A Laravel package for systematic construction of symbolic intelligent systems

Alberto Paparella

Universitá degli Studi di Ferrara
Dipartimento di Matematica e Informatica

Relatore
Prof. **Guido Sciavicco**

Correlatori
Dott. **Giovanni Pagliarini**
Prof. **Giacomo Piva**

Università degli Studi di Ferrara

4 ottobre 2021

Contents

1. Introduction
   - Pitón
2. Preliminaries
   - Hierarchical problems
   - Rule-based models
   - Rule-extraction
3. How to use the package
   - Install
   - Workflow
   - Configuring the database
   - Training
   - Prediction
4. Automated treatment suggestion
   - Problem
   - Database
   - Experiment results
5. Backup slides

Università degli Studi di Ferrara

Pitón

Pitón

Pitón 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**.

Setting

The package can be placed in the *supervised learning* area of machine learning. The target of the package is learning from a *dataset* (e.g. a set of data representable in tabular form), making use of an *algorithm* we refer to as *learner*, which synthesize a *model* that can be used to classify new *instances*.

Specifically, an algorithm divides the dataset into a *training* set and a *test* set, and it *fits* a model based on *input* and *output* values (e.g. features values and respective classification label in the *train* set); then, it tests the extracted model using only the *input* values of the *test* set and comparing the results to their original classification.

Hierarchical problems

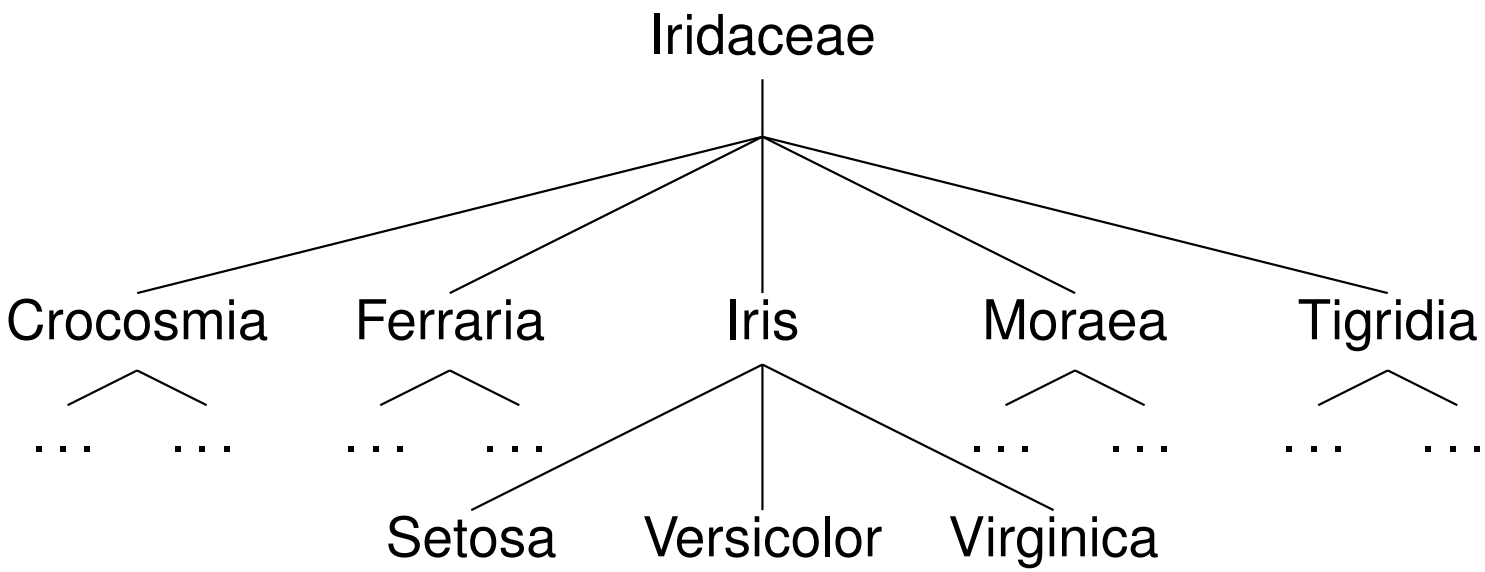


Figure: Hierarchy of problems for Iridaceae classification

Rule-based models

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

Figure: Example of a generic rule-based model

Binary rule-based models

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

Figure: Example of a generic *binary* rule-based model

Università degli Studi di Ferrara

Binary classification

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

Figure: Example of a generic *binary* rule-based model

Classification with binary rule-based models:

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 $\mathcal{C}$.

Binary classification

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

Figure: Example of a generic *binary* rule-based model

Classification with binary rule-based models:

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 $\mathcal{C}$.

Università degli Studi di Ferrara

RIPPERk

To build a rule:

1. Randomly partition all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.
2. *Grow* a rule by greedily adding conditions until the rule reaches a confidence of 100% using only the growing set.
3. To prevent growing set *overfitting*, immediately prune the rule, deleting some conditions based on a pruning criterion, using the pruning data.
4. Lastly, all the positive and negative examples covered by the rule must be removed; then repeat 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* according to some criterion.

RIPPERk

To build a rule:

1. Randomly partition all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.
2. *Grow* a rule by greedily adding conditions until the rule reaches a confidence of 100% using only the growing set.
3. To prevent growing set *overfitting*, immediately prune the rule, deleting some conditions based on a pruning criterion, using the pruning data.
4. Lastly, all the positive and negative examples covered by the rule must be removed; then repeat 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* according to some criterion.

RIPPERk

To build a rule:

1. Randomly partition all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.
2. *Grow* a rule by greedily adding conditions until the rule reaches a confidence of 100% using only the growing set.
3. To prevent growing set *overfitting*, immediately prune the rule, deleting some conditions based on a pruning criterion, using the pruning data.
4. Lastly, all the positive and negative examples covered by the rule must be removed; then repeat 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* according to some criterion.

RIPPERk

To build a rule:

1. Randomly partition all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.
2. *Grow* a rule by greedily adding conditions until the rule reaches a confidence of 100% using only the growing set.
3. To prevent growing set *overfitting*, immediately prune the rule, deleting some conditions based on a pruning criterion, using the pruning data.
4. Lastly, all the positive and negative examples covered by the rule must be removed; then repeat 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* according to some criterion.

RIPPERk

To build a rule:

1. Randomly partition all the examples which have not been covered by any rule yet into two subsets: a *growing set* and a *pruning set*.
2. *Grow* a rule by greedily adding conditions until the rule reaches a confidence of 100% using only the growing set.
3. To prevent growing set *overfitting*, immediately prune the rule, deleting some conditions based on a pruning criterion, using the pruning data.
4. Lastly, all the positive and negative examples covered by the rule must be removed; then repeat 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* according to some criterion.

Università degli Studi di Ferrara

Decision Tree

Example data: hospitalization context where patients are *instances* described by their *age*, *weight* and *gender*. Data are normally collected in tabular form:

# Sample	Age	Weight	Gender	Class label
1	37	70	M	ILL
2	49	81	M	HEALTHY
3	20	55	F	ILL
...	...	...	...	...

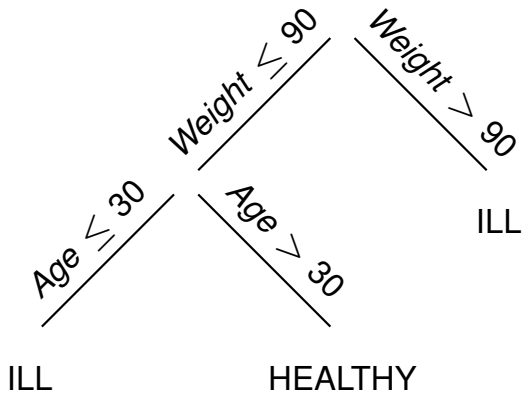
By means of learning algorithms, a decision tree classifier can be *trained* from such data:

Decision Tree

Example data: hospitalization context where patients are *instances* described by their *age*, *weight* and *gender*. Data are normally collected in tabular form:

# Sample	Age	Weight	Gender	Class label
1	37	70	M	ILL
2	49	81	M	HEALTHY
3	20	55	F	ILL
...	...	...	...	...

By means of learning algorithms, a decision tree classifier can be *trained* from such data:



CART

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.

CART

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.

CART

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.

Install

Add the GitHub project as a repository in the `composer.json` file of your Laravel application as follows:

```
1 "repositories": [
2     {
3         "type": "vcs",
4         "url": "https://github.com/aclai-lab/piton"
5     }
6 ]
```

Install

And then add the package in the require section:

```
1 "require": {
2     "aclai/piton": "master"
3 }
```

Finally, run in the terminal the `composer update` command.

Workflow

- The package reads data directly from a MySQL database following the parameters specified into a *problem* configuration file, which information will be stored into the *Problems* table, and creating a dataset.
- Then, it derives the hierarchy of problems, and creates a new instance in the *Model version* table that will store information about the extracted hierarchy.
- Lastly, it will launch the *training* process for each problem, associatiating each extracted *rule-based model* to its correspondent problem in the hierarchy; these models are stored in the *Class model* table, while their rules are stored in the *Rules* table.
- Note that the database is dynamic, and therefore the process can be iterated over time (e.g. once a week) with new data.

Workflow

- The package reads data directly from a MySQL database following the parameters specified into a *problem* configuration file, which information will be stored into the *Problems* table, and creating a dataset.
- Then, it derives the hierarchy of problems, and creates a new instance in the *Model version* table that will store information about the extracted hierarchy.
- Lastly, it will launch the *training* process for each problem, associatiating each extracted *rule-based model* to its correspondent problem in the hierarchy; these models are stored in the *Class model* table, while their rules are stored in the *Rules* table.
- Note that the database is dynamic, and therefore the process can be iterated over time (e.g. once a week) with new data.

Workflow

- The package reads data directly from a MySQL database following the parameters specified into a *problem* configuration file, which information will be stored into the *Problems* table, and creating a dataset.
- Then, it derives the hierarchy of problems, and creates a new instance in the *Model version* table that will store information about the extracted hierarchy.
- Lastly, it will launch the *training* process for each problem, associatiating each extracted *rule-based model* to its correspondent problem in the hierarchy; these models are stored in the *Class model* table, while their rules are stored in the *Rules* table.
- Note that the database is dynamic, and therefore the process can be iterated over time (e.g. once a week) with new data.

Università degli Studi di Ferrara

Workflow

- The package reads data directly from a MySQL database following the parameters specified into a *problem* configuration file, which information will be stored into the *Problems* table, and creating a dataset.
- Then, it derives the hierarchy of problems, and creates a new instance in the *Model version* table that will store information about the extracted hierarchy.
- Lastly, it will launch the *training* process for each problem, associatiating each extracted *rule-based model* to its correspondent problem in the hierarchy; these models are stored in the *Class model* table, while their rules are stored in the *Rules* table.
- Note that the database is dynamic, and therefore the process can be iterated over time (e.g. once a week) with new data.

Università degli Studi di Ferrara

Pitón database

Problems
ID
Name
Input tables
Input columns
Output columns
Where clauses
Order by clauses
Limit
Identifier column name

Model version
ID
Problem ID
Author ID
Learner
Training mode
Cut off value
Experiment ID
Date
Hierarchy
Test results
Test date

Class model
ID
Model version ID
Recursion level
Father node
Class
Rules
Json logic rules
Attributes
Test results*
Test date
Additional infos

Rules
ID
Class model ID
Antecedents
Consequent
Covered
Support
Confidence
Lift
Conviction
Global covered
Global support
Global confidence
Global lift
Global conviction

Figure: Database structure

Configuring the database

First, one should add the following new connection in the `connections` array inside the `config/database.php` file:

```
'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,
        ],
```

Configuring the database

Then, one should add the following to the `.env` file of the project, to specify how to access this database:

```
1 DB_CONNECTION_PITON=mysql
2 DB_HOST_PITON=127.0.0.1
3 DB_PORT_PITON=3306
4 DB_DATABASE_PITON=<your_piton_database>
5 DB_USERNAME_PITON=<your_mysql_username>
6 DB_PASSWORD_PITON=<your_mysql_password>
```

After the database has been configured, it is finally possible to launch the `php artisan migrate` command from the command line to populate it.

Università degli Studi di Ferrara

Training

1. Publish the *problem-config* file via the following command:
   `php artisan vendor:publish --tag=problem-config`
2. Fill it, and rename it after the problem to be solved.
3. Publish the configuration file for the specified learner (*between prip, sklearn_cart, wittgenstein_irep, wittgenstein_ripperk*):
   `php artisan vendor:publish --tag=<learner_name>-config`
4. Run the `piton:update_models` command, which accepts as parameters a *problem* name, an *author ID*, the name of the learner to be used and, eventually, the specific algorithm to be used for the training.

Training

1. Publish the *problem-config* file via the following command:
   `php artisan vendor:publish --tag=problem-config`
2. Fill it, and rename it after the problem to be solved.
3. Publish the configuration file for the specified learner (*between prip, sklearn_cart, wittgenstein_irep, wittgenstein_ripperk*):
   `php artisan vendor:publish --tag=<learner_name>-config`
4. Run the `piton:update_models` command, which accepts as parameters a *problem* name, an *author ID*, the name of the learner to be used and, eventually, the specific algorithm to be used for the training.

Training

1. Publish the *problem-config* file via the following command:
   `php artisan vendor:publish --tag=problem-config`
2. Fill it, and rename it after the problem to be solved.
3. Publish the configuration file for the specified learner (*between prip, sklearn_cart, wittgenstein_irep, wittgenstein_ripperk*):
   `php artisan vendor:publish --tag=<learner_name>-config`
4. Run the `piton:update_models` command, which accepts as parameters a *problem* name, an *author ID*, the name of the learner to be used and, eventually, the specific algorithm to be used for the training.

Training

1. Publish the *problem-config* file via the following command:
   `php artisan vendor:publish --tag=problem-config`
2. Fill it, and rename it after the problem to be solved.
3. Publish the configuration file for the specified learner (*between prip, sklearn_cart, wittgenstein_irep, wittgenstein_ripperk*):
   `php artisan vendor:publish --tag=<learner_name>-config`
4. Run the `piton:update_models` command, which accepts as parameters a *problem* name, an *author ID*, the name of the learner to be used and, eventually, the specific algorithm to be used for the training.

Prediction

Prediction can be done in two different ways:

- Using the `php artisan piton:predict_by_identifier` command.
- Using the `predictByIdentifier()` function of the `DBFit` class. This function accepts as parameters an instance id, and 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 a web page (even using AJAX).

Example:

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

Prediction

Prediction can be done in two different ways:

- Using the `php artisan piton:predict_by_identifier` command.
- Using the `predictByIdentifier()` function of the `DBFit` class. This function accepts as parameters an instance id, and 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 a web page (even using AJAX).

Example:

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

Problem

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.

Problem

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

Table: Possible therapies and respective active principles.

Problem

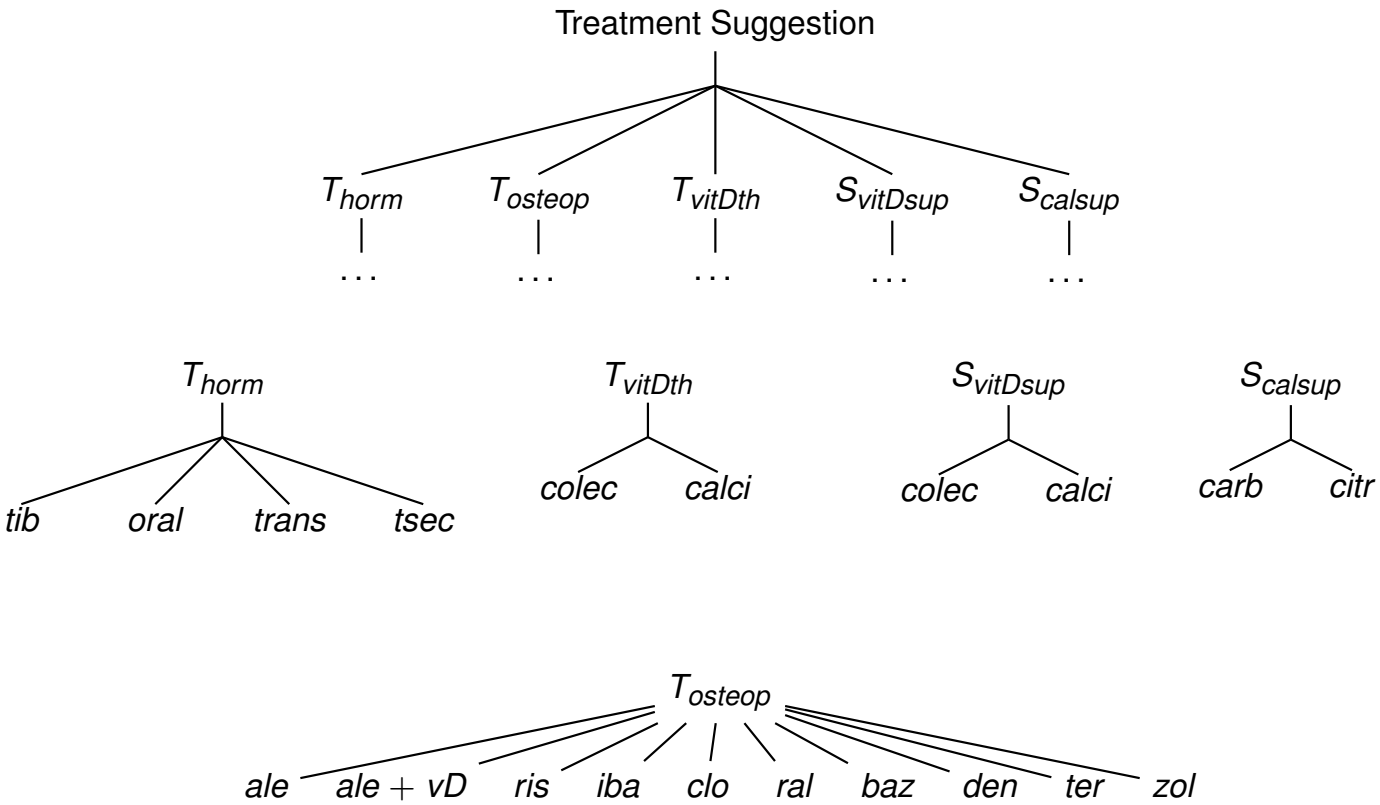


Figure: Possible hierarchy to solve the treatment suggestion problem

Database

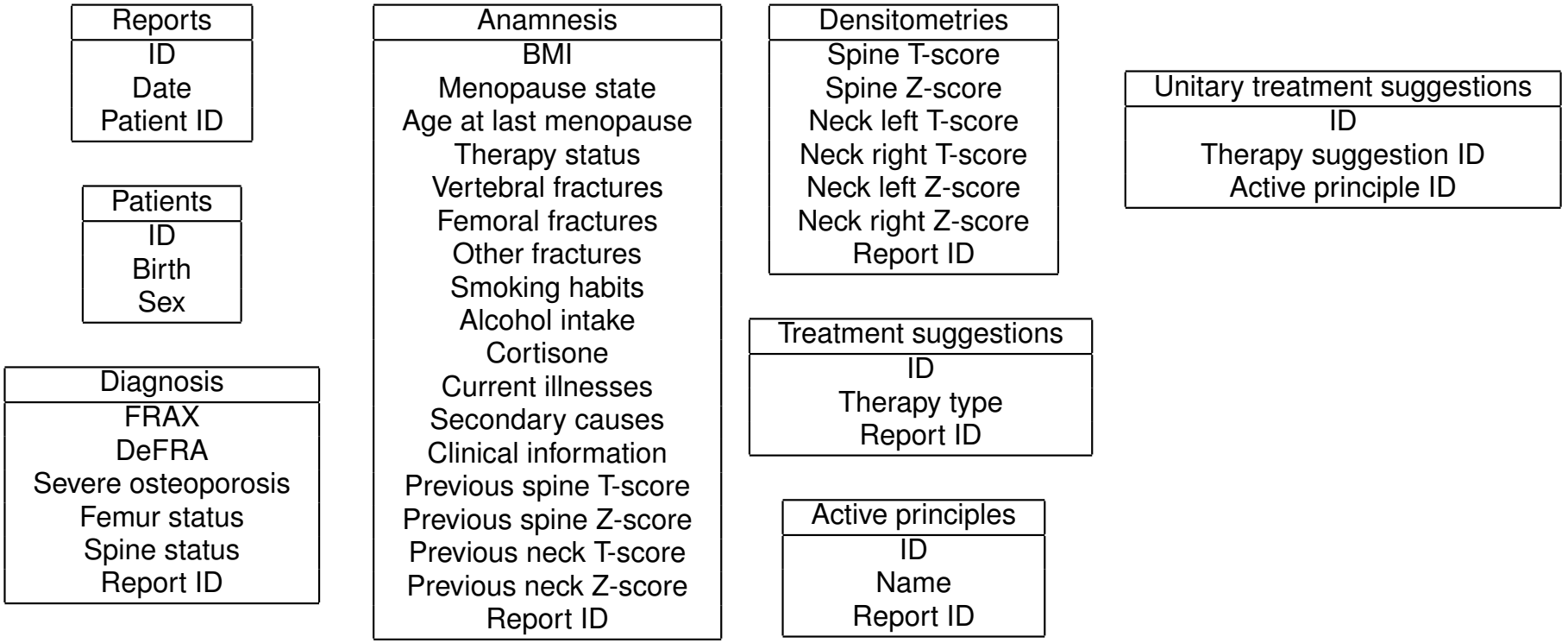


Figure: Sub-set of tables of the CMO database.

Experiment results

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.

In this experiment, only three data set have been considered for classifier extraction at the first level and seven at the second level; this is becaues the others were *unbalanced* (e.g. one class appeared for less than 10% of the instances).

For each data set, 80% of the instances are used as a *training set*, while the 20% stratified most recent records of each data set are used for *testing* purposes.

Experiment results

For each problem and its corresponding test, we reported the following values:

- *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;
- *negative predicted value*, that is, the inverse of the false omission rate;
- *F1* score, that is, the harmonic mean of sensitivity and positive predicted value.

Experiment results

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

Experiment results

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

Experiment results

For local evaluation, we consider the number of rules of each classifier and their distribution among the following four types, based on *support* and *confidence*:

I. Relevant and reliable, showing *support* > 0.2 and *confidence* > 0.7;
II. Relevant, but unreliable, showing *support* > 0.2 but *confidence* ≤ 0.7;
III. Irrelevant, but reliable, showing *support* ≤ 0.2 but *confidence* > 0.7;
IV. Irrelevant and unreliable, showing *support* ≤ 0.2 and *confidence* ≤ 0.7.

Experiment results

Types I and III (which include only reliable rules) are the most common ones, indicating that our approach is relatively stable.

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

Thank you for your attention

Università degli Studi di Ferrara

Thank you for your attention

Università degli Studi di Ferrara

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into *N* folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into *N* folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

Università degli Studi di Ferrara

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into N folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

Università degli Studi di Ferrara

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into N folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into N folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

CART

Pruning

Then, in order to enhance the generalization of the resulting decision tree, pruning is applied:

1. Split randomly training data into N folds.
2. Select a pruning level for the tree (level 0 equals to the full 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 (e.g. the class with the higher number of cases covered by the node) to their parent node.

Università degli Studi di Ferrara

Experiment results

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

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