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

```
  Activating project at `~/logic-and-machine-learning`
```

In [2]:
```
using Random

Random.seed!(1235)
```

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

Many-Expert Decision Trees

`ManyExpertDecisionTrees.jl` is still in development and has not been released yet!

In [3]:
```
using ManyExpertDecisionTrees
```

"Many-Expert Decision Trees" sounds like a very general name...

Let's start with some motivation (and only one expert):

- we want to move from "hard" to "soft" decisions
- we want a better treatment of uncertainty

How will we achieve that?

- evaluating all the branches in our tree, and choosing the one(s) with higher values - i.e., we do not comply to one, strict, crisp decision at each step, but we take into consideration the contribution of each node
- for each node, we won't have that a feature is "true" or "false"; rather, we will assign a value between 0 and 1, and combine these values using the t-norm
- at the end, we do not constrain the model to always give a (single) class - it can also say "I do not know which class, but """surely""" (for the model) it is between those classes"

Let's load, once again, the "iris" dataset.

In [4]:
```
using RDatasets # used to load the iris dataset

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

In [5]:
```
using MLJ

y, X = unpack(data, ==(:Species));
```

And let's split out data into training and test.

In [6]:
```
(X_train, X_test), (y_train, y_test) = partition(
    (X, y),
    0.8,
```

```
    rng=13,
    shuffle=true,
    multi=true
);
```

Our approach works in the following way:

- we will further divide our training dataset into n+1 slices, were n is the number of experts (in this first example, just one - so we'll have 2 slices)
- then, we will learn a classical (crisp) decision tree, using the first slice of the training set
- finally, we will use each of the other `n` slices to train some parameters characterising a "soft" version of the learnt decision tree

In [7]:
```
(X_train_dt, X_train_exp), (y_train_dt, y_train_exp) = partition(
    (X_train, y_train),
    0.4,
    rng=42,
    shuffle=true,
    multi=true
);
```

Let's build a classical (crisp) decision tree on the first slice.

(Remember: we already shuffled out instances when splitting into train/test)

In [8]:
```
using DecisionTree

# Build a standard decision tree (explicitly)
dt = build_tree(y_train_dt, Matrix(X_train_dt))
```

Out[8]:
```
Decision Tree
Leaves: 5
Depth:  4
```

In [9]:
```
# Prune tree: merge leaves having >= 90% combined purity
dt = prune_tree(dt, 0.9)
```

Out[9]:
```
Decision Tree
Leaves: 5
Depth:  4
```

In [10]:
```
print_tree(dt)
```

```
Feature 4 < 0.75 ?
├─ setosa : 15/15
└─ Feature 3 < 4.9 ?
    ├─ Feature 4 < 1.7 ?
        ├─ versicolor : 14/14
        └─ Feature 2 < 3.1 ?
            ├─ virginica : 1/1
            └─ versicolor : 1/1
    └─ virginica : 17/17
```

In [11]:
```
y_pred = apply_tree(dt, Matrix(X_test));
```

In [12]:
```
cm = confusion_matrix(y_test, y_pred)
```

Out[12]:

	Ground Truth		
Predicted	setosa	versicol…	virginica
setosa	12	0	0
versicol…	0	6	3
virginica	0	0	9

In [13]:
```
accuracy(cm)
```

Out[13]:
```
0.9
```

Let's try to soften this decision tree!

First, we need to define a new structure (we need to add more information about each node of our decision tree); namely, a `ManyExpertDecisionTree` .

Not only that: we need to specify a `ManyExpertAlgebra` (more on that in a minute!) specifying a fuzzy logic to use for each expert.

In [14]:
```
using SoleLogics.ManyValuedLogics

mxa = ManyExpertAlgebra(ProductLogic)
```

Out[14]:
```
ManyExpertAlgebra with 1 expert:
  [1] FuzzyLogic(t-norm: Product (*))
```

The idea is to soften the original decision tree treating each node as a "membership" to a "fuzzy set" (note that if we use only `true` and `false` , we obtain the original split).

Hence, we will leverage membership functions, associating one for each node to each expert.

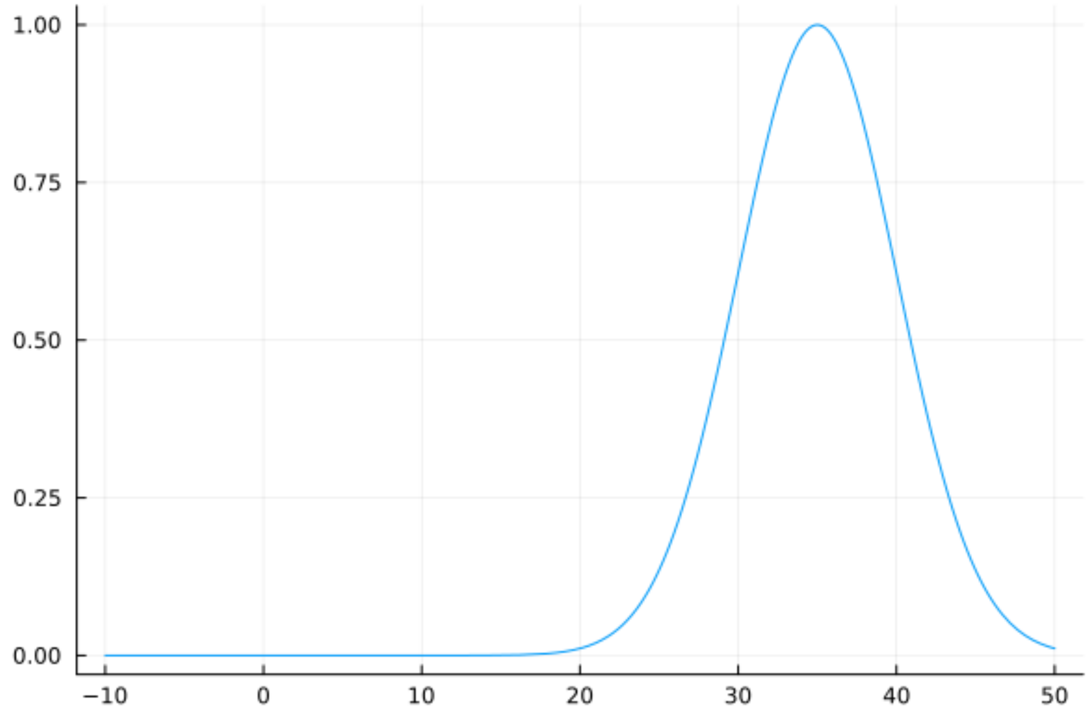
For membership functions, we will leverage the `FuzzyLogic.jl` package.

Watch out! Even if it is called `FuzzyLogic.jl` , this package offers classical tools (like membership functions) to work with fuzzy sets and system, and it is NOT a package to manipulate mathematical fuzzy logic.

Moreover, since we already have `FuzzyLogic` as a type in our naming space, we provide an alias to load the package, as follows.

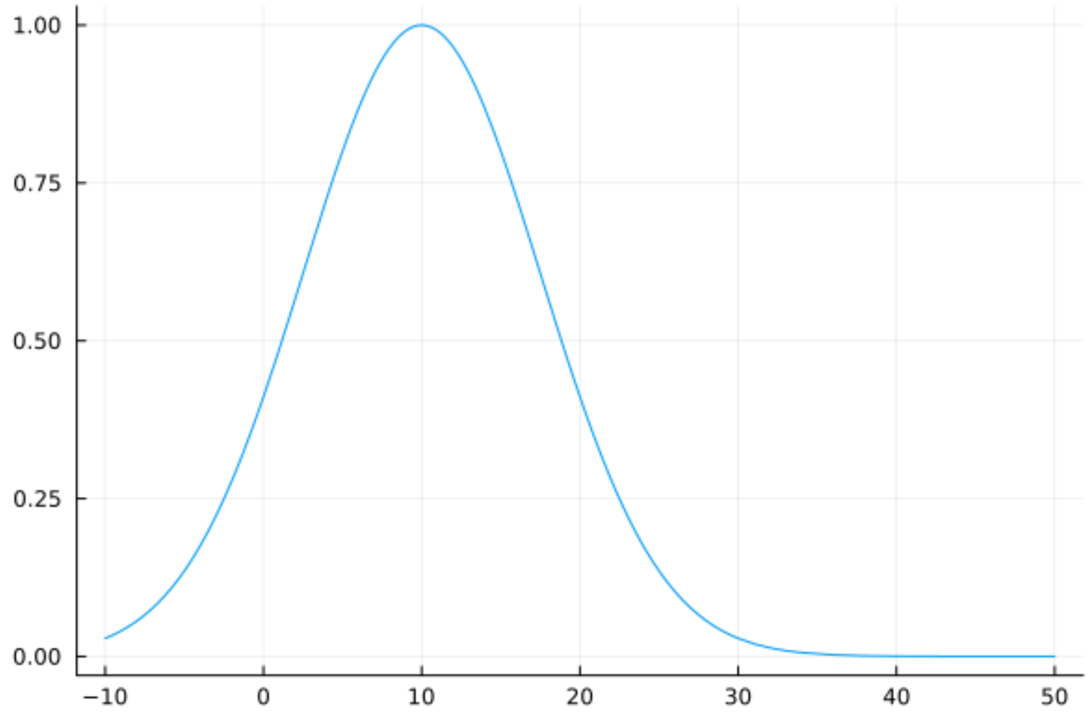
In [15]:
```
using ManyExpertDecisionTrees: FL    # This is an alias for `FuzzyLogic.jl
using Plots

hot = FL.GaussianMF(35.0, 5.0)  # temp>25
plot(hot, -10, 50)
```

Out[15]:



In [16]:
```
cold = FL.GaussianMF(10.0, 7.5) # temp≤25
plot(cold, -10, 50)
```

Out[16]:



In [17]:
```
hot(32)
```

Out[17]: 0.835270211411272

In [18]:
```
cold(32)
```

Out[18]: 0.013538440136652357

In [19]:
```
hot(12)
```

Out[19]: 2.5419346516199247e-5

In [20]:
```
cold(12)
```

Out[20]: 0.9650691177896804

In [21]:
```
hot(25)
```

Out[21]: 0.1353352832366127

In [22]:
```
cold(25)
```

Out[22]: 0.1353352832366127

To soften the decision tree, we use the manify function, specifying:

- the original decision tree
- the portion of the training set to use
- a tuple of kind of "membership functions" to use (one for each expert)

The parameters I'm learning are the parameters of the chosen function; in our case, we will only use Gaussian functions.

In [23]:
```
medt = manify(dt, X_train_exp, (FL.GaussianMF))
```

```
Many-Expert DecisionTree Root
Experts: DataType[FuzzyLogic.GaussianMF{Float64}]
```

Out[23]:

In [24]:
```
y_pred_mxa = map(eachrow(X_test)) do row
    result = ManyExpertDecisionTrees.apply(
        medt,
        mxa,
        Vector{Float64}(row)
    )
    return length(result) != 1 ? :vague : first(result)
end;
```

In [25]:
```
n_total = length(y_test)

n_correct = count(i -> y_pred_mxa[i] == y_test[i], 1:n_total)
(n_correct / n_total) * 100
```

Out[25]: 93.33333333333333

In [26]:
```
n_vague = count(==(:vague), y_pred_mxa)
(n_vague / n_total) * 100
```

Out[26]: 0.0

In [27]:
```
n_wrong = n_total - n_correct - n_vague
(n_wrong / n_total) * 100
```

Out[27]: 6.666666666666667

Wow, this was lucky! This improved performance!

Probably, the heuristic didn't choose the "best" attribute at each step: with softening, we can make up for it!

Now, let's try to use more than one expert!

In [28]:
```
# Since we'd need to further split our data between experts, let's give t
# a bigger slice!
(X_train_dt, X_train_exp), (y_train_dt, y_train_exp) = partition(
    (X_train, y_train),
    0.2,
    rng=42,
    shuffle=true,
    multi=true
);
```

In [29]:
```
# Build a standard decision tree (explicitly)
dt = build_tree(y_train_dt, Matrix(X_train_dt))
# Prune tree: merge leaves having >= 90% combined purity
dt = prune_tree(dt, 0.9)
print_tree(dt)
```

```
Feature 4 < 0.75 ?
├─ setosa : 7/7
└─ Feature 3 < 4.9 ?
    ├─ Feature 3 < 4.7 ?
        ├─ versicolor : 5/5
        └─ Feature 1 < 5.95 ?
            ├─ versicolor : 1/1
            └─ virginica : 1/1
    └─ virginica : 10/10
```

In [30]:
```
# Evaluate performance (crisp decision tree)
y_pred = apply_tree(dt, Matrix(X_test));
cm = confusion_matrix(y_test, y_pred);
accuracy(cm)
```

Out[30]:
```
0.8333333333333334
```

We will use 2 experts: one characterised by `ProductLogic`, the second by `GodelLogic`.

What is `ManyExpertAlgebra`? Now we can answer this question (but not mind too much about it - if it's confusing, skip this part): it is a product of fuzzy logics!

The result is a many-valued logic, characterised by partially ordered tuples.

In [31]:
```
# Algebra characterising each expert
mxa = ManyExpertAlgebra(ProductLogic, GodelLogic)
```

Out[31]:
```
ManyExpertAlgebra with 2 experts:
  [1] FuzzyLogic(t-norm: Product (*))
  [2] FuzzyLogic(t-norm: Gödel (min))
```

In [32]:
```
# Manify decision tree
medt = manify(dt, X_train_exp, FL.GaussianMF, FL.GaussianMF)
```

Out[32]:

```
Many-Expert DecisionTree Root
Experts: DataType[FuzzyLogic.GaussianMF{Float64}, FuzzyLogic.GaussianMF{Fl
oat64}]
```

In [33]:
```
# Evaluate performance (many-expert decision tree)
y_pred_mxa = map(eachrow(X_test)) do row
    result = ManyExpertDecisionTrees.apply(
        medt,
        mxa,
        Vector{Float64}(row)
    )
    return length(result) != 1 ? :vague : first(result)
end;
```

In [34]:
```
n_correct = count(i -> y_pred_mxa[i] == y_test[i], 1:n_total)
(n_correct / n_total) * 100
```

Out[34]:
```
86.66666666666667
```

In [35]:
```
n_vague = count(==(:vague), y_pred_mxa)
(n_vague / n_total) * 100
```

Out[35]:
```
3.3333333333333335
```

In [36]:
```
n_wrong = n_total - n_correct - n_vague
(n_wrong / n_total) * 100
```

Out[36]:
```
10.0
```

Not only we slightly improved our performance...

We also reduced wrong predictions, in favor of "uknowns": this is important, especially in critical scenarios!

For instance, suppose we are predicting if it's safe to travel by plane on a specific day, based on weather forecasting: with a crisp decision tree, I would have an error of 17%!

Instead, if I know it's unsure, if it's safe or not, I can default to "it is not", hence, my error is only 10%.

This is good, even at the risk of loosing some accuracy: better be safe than sorry!

The following cell represents a simple experiment, comparing various combinations of experts on the `iris` dataset.

In [37]:
```
using Combinatorics

allexperts = (GodelLogic, LukasiewiczLogic, ProductLogic);

# Compute all possible expert compbinations (with replacement)
expertcomb = begin
    c = Vector{Vector{FuzzyLogic}}()
    for i in 1:length(allexperts)
        append!(
            c,
```

```julia
            collect(Combinatorics.with_replacement_combinations(allexpert
        )
    end
    c
end;

# This is useful to read results later
expertcombreadable = map(expertcomb) do experts
    result = ""
    for expert in experts
        if (expert === GodelLogic)
            result *= "G"
        end
        if (expert === LukasiewiczLogic)
            result *= "L"
        end
        if (expert === ProductLogic)
            result *= "P"
        end
    end

    return result
end;

correct = [[0.0, 0.0] for _ in 1:length(expertcomb)];
wrong = [[0.0, 0.0] for _ in 1:length(expertcomb)];
vague = [[0.0, 0.0] for _ in 1:length(expertcomb)];

n_runs = 10

for i in 1:n_runs
    # Partition set into training and validation
    X_train, y_train, X_test, y_test = begin
        train, test = partition(eachindex(y), 0.8, shuffle=true, rng=i)
        X_train, y_train = X[train, :], y[train]
        X_test, y_test = X[test, :], y[test]
        X_train, y_train, X_test, y_test
    end

    # Build a standard decision tree
    dt = build_tree(y_train, Matrix(X_train))
    dt = prune_tree(dt, 0.9)

    # For each expert combination, build a ManyExpertDecisionTree
    Threads.@threads for k in eachindex(expertcomb)
        mf_experts = ntuple(_ -> FL.GaussianMF, length(expertcomb[k]))
        MXA = ManyExpertAlgebra(expertcomb[k]...)

        medt = manify(dt, X_train, mf_experts...)

        y_pred = map(eachrow(X_test)) do row
            result = ManyExpertDecisionTrees.apply(
                medt,
                MXA,
                Vector{Float64}(row)
            )
            return length(result) != 1 ? :vague : first(result)
        end

        # Extrapolating statistics
```

```julia
        n_total = length(y_test)

        n_vague = count(==(:vague), y_pred)
        pvague = (n_vague / n_total) * 100

        n_correct = count(i -> y_pred[i] == y_test[i], 1:n_total)
        pcorrect = (n_correct / n_total) * 100

        n_wrong = n_total - n_correct - n_vague
        pwrong = (n_wrong / n_total) * 100

        deltacorrect = (pcorrect - correct[k][1])
        correct[k][1] += deltacorrect / i
        correct[k][2] += deltacorrect * (pcorrect - correct[k][1])

        deltawrong = (pwrong - wrong[k][1])
        wrong[k][1] += deltawrong / i
        wrong[k][2] += deltawrong * (pwrong - wrong[k][1])

        deltavague = (pvague - vague[k][1])
        vague[k][1] += deltavague / i
        vague[k][2] += deltavague * (pvague - vague[k][1])

    end
end

# Process results: extract means and compute standard deviations (sample
correct_mean = [x[1] for x in correct]
correct_std = [sqrt(x[2] / (n_runs - 1)) for x in correct]

wrong_mean = [x[1] for x in wrong]
wrong_std = [sqrt(x[2] / (n_runs - 1)) for x in wrong]

vague_mean = [x[1] for x in vague]
vague_std = [sqrt(x[2] / (n_runs - 1)) for x in vague]

df = DataFrame(
    experts=expertcombreadable,
    correct_mean=correct_mean,
    correct_std=correct_std,
    wrong_mean=wrong_mean,
    wrong_std=wrong_std,
    vague_mean=vague_mean,
    vague_std=vague_std
)
```

Out[37]: 19×7 DataFrame

Row	experts	correct_mean	correct_std	wrong_mean	wrong_std	vague_mean
	String	Float64	Float64	Float64	Float64	Float64
1	G	95.3333	3.22031	3.66667	2.91865	1.0
2	L	71.0	18.3955	6.0	4.66137	23.0
3	P	94.3333	3.53117	5.66667	3.53117	0.0
4	GG	86.3333	5.97319	2.66667	3.44265	11.0
5	GL	65.6667	21.2016	1.66667	2.35702	32.6667
6	GP	88.0	5.01848	3.0	2.45955	9.0
7	LL	64.3333	18.9899	5.0	4.23099	30.6667
8	LP	70.0	17.3561	3.0	2.91865	27.0
9	PP	93.0	4.28895	5.33333	2.81091	1.66667
10	GGG	84.0	7.66586	1.33333	1.72133	14.6667
11	GGL	64.0	21.3032	1.0	1.61015	35.0
12	GGP	85.3333	6.32456	1.33333	1.72133	13.3333
13	GLL	59.3333	21.0701	1.66667	2.35702	39.0
14	GLP	64.6667	17.0837	1.66667	2.35702	33.6667
15	GPP	86.0	6.0451	2.33333	2.24983	11.6667
16	LLL	59.0	19.5031	3.66667	4.28895	37.3333
17	LLP	63.0	16.8105	3.66667	4.28895	33.3333
18	LPP	72.0	13.0715	3.33333	2.72166	24.6667
19	PPP	91.0	4.45831	4.66667	2.81091	4.33333

Exercise: play some more with the iris dataset, trying different combinations of experts. Which is the one that works better?

Exercise: put into practice what you learned using the following dataset!

In [38]:
```
using CSV
using DataFrames

PENGUINS_PATH = joinpath("..", "datasets", "penguins.csv")
data = DataFrame(CSV.File(PENGUINS_PATH))
```

Out[38]: 344×9 DataFrame *319 rows omitted*

Row	rownames	species	island	bill_length_mm	bill_depth_mm	flipper_le
	Int64	String15	String15	Float64?	Float64?	Int64?
1	1	Adelie	Torgersen	39.1	18.7	
2	2	Adelie	Torgersen	39.5	17.4	
3	3	Adelie	Torgersen	40.3	18.0	
4	4	Adelie	Torgersen	*missing*	*missing*	
5	5	Adelie	Torgersen	36.7	19.3	
6	6	Adelie	Torgersen	39.3	20.6	
7	7	Adelie	Torgersen	38.9	17.8	
8	8	Adelie	Torgersen	39.2	19.6	
9	9	Adelie	Torgersen	34.1	18.1	
10	10	Adelie	Torgersen	42.0	20.2	
11	11	Adelie	Torgersen	37.8	17.1	
12	12	Adelie	Torgersen	37.8	17.3	
13	13	Adelie	Torgersen	41.1	17.6	
⋮	⋮	⋮	⋮	⋮	⋮	
333	333	Chinstrap	Dream	45.2	16.6	
334	334	Chinstrap	Dream	49.3	19.9	
335	335	Chinstrap	Dream	50.2	18.8	
336	336	Chinstrap	Dream	45.6	19.4	
337	337	Chinstrap	Dream	51.9	19.5	
338	338	Chinstrap	Dream	46.8	16.5	
339	339	Chinstrap	Dream	45.7	17.0	
340	340	Chinstrap	Dream	55.8	19.8	
341	341	Chinstrap	Dream	43.5	18.1	
342	342	Chinstrap	Dream	49.6	18.2	
343	343	Chinstrap	Dream	50.8	19.0	
344	344	Chinstrap	Dream	50.2	18.7	

We need a bit of data preprocessing...

(We will see more about it tomorrow!)

In [39]:
```
using Impute
```

```
data_nomissing = Impute.filter(data; dims=:rows);
```

In [40]:
```
schema(data_nomissing)
```

Out[40]:

names	scitypes	types
rownames	Count	Int64
species	Textual	String15
island	Textual	String15
bill_length_mm	Continuous	Float64
bill_depth_mm	Continuous	Float64
flipper_length_mm	Count	Int64
body_mass_g	Count	Int64
sex	Textual	String7
year	Count	Int64

In [41]:
```
data_drop_cols = select!(data_nomissing, Not(:island, :sex))
```

Out[41]: 333×7 DataFrame *308 rows omitted*

Row	rownames	species	bill_length_mm	bill_depth_mm	flipper_length_mm	I
	Int64	String15	Float64	Float64	Int64	I
1	1	Adelie	39.1	18.7	181	
2	2	Adelie	39.5	17.4	186	
3	3	Adelie	40.3	18.0	195	
4	5	Adelie	36.7	19.3	193	
5	6	Adelie	39.3	20.6	190	
6	7	Adelie	38.9	17.8	181	
7	8	Adelie	39.2	19.6	195	
8	13	Adelie	41.1	17.6	182	
9	14	Adelie	38.6	21.2	191	
10	15	Adelie	34.6	21.1	198	
11	16	Adelie	36.6	17.8	185	
12	17	Adelie	38.7	19.0	195	
13	18	Adelie	42.5	20.7	197	
⋮	⋮	⋮	⋮	⋮	⋮	
322	333	Chinstrap	45.2	16.6	191	
323	334	Chinstrap	49.3	19.9	203	
324	335	Chinstrap	50.2	18.8	202	
325	336	Chinstrap	45.6	19.4	194	
326	337	Chinstrap	51.9	19.5	206	
327	338	Chinstrap	46.8	16.5	189	
328	339	Chinstrap	45.7	17.0	195	
329	340	Chinstrap	55.8	19.8	207	
330	341	Chinstrap	43.5	18.1	202	
331	342	Chinstrap	49.6	18.2	193	
332	343	Chinstrap	50.8	19.0	210	
333	344	Chinstrap	50.2	18.7	198	