Symbolic Machine Learning

The main way-to-go for implementing a machine learning pipeline in Julia is via the MLJ.jl package.

We are going to work with the `iris` dataset, trying to discover the relation between the specific attribute values of an iris flower and the family to which the same flower belongs to.

More generally, we want to find the relation between the values of the *attributes* of each instance (`X`) and the corresponding *labels* (`y`).

In order to do so, we are going to train a (classification) decision tree, leveraging the `DecisionTree` package, which can be easily integrated within an `MLJ` pipeline.

Later in this notebook, we will repeat this process leveraging the `Sole.jl` library, which will allow us to explicitly model the problem through the lens of logic.

In [1]:
```
using Pkg
Pkg.activate("..")
Pkg.instantiate()
Pkg.update()
```

```
  Activating project at `~/logic-and-machine-learning`
    Updating registry at `~/.julia/registries/General.toml`
    Updating git-repo `https://github.com/aclai-lab/ManyExpertDecisionTree
s.jl`
    Updating git-repo `https://github.com/aclai-lab/SoleReasoners.jl#embed
ding`
  No Changes to `~/logic-and-machine-learning/Project.toml`
  No Changes to `~/logic-and-machine-learning/Manifest.toml`
```

In [2]:
```
# for reproducibility purposes
using Random
Random.seed!(1605)
```

Out[2]:
```
TaskLocalRNG()
```

Learning with MLJ.jl

Data Loading and Description

In [3]:
```
using MLJ
using RDatasets # used to load the iris dataset


data = RDatasets.dataset("datasets", "iris");
```

In [4]:
```
schema(data)
```

Out[4]:

names	scitypes	types
SepalLength	Continuous	Float64
SepalWidth	Continuous	Float64
PetalLength	Continuous	Float64
PetalWidth	Continuous	Float64
Species	Multiclass{3}	CategoricalValue{String, UInt8}

In [5]:

```
data
```

Out[5]: 150×5 DataFrame *125 rows omitted*

Row	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	Cat…
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
⋮	⋮	⋮	⋮	⋮	⋮
139	6.0	3.0	4.8	1.8	virginica
140	6.9	3.1	5.4	2.1	virginica
141	6.7	3.1	5.6	2.4	virginica
142	6.9	3.1	5.1	2.3	virginica
143	5.8	2.7	5.1	1.9	virginica
144	6.8	3.2	5.9	2.3	virginica
145	6.7	3.3	5.7	2.5	virginica
146	6.7	3.0	5.2	2.3	virginica
147	6.3	2.5	5.0	1.9	virginica
148	6.5	3.0	5.2	2.0	virginica
149	6.2	3.4	5.4	2.3	virginica
150	5.9	3.0	5.1	1.8	virginica

In [6]:
```
y, X = unpack(data, ==(:Species))
```

Out[6]:
```
(CategoricalArrays.CategoricalValue{String, UInt8}["setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa"  …  "virginica", "virginica", "virginica", "virginica", "virginica", "virginica", "virginica", "virginica", "virginica", "virginica"], 150×4 DataFrame
 Row │ SepalLength  SepalWidth  PetalLength  PetalWidth
     │ Float64      Float64     Float64      Float64
─────┼──────────────────────────────────────────────────
   1 │         5.1         3.5          1.4         0.2
   2 │         4.9         3.0          1.4         0.2
   3 │         4.7         3.2          1.3         0.2
   4 │         4.6         3.1          1.5         0.2
   5 │         5.0         3.6          1.4         0.2
   6 │         5.4         3.9          1.7         0.4
   7 │         4.6         3.4          1.4         0.3
   8 │         5.0         3.4          1.5         0.2
   9 │         4.4         2.9          1.4         0.2
  10 │         4.9         3.1          1.5         0.1
  11 │         5.4         3.7          1.5         0.2
  ⋮  │      ⋮           ⋮            ⋮           ⋮
 141 │         6.7         3.1          5.6         2.4
 142 │         6.9         3.1          5.1         2.3
 143 │         5.8         2.7          5.1         1.9
 144 │         6.8         3.2          5.9         2.3
 145 │         6.7         3.3          5.7         2.5
 146 │         6.7         3.0          5.2         2.3
 147 │         6.3         2.5          5.0         1.9
 148 │         6.5         3.0          5.2         2.0
 149 │         6.2         3.4          5.4         2.3
 150 │         5.9         3.0          5.1         1.8
                                        129 rows omitted)
```

In [7]:
```julia
# categorical vectors are lighter than raw vectors; can you guess why?
typeof(y)
```

Out[7]:
```
CategoricalVector{String, UInt8, String, CategoricalValue{String, UInt8}, Union{}} (alias for CategoricalArrays.CategoricalArray{String, 1, UInt8, String, CategoricalArrays.CategoricalValue{String, UInt8}, Union{}})
```

In [8]:
```julia
typeof(X)
```

Out[8]:
```
DataFrame
```

In [9]:
```julia
# to ensure that classes are balanced
for class in unique(y)
    println("$(class) - $(count(yi -> yi == class, y))")
end
```

```
setosa - 50
versicolor - 50
virginica - 50
```

Data Preprocessing

In the limited scenario of this exercise, there is not much space for complex preprocessing of our data. For example, we are not dealing with unbalanced classes, missing data, or complex encodings.

The usual workflow, at this point, is to partition the data into a training and a test bucket, keeping a balanced class diversity.

With this distinction, we can train a model on the initial training data and leverage the test one for simulating a real-world scenario, obtaining reliable performance.

MLJ makes our work *much* easier, even providing us with a more sophisticated training strategy, as we will see later.

Model Training

We will integrate an external model, coming from the `DecisionTree` package, into the MLJ workflow.

In the next lessons, we will be doing something similar with another model called `ModalDecisionTree`.

In [10]:
```
try
    DecisionTreeClassifier = @load DecisionTreeClassifier pkg=DecisionTre
catch
    println("The DecisionTreeClassifier symbol has already been imported.
end
```

```
[ Info: For silent loading, specify `verbosity=0`.
import MLJDecisionTreeInterface ✔
```

Out[10]:
```
MLJDecisionTreeInterface.DecisionTreeClassifier
```

In [11]:
```
model = MLJDecisionTreeInterface.DecisionTreeClassifier(
    max_depth=5,
    min_samples_leaf=1,
    min_samples_split=2
)
```

Out[11]:
```
DecisionTreeClassifier(
  max_depth = 5,
  min_samples_leaf = 1,
  min_samples_split = 2,
  min_purity_increase = 0.0,
  n_subfeatures = 0,
  post_prune = false,
  merge_purity_threshold = 1.0,
  display_depth = 5,
  feature_importance = :impurity,
  rng = TaskLocalRNG())
```

A machine is a binding between a model and the data it works with.

It also keeps track of other information we might want to inspect, such as the specific parameters learned by a model.

In the cell below, we bind the decision tree model to all the instances we have available. This is not a good idea, but we will return on the topic in a moment.

In [12]:
```
mach = machine(model, X, y)
```

Out[12]:
```
untrained Machine; caches model-specific representations of data
  model: DecisionTreeClassifier(max_depth = 5, …)
  args:
    1:  Source @299 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @769 ⏎ AbstractVector{Multiclass{3}}
```

In [13]:
```
fit!(mach)
```

```
[ Info: Training machine(DecisionTreeClassifier(max_depth = 5, …), …).
```

Out[13]:
```
trained Machine; caches model-specific representations of data
  model: DecisionTreeClassifier(max_depth = 5, …)
  args:
    1:  Source @299 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @769 ⏎ AbstractVector{Multiclass{3}}
```

In [14]:
```
y_predict_probabilities = MLJ.predict(mach, X)
y_predict = mode.(y_predict_probabilities)
```

Out[14]:
```
150-element CategoricalArrays.CategoricalArray{String,1,UInt8}:
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 "setosa"
 ⋮
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
 "virginica"
```

In [15]:
```
fitted_params(mach).tree
```

```
Out[15]: PetalLength < 2.45
         ├─ setosa (50/50)
         └─ PetalWidth < 1.75
            ├─ PetalLength < 4.95
            │  ├─ PetalWidth < 1.65
            │  │  ├─ versicolor (47/47)
            │  │  └─ virginica (1/1)
            │  └─ PetalWidth < 1.55
            │     ├─ virginica (3/3)
            │     └─ SepalLength < 6.95
            │        ├─ versicolor (2/2)
            │        └─ virginica (1/1)
            └─ PetalLength < 4.85
               ├─ SepalWidth < 3.1
               │  ├─ virginica (2/2)
               │  └─ versicolor (1/1)
               └─ virginica (43/43)
```

Confusion Matrix and Overfitting

It is common practice to summarize the performance of a model using a *confusion matrix*, containing the true positives and negatives found by our model on the test data, as well as the false positives and negatives.

In the case of binary classification, a confusion matrix is shaped as follows.

Predicted / Ground truth	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Among the many, three important measures can be obtained by the matrix above: accuracy, precision, and recall. In the binary classification scenario, they are defined as follows.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

In the multi-class scenario, as in our case, we can compute precision and recall individually for each class. For obtaining a unique scalar, we can average all the results.

```
In [16]: cm = confusion_matrix(y_predict, y)
```

Out[16]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	50	0	0
versicol…	0	50	0
virginica	0	0	50

In [17]:
```
# wow! our model is so good!
accuracy(cm)
```

Out[17]: 1.0

How awful! The model we just trained is bad, for sure.

Can you tell why?

Answer (decode from base64encode):
VGhlIGNvZGUgaXMgbm90IGdlbmVyYWxpemluZyEKVGhlIHNwbGl0cyBpbiB0aGUgdHJ

Model Evaluation

Imagine projecting the data points on a bidimensional plane: can you provide a graphical sketch of what is happening during the inference process of the tree trained above?

Let us to obtain a more reliable model.

In [18]:
```
(X_train, X_test), (y_train, y_test) = partition((X, y), 0.7, rng=121, sh
```

In [19]:
```
mach = machine(model, X_train, y_train)
fit!(mach)
y_predict_probabilities = MLJ.predict(mach, X_test)
y_predict = mode.(y_predict_probabilities)
cm = confusion_matrix(y_predict, y_test)
```

[Info: Training machine(DecisionTreeClassifier(max_depth = 5, …), …).

Out[19]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	14	0	0
versicol…	0	13	1
virginica	0	2	15

We can iterate the process above on multiple *folds*, to assess the overall quality of a machine learning training strategy. This technique is commonly called *cross-*

validation.

In the following, the iris dataset will be shuffled and divided into training and test in different ways, and each time a decision tree will be learned and tested over a different portion of the data.

In [20]:
```
acc = evaluate!(
    mach,
    resampling=StratifiedCV(; nfolds = 5, shuffle=true),     # cross valid
    measures=[accuracy]
)
```

```
Evaluating over 5 folds: 100%[=========================] Time: 0:00:02
```

Out[20]:
```
PerformanceEvaluation object with these fields:
  model, measure, operation,
  measurement, per_fold, per_observation,
  fitted_params_per_fold, report_per_fold,
  train_test_rows, resampling, repeats
Extract:
```

measure	operation	measurement
Accuracy()	predict_mode	0.943

per_fold	1.96*SE
[0.952, 0.857, 1.0, 0.952, 0.952]	0.0511

Training with Hyperparameters Tuning

The arguments of `DecisionTreeClassifier(...)` are said to be `hyperparameters`, as they are the meta-parameters exploited for creating a specific algorithm (i.e., the if-else cascade we call decision tree).

Which combination of hyperparameters should we provide?

In this rather lightweight example, we can systematically try many combinations and keep the one which expresses the highest performance.

This technique goes under the name of *grid search*.

In [21]:
```
max_depth_range = range(Int, :max_depth, lower=2, upper=10)
min_samples_leaf_range = range(Int, :min_samples_leaf, lower=1, upper=5)
min_samples_split_range = range(Int, :min_samples_split, lower=2, upper=1
```

In [22]:
```
tuned_tree = TunedModel(
    model = MLJDecisionTreeInterface.DecisionTreeClassifier(),
    resampling = StratifiedCV(nfolds = 5, shuffle = true),
    range = [max_depth_range, min_samples_leaf_range, min_samples_split_r
    measure = accuracy,
    tuning = Grid()
)
```

Out[22]:
```
ProbabilisticTunedModel(
  model = DecisionTreeClassifier(
        max_depth = -1,
        min_samples_leaf = 1,
        min_samples_split = 2,
        min_purity_increase = 0.0,
        n_subfeatures = 0,
        post_prune = false,
        merge_purity_threshold = 1.0,
        display_depth = 5,
        feature_importance = :impurity,
        rng = TaskLocalRNG()),
  tuning = Grid(
        goal = nothing,
        resolution = 10,
        shuffle = true,
        rng = TaskLocalRNG()),
  resampling = StratifiedCV(
        nfolds = 5,
        shuffle = true,
        rng = TaskLocalRNG()),
  measure = Accuracy(),
  weights = nothing,
  class_weights = nothing,
  operation = nothing,
  range = MLJBase.NumericRange{Int64, MLJBase.Bounded, Symbol}[NumericRa
nge(2 ≤ max_depth ≤ 10; origin=6.0, unit=4.0), NumericRange(1 ≤ min_samp
les_leaf ≤ 5; origin=3.0, unit=2.0), NumericRange(2 ≤ min_samples_split
≤ 10; origin=6.0, unit=4.0)],
  selection_heuristic = MLJTuning.NaiveSelection(nothing),
  train_best = true,
  repeats = 1,
  n = nothing,
  acceleration = CPU1{Nothing}(nothing),
  acceleration_resampling = CPU1{Nothing}(nothing),
  check_measure = true,
  cache = true,
  compact_history = true,
  logger = nothing)
```

In [23]:
```
# find the best model, exploring different hyperparameterizations leverag
# cross validation
mach = machine(tuned_tree, X, y)
fit!(mach)
y_predict_probabilities = MLJ.predict(mach, X_test)
y_predict = mode.(y_predict_probabilities)
cm = confusion_matrix(y_predict, y_test)
```

```
[ Info: Training machine(ProbabilisticTunedModel(model = DecisionTreeClass
ifier(max_depth = -1, …), …), …).
[ Info: Attempting to evaluate 405 models.
Evaluating over 405 metamodels: 100%[=========================] Time: 0:0
0:00
```

Out[23]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	14	0	0
versicol…	0	13	0
virginica	0	2	16

Training Forests

In [24]:

```
try
    RandomForestClassifier = @load RandomForestClassifier pkg=DecisionTre
catch
    println("The RandomForestClassifier symbol has already been imported.
end
```

```
import MLJDecisionTreeInterface ✔
[ Info: For silent loading, specify `verbosity=0`.
```

Out[24]: MLJDecisionTreeInterface.RandomForestClassifier

In [25]:

```
forest = MLJDecisionTreeInterface.RandomForestClassifier(;
    max_depth=3, min_samples_leaf=1, min_samples_split=2, n_trees=10)

forestmach = machine(forest, X_train, y_train)

MLJ.fit!(forestmach, verbosity=0)

y_predict_probabilities = MLJ.predict(forestmach, X_test)
y_predict = mode.(y_predict_probabilities)

cm = confusion_matrix(y_predict, y_test)
```

Out[25]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	14	0	0
versicol…	0	12	1
virginica	0	3	15

Learning with Sole.jl

Tabular Datasets and Logisets

Symbolic AI treats tabular datasets, such as the iris flower, as sets of propositional interpretations, onto which formulas of propositional logic are interpreted.

Look at the (classical) tabular dataset $\mathcal{I}$ below. We denote instances with I, and *variables*[1], as V_i.

	V_1	V_2	V_3
I_1	1.2	$[1,2,3]$	A
I_2	1.3	$[9,7,6]$	B
I_3	0.8	$[2,8,2]$	C
I_4	1.1	$[1,3,7]$	B
I_5	1.2	$[4,3,3]$	B

We can change the point of view on the table above from a statistical to a logical one, called a *logiset*.

This requires the definition of a propositional alphabet $\mathcal{P}$.

Consider $\mathcal{P} = \{p, q, r\}$, with:

$$p \coloneqq V_1 \geq 1$$

$$q \coloneqq \mathrm{sum}(V_2) < 13$$

$$r \coloneqq V_3 = \mathrm{B}$$

We denote the truth constant with $\top$ (top), and the false constant with $\bot$ (bot).

The resulting (propositional) logiset $\mathcal{I}_{\mathcal{P}}$ is:

	p	q	r
I_1	$\top$	$\top$	$\bot$
I_2	$\top$	$\bot$	$\top$
I_3	$\bot$	$\top$	$\bot$
I_4	$\top$	$\top$	$\top$
I_5	$\top$	$\top$	$\top$

[1] We use the term "variable" to denote, in general, a column of the tabular dataset: this corresponds to a raw attribute or a *feature* (a processed attribute).

In [26]:
```
using MLJBase
using SoleData
```

In [27]:
```
X_logiset = PropositionalLogiset(data);
X_logiset.tabulardataset == data
```

Out[27]:
```
true
```

In [28]:
```
phi = parseformula(
    "SepalLength > 5.8 ∧ SepalWidth < 3.0 ∨ Species == \"setosa\"";
    atom_parser = a->Atom(
        parsecondition(
            SoleData.ScalarCondition, a;
            featuretype = SoleData.VariableValue
        )
    ),
```

```
)
```

```
┌ Warning: Please, specify a type for the feature values (featvaltype = ...). Real will be used, but note that this may raise type errors. (expr = "SepalLength > 5.8")
└ @ SoleData ~/.julia/packages/SoleData/N5aE8/src/scalar/conditions.jl:330
┌ Warning: Please, specify a type for the feature values (featvaltype = ...). Real will be used, but note that this may raise type errors. (expr = "SepalWidth < 3.0")
└ @ SoleData ~/.julia/packages/SoleData/N5aE8/src/scalar/conditions.jl:330
┌ Warning: Please, specify a type for the feature values (featvaltype = ...). Real will be used, but note that this may raise type errors. (expr = "Species == \"setosa\"")
└ @ SoleData ~/.julia/packages/SoleData/N5aE8/src/scalar/conditions.jl:330
```

Out[28]:
```
SyntaxBranch: (SepalLength > 5.8 ∧ SepalWidth < 3.0) ∨ Species == setosa
```

In [29]:
```julia
# check(phi, SoleLogics.LogicalInstance(X_logiset, 1))
check(phi, X_logiset, 1)
```

Out[29]:
```
true
```

From DecisionTree.jl to SoleModels.jl

If we manage to make an existing model compliant with the interface of `SoleModels` package, then we can play with it from a logical standpoint.

In [30]:
```julia
using SoleModels
```

In [31]:
```julia
mach = machine(model, X_train, y_train)
fit!(mach)

# \:seedling:
🌱 = fitted_params(mach).tree
```

```
[ Info: Training machine(DecisionTreeClassifier(max_depth = 5, …), …).
```

Out[31]:
```
PetalLength < 2.6
├─ setosa (36/36)
└─ PetalWidth < 1.65
    ├─ PetalLength < 4.95
    │   ├─ versicolor (34/34)
    │   └─ SepalLength < 6.15
    │       ├─ SepalWidth < 2.45
    │       │   ├─ virginica (1/1)
    │       │   └─ versicolor (1/1)
    │       └─ virginica (2/2)
    └─ virginica (31/31)
```

In [32]:
```julia
# we encode the model in such a way that it can be investigated via SoleM
# \:evergreen_tree:
🌲 = solemodel(🌱)
printmodel(🌲)
```

```
▣ ([PetalLength] < 2.5999999999999996)
├✔ setosa
└✘ ([PetalWidth] < 1.65)
  ├✔ ([PetalLength] < 4.95)
  │ ├✔ versicolor
  │ └✘ ([SepalLength] < 6.15)
  │   ├✔ ([SepalWidth] < 2.45)
  │   │ ├✔ virginica
  │   │ └✘ versicolor
  │   └✘ virginica
  └✘ virginica
```

In [33]:

```
# these are all the logical rules encoded by the tree
listrules(🌲)
```

Out[33]:

```
6-element Vector{ClassificationRule{String}}:
 ▣ ([PetalLength] < 2.5999999999999996)  ↣  setosa

 ▣ (([PetalLength] ≥ 2.5999999999999996)) ∧ (([PetalWidth] < 1.65)) ∧
(([PetalLength] < 4.95))  ↣  versicolor

 ▣ (([PetalLength] ≥ 2.5999999999999996)) ∧ (([PetalWidth] < 1.65)) ∧
(([PetalLength] ≥ 4.95)) ∧ (([SepalLength] < 6.15)) ∧ (([SepalWidth] <
2.45))  ↣  virginica

 ▣ (([PetalLength] ≥ 2.5999999999999996)) ∧ (([PetalWidth] < 1.65)) ∧
(([PetalLength] ≥ 4.95)) ∧ (([SepalLength] < 6.15)) ∧ (([SepalWidth] ≥
2.45))  ↣  versicolor

 ▣ (([PetalLength] ≥ 2.5999999999999996)) ∧ (([PetalWidth] < 1.65)) ∧
(([PetalLength] ≥ 4.95)) ∧ (([SepalLength] ≥ 6.15))  ↣  virginica

 ▣ (([PetalLength] ≥ 2.5999999999999996)) ∧ (([PetalWidth] ≥ 1.65))  ↣
virginica
```

In [34]:

```
metricstable(🌲)
```

```
┌──────────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────────────┬──────────
──┬─────────────┬──────────┬─────────────┬─────────────┬──────────┬────────┐
│
Antecedent │ Consequent │ ninstances │ ncovered │   coverage │ confidence
│    lift │ natoms │
├──────────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────────────┼──────────
──┼─────────────┼──────────┼─────────────┼─────────────┼──────────┼────────┤
│
[PetalLength] < 2.5999999999999996 │     setosa │        105 │       36 │
0.342857 │        1.0 │ 2.91667 │      1 │
│                                                   ([PetalLength] ≥ 2.5999
999999999996) ∧ ([PetalWidth] < 1.65) ∧ ([PetalLength] < 4.95) │ versicolo
r │        105 │        34 │    0.32381 │         1.0 │     3.0 │       3 │
│ ([PetalLength] ≥ 2.5999999999999996) ∧ ([PetalWidth] < 1.65) ∧ ([PetalLe
ngth] ≥ 4.95) ∧ ([SepalLength] < 6.15) ∧ ([SepalWidth] < 2.45) │  virginic
a │        105 │         1 │ 0.00952381 │         1.0 │ 3.08824 │       5 │
│ ([PetalLength] ≥ 2.5999999999999996) ∧ ([PetalWidth] < 1.65) ∧ ([PetalLe
ngth] ≥ 4.95) ∧ ([SepalLength] < 6.15) ∧ ([SepalWidth] ≥ 2.45) │ versicolo
r │        105 │         1 │ 0.00952381 │         1.0 │     3.0 │       5 │
│                          ([PetalLength] ≥ 2.5999999999999996) ∧ ([PetalWi
dth] < 1.65) ∧ ([PetalLength] ≥ 4.95) ∧ ([SepalLength] ≥ 6.15) │  virginic
a │        105 │         2 │  0.0190476 │         1.0 │ 3.08824 │       4 │
│
([PetalLength] ≥ 2.5999999999999996) ∧ ([PetalWidth] ≥ 1.65) │  virginica
│        105 │       31 │   0.295238 │        1.0 │ 3.08824 │      2 │
└──────────────────────────────────────────────────────────────────────────
───────────────────────────────────────────────────────────────┴──────────
──┴─────────────┴──────────┴─────────────┴─────────────┴──────────┴────────┘
```

In [35]:
```
# show all the testing instances to the tree, and compare the metrics
# with the testing samples
apply!(🌲, X_test, y_test);
```

In [36]:
```
# we can visualize how our model behaved at testing time
metricstable(
    🌲;
    normalize = true,
    metrics_kwargs = (;
        additional_metrics = (;
            height = r->SoleLogics.height(antecedent(r))
        )
    )
)
```

Antecedent	Consequent	ninstances	ncovered	coverage	confidence	lift	natoms	height
[PetalLength] < 2.5999999999999996	setosa	45	14	0.311111	1.0	3.21429	1	0
([PetalLength] ∈ [2.5999999999999996,4.95)) ∧ ([PetalWidth] < 1.65)	versicolor	45	13	0.288889	1.0	3.0	2	1
([PetalLength] ≥ 4.95) ∧ ([PetalWidth] < 1.65) ∧ ([SepalLength] < 6.15) ∧ ([SepalWidth] < 2.45)	virginica	45	0	0.0	NaN	NaN	4	3
([PetalLength] ≥ 4.95) ∧ ([PetalWidth] < 1.65) ∧ ([SepalLength] < 6.15) ∧ ([SepalWidth] ≥ 2.45)	versicolor	45	1	0.0222222	0.0	0.0	4	3
([PetalLength] ≥ 4.95) ∧ ([PetalWidth] < 1.65) ∧ ([SepalLength] ≥ 6.15)	virginica	45	0	0.0	NaN	NaN	3	2
([PetalLength] ≥ 2.5999999999999996) ∧ ([PetalWidth] ≥ 1.65)	virginica	45	17	0.377778	0.882353	2.48162	2	1

In [37]:
```
# join some rules for the same class into a single, sufficient and necess
# condition for the same class
metricstable(joinrules(🌲; min_ncovered = 1, normalize = true))
```

Antecedent	Consequent	ninstances	ncovered	coverage	confidence	lift	natoms
[PetalLength] < 2.5999999999999996	setosa	45	14	0.311111	1.0	3.21429	1
(([PetalLength] ∈ [2.5999999999999996,4.95)) ∧ ([PetalWidth] < 1.65)) ∨ (([PetalLength] ≥ 4.95) ∧ ([PetalWidth] < 1.65) ∧ ([SepalLength] < 6.15) ∧ ([SepalWidth] ≥ 2.45))	versicolor	90	14	0.155556	0.928571	2.78571	6
[PetalLength] ≥ 2.5999999999999996 ∧ [PetalWidth] ≥ 1.65	virginica	45	17	0.377778	0.882353	2.48162	2

Here, we are just scratching the surface of Sole framework, limiting ourselves to pretty printings.

In the next lessons, we will enhance the machine learning pipeline we introduced today, with spatial reasoning considerations.

Below, there is a little spoiler about a fancy machine learning model, which is general enough for dealing with more-than-propositional logics.

In [38]:
```
using ModalDecisionTrees

mdt_model = ModalDecisionTree()
mach = machine(mdt_model, X_test, y_test)
fit!(mach)
y_pred = predict_mode(mach)
cm = confusion_matrix(y_predict, y_test)
```

```
[ Info: Precomputing logiset...
[ Info: Training machine(ModalDecisionTree(max_depth = nothing, …), …).
```

Out[38]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	14	0	0
versicol…	0	12	1
virginica	0	3	15

Exercise

Try to write your own pipeline, considering the seeds dataset.

In [39]:
```
using CSV
```

In [40]:
```
SEEDS_PATH = joinpath("..", "datasets", "seeds.csv")
data = DataFrame(
    CSV.File(SEEDS_PATH; header=false)
)
```

Out[40]: 210×8 DataFrame *185 rows omitted*

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Co
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Int
1	15.26	14.84	0.871	5.763	3.312	2.221	5.22	
2	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	
3	14.29	14.09	0.905	5.291	3.337	2.699	4.825	
4	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	
5	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	
6	14.38	14.21	0.8951	5.386	3.312	2.462	4.956	
7	14.69	14.49	0.8799	5.563	3.259	3.586	5.219	
8	14.11	14.1	0.8911	5.42	3.302	2.7	5.0	
9	16.63	15.46	0.8747	6.053	3.465	2.04	5.877	
10	16.44	15.25	0.888	5.884	3.505	1.969	5.533	
11	15.26	14.85	0.8696	5.714	3.242	4.543	5.314	
12	14.03	14.16	0.8796	5.438	3.201	1.717	5.001	
13	13.89	14.02	0.888	5.439	3.199	3.986	4.738	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
199	12.62	13.67	0.8481	5.41	2.911	3.306	5.231	
200	12.76	13.38	0.8964	5.073	3.155	2.828	4.83	
201	12.38	13.44	0.8609	5.219	2.989	5.472	5.045	
202	12.67	13.32	0.8977	4.984	3.135	2.3	4.745	
203	11.18	12.72	0.868	5.009	2.81	4.051	4.828	
204	12.7	13.41	0.8874	5.183	3.091	8.456	5.0	
205	12.37	13.47	0.8567	5.204	2.96	3.919	5.001	
206	12.19	13.2	0.8783	5.137	2.981	3.631	4.87	
207	11.23	12.88	0.8511	5.14	2.795	4.325	5.003	
208	13.2	13.66	0.8883	5.236	3.232	8.315	5.056	
209	11.84	13.21	0.8521	5.175	2.836	3.598	5.044	
210	12.3	13.34	0.8684	5.243	2.974	5.637	5.063	

In [41]:

```
# write your pipeline here
```