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

```
  Activating project at `~/logic-and-machine-learning`
Status `~/logic-and-machine-learning/Project.toml`
^ [da404889] ARFFFiles v1.5.0
  [6e4b80f9] BenchmarkTools v1.6.3
  [336ed68f] CSV v0.10.15
  [159f3aea] Cairo v1.1.1
  [861a8166] Combinatorics v1.1.0
  [a93c6f00] DataFrames v1.8.1
⌅ [864edb3b] DataStructures v0.18.22
  [7806a523] DecisionTree v0.12.4
  [186bb1d3] Fontconfig v0.4.1
  [271df9f8] FuzzyLogic v0.1.3
  [a2cc645c] GraphPlot v0.6.2
^ [86223c79] Graphs v1.13.1
  [6a3955dd] ImageFiltering v0.7.12
  [f7bf1975] Impute v0.6.13
  [23992714] MAT v0.11.4
⌅ [add582a8] MLJ v0.20.9
^ [a7f614a8] MLJBase v1.9.2
^ [c6f25543] MLJDecisionTreeInterface v0.4.2
  [b59e7f69] ManyExpertDecisionTrees v1.0.0 `https://github.com/aclai-lab/
ManyExpertDecisionTrees.jl#`
  [24e37439] MatrixProfile v1.1.1
  [fb95e5f7] ModalAssociationRules v0.2.1
  [e54bda2e] ModalDecisionTrees v0.5.2
  [8cc5100c] MultiData v0.1.4
  [91a5bcdd] Plots v1.41.4
  [ce6b1742] RDatasets v0.8.1
  [4475fa32] SoleBase v0.13.4
  [123f1ae1] SoleData v0.16.7
  [b002da8f] SoleLogics v0.13.7
  [4249d9c7] SoleModels v0.10.6
  [eb5c4719] SoleReasoners v0.1.0 `https://github.com/aclai-lab/SoleReason
ers.jl#embedding#main`
  [2913bbd2] StatsBase v0.34.10
  [9a3f8284] Random v1.11.0
  [9e88b42a] Serialization v1.11.0
Info Packages marked with ^ and ⌅ have new versions available. Those with
^ may be upgradable, but those with ⌅ are restricted by compatibility cons
traints from upgrading. To see why use `status --outdated`
```

In [2]:
```julia
using Random
Random.seed!(1605)
```

Out[2]: `TaskLocalRNG()`

Learning with Modal Decision Trees

Let us try to tackle the Natops dataset with what we learned in the previous days.

In [3]:
```julia
using ARFFFiles
```

```julia
using DataFrames
using MLJ
using Plots
using Random
using StatsBase
using SoleData
using SoleModels
```

In [4]:
```julia
function parse_natops(arffstring::String)
    df = DataFrame()
    classes = String[]

    lines = split(arffstring, "\n")
    for i in 1:length(lines)
        line = lines[i]

        # split the current line;
        # if it is not a data line, starting with DATA_MARK, continue;
        # continue even in the case where checking the first character th
        # out an error.
        sline = nothing
        try
            sline = split(line, " ")
            if sline[1][1] != '\''
                continue
            end
        catch
            continue
        end

        # skip the initial hypen an read the data
        sline[1] = sline[1][2:end]
        data_and_class = split(sline[1], "\'")
        string_data = split(data_and_class[1], "\\n")
        class = data_and_class[2][2:end]

        if isempty(names(df))
            for i in 1:length(string_data)
                insertcols!(df, Symbol("V$(i)") => Array{Float64, 1}[])
            end
        end

        float_data = Dict{Int,Vector{Float64}}()

        for i in 1:length(string_data)
            float_data[i] = map(x->parse(Float64,x), split(string_data[i]
        end

        push!(df, [float_data[i] for i in 1:length(string_data)])
        push!(classes, class)

    end

    p = sortperm(eachrow(df), by=x->classes[rownumber(x)])

    return df[p, :], classes[p]
end
```

Out[4]: parse_natops (generic function with 1 method)

In [5]:
```
X, y = read(
    joinpath(@__DIR__, "..", "datasets", "natops.arff"),
    String
) |> parse_natops
```

Out[5]:
```
(360×24 DataFrame
 Row │ V1                                 V2
V …
     │ Array…                             Array…
A …
─────┼───────────────────────────────────────────────────────────────────
────────
   1 │ [-0.519771, -0.52758, -0.531415,…  [-2.14011, -2.18043, -2.18425,
-…  [ …
   2 │ [-0.489753, -0.48607, -0.484529,…  [-1.55293, -1.54966, -1.55206,
-…  [
   3 │ [-0.521346, -0.518394, -0.522321…  [-1.72326, -1.72407, -1.72326,
-…  [
   4 │ [-0.57022, -0.562064, -0.565967,…  [-1.91196, -1.90369, -1.90527,
-…  [
   5 │ [-0.624417, -0.626031, -0.625388…  [-1.84287, -1.84026, -1.84688,
-…  [ …
   6 │ [-0.502501, -0.502525, -0.499415…  [-2.17556, -2.15613, -2.18516,
-…  [
   7 │ [-0.488461, -0.489463, -0.487539…  [-2.17242, -2.18203, -2.18057,
-…  [
   8 │ [-0.468105, -0.410602, -0.473909…  [-1.86535, -1.89011, -1.87105,
-…  [
   9 │ [-0.568195, -0.572936, -0.571337…  [-1.79059, -1.78162, -1.78303,
-…  [ …
  10 │ [-0.517579, -0.515374, -0.517325…  [-1.73887, -1.74072, -1.7397,
-1…  [
  11 │ [-0.631494, -0.629032, -0.630474…  [-1.98071, -1.98581, -1.98407,
-…  [
  ⋮  │                 ⋮                                  ⋮
⋱
 351 │ [-0.937442, -1.02995, -0.985338,…  [-2.10495, -1.96835, -1.95082,
-…  [
 352 │ [-0.595625, -0.593605, -0.588495…  [-1.93897, -1.93641, -1.93761,
-…  [ …
 353 │ [-0.466582, -0.469372, -0.43454,…  [-1.71874, -1.72861, -1.58691,
-…  [
 354 │ [-0.500404, -0.502824, -0.504771…  [-1.89209, -1.88986, -1.89188,
-…  [
 355 │ [-0.686893, -0.690966, -0.710514…  [-2.04375, -2.05011, -2.07035,
-…  [
 356 │ [-0.525938, -0.516073, -0.5177, …  [-1.69259, -1.70519, -1.70824,
-…  [ …
 357 │ [-0.440887, -0.452221, -0.447185…  [-1.90444, -1.91917, -1.91652,
-…  [
 358 │ [-0.647672, -0.653511, -0.642305…  [-1.6173, -1.61051, -1.60491,
-1…  [
 359 │ [-0.476117, -0.4705, -0.474443, …  [-1.70846, -1.7099, -1.70912,
-1…  [
 360 │ [-0.553245, -0.551704, -0.548044…  [-1.69493, -1.69349, -1.69951,
-…  [ …
                                                22 columns and 339 rows
omitted, ["1.0", "1.0", "1.0", "1.0", "1.0", "1.0", "1.0", "1.0", "1.0",
"1.0"  …  "6.0", "6.0", "6.0", "6.0", "6.0", "6.0", "6.0", "6.0", "6.0",
"6.0"])
```

In [6]:
```
variablenames = [
    "X[Hand tip l]", "Y[Hand tip l]", "Z[Hand tip l]",
    "X[Hand tip r]", "Y[Hand tip r]", "Z[Hand tip r]",
    "X[Elbow l]", "Y[Elbow l]", "Z[Elbow l]",
```

```julia
    "X[Elbow r]", "Y[Elbow r]", "Z[Elbow r]",
    "X[Wrist l]", "Y[Wrist l]", "Z[Wrist l]",
    "X[Wrist r]", "Y[Wrist r]", "Z[Wrist r]",
    "X[Thumb l]", "Y[Thumb l]", "Z[Thumb l]",
    "X[Thumb r]", "Y[Thumb r]", "Z[Thumb r]",
]

classnames = [
    "I have command",
    "All clear",
    "Not clear",
    "Spread wings",
    "Fold wings",
    "Lock wings"
]

try
    X = map(i -> variablenames[round(Int, parse(Float64, i))], X)
    y = map(i -> classnames[round(Int, parse(Float64, i))], y)
catch
    println(
        "You already converted the variable and class names to human " *
        "readable strings."
    )
end
```

```
You already converted the variable and class names to human readable strin
gs.
```

In [7]:

```julia
X_ninstances, X_nattributes = size(X)
X_ndatapoints = length(X[1,1])

println("Number of instances: $(X_ninstances)")
println("Number of attributes: $(X_nattributes)")
println("Number of datapoints for each attribute: $(X_ndatapoints)")
```

```
Number of instances: 360
Number of attributes: 24
Number of datapoints for each attribute: 51
```

In [8]:

```julia
# for every combination of instance and attributes,
# we are still dealing with the same number of datapoints (51)
all(
    i -> length(X[i[1],i[2]]) == X_ndatapoints,
    Iterators.product(1:X_ninstances, 1:X_nattributes)
)
```

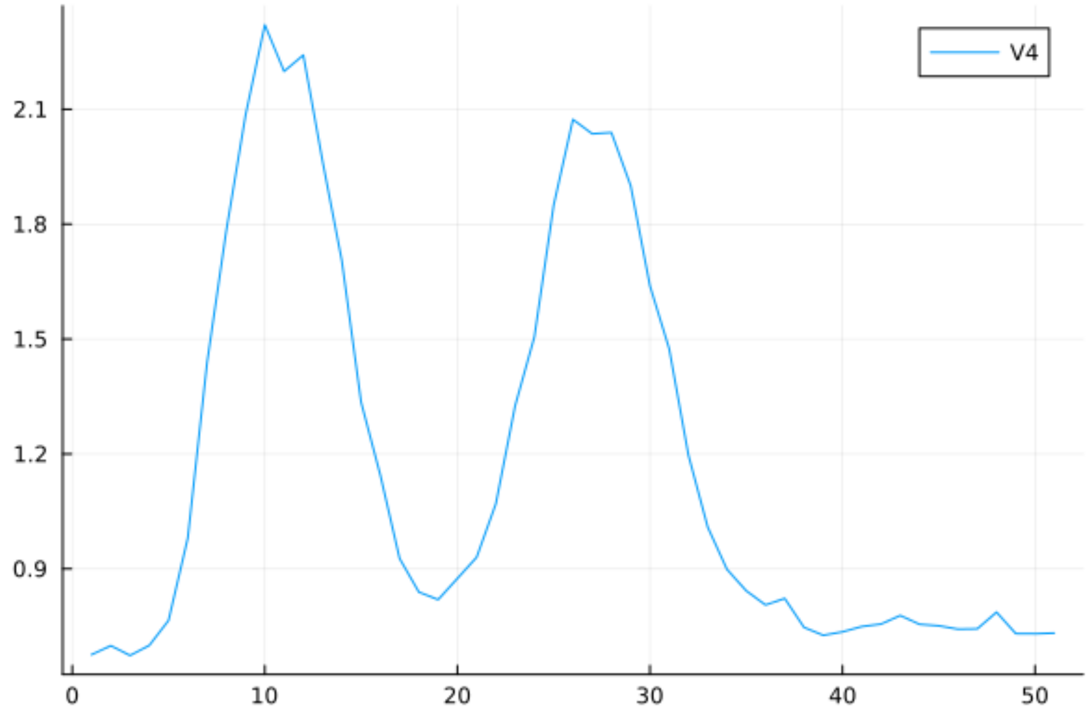
Out[8]:

```
true
```

In [9]:

```julia
# try to change the target attribute
_attribute = 4
plot(X[1,_attribute], label = names(X)[_attribute])
```

Out[9]:

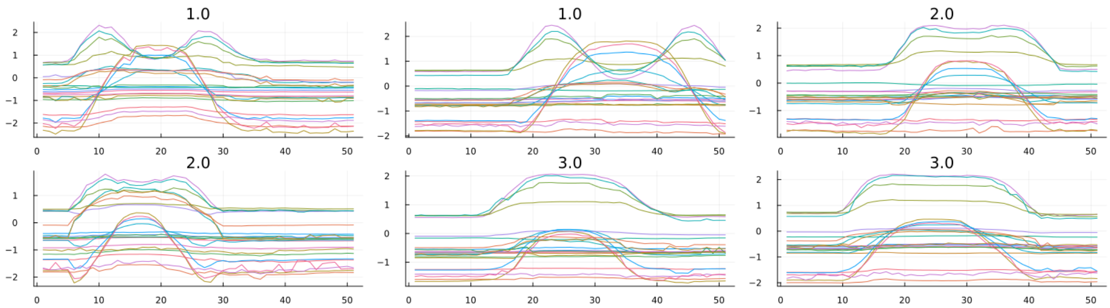


In [10]:
```
countmap(y)
```

Out[10]:
```
Dict{String, Int64} with 6 entries:
  "6.0" => 60
  "2.0" => 60
  "3.0" => 60
  "4.0" => 60
  "1.0" => 60
  "5.0" => 60
```

In [11]:
```
 # let us summarize one instance for each class
plot(map(i ->
    plot(collect(X[i,:]),
        labels=nothing,
        title=y[i]),
        1:30:180
    )...,
    layout = (2, 3),
    size = (1500,400)
)
```

Out[11]:



In [12]:
```
# length of X[hand tip l] of the first instance
length(X[1,1])
```

Out[12]: 51

In [13]:
```
# each instance can be shaped as a Kripke Frame, whose worlds encode all
```

```
# intervals in the range [1, 51] (including the degenerate, punctual case
# as [1, 1])
fr = SoleLogics.frame(X, 1)
```

Out[13]:
```
FullDimensionalFrame{1, SoleLogics.Interval{Int64}}((51,))
```

In [14]:
```
allworlds(fr) |> collect
```

Out[14]:
```
1326-element Vector{SoleLogics.Interval{Int64}}:
 SoleLogics.Interval{Int64}(1, 2)
 SoleLogics.Interval{Int64}(1, 3)
 SoleLogics.Interval{Int64}(2, 3)
 SoleLogics.Interval{Int64}(1, 4)
 SoleLogics.Interval{Int64}(2, 4)
 SoleLogics.Interval{Int64}(3, 4)
 SoleLogics.Interval{Int64}(1, 5)
 SoleLogics.Interval{Int64}(2, 5)
 SoleLogics.Interval{Int64}(3, 5)
 SoleLogics.Interval{Int64}(4, 5)
 SoleLogics.Interval{Int64}(1, 6)
 SoleLogics.Interval{Int64}(2, 6)
 SoleLogics.Interval{Int64}(3, 6)
 ⋮
 SoleLogics.Interval{Int64}(40, 52)
 SoleLogics.Interval{Int64}(41, 52)
 SoleLogics.Interval{Int64}(42, 52)
 SoleLogics.Interval{Int64}(43, 52)
 SoleLogics.Interval{Int64}(44, 52)
 SoleLogics.Interval{Int64}(45, 52)
 SoleLogics.Interval{Int64}(46, 52)
 SoleLogics.Interval{Int64}(47, 52)
 SoleLogics.Interval{Int64}(48, 52)
 SoleLogics.Interval{Int64}(49, 52)
 SoleLogics.Interval{Int64}(50, 52)
 SoleLogics.Interval{Int64}(51, 52)
```

In [15]:
```
using SoleLogics: Interval

# enumerate the intervals that are "Later" than [1,10]
collect(accessibles(fr, Interval(1,10), IA_L))
```

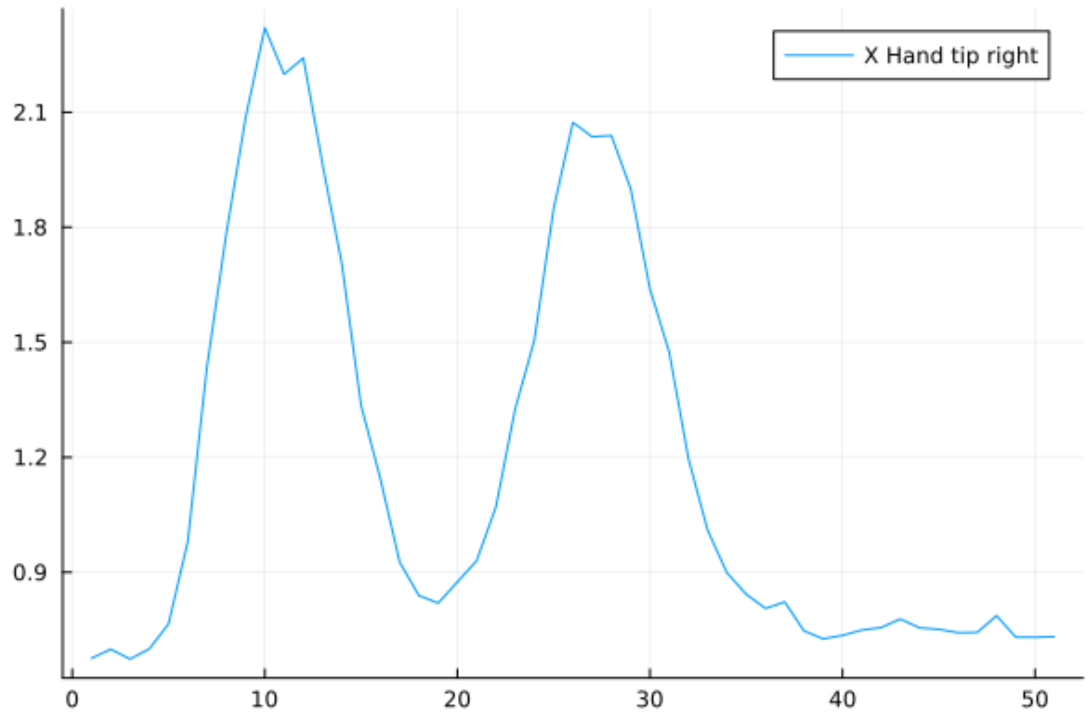
Out[15]:
```
861-element Vector{Interval{Int64}}:
 Interval{Int64}(11, 12)
 Interval{Int64}(11, 13)
 Interval{Int64}(12, 13)
 Interval{Int64}(11, 14)
 Interval{Int64}(12, 14)
 Interval{Int64}(13, 14)
 Interval{Int64}(11, 15)
 Interval{Int64}(12, 15)
 Interval{Int64}(13, 15)
 Interval{Int64}(14, 15)
 Interval{Int64}(11, 16)
 Interval{Int64}(12, 16)
 Interval{Int64}(13, 16)
 ⋮
 Interval{Int64}(40, 52)
 Interval{Int64}(41, 52)
 Interval{Int64}(42, 52)
 Interval{Int64}(43, 52)
 Interval{Int64}(44, 52)
 Interval{Int64}(45, 52)
 Interval{Int64}(46, 52)
 Interval{Int64}(47, 52)
 Interval{Int64}(48, 52)
 Interval{Int64}(49, 52)
 Interval{Int64}(50, 52)
 Interval{Int64}(51, 52)
```

In [16]:
```
# we compute the value of a certain feature on each world where we can
feature = SoleData.VariableMax(4)
```

Out[16]:
```
VariableMax{Int64}: max[V4]
```

In [17]:
```
plot(X[1, 4], labels="X Hand tip right")
```

Out[17]:



In [18]:
```
SoleData.featvalue(feature, X, 1, Interval(10, 30))
```

Out[18]: 2.320374

In [19]:
```julia
# when we are interested in windowing the data, it is easy to transform a
# dataset into a Kripke Model
Xk = scalarlogiset(X)
```

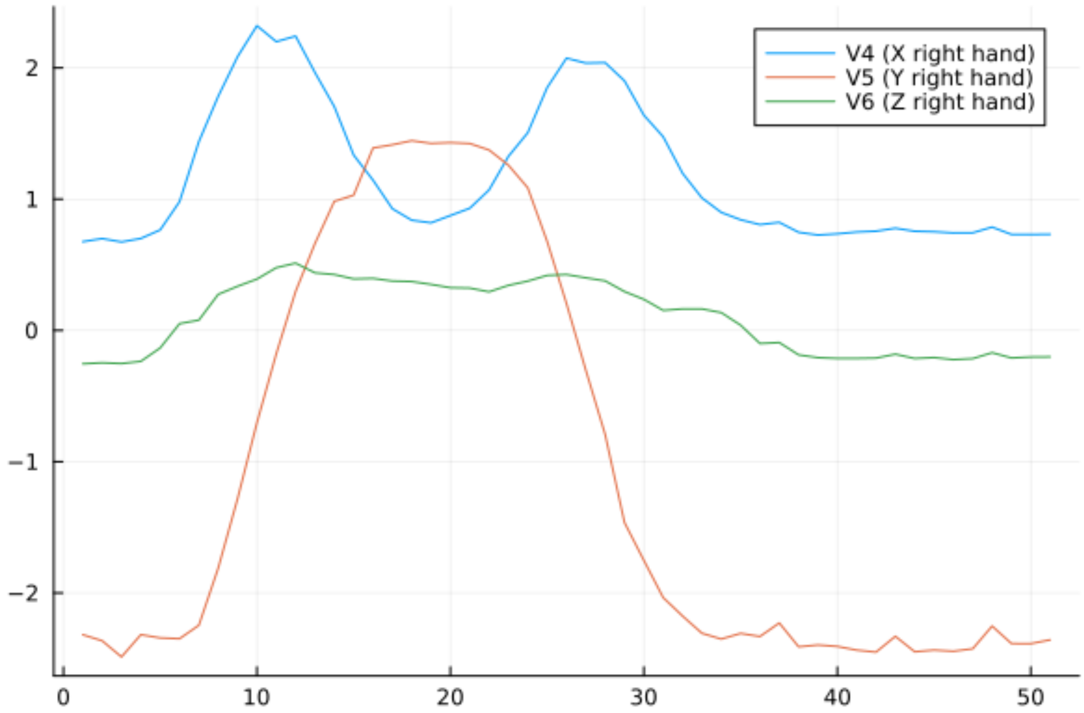
Out[19]:
```
SupportedLogiset with 1 support (342.94 MBs)
├ worldtype:                   Interval{Int64}
├ featvaltype:                 Float64
├ featuretype:                 SoleData.AbstractUnivariateFeature
├ frametype:                   FullDimensionalFrame{1, Interval{Int64}}
├ # instances:                 360
├ usesfullmemo:                true
├[BASE] UniformFullDimensionalLogiset of channel size (51,) (342.91 MBs)
│ ├ size × eltype:              (51, 51, 360, 48) × Float64
│ └ features:                   48 -> SoleData.AbstractUnivariateFeature
[max[V1], min[V1], max[V2], min[V2], ..., min[V22], max[V23], min[V23],
max[V24], min[V24]]
└[SUPPORT 1] FullMemoset (0 memoized values, 31.03 KBs))
```

In [20]:
```julia
# we can check custom conditions over the logiset we just created
p = Atom(ScalarCondition(feature, <, 1.0))
check(p, Xk, 1, Interval(10, 30))
```

Out[20]: false

In [21]:
```julia
plot(
    collect(X[1, 4:6]),
    labels=["V4 (X right hand)" "V5 (Y right hand)" "V6 (Z right hand)"]
)
```

Out[21]:



In [22]:
```julia
p = Atom(ScalarCondition(VariableMin(4), >, 1.0))
q = Atom(ScalarCondition(VariableMax(5), <=, 3.0))
r = Atom(ScalarCondition(VariableMax(6), <=, 0.0))

phi = ¬p ∨ (q ∧ r)
```

```julia
println(syntaxstring(phi))

check(phi, SoleLogics.LogicalInstance(Xk, 1), Interval(10, 30))
```

```
¬min[V4] > 1.0 ∨ (max[V5] ≤ 3.0 ∧ max[V6] ≤ 0.0)
```

Out[22]:
```
true
```

Let us try to check some modal formulae.

In [23]:
```julia
boxlater = box(SoleLogics.IA_A)
```

Out[23]:
```
BoxRelationalConnective{SoleLogics._IA_A}: [A]
```

In [24]:
```julia
later_always_phi = boxlater(phi)
```

Out[24]:
```
SyntaxBranch: [A](¬min[V4] > 1.0 ∨ (max[V5] ≤ 3.0 ∧ max[V6] ≤ 0.0))
```

In [25]:
```julia
check(later_always_phi, SoleLogics.LogicalInstance(Xk, 1), Interval(10, 3
```

Out[25]:
```
false
```

In [26]:
```julia
SoleLogics.getinstance(Xk, 1)
```

Out[26]:
```
SoleLogics.LogicalInstance{SupportedLogiset{Interval{Int64}, Float64, So
leData.AbstractUnivariateFeature, FullDimensionalFrame{1, Interval{Int6
4}}, SoleData.DimensionalDatasets.UniformFullDimensionalLogiset{Float64,
Interval{Int64}, 1, Array{Float64, 4}, SoleData.AbstractUnivariateFeatur
e, FullDimensionalFrame{1, Interval{Int64}}}, 1, Tuple{SoleData.FullMemo
set{Interval{Int64}, Vector{ThreadSafeDicts.ThreadSafeDict{SyntaxTree, V
ector{Interval{Int64}}}}}}}}(SupportedLogiset with 1 support (343.06 MB
s)
├ worldtype:                   Interval{Int64}
├ featvaltype:                 Float64
├ featuretype:                 SoleData.AbstractUnivariateFeature
├ frametype:                   FullDimensionalFrame{1, Interval{Int64}}
├ # instances:                 360
├ usesfullmemo:                true
├[BASE] UniformFullDimensionalLogiset of channel size (51,) (342.91 MBs)
│ ├ size × eltype:              (51, 51, 360, 48) × Float64
│ └ features:                   48 -> SoleData.AbstractUnivariateFeature
[max[V1], min[V1], max[V2], min[V2], ..., min[V22], max[V23], min[V23],
max[V24], min[V24]]
└[SUPPORT 1] FullMemoset (12 memoized values, 157.27 KBs))
, 1)
```

In [27]:
```julia
# let us try with an even more complex scenario
check_mask = zeros(Int64, 51)
for i in 1:X_ndatapoints
    check_mask[i] = check(
        phi,
        SoleLogics.LogicalInstance(Xk, i),
        Interval(1,30)
    )
end

println(check_mask)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]
```

Modal Decision Trees

In [28]:
```julia
using SoleBase
using ModalDecisionTrees
```

In [29]:
```julia
# the experiment we are just going to execute could be too
# heavy for standard commodity hardware;
# we can reduce data dimensionality via a moving window
X_small = broadcast(
    x -> movingwindow(mean, x; nwindows = 10, relative_overlap = 0.2),
    X
)

X_small_ninstances, X_small_nattributes = size(X_small)
X_small_ndatapoints = length(X_small[1,1])

println(
    "The number of datapoints changed from $(X_ndatapoints) to " *
    "$(X_small_ndatapoints)"
)
```

```
The number of datapoints changed from 51 to 10
```

In [30]:
```julia
features = [maximum, minimum]
Xk_small = scalarlogiset(X_small, features)
```

Out[30]:
```
SupportedLogiset with 1 support (13.22 MBs)
├ worldtype:                   Interval{Int64}
├ featvaltype:                 Float64
├ featuretype:                 SoleData.AbstractUnivariateFeature
├ frametype:                   FullDimensionalFrame{1, Interval{Int64}}
├ # instances:                 360
├ usesfullmemo:                true
├[BASE] UniformFullDimensionalLogiset of channel size (10,) (13.19 MBs)
│ ├ size × eltype:              (10, 10, 360, 48) × Float64
│ └ features:                   48 -> SoleData.AbstractUnivariateFeature
[max[V1], min[V1], max[V2], min[V2], ..., min[V22], max[V23], min[V23],
max[V24], min[V24]]
└[SUPPORT 1] FullMemoset (0 memoized values, 31.03 KBs))
```

In [31]:
```julia
model = ModalDecisionTree(; relations = :IA, features = [minimum, maximum
```

Out[31]:
```
ModalDecisionTree(
  max_depth = nothing,
  min_samples_leaf = 4,
  min_purity_increase = 0.002,
  max_purity_at_leaf = Inf,
  max_modal_depth = nothing,
  relations = :IA,
  features = nothing,
  conditions = Function[minimum, maximum],
  featvaltype = Float64,
  initconditions = nothing,
  downsize = SoleData.var"#downsize#541"(),
  force_i_variables = true,
  fixcallablenans = false,
  print_progress = false,
  rng = TaskLocalRNG(),
  display_depth = nothing,
  min_samples_split = nothing,
  n_subfeatures = identity,
  post_prune = false,
  merge_purity_threshold = nothing,
  feature_importance = :split)
```

In [32]:
```
(X_small_train, X_small_test), (y_small_train, y_small_test) = partition(
    (X_small, y),
    0.7,
    rng=121,
    shuffle=true,
    multi=true
);
```

In [33]:
```
# bind the modal decision tree to the logiset;
# then train it and compute the accuracy

mach = machine(model, X_small_train, y_small_train)
@time fit!(mach);

y_small_predict_probabilities = MLJ.predict(mach, X_small_test)
y_small_predict = mode.(y_small_predict_probabilities)

MLJ.accuracy(y_small_predict, y_small_test)
```

```
[ Info: Precomputing logiset...
[ Info: Training machine(ModalDecisionTree(max_depth = nothing, …), …).
 34.692910 seconds (340.64 M allocations: 11.905 GiB, 7.38% gc time, 73.7
6% compilation time)
```

Out[33]:
```
0.8981481481481481
```

In [34]:
```
# show the restricted modal decision tree learned
printmodel(report(mach).rawmodel_full; hidemodality = true)
```

```
{1} RestrictedDecision(⟨G⟩min[V1] ≥ -0.036200142857142854)   2.0 : 46/252
(conf = 0.1825)
✔ {1} RestrictedDecision(⟨G⟩min[V13] < -1.3298954285714284)  6.0 : 44/129
(conf = 0.3411)
│✔ {1} RestrictedDecision(⟨B⟩max[V4] < 0.37886)              4.0 : 43/85
(conf = 0.5059)
││✔ 4.0 : 43/43 (conf = 1.0000)
││✘ 5.0 : 42/42 (conf = 1.0000)
│✘ 6.0 : 44/44 (conf = 1.0000)
✘ {1} RestrictedDecision(⟨G⟩min[V5] ≥ 0.9855877142857142)    2.0 : 46/123
(conf = 0.3740)
 ✔ {1} RestrictedDecision(⟨=⟩min[V4] < 1.412236375)          1.0 : 36/37
(conf = 0.9730)
 │✔ 1.0 : 33/33 (conf = 1.0000)
 │✘ 1.0 : 3/4 (conf = 0.7500)
 ✘ {1} RestrictedDecision(⟨G⟩min[V11] ≥ 0.188145)            2.0 : 46/86
(conf = 0.5349)
  ✔ {1} RestrictedDecision(⟨=⟩min[V5] ≥ 0.6772125714285714)  3.0 : 23/29
(conf = 0.7931)
  │✔ 2.0 : 6/6 (conf = 1.0000)
  │✘ 3.0 : 23/23 (conf = 1.0000)
  ✘ {1} RestrictedDecision(⟨G⟩min[V5] ≥ 0.383966)            2.0 : 40/57
(conf = 0.7018)
   ✔ {1} RestrictedDecision(⟨=⟩min[V5] < 0.6313968571428571) 2.0 : 20/22
(conf = 0.9091)
   │✔ 2.0 : 18/18 (conf = 1.0000)
   │✘ 2.0 : 2/4 (conf = 0.5000)
   ✘ {1} RestrictedDecision(⟨G⟩min[V11] ≥ 0.0013174285714285718) 2.0 : 2
0/35 (conf = 0.5714)
    ✔ {1} RestrictedDecision(⟨=⟩min[V5] < 0.232145375)       3.0 : 12/15
(conf = 0.8000)
    │✔ 3.0 : 11/11 (conf = 1.0000)
    │✘ 2.0 : 3/4 (conf = 0.7500)
    ✘ {1} RestrictedDecision(⟨G⟩min[V5] ≥ 0.059778)          2.0 : 17/20
(conf = 0.8500)
     ✔ 2.0 : 14/14 (conf = 1.0000)
     ✘ 2.0 : 3/6 (conf = 0.5000)
```

In [35]:

```
# show its *pure* version
printmodel(
    report(mach).solemodel_full;
    show_metrics = true,
    hidemodality = true
)
```

```
▣ (⟨G⟩(min[V1] ≥ -0.036200142857142854))
├✔ (⟨G⟩((min[V1] ≥ -0.036200142857142854) ∧ ⟨G⟩(min[V13] < -1.329895428571
4284)))
│ ├✔ (⟨G⟩((min[V1] ≥ -0.036200142857142854) ∧ ⟨G⟩((min[V13] < -1.329895428
5714284) ∧ ⟨B⟩(max[V4] < 0.37886))))
│ │ ├✔ 4.0 : (ninstances = 43, ncovered = 43, confidence = 1.0, lift =
1.0)
│ │ └✘ 5.0 : (ninstances = 42, ncovered = 42, confidence = 1.0, lift =
1.0)
│ └✘ 6.0 : (ninstances = 44, ncovered = 44, confidence = 1.0, lift = 1.0)
└✘ (⟨G⟩(min[V5] ≥ 0.9855877142857142))
  ├✔ (⟨G⟩((min[V5] ≥ 0.9855877142857142) ∧ (min[V4] < 1.412236375)))
  │ ├✔ 1.0 : (ninstances = 33, ncovered = 33, confidence = 1.0, lift =
1.0)
  │ └✘ 1.0 : (ninstances = 4, ncovered = 4, confidence = 0.75, lift = 1.0)
  └✘ (⟨G⟩(min[V11] ≥ 0.188145))
    ├✔ (⟨G⟩((min[V11] ≥ 0.188145) ∧ (min[V5] ≥ 0.6772125714285714)))
    │ ├✔ 2.0 : (ninstances = 6, ncovered = 6, confidence = 1.0, lift =
1.0)
    │ └✘ 3.0 : (ninstances = 23, ncovered = 23, confidence = 1.0, lift =
1.0)
    └✘ (⟨G⟩(min[V5] ≥ 0.383966))
      ├✔ (⟨G⟩((min[V5] ≥ 0.383966) ∧ (min[V5] < 0.6313968571428571)))
      │ ├✔ 2.0 : (ninstances = 18, ncovered = 18, confidence = 1.0, lift =
1.0)
      │ └✘ 2.0 : (ninstances = 4, ncovered = 4, confidence = 0.5, lift =
1.0)
      └✘ (⟨G⟩(min[V11] ≥ 0.0013174285714285718))
        ├✔ (⟨G⟩((min[V11] ≥ 0.0013174285714285718) ∧ (min[V5] < 0.23214537
5)))
        │ ├✔ 3.0 : (ninstances = 11, ncovered = 11, confidence = 1.0, lift
= 1.0)
        │ └✘ 2.0 : (ninstances = 4, ncovered = 4, confidence = 0.75, lift
= 1.0)
        └✘ (⟨G⟩(min[V5] ≥ 0.059778))
          ├✔ 2.0 : (ninstances = 14, ncovered = 14, confidence = 1.0, lift
= 1.0)
          └✘ 2.0 : (ninstances = 6, ncovered = 6, confidence = 0.5, lift =
1.0)
```

In [36]:

```
simplified_restricted_tree = ModalDecisionTrees.prune(
    report(mach).rawmodel_full;
    simplify = true
)

puretree = ModalDecisionTrees.translate(simplified_restricted_tree)
printmodel(
    puretree;
    threshold_digits = 2,
    use_feature_abbreviations = true,
    parenthesize_atoms = false,
    variable_names_map = [names(X)],
    hidemodality = true
)

println("# Leaves: ", SoleModels.nsubmodels(puretree))
println("# Classes: ", length(unique(y)))
```

```
▣ (⟨G⟩V1 ≥ -0.04)
├✔ (⟨G⟩(V1 ≥ -0.04 ∧ ⟨G⟩V13 ↓ -1.33))
│ ├✔ (⟨G⟩(V1 ≥ -0.04 ∧ ⟨G⟩(V13 ↓ -1.33 ∧ ⟨B⟩V4 < 0.38)))
│ │ ├✔ 4.0
│ │ └✘ 5.0
│ └✘ 6.0
└✘ (⟨G⟩V5 ≥ 0.99)
  ├✔ 1.0
  └✘ (⟨G⟩V11 ≥ 0.19)
    ├✔ (⟨G⟩(V11 ≥ 0.19 ∧ V5 ≥ 0.68))
    │ ├✔ 2.0
    │ └✘ 3.0
    └✘ (⟨G⟩V5 ≥ 0.38)
      ├✔ 2.0
      └✘ (⟨G⟩V11 ≥ 0.0)
        ├✔ (⟨G⟩(V11 ≥ 0.0 ∧ V5 ↓ 0.23))
        │ ├✔ 3.0
        │ └✘ 2.0
        └✘ 2.0
# Leaves: 18
# Classes: 6
```

In [37]:

```
# print the leaf rules and their training performances
ruleset = listrules(puretree)
printmodel.(
    ruleset;
    show_metrics = true,
    threshold_digits = 2,
    use_feature_abbreviations = true,
    parenthesize_atoms = false,
    hidemodality = true
);
```

```
▣ (⟨G⟩(V1 ≥ -0.04 ∧ ⟨G⟩(V13 ↓ -1.33 ∧ ⟨B⟩V4 < 0.38)))  ↣  4.0 : (ninstance
s = 252, ncovered = 43, coverage = 0.17, confidence = 1.0, lift = 5.86, na
toms = 3)
▣ ⟨G⟩(V1 ≥ -0.04 ∧ ⟨G⟩V13 ↓ -1.33) ∧ [G](V1 ≥ -0.04 → [G](V13 ↓ -1.33 →
[B]V4 ⤒ 0.38))  ↣  5.0 : (ninstances = 252, ncovered = 42, coverage = 0.1
7, confidence = 1.0, lift = 6.0, natoms = 5)
▣ ⟨G⟩V1 ≥ -0.04 ∧ [G](V1 ≥ -0.04 → [G]V13 ≥ -1.33)  ↣  6.0 : (ninstances =
252, ncovered = 44, coverage = 0.17, confidence = 1.0, lift = 5.73, natoms
= 3)
▣ ⟨G⟩V5 ≥ 0.99 ∧ [G]V1 ↓ -0.04  ↣  1.0 : (ninstances = 252, ncovered = 37,
coverage = 0.15, confidence = 0.97, lift = 6.45, natoms = 2)
▣ ⟨G⟩(V11 ≥ 0.19 ∧ V5 ≥ 0.68) ∧ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99  ↣  2.0 : (ni
nstances = 252, ncovered = 6, coverage = 0.02, confidence = 1.0, lift = 5.
48, natoms = 4)
▣ ⟨G⟩V11 ≥ 0.19 ∧ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99 ∧ [G](V11 ≥ 0.19 → V5 ↓ 0.6
8)  ↣  3.0 : (ninstances = 252, ncovered = 23, coverage = 0.09, confidence
= 1.0, lift = 6.46, natoms = 5)
▣ ⟨G⟩V5 ≥ 0.38 ∧ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99 ∧ [G]V11 ↓ 0.19  ↣  2.0 : (n
instances = 252, ncovered = 22, coverage = 0.09, confidence = 0.91, lift =
4.98, natoms = 4)
▣ ⟨G⟩(V11 ≥ 0.0 ∧ V5 ↓ 0.23) ∧ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99 ∧ [G]V11 ↓ 0.1
9 ∧ [G]V5 ↓ 0.38  ↣  3.0 : (ninstances = 252, ncovered = 11, coverage = 0.
04, confidence = 1.0, lift = 6.46, natoms = 6)
▣ ⟨G⟩V11 ≥ 0.0 ∧ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99 ∧ [G]V11 ↓ 0.19 ∧ [G]V5 ↓ 0.
38 ∧ [G](V11 ≥ 0.0 → V5 ≥ 0.23)  ↣  2.0 : (ninstances = 252, ncovered = 4,
coverage = 0.02, confidence = 0.75, lift = 4.11, natoms = 7)
▣ [G]V1 ↓ -0.04 ∧ [G]V5 ↓ 0.99 ∧ [G]V11 ↓ 0.19 ∧ [G]V5 ↓ 0.38 ∧ [G]V11 ↓ 0
.0  ↣  2.0 : (ninstances = 252, ncovered = 20, coverage = 0.08, confidence
= 0.85, lift = 4.66, natoms = 5)
```

In [38]:

```
println("IF\n\t",
    SoleLogics.experimentals.formula2natlang(
        antecedent(ruleset[4]);
        threshold_digits = 2,
        variable_names_map = [names(X)]
    )
)

println("THEN\n\t", consequent(ruleset[4]))
```

```
IF
	(∃ interval where (min[V5] ≥ 0.99)) and (∀ intervals (min[V1] <
-0.04))
THEN
	▣ 1.0
```