Decision Trees (DT)

ref. from book **"Data Science from Scratch"**, Chap 17

```
begin
    using Test
    using Random
    using PlutoUI
```

```
AbstractArray{T1,1} where T1
```

```
begin
    const F = Float64
    const T = Any
    const VF = AbstractVector{F}
    const VT = AbstractVector{T1} where {T1 <: Any};
```

TOC

back to TOC

Entropy

In mathematical terms, if p_i is the proportion of data labeled as class c_i, then the entropy is defined as:

$$H(S) = \sum_i -p_i \times log_2(p_i)$$

With the standard convention: $0 \times log_2(0) = 0$

Each term is non-negative and is close to 0 when p_i is either close 0 or close to 1. This means the entropy will be small when every p_i is close to 0 or 1 (*i.e.* when most of the data is in 1 class) and it will be larger when many of the p_i's are close to 0 (*i.e.* when the data is spread across multiple classes).

```
entropy (generic function with 1 method)
```

```
function entropy(class_prob::VF)::F
    λ = p -> p > zero(F) ? -p * log(2, p) : zero(F)
    sum(λ.(class_prob))
```

```
Test Passed
```

```
begin
    @test entropy([1.]) ≈ 0.        # minimal entropy (max. certainty)
    @test entropy([.5, .5]) ≈ 1.    # maximal entropy for 2 classes
    @test 0.81 < entropy([.25, .75]) < 0.82
```

Our data will consist of pairs(input, label) for which we will need to compute the class probabilities.

```
data_entropy (generic function with 1 method)
```

```
begin
    function counter(labels::VT)::Dict{T, Integer}
        h = Dict{T, Integer}()
        for v ∈ labels
            h[v] = get(h, v, 0) + 1
        end
        h
    end

    function class_prob(labels::VT)::VF
        @assert length(labels) > 0
        tot_cnt = length(labels)
        [cnt / tot_cnt for cnt ∈ values(counter(labels))]
    end

    function data_entropy(labels::VT)::F
        class_prob(labels) |> entropy
    end
```

```
Test Passed
```

```
begin
    @test data_entropy(["a"]) ≈ 0. # == 0.
    @test data_entropy([:a]) ≈ 0.
    @test data_entropy([:b, :a, :b]) ≈ entropy([1. / 3., 2. / 3.])
    @test data_entropy([true, false]) == 1.
    @test data_entropy([2, 1, 2, 2]) == entropy([0.25, 0.75])
```

```
Float64[1.0]
```

```
(Float64[0.6, 0.4], Float64[0.4, 0.6])
```

```
(0.970951, 0.970951)
```

back to TOC

Entropy Partition

Mathematically, if we split our data S into partitions $S_1, \ldots S_m$ containing proportions $q_1, \ldots q_m$ of the data, then we compute the entropy of the partition as a weighted sum:

$$H = \sum_{i=1}^{m} q_i \times H(S_i)$$

partition_entropy (generic function with 1 method)

```
function partition_entropy(subsets::Vector{VT})::F
    """Given the partition into subsets, calc. its entropy"""
    # @show subsets
    tot_cnt = sum(length.(subsets))
    # @show tot_cnt
    λ = s -> data_entropy(s) * length(s) / tot_cnt
    sum(λ.(subsets))
```

Test Passed

```
begin
    a_ = BitArray{1}[[1, 1, 1, 1], [0, 0, 0, 1, 1], [1, 1, 0, 1, 0]]
    @test abs(partition_entropy(VT[a_...]) - 0.69353613) ≤ 1e-6
```

back to TOC

Creating our DT

```
begin
const NB = Union{Nothing, Bool}

struct Candidate
    level::Symbol
    lang::Symbol
    tweets::Bool
    phd::Bool
    did_well::NB
    function Candidate(level::Symbol, lang::Symbol, tweets::Bool, phd::Bool,
did_well::NB=nothing)
        new(level, lang, tweets, phd, did_well)
    end
end

```

```
inputs = [
    #         level    lang     tweets phd    did_well
    Candidate(:Senior, :Java,   false, false, false),
    Candidate(:Senior, :Java,   false, true,  false),
    Candidate(:Mid,    :Python, false, false, true),
    Candidate(:Junior, :Python, false, false, true),
    Candidate(:Junior, :R,      true,  false, true),
    Candidate(:Junior, :R,      true,  true,  false),
    Candidate(:Mid,    :R,      true,  true,  true),
    Candidate(:Senior, :Python, false, false, false),
    Candidate(:Senior, :R,      true,  false, true),
    Candidate(:Junior, :Python, true,  false, true),
    Candidate(:Senior, :Python, true,  true,  true),
    Candidate(:Mid,    :Python, false, true,  true),
    Candidate(:Mid,    :Java,   true,  false, true),
    Candidate(:Junior, :Python, false, true,  false)
```

We will build a decision tree (DT) following ID3 algorithm, which works as follows:

- if the data have all the same label, create a leaf node that predicts that label and stops.

- if the list of attributes is empty (*i.e* no more questions to split the data on), create a leaf that predicts the most common lable and stops
- otherwise try partitionning the data by each of the attributes
- choose the partition with the lowest entropy
- add a decision node based on the chosen attribute
- using the remaining attributes, recursively apply previous steps on each subset

First let's go manually through those steps using our toy dataset.

```
(:level, :lang, :tweets, :phd)
```

```
## Tuple
```

```
Symbol[:level, :lang, :tweets, :phd]
```

```
## Vector
```

```
partition_by (generic function with 1 method)
```

```
function partition_by(inputs::VT, attr::Symbol)::Dict{T, VT}
    part = Dict{T, VT}()
    for inp ∈ inputs
        key = getfield(inp, attr)
        part[key] = push!(get(part, key, []), inp)
    end
    part
```

```
partition_entropy_by (generic function with 1 method)
```

```
function partition_entropy_by(inputs::VT, attr::Symbol, label_attr::Symbol)::F
    """Given the partition, calc. its entropy"""
    parts = partition_by(inputs, attr)
    # @show(parts, attr)
    # println("----------------------")

    λ = inp -> getfield(inp, label_attr)
    labels = [λ.(p) for p ∈ values(parts)]
    # @show(labels)
    # println("----------------------")

    partition_entropy(VT[labels...])
```

```
level => 0.6935361388961919
lang => 0.8601317128547441
tweets => 0.7884504573082896
phd => 0.8921589282623617
```

```
with_terminal() do
    for key ∈ fieldnames(Candidate)[1:end-1]
        r = partition_entropy_by(inputs, key, :did_well)
        println("$(key) => $(r)")
    end
```

```
Test Passed
```

```
begin
    @test 0.69 ≤ partition_entropy_by(inputs, :level, :did_well) < 0.7
    @test 0.86 ≤ partition_entropy_by(inputs, :lang, :did_well) < 0.87
    @test 0.78 ≤ partition_entropy_by(inputs, :tweets, :did_well) < 0.79
```

```julia
    @test 0.89 ≤ partition_entropy_by(inputs, :phd, :did_well) < 0.90
```

Test Passed

```julia
begin
    senior_inputs = filter(c -> getfield(c, :level) == :Senior, inputs)

    @test partition_entropy_by(senior_inputs, :lang, :did_well) ≈ 0.4
    @test partition_entropy_by(senior_inputs, :tweets, :did_well) ≈ 0.0
    @test 0.95 ≤ partition_entropy_by(senior_inputs, :phd, :did_well) ≤ 0.96
```

back to TOC

Putting it all Together

We are going to define out tree as either:

- a :leaf that predicts a single value xor
- a :split containing an attribute to split on, subtrees for specific values of that attribute and possibly a default value (if we see an unknown value)

```julia
struct Leaf
    value::T
```

```julia
struct Split
    attr::Symbol
    subtrees::Dict
    defval::T
```

Union{Leaf, Split}

classify (generic function with 1 method)

```julia
function classify(dt::DT, input::T)::T
    """
    Classify given input using given decision tree (dt)
    """
    typeof(dt) == Leaf && (return dt.value)

    ## Otherwise this tree consists of an attr to split on and a
    ## dictionary whose keys are values of that attribute and whose
    ## values are subtrees to consider next
    sdt_key = getfield(input, dt.attr)

    if !haskey(dt.subtrees, sdt_key)
        return dt.defval       ## no subtree for key => default value
    end

    sdt = dt.subtrees[sdt_key] ## choose appropriate subtree and
    classify(sdt, input)       ## use it to classify the input
```

build_tree_id3 (generic function with 1 method)

```julia
function build_tree_id3(inputs::VT, split_attrs::Vector{Symbol},
        target::Symbol)::DT
    λ₁ = inp -> getfield(inp, target)
    label_cnt = λ₁.(inputs) |> counter
```

```julia

    most_common_label = reduce((m, x) -> m = x[2] > m[2] ? x : m, label_cnt;
        init=(nothing, -1))[1]
    # sort(collect(label_cnt), by=t -> t[2], rev=true)[1][1]

    ## If unique label, predict it
    length(label_cnt) == 1 && (return Leaf(most_common_label))

    ## no split attributes left => return the majority label
    length(split_attrs) == 0 && (return Leaf(most_common_label))

    ## otherwise split by best attribute
    best_attr = reduce(
        (t_attr, c_attr) -> (p = partition_entropy_by(inputs, c_attr, target);
        t_attr = p < t_attr[2] ? (c_attr, p) : t_attr), split_attrs;
        init=(nothing, Inf)
    )[1]

    parts = partition_by(inputs, best_attr)
    new_attrs = filter(a -> a ≠ best_attr, split_attrs)

    ## Recursively build the subtrees
    subtrees = Dict(attr_val => build_tree_id3(subset, new_attrs, target)
        for (attr_val, subset) ∈ parts)

    return Split(best_attr, subtrees, most_common_label)
```

```
dtree =   Split(
          attr = :level
          subtrees =   Dict(
                       :Mid ⇒   Leaf(
                                value = true
                              )
                       :Senior ⇒   Split(
                                   attr = :tweets
                                   subtrees =   Dict(
                                                false ⇒   Leaf(false)
                                                true ⇒   Leaf(true)
                                              )
                                   defval = false
                                 )
                       :Junior ⇒   Split(
                                   attr = :phd
                                   subtrees =   Dict(
                                                false ⇒   Leaf(true)
                                                true ⇒   Leaf(false)
                                              )
                                   defval = true
                                 )
                     )
          defval = true
        )
```

```
Test Passed
```

```julia
@test classify(dtree, Candidate(:Junior, :Java, true, false))
```

```
Test Passed
```

```julia
@test !classify(dtree, Candidate(:Junior, :Java, true, true))
```

```
Test Passed
```

```julia
@test classify(dtree, Candidate(:Intern, :Java, true, true))
```