```julia
using Statistics
```

```julia
using LinearAlgebra
```

```julia
using Plots
```

```julia
using PlutoUI
```

```
PlotlyBackend()
```

```julia
plotly()
```

```
For saving to png with the Plotly backend PlotlyBase has to be installed.
```

# *Decision Trees!*

Decision trees in Julia.

## Tree Structure

Defining the base tree structure for the decision tree.

```julia
mutable struct DecisionNode
    threshold::Union{Number, Missing}
    attribute::Union{Integer, Missing}
    parent::Union{DecisionNode, Missing}
    leftChild::Union{DecisionNode, LeafNode, Missing}
    rightChild::Union{DecisionNode, LeafNode, Missing}

    DecisionNode() = new(missing, missing, missing, missing, missing)
    DecisionNode(threshold::Number, attribute::Integer) = new(threshold, attribute, missing, missing, missing)
    DecisionNode(threshold::Number, attribute::Integer, parent::DecisionNode) = new(threshold, attribute, parent, missing, missing)
end
```

```julia
mutable struct LeafNode
    class::Bool

    LeafNode(class::Bool) = new(class)
end
```

### `setleftchild!`

setleftchild!(node::DecisionNode, child::DecisionNode)

Sets the `leftChild` of the `node` to `child`, if it has not been already set.

```
"""setleftchild!(node::DecisionNode, child::DecisionNode)

Sets the `leftChild` of the `node` to `child`, if it has not been already set.
"""
function setleftchild!(node::DecisionNode, child::DecisionNode)
    if !isequal(node.leftChild, missing)
        error("Left child has already been set!")
    else
        node.leftChild = child
    end
end
```

### `setrightchild!`

setrightchild!(node::DecisionNode, child::DecisionNode)

Sets the `rightChild` of the `node` to `child`, if it has not been already set.

```
"""setrightchild!(node::DecisionNode, child::DecisionNode)

Sets the `rightChild` of the `node` to `child`, if it has not been already set.
"""
function setrightchild!(node::DecisionNode, child::DecisionNode)
    if !isequal(node.rightChild, missing)
        error("Right child has already been set!")
    else
        node.rightChild = child
    end
end
```

### hasleftchild

hasleftchild(node::DecisionNode)

Checks whether the node has a left child.

```
"""hasleftchild(node::DecisionNode)

Checks whether the node has a left child.
"""
function hasleftchild(node::DecisionNode)
    return !isequal(node.leftChild, missing)
end
```

### hasrightchild

hasrightchild(node::DecisionNode)

Checks whether the node has a left child.

```
"""hasrightchild(node::DecisionNode)

Checks whether the node has a left child.
"""
function hasrightchild(node::DecisionNode)
    return !isequal(node.rightChild, missing)
end
```

### inorder_rec

inorder_rec(node::DecisionNode)

Recursive function printing the threshold values within the tree while traversing it in order.

```
"""inorder_rec(node::DecisionNode)

Recursive function printing the threshold values within the tree while
traversing it in order.
"""
function inorder_rec(node::DecisionNode)
    if hasleftchild(node)
        inorder_rec(node.leftChild)
    end
    print(node.threshold, " ")
    if hasrightchild(node)
        inorder_rec(node.rightChild)
    end
end
```

### inorder

inorder(node::DecisionNode)

Function printing the threshold values within the tree while traversing it in order.

```
"""inorder(node::DecisionNode)

Function printing the threshold values within the tree while traversing it in
order.
"""
function inorder(node::DecisionNode)
    with_terminal() do
        inorder_rec(node)
    end
end
```

# Entropy and Information Gain

### entropy

entropy(Y::AbstractArray{Bool})

Computes the Shannon's entropy of a given boolean array.

```julia
"""entropy(Y::AbstractArray{Bool})

Computes the Shannon's entropy of a given boolean array.
"""
function entropy(Y::AbstractArray{Bool})
    positiveProb = sum(Y) / length(Y)
    negativeProb = 1 - positiveProb

    if length(Y) == 0
        return 0.0
    elseif positiveProb == 1 || negativeProb == 1
        return 0.0
    end

    return -(positiveProb * log(positiveProb) + negativeProb * log(negativeProb))
end
```

### informationgain

informationgain(Y::AbstractArray{Bool}, Y1::AbstractArray{Bool}, Y2::AbstractArray{Bool})

Computes the information gain obtained by dividing the boolean array `Y` into the two boolean arrays `Y1` and `Y2`.

```julia
"""informationgain(Y::AbstractArray{Bool}, Y1::AbstractArray{Bool}, Y2::AbstractArray{Bool})

Computes the information gain obtained by dividing the boolean array `Y` into the two boolean arrays `Y1` and `Y2`.
"""
function informationgain(Y::AbstractArray{Bool}, Y1::AbstractArray{Bool}, Y2::AbstractArray{Bool})
    return entropy(Y) - entropy(Y2) - entropy(Y1)
end
```

### Dataset generation

Generating a simple random dataset.

```
md"### Dataset generation
Generating a simple random dataset.
"
```

**rowsnorm**

rowsnorm(X::AbstractMatrix)

Computes the norms of vectors forming the rows of the matrix `X`.

```
"""rowsnorm(X::AbstractMatrix)

Computes the norms of vectors forming the rows of the matrix `X`.
"""
function rowsnorm(X::AbstractMatrix)
    rowsCount = size(X, 1)
    return [ norm(X[i, :]) for i in 1:rowsCount ]
end
```

### generatedataset

generatedataset(examplesCount::Integer)

Generates a simple dataset.

```julia
"""generatedataset(examplesCount::Integer)

Generates a simple dataset.
"""
function generatedataset(examplesCount::Integer)
    positiveCount = div(examplesCount, 2)
    negativeCount = examplesCount - positiveCount

    positiveExamples = randn(positiveCount, 2)
    negativeExamples = randn(negativeCount, 2) .* 4

    minNegativeNorm = maximum(rowsnorm(positiveExamples)) * 1.2

    negativeExamples = negativeExamples[filter(i -> norm(negativeExamples[i,
:]) > minNegativeNorm, 1:negativeCount), :]

    while size(negativeExamples, 1) < negativeCount

        newNegativeExamples = randn(negativeCount, 2) .* 4

        newNegativeExamples = newNegativeExamples[filter(i ->
norm(newNegativeExamples[i, :]) > minNegativeNorm, 1:negativeCount), :]

        negativeExamples = vcat(negativeExamples, newNegativeExamples)
    end

    negativeExamples = negativeExamples[1:negativeCount, :]

    X = vcat(positiveExamples, negativeExamples)
    Y = [ trues(positiveCount); falses(negativeCount) ]

    return X, Y
end
```

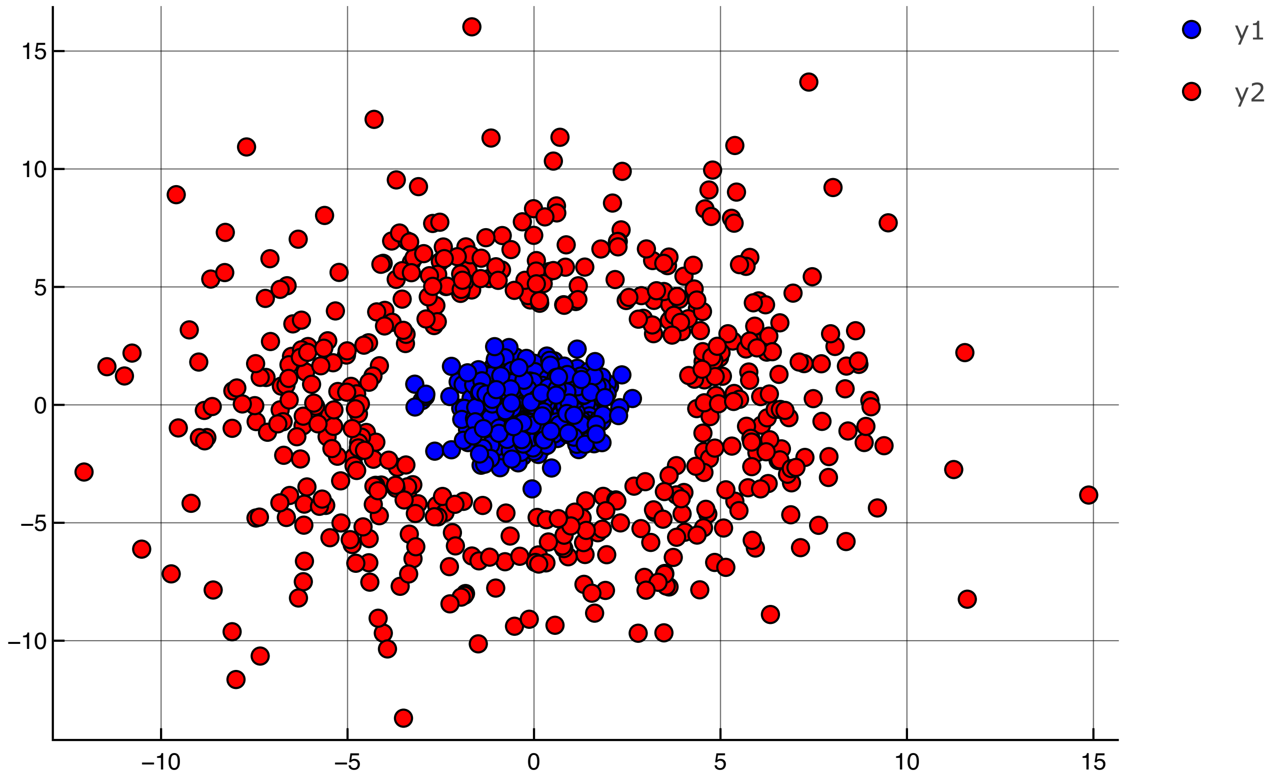
```
(1000×2 Matrix{Float64}:, BitVector: [true, true, true, true, true, true, true, true
 -1.06379    -1.04833
 -1.13407    -0.604511
  0.375964   -1.07772
  1.05079     0.387137
  0.222797   -1.82657
  0.22561     0.128053
  0.345356    2.06891
  ⋮
 -6.3595     -1.35214
 -3.16996    -6.0156
  3.47938     5.99875
 -3.92912   -10.3486
 -4.77815    -6.71559
  7.10987     1.81458
```

```julia
X, Y = generatedataset(1000)
```

### plot_examples

plot_examples(X::AbstractMatrix, Y::BitVector)

Plots the dataset `X`, `Y`. Meant to be used with 2-dimensional datasets.




```julia
plot_examples(X, Y)
```

# Building the Trees

```
mutable struct DecisionTree
    root::DecisionNode

    function DecisionTree(root::DecisionNode)
        new(root)
    end
end
```

### `splitonvalue`

splitonvalue(X::AbstractVector, Y::BitVector, value::Number)

Splits the vector `Y` into two parts, `first` and `second`, based on the provided `value` and the vector `X`.

```
"""splitonvalue(X::AbstractVector, Y::BitVector, value::Number)

Splits the vector `Y` into two parts, `first` and `second`, based on the
provided `value` and the vector `X`.
"""
function splitonvalue(X::AbstractVector, Y::BitVector, value::Number)
    first = [ Y[i] for i in eachindex(X) if X[i] <= value ]
    second = [ Y[i] for i in eachindex(X) if X[i] > value ]
    return first, second
end
```

### splitonvalue

splitonvalue(X::AbstractVector, Y::BitVector, value::Number)

Splits the vector `Y` into two parts, `first` and `second`, based on the provided `value` and the vector `X`.

splitonvalue(X::AbstractMatrix, Y::BitVector, attrib::Integer, value::Number)

Splits the dataset `(X, Y)` into two parts, `first` and `second`, based on the provided `value` and `attrib` indicating the column of the matrix `X`.

```
"""splitonvalue(X::AbstractMatrix, Y::BitVector, attrib::Integer, value::Number)

Splits the dataset `(X, Y)` into two parts, `first` and `second`, based on the
provided `value` and `attrib` indicating the column of the matrix `X`.
"""
function splitonvalue(X::AbstractMatrix, Y::BitVector, attrib::Integer,
value::Number)
    firstX = X[filter(i -> X[i, attrib] <= value, 1:size(X, 1)), :]
    secondX = X[filter(i -> X[i, attrib] > value, 1:size(X, 1)), :]
    firstY = Y[filter(i -> X[i, attrib] <= value, 1:size(X, 1))]
    secondY = Y[filter(i -> X[i, attrib] > value, 1:size(X, 1))]
    return firstX, secondX, firstY, secondY
end
```

### findbestsplitvalue

findbestsplitvalue(X::AbstractVector, Y::BitVector)

Finds the split on value that yields the largest information gain.

```julia
"""findbestsplitvalue(X::AbstractVector, Y::BitVector)

Finds the split on value that yields the largest information gain.
"""
function findbestsplitvalue(X::AbstractVector, Y::BitVector)
    bestSplitValue = X[1]
    bestInformationGain = -Inf
    for value in X
        first, second = splitonvalue(X, Y, value)
        informationGain = informationgain(Y, first, second)
        if informationGain > bestInformationGain
            bestSplitValue = value
            bestInformationGain = informationGain
        end
    end
    return bestSplitValue, bestInformationGain
end
```

### findbestsplit

findbestsplit(X::AbstractMatrix, Y::BitVector)

Finds the best attribute and its value on which to split the dataset `(X, Y)` in order to obtain to largest information gain.

```julia
"""findbestsplit(X::AbstractMatrix, Y::BitVector)

Finds the best attribute and its value on which to split the dataset `(X, Y)`
in order to obtain to largest information gain.
"""
function findbestsplit(X::AbstractMatrix, Y::BitVector)
    bestSplitAttrib = 0
    bestSplitValue::Number = 0.0
    bestInformationGain = -Inf
    for i in 1:size(X, 2)
        splitValue, informationGain = findbestsplitvalue(X[:, i], Y)
        if informationGain > bestInformationGain
            bestSplitAttrib = i
            bestSplitValue = splitValue
            bestInformationGain = informationGain
        end
    end

    return bestSplitAttrib, bestSplitValue
end
```

### split

```
split(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
split(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` into an array of substrings on occurrences of the delimiter(s) `dlm`. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false`

without a `dlm` argument, `true` with a `dlm` argument.

See also `rsplit`.

# Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
2-element Vector{SubString{String}}:
 "Ma"
 "rch"
```

split(node::DecisionNode, X::AbstractMatrix, Y::BitVector, depth::Integer, maxDepth::Number)

Constructs the decision node `node`, so that it makes such a decision that yields the largest information gain. Recursively constructs its children until no decisions can be made or until the maximal depth of the tree has been reached.

```julia
"""split(node::DecisionNode, X::AbstractMatrix, Y::BitVector, depth::Integer,
maxDepth::Number)

Constructs the decision node `node`, so that it makes such a decision that
yields the largest information gain. Recursively constructs its children until
no decisions can be made or until the maximal depth of the tree has been
reached.
"""
function split(node::DecisionNode, X::AbstractMatrix, Y::BitVector,
depth::Integer, maxDepth::Number)
    class = mean(Y) > 0.5
    if depth >= maxDepth
        return LeafNode(class)
    elseif entropy(Y) == 0
        return LeafNode(class)
    end

    splitAttrib, splitValue = findbestsplit(X, Y)
    firstX, secondX, firstY, secondY = splitonvalue(X, Y, splitAttrib,
splitValue)

    node.attribute = splitAttrib
    node.threshold = splitValue

    leftChild = DecisionNode()
    leftChild.parent = node
    leftChild = split(leftChild, firstX, firstY, depth + 1, maxDepth)

    rightChild = DecisionNode()
    rightChild.parent = node
    rightChild = split(rightChild, secondX, secondY, depth + 1, maxDepth)

    node.leftChild = leftChild
    node.rightChild = rightChild

    return node

end
```

### decisiontree

decisiontree(X::AbstractMatrix, Y::BitVector, maxDepth::Number = Inf)

Constructs a decision tree based on the given dataset `(X, Y)`.

```julia
"""decisiontree(X::AbstractMatrix, Y::BitVector, maxDepth::Number = Inf)

Constructs a decision tree based on the given dataset `(X, Y)`.
"""
function decisiontree(X::AbstractMatrix, Y::BitVector, maxDepth::Number = Inf)
    root = DecisionNode()
    split(root, X, Y, 0, maxDepth)
    return DecisionTree(root)
end
```

# Classification using the Decision Tree

### decisiontree_classify

decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)

Classifies the observation `X` using the decision tree `decisionTree`.

```julia
"""decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)

Classifies the observation `X` using the decision tree `decisionTree`.
"""
function decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)
    currentNode::Union{DecisionNode, LeafNode} = decisionTree.root
    while isequal(typeof(currentNode), DecisionNode)
        currentNode = X[currentNode.attribute] <= currentNode.threshold ?
currentNode.leftChild : currentNode.rightChild
    end

    return currentNode.class
end
```

### decisiontree_classify

decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)

Classifies the observation `X` using the decision tree `decisionTree`.

decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)

Classifies all of the observations contained in the matrix `X` using the decision tree `decisionTree`.

```
"""decisiontree_classify(decisionTree::DecisionTree, X::AbstractVector)

Classifies all of the observations contained in the matrix `X` using the
decision tree `decisionTree`.
"""
function decisiontree_classify(decisionTree::DecisionTree, X::AbstractMatrix)
    return [ decisiontree_classify(decisionTree, X[i, :]) for i in 1:size(X, 1)
]
end
```

### accuracy

accuracy(expected::AbstractVector{Bool}, predicted::AbstractVector{Bool})

Computes the classification accuracy given the `expected` and `predicted` vectors.

```
"""accuracy(expected::AbstractVector{Bool}, predicted::AbstractVector{Bool})

Computes the classification accuracy given the `expected` and `predicted`
vectors.
"""
function accuracy(expected::AbstractVector{Bool},
predicted::AbstractVector{Bool})
    return mean(expected .== predicted)
end
```

**datasetaccuracy**

datasetaccuracy(decisionTree::DecisionTree, X::AbstractMatrix, Y::AbstractVector{Bool})

Computes the accuracy of the decision tree `decisionTree` on the dataset `(X, Y)`.

```
"""datasetaccuracy(decisionTree::DecisionTree, X::AbstractMatrix,
Y::AbstractVector{Bool})

Computes the accuracy of the decision tree `decisionTree` on the dataset `(X,
Y)`.
"""
function datasetaccuracy(decisionTree::DecisionTree, X::AbstractMatrix,
Y::AbstractVector{Bool})
    return mean(decisiontree_classify(decisionTree, X) .== Y)
end
```

# Check the Accuracy on the Training Dataset

```
basic_tree =
  DecisionTree(DecisionNode(2.638953901238894, 1, missing, DecisionNode(-3.2179346619
```

```
basic_tree = decisiontree(X, Y, Inf)
```

```
1.0
```

```
datasetaccuracy(basic_tree, X, Y)
```

# Training and Testing datasets

```
md"### Training and Testing datasets"
```

```
datasetSize = 1000
```

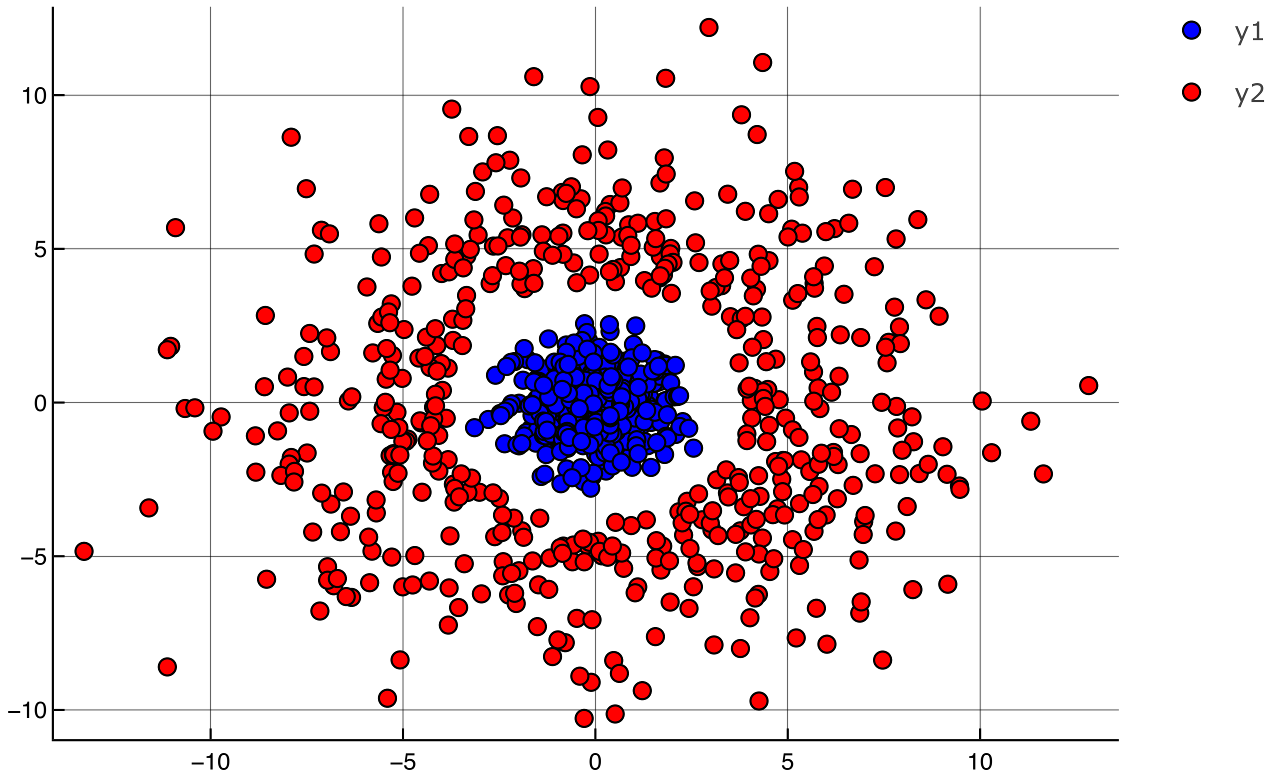
```
datasetSize = 1000
```

```
trainDatasetFraction = 0.7
```

```
trainDatasetFraction = 0.7
```

```
(1000×2 Matrix{Float64}:, BitVector: [true, true, true, true, true, true, true, true
  1.51648     0.887544
  0.0362734  -0.54108
 -0.844505   -1.40261
  0.13087     1.08185
 -1.68615    -0.86412
 -0.157102   -0.544819
 -0.0827385  -1.11503
  ⋮
  4.41459    -0.12504
  6.68291     6.9396
  8.38081     5.95054
 -1.11468     4.7919
 -5.42384     1.7565
  7.80637    -4.18129
```

```
datasetX, datasetY = generatedataset(datasetSize)
```




```
plot_examples(datasetX, datasetY)
```

```
BitVector: [true, true, true, true, true, true, true, true, true, true, true, true, t
```

```
begin
    trainMask = rand(datasetSize) .< trainDatasetFraction
    testMask = map(x -> !x, trainMask)
    trainX = datasetX[trainMask, :]
    trainY = datasetY[trainMask]
    testX = datasetX[testMask, :]
    testY = datasetY[testMask]
end
```

# Testing the Tree

```
tree =
  DecisionTree(DecisionNode(2.55241689484124, 1, missing, DecisionNode(-3.15711262695
```

```
tree = decisiontree(trainX, trainY, 4)
```

```
1.0
```

```
datasetaccuracy(tree, trainX, trainY)
```

```
0.9932885906040269
```

```
datasetaccuracy(tree, testX, testY)
```