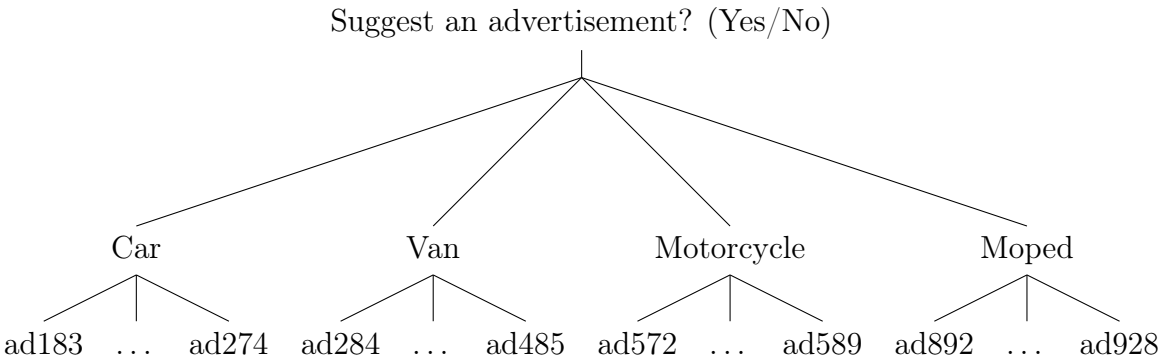

Figure 6.1: Possible hierarchy to solve the automatic advertisement problem

Of course, the number of searches for each single user would be too little. Furthermore, we would like the suggestion to also have effect to users that have not showed interest in buying a vehicle yet. To do this, we categorize the users of the social network in sets of users with the same characteristics, and associate a user category to each user and to each search. This way, the suggestion will be based on the searches of users with similar characteristics. Let's suppose the tables of Figure 6.2 to be a sub-part of a possible social network database.

Users
ID
Birth
Social status
Has children
Job
Interests
...
User category ID

User category
ID
Age
Social status
Has children
Job
...

Advertisements
ID
Name
Vehicle type
Vehicle model
Kilometers
Year
Smoker driver
...

Searches
ID
User category ID
Advertisement ID
Date
...

Figure 6.2: Part of the social network database

6.2 How to effectively use the package to solve the problem

Let's consider a Laravel project associated with a social network and with a database containing the tables illustrated in Figure 6.2. The package has been added to the `composer.json` file as seen in par. 5.3.1, the connection for the package database has been added to the `config/database.php` file and the access parameters to access it to the `.env` file as shown in par. 5.3.2. Finally, the problem configuration file has been published in `config/problem.php` as seen in par. 5.3.3 and has been manually renamed into `automaticAdvertisement.php`.
Let's discuss a possible configuration to solve the automatic advertisement problem. First of all, one should specify how to join the various tables to obtain a single table (or a *dataset*) where every instance corresponds to a single research, but with attributes from all the other tables. It is important to remember that the extraction happens on the user categories rather than the singles users, so the *Users* table is not needed in this stage, and the association of a user to a user category should be managed outside of the package by the specific application. By definition, each instance of the *Searches* table is associated with exactly one instance of the *User category* table, while many different searches can be associated with each user category, and all the needed information about the input attributes is contained in these tables. So one possible configuration for the `inputTables` array could be the following (comprising only one left join):

```
'inputTables' => [
        "searches",
        [
            "user_category",
            "user_category.id = searches.user_category_id",
            "LEFT JOIN"
        ]
    ],
```

Then, the `whereClauses` array is used to specify some structural constraints. For example, in this case, one should think about removing from the classification users younger than 14 years old (considering also mopeds) as younger users won't

be actually interested in the purchase of a vehicle, and at training not to consider searches that are too old (i.e. removing searches previous to 2020). So the `whereClauses` array could be filled as follows:

```
'whereClauses' => [
        'default' => [
            /* Structural constraints */
            "user_category.age > '13'",
        ],
        'onlyTraining' => [
            "searches.date > '2020-01-01'",
        ]
    ],
```

Then, it is possible to specify how to order the instances, for example by the date of the search, using the `orderByClauses` array:

```
'orderByClauses' => [
        [
            "searches.date",
            "ASC"
        ]
    ],
```

And which column serves as a global identifier for the instances, using the `identifierColumnName` array:

```
'identifierColumnName' => "searches.id",
```

At this point, one must specify which columns of the specified tables are to be used for the rule-extraction, and how to treat them. The following syntax is used: `"0+..."` implies that the column will be treated as a continuous attribute, `"CONCAT('',...)"` implies that the column will be treated as a discrete attribute; furthermore, the second parameter for each column can be used to specify a certain treatment method, such as *forceCategoricalBinary* for discrete attributes, that will convert the column in many binary categorical attributes, one for each possible value in its domain. This is specified using the `inputColumns` array.

```
'inputColumns' => [
        /* Age */
        [
            "0+user_category.age",
            NULL,
            "age"
        ],
        /* Social status */
        [
            "CONCAT('', user_category.social_status)",
            "ForceCategorical",
            "gender""
        ],
        /* Has children */
        [
            "CONCAT('', user_category.has_children)",
            "ForceCategorical",
            "has_children"
        ],
        /* Job */
        [
            "CONCAT('', user_category.job)",
            "ForceCategorical",
            "job"
        ],
                ...
        ],
```

Lastly, one should, of course, specify on which attributes the classification should happen. This is done using the `outputColumns` array (eventually, if new joins are needed to add these attributes, they can be done here).

```
'outputColumns' => [
        [
            "advertisements.vehicle_type",
            [
                [
                    "advertisements",
                    [
          "searches.advertisement_id = advertisements.id",
                    ],
                    "LEFT JOIN"
                ]
            ],
            "ForceCategoricalBinary",
            "VehicleType"
        ],
        [
            "advertisements.name",
            [
                [
                    "advertisements",
                    [
          "searches.advertisement_id = advertisements.id",
                    ],
                    "LEFT JOIN"
                ],
            ],
            "ForceCategoricalBinary",
            "Advertisement"
        ]
    ],
```

After filling the arrays in the problem configuration file, it is possible to publish a learner configuration file and to launch the rule-extraction as discussed in par. 5.3.3.

Chapter 7

A real application: automated treatment suggestion

It is finally time to introduce the environment that gave birth to the package.
That is, the second version of the physicians' platform of the Research Center for the Study of Menopause and Osteoporosis within the University of Ferrara (Italy), abbreviated into CMO. This platform served primarly as a way to access the patients' database, offering tools to insert or retrieve patients and reports, densitometries, therapies, and so on. Furthermore, it offered the physicians an interesting tool for treatment suggestion, which based on the therapies given by the physicians in the past allowed the physician to receive suggestions for the prevention and treatment of osteoporosis within patients with similar characteristics and symptoms.
The new version of the platform was entirely developed using the Laravel framework during the last year. However, the idea came to mind to separate the machine learning logic from the project, which at the time only consisted of one implementation of the RIPPERk algorithm, so that it would be easier to scale and, more importantly, to use it in a completely different environment. In fact, the package finds a great fit in the CMO project serving as a clinical decision support system,

especially given the structure of the problem: that is, a two levels hierarchy, the first concerning the treatment type, the second concerning the specific active principle to be used.
A lot has changed from its first version, starting from the support for Scikit-Learn and Wittgenstein Python packages, which give the physician the possibility to switch between models based on accuracies, or to confront the results of more models for the same patient before taking a decision for a specific therapy. Moreover, while its first version only supported one problem at a time, the package now gives support to more indipendent hierarchical problems at a time.

7.1 What is the CMO

The Center for the Research and for the Study of Menopause and Osteoporosis, or CMO, is an interdepartmental center of the University of Studies of Ferrara which availing of multidisciplinary clinical skills such as gynecology, ortophedy, radiology, intertistic and biochemic studies all the thematic concerning menopause and menopausal and senile osteoporosis. It represents an autonomous scientific articolation among all of its composing structures: the Department for Translational Medicine and for Romagna and the Department of Neuroscience and Rehabilitation. Its actual headquarters can be found at the woman health center of Ferrara's AUSL, convention for the women health in menopausal age subscribed among the University for the Studies of Ferrara, the AUSL of Ferrara and the Hospital Agency.
Osteoporosis is a skeletal system disease mainly categorized by alterations in bone mass density and its structure, that makes the bone prone to fracture, and which has proven to be strongly correlated with women menopausal state.

7.2 The problem: treatment suggestion for osteoporosis

Given information about the patient, such as its anamnesis, its menopausal state, its densitometry, we would like to predict which therapies and respective active principles the physician could suggest the patient.

This can easily be seen as a hierarchy of problems consisting of two levels: the first concerning the type of the suggested therapy, the second concerning which active principle to suggest. The possible therapies and respective active principles are summarized by Table 7.1.

Name	Therapy type	Active principles	Abbreviation
T_{horm}	Hormonal Therapy	MHT (tibolone)	tib
		MHT (oral)	$oral$
		MHT transdermal	$trans$
		MHT (TSEC)	$tsec$
T_{osteop}	Osteoprotective Therapy	Alendronate	ale
		Alendronate + vit D	$ale + vD$
		Risendronate	ris
		Ibandronate	iba
		Clodronate	clo
		Raloxifene	ral
		Bazedoxifene	baz
		Denomasub	den
		Teriparatide	ter
		Zoledronate	zol
T_{vitDth}	Vitamin D Therapy	Colecalciferol	$colec$
		Calcifediol	$calci$
$S_{vitDsup}$	Vitamin D Supplementation	Colecalciferol	$colec$
		Calcifediol	$calci$
S_{calsup}	Calcium Supplementation	Carbonated calcium	$carb$
		Citrated calcium	$citr$

Table 7.1: Possible therapies and respective active principles.

Meanwhile, the hierarchy can be visualized as represented in Figure 7.1 (each sub-problem of the first level has been separated for a cleaner visualization).

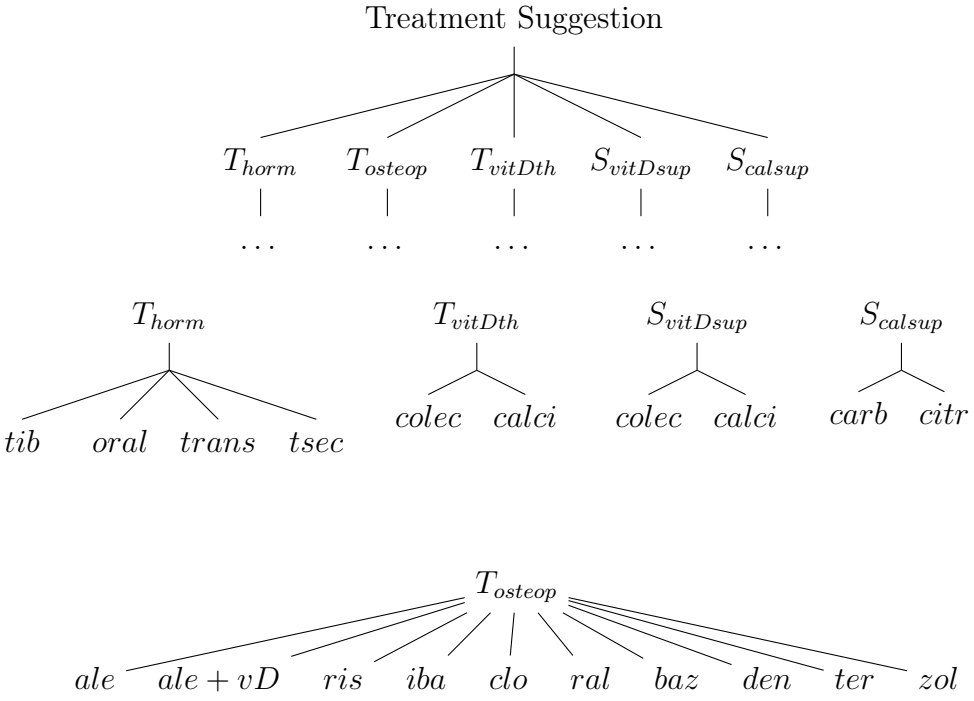

Figure 7.1: Possible hierarchy to solve the treatment suggestion problem

The application is associated with a MySQL database, containing a great amount of data about the patients served to this moment and the given therapies.
Specifically, we consider the attributes of the tables represented in Figure 7.2, with the following considerations:

- each report is associated with one patient, and each diagnosis, anamnesis, densitometry, treatment suggestion and active principle are associated with one report;
- despite many reports could be associated with one patient, at the moment of writing the platform considers each served patient as a new patient, so only one report is associated with each patient, as well as each report is associated with only one diagnosis, one anamnesis and one densitometry;
- many treatment suggestions and active principles can be associated with one report, while each unitary treatment suggestion is associated with one treatment suggestion and one active principle, and viceversa.

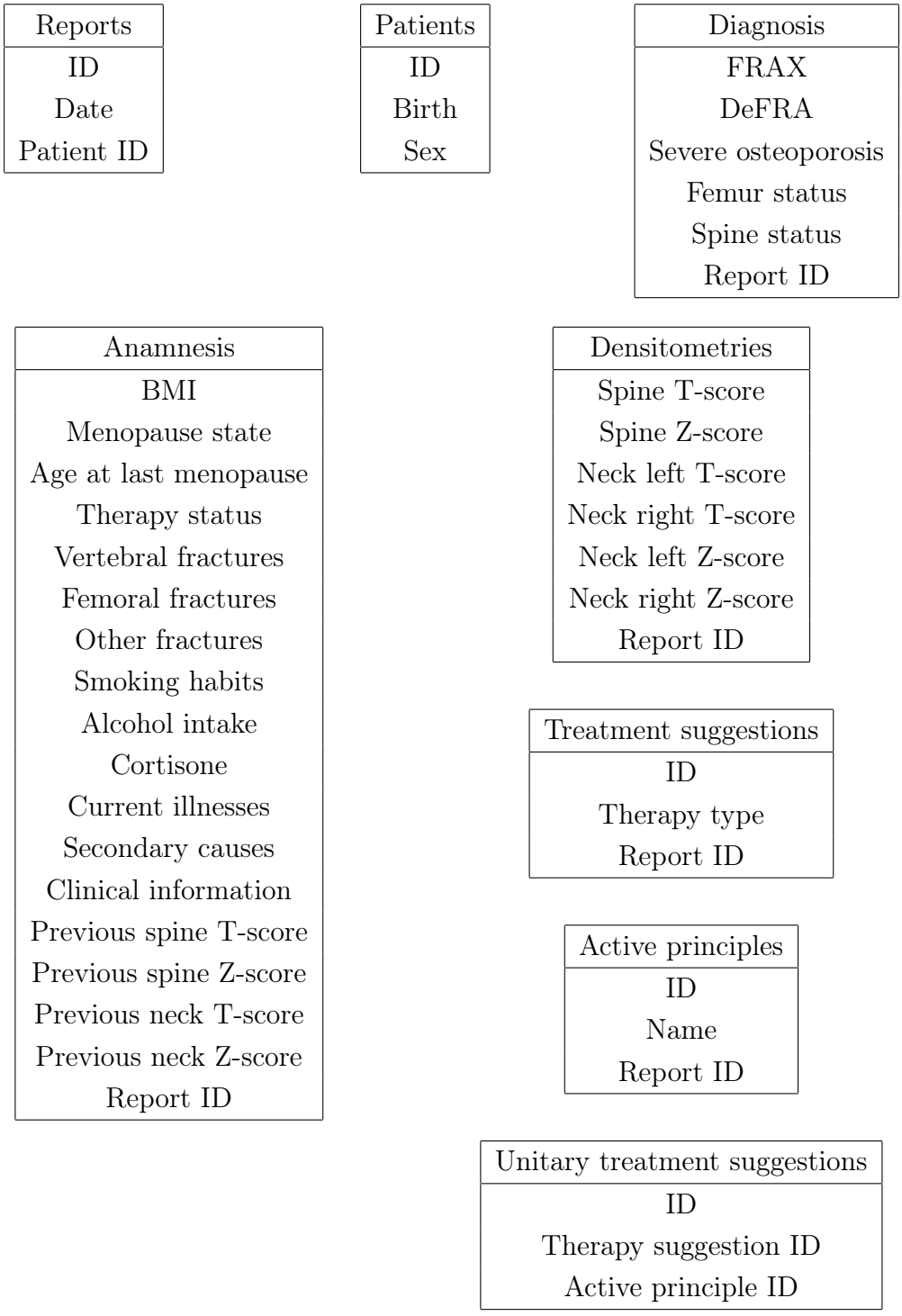

Figure 7.2: Sub-set of tables of the CMO database.

7.3 How to use the package to solve the problem

We will now propose a configuration for the `problem.php` file, which in this application we have renamed `treatmentSuggestion`, to solve the problem.
First, we notice that we have to join the reports, the patients, the anamnesis, the diagnosis and the densitometries tables. To do so, we notice that the anamnesis, the diagnosis and the densitometries tables have a column *Report ID*, while the reports' table has a column *Patient ID*.
So we can fill the `inputTables` array as follows:

```
'inputTables' => [
        "reports",
        [
            "patients",
            "patients.id = reports.patient_id",
            "LEFT JOIN"
        ],
        [
            "anamnesis",
            "anamnesis.report_id = reports.id",
            "LEFT JOIN"
        ],
        [
            "diagnosis",
            "diagnosis.report_id = reports.id",
            "LEFT JOIN"
        ],
        [
            "densitometries",
            "densitometries.report_id = reports.id",
            "LEFT JOIN"
        ]
    ],
```

Then, we could specify some structural constraints. To do so, we must fill the

`whereClauses` array. Specifically, we can be interested only in reports which have been stored after 2018-09-01, and being the research about the connection between menopause and osteoporosis, we are interested only in female patients, and among them we need information about their menopause state and their body mass index value (so they cannot be null nor -1). Then, we could be more interested in the age of the patient at the time of the visit rather than its date of birth. Finally, at training time (but not at prediction time) the output values, and so the treatment type and the active principle, for one instance cannot be null, and should have been suggested at least one time.
So the `whereClauses` array could be filled as follows:

```
'whereClauses' => [
        'default' => [
            /* Structural constraints */
            "reports.date > '2018-09-01'",
            "patients.sex = 'F'",
            "!ISNULL(anamnesis.menopause_state)",
            "DATEDIFF(reports.date,patients.date_of_birth)
                                        / 365 >= 40",
            /* Constraints for manual cleaning */
            "anamnesis.bmi is NOT NULL",
            "anamnesis.bmi != -1",
        ],
        'onlyTraining' => [
            [
                "reports.id",
                "NOT IN",
                [
                    "reuse_current_query",
                    1,
                    [
                      "!ISNULL(treatment_suggestions.type)",
                      "ISNULL(active_principles.name)"
                    ]
```

```
                ]
            ],
            [
                "reports.id",
                "NOT IN",
                [
                    "reuse_current_query",
                    1,
                    [],
                    [
                        "GROUP BY" => [
                            "treatment_suggestions.type",
                            "active_principles.name",
                            "reports.id"
                        ],
                        "HAVING" => "COUNT(*) > 1"
                    ]
                ]
            ]
        ]
    ],
```

Then, we can specify how to order the instances, for example by the date of the report:

```
'orderByClauses' => [
        [
            "reports.date",
            "ASC"
        ]
    ],
```

And which column serves as a global identifier for the instances:

```
'identifierColumnName' => "reports.id",
```

Then we need to specify which columns of these tables are to be used for the rule-extraction. We will write just few of them, to make some examples.
Note that with the syntax "0+..." we declare that column should be treated as a numerical attribute, with "CONCAT(",...)" we declare that it should be treated as a categorical attribute using, eventually, the second parameter for each column to specify how to treat this attribute (i.e. ForceCategoricalBinary force the split of the attribute in many binary categorical attributes).
We can also specify other information, such as how to deal with null values or to create new attributes, making use of the MySQL syntax. The third parameter is simply the new name we give to the attribute (i.e. we have the date of birth in the database, but not an age attribute, so we create it and give it the name *age*).

```
'inputColumns' => [
        /* Age */
        [
            "0+DATEDIFF(reports.date,patients.date_of_birth)
                                    / 365",
            NULL,
            "age"
        ],
        /* Body Mass Index */
        [
            "0+IF(ISNULL(anamnesis.bmi) OR anamnesis.bmi =
                                    -1, NULL, anamnesis.bmi)",
            NULL,
            "body mass index"
        ],
        /* Gender */
        [
            "CONCAT('', patients.sex)",
            "ForceCategorical",
            "gender"
        ],
```

```
        /* Menopause State */
        [
            "CONCAT('', anamnesis.menopause_state)",
            "ForceCategorical",
            "menopause state"
        ],
            /* Age at last menopause */
        [
            "0+anamnesi.eta_menopausa",
            NULL,
            "age at last menopause"
        ],
                ...
        ],
```

Finally, we specify which columns contain information about the classification, using the `outputColumns` array. In this case, we have two levels: one for the type of the therapy and one for the active principle.

```
'outputColumns' => [
        [
            "treatment_suggestions.type",
            [
                [
                    "treatment_suggestions",
                    [
                        "treatment_suggestions.report_id =
                                report_id",
                        "treatment_suggestions.type !=
                                'In-depth investigation'"
                    ],
                    "LEFT JOIN"
                ]
            ],
```

```
            "ForceCategoricalBinary",
            "Therapy"
        ],
        [
            "active_principles.name",
            [
                [
                    "treatment_suggestions",
                    [
                        "treatment_suggestions.id =
  unitary_treatment_suggestions.treatment_suggestion_id"
                    ],
                    "LEFT JOIN"
                ],
                [
                    "active_principles",
                    [
   "unitary_treatment_suggestions.active_principle_id =
                                active_principle.id"
                    ],
                    "LEFT JOIN"
                ]
            ],
            "ForceCategoricalBinary",
            "PrincipioAttivo"
        ]
    ],
```

Now, it is possible to extract rule-based model as seen in par. 5.3.3.
Specifically, these models are updated offline once a week, so that they can improve over time thanks to the greater amount of data.
The prediction feature is implemented as a button in the page where the physician fills the report for a specific patient, and it appears after all the data about anamnesis, densitometries and diagnosis has been inserted. Compiling the report,

a new instance in the *Reports* table is inserted into the database, and the information about its id is kept while filling the form, and clicking the *Suggest a Therapy* button it is passed to the function `dbfit()->predictByIdentifier()` as a parameter. This function will then return the prediction about this instance in json form, which will be displayed to the physician, to help him to find a treatment for the patient. It is important to remember that the models **don't** suggest a therapy to a patient. Instead, they are meant to help the physician to find a therapy for the patient. The physician has always the final word.

7.4 Experiment

To estimate the effectiveness of the application of the package to the CMO system, a complete experiment has been executed, considering the recommendations that have been given from Sept. the 1st, 2018 to Aug. the 31th, 2020. During this period, 2052 postmenopausal women over 40 years of age underwent a DXA (*Dual-energy X-ray absorptiometry*, which data is stored in the *Densitometries* table of Figure 7.2) examination at CMO. Of these, 18 patients returned more than once; our original data set, then, is composed by 2070 reports. Out of these reports, 16 presented a null BMI, 4 of them presented a recommendation of some type without drug or supplement specification, and 2 of them presented the same recommendation more than once (with the same drug or supplementation) because of human error during data insertion; these have been filtered out, leaving us with 2048 instances. Observe that at the first level, it holds that each instance is a record; at the second level, however, after selecting only those reports in which a particular recommendation type has been given, more than one recommendation for the same patient may have been selected, and therefore it holds that each instance is a recommendation. Furthermore, there are several instances in which FRAX and/or DeFRA was less than 0.1 or more than 50 (for DeFRA only): these values have been replaced by 0 and 50, respectively. An important observation at this point is that the single instance does not always end in a therapy recommendation, but it may also end in a recommendation of further exams and investigations; in this experiment, such cases have been computed as negative cases for therapy recommendation, causing some imbalance between osteoporosis/severe osteoporosis cases and effectively recommended therapies.

In this experiment we fixed $cutOffValue = 10\%$; thus, only three data set have been considered for classifier extraction at the first level, namely D_{osteop}, $D_{vitDsup}$, and D_{calsup}. Under the same parameter, at the second level the following data set have been considered: D^{ale}_{osteop}, D^{den}_{osteop}, D^{ris}_{osteop}, $D^{colec}_{vitDsup}$, $D^{calci}_{vitDsup}$, D^{carb}_{calsup}, and D^{citr}_{calsup}.

After running the entire system, having fixed $trainingMode = [0.8, 0.2]$ (so that the 20% stratified most recent records of each data set are used for testing purposes) the results are as in Tables 7.2 and 7.3. For each problem and its corresponding test, we reported the following values (most of them already discussed in par. 5.2): accuracy, that is, the rate of corrected classification, sensitivity, that is, the rate of true positives, specificity, that is, the rate of true negatives, positive predicted value, that is, the inverse of the false discovery rate, the negative predicted value, that is, the inverse of the false omission rate, and the F1 score, that is, the harmonic mean of sensitivity and positive predicted value

Let us focus, first, on the behaviour of Γ_{osteop}. Out of 2048 instances, in 447 a osteoprotective therapy has been prescribed; our system is able to correctly predict if that is the case for a new instance in the 86% of the cases. If, in fact, an osteoprotective therapy should be recommended, the system returns the correct suggestion in the 55% of the cases, while if the therapy should not be recommended, the system gives a correct prediction in the 95% of the cases. In the case of predicting if a patient needs calcium supplementation (580 positive instances), our system gives a correct prediction in the 83% of the cases, which drops to 47% in the positive cases and raises to the 97% in the negative ones. Finally, in the case of supplementation of vitamin D (1125 positive cases), the rate of overall correct prediction is 75%, which is 74% and 76% in the positive and the negative cases, respectively. As it turns out, predicting the correct drug or supplement is quite more difficult at least in some cases, because the data set are still unbalanced even after our initial screening. Yet, in at least three cases (predicting a recommendation of risedronate, calcifediol, and colecalciferol), the accuracy is still between 83% and 93%.

Local evaluation is shown in Tables 7.4 and 7.5, where we have displayed, for each type, the number of rules of each classifier, and their distribution among the following four types (calculated on *support*, that is, the ratio of cases in which both the antecedent and the consequent of the rule are verified over the total number of

cases, and *confidence*, that is, the ratio of the cases in which only the antecedent is verified over the cases in which both the antecedent and the consequent are verified):

- (I) relevant and reliable, showing $support > 0.2$ and $confidence > 0.7$;
- (II) relevant, but unreliable, showing $support > 0.2$ but $confidence \leq 0.7$;
- (III) irrelevant, but reliable, showing $support \leq 0.2$ but $confidence > 0.7$;
- (IV) irrelevant and unreliable, showing $support \leq 0.2$ and $confidence \leq 0.7$.

Among these, types I and III (which include only reliable rules) are the most common ones, indicating that our approach is relatively stable.
These results have also been integrated into a scientific paper, named *Predicting Therapy Recommendations in Postmenopausal Osteoporosis* [13].

Classifier	Accuracy	F_1	Sensitivity	Specificity	PPV	NPV
Γ_{osteop}	0.86	0.63	0.55	0.95	0.74	0.88
$\Gamma_{vitDsup}$	0.75	0.76	0.74	0.76	0.79	0.71
Γ_{calsup}	0.83	0.61	0.47	0.97	0.85	0.82

Table 7.2: Results of the experiment: global evaluation of first level classifiers.

Classifier	Accuracy	F_1	Sensitivity	Specificity	PPV	NPV
Γ^{ale}_{osteop}	0.72	0.39	0.27	0.95	0.73	0.72
Γ^{den}_{osteop}	0.71	0.35	0.25	0.92	0.58	0.73
Γ^{ris}_{osteop}	0.83	0.12	0.10	0.92	0.14	0.89
$\Gamma^{calci}_{vitDsup}$	0.93	0.65	0.54	0.98	0.83	0.94
$\Gamma^{colec}_{vitDsup}$	0.92	0.96	0.98	0.54	0.94	0.79
Γ^{citr}_{calsup}	0.62	0.52	0.46	0.75	0.60	0.63
Γ^{carb}_{calsup}	0.62	0.68	0.75	0.46	0.63	0.60

Table 7.3: Results of the experiment: global evaluation of second level classifiers.

Classifier	#	#I	#II	#III	#IV
Γ_{osteop}	6	1	0	5	0
$\Gamma_{vitDsup}$	6	2	2	2	0
Γ_{calsup}	4	1	0	2	1

Table 7.4: Results of the experiment: local evaluation of first level classifiers.

Classifier	#	#I	#II	#III	#IV
Γ^{ale}_{osteop}	2	0	2	0	0
Γ^{den}_{osteop}	4	0	1	1	2
Γ^{ris}_{osteop}	2	1	0	0	1
$\Gamma^{calci}_{vitDsup}$	5	1	0	2	2
$\Gamma^{colec}_{vitDsup}$	5	1	0	2	2
Γ^{citr}_{calsup}	3	0	2	1	0
Γ^{carb}_{calsup}	3	0	2	1	0

Table 7.5: Results of the experiment: local evaluation of second level classifiers.

Chapter 8

Conclusions

This thesis discussed the implementation of a Laravel package for the systematic construction of symbolic intelligent systems for solving hierarchically-arranged problems through the creation of rule-based models using data stored in a MySQL database.
The package is open-source, and it is currently used by the CMO of Ferrara as a clinical decision support system to suggest therapies for the cure of osteoporosis to the physicians.
However, the package is certainly not flawless, and there may be many potential improvements. As an example, at the moment of writing all the attributes used for training the models are associated to the extracted rule-based model, and to make predictions about a new instance it must have at least the same attributes as the ones associated to the rule-based model. To improve this, the package could save only the attributes used by its rules, and, as a further improvement, it could associate them to each single rule rather than to the entire model. This way the models would be more flexible and predict on much more instances.
Then, at the moment of writing, all the main features of the package (that are, reading from the database and creating the hierarchies, fitting the rule-based models and making predictions on new instances) are comprised only in one class, that

is DBFit. This class could therefore be divided into three lighter classes, each one containing the logic for one of these functionalities: one for reading from the MySQL database and instancing the hierarchy of problems, one dedicated to the fitting of the rule-based models, and the last one comprising the logic to make prediction on new instances.

Finally, the package could be scaled to support even more learners, both from Scikit-Learn and brand new, to be used in a grater range of different applications.

In this internship, I have developed my skill set and gained valuable applied experience. I have broadened my knowledge about machine learning and interpretable AI, as well as many areas such as package development, Laravel, python, sci-kit learn, Wittgenstein and pandas libraries, version control, regular expressions, decision trees and rule-based classification.

I believe this internship had a significant impact on my professional development and heightened my interest the fields of machine learning and explainable AI, especially applied to the biomedical field.

Bibliography

[1] Iridaceae - The plant list:
http://www.theplantlist.org/browse/A/Iridaceae/

[2] Iris - The plant list:
http://www.theplantlist.org/browse/A/Iridaceae/Iris/

[3] 14 Different Types of Learning in Machine Learning:
https://machinelearningmastery.com/types-of-learning-in-machine-learning/

[4] W.W. Cohen. Fast effective rule induction. In Proc. of the 20th International Conference on Machine Learning, pages 115–123. 1995.

[5] Breiman L (1984) Classification and regression trees. The Wadsworth and Brooks-Cole statisticsprobability series. Chapman & Hall.

[6] Laravel - The PHP Framework For Web Artisans:
https://laravel.com/

[7] Installation - Laravel - The PHP Framework For Web Artisans:
https://laravel.com/docs/8.x

[8] Scikit-Learn: machine learning in python:
https://scikit-learn.org/stable/

[9] GitHub - scikit-learn/scikit-learn: machine learning in python:
https://github.com/scikit-learn/scikit-learn

[10] Getting Started - Scikit-Learn 1.0 documentation:
https://scikit-learn.org/stable/getting_started.html

[11] 1.10. Decision Trees - Scikit-Learn 1.0 documentation:
https://scikit-learn.org/stable/modules/tree.html

[12] Wittgenstein - PyPI:
https://pypi.org/project/wittgenstein/

[13] G. Bonaccorsi, M. Giganti, M. Nitsenko, G. Pagliarini, G. Piva, and G. Sciavicco. Predicting Therapy Recommendations in Postmenopausal Osteoporosis, pages 15-21. 2020.