Week 7:

K-Fold Cross Validation and K- Nearest Neighbour Classification

Preparing the data and initial analysis:

julia> using DataFrames

julia> using GLM

julia> using Random

julia> using Statistics

julia> Random.seed!(1234);

julia> df = DataFrame(randn(50, 9), :auto);

julia> df.y = sum(eachcol(df)) .+ 1.0 .+ randn(50);

julia> df

50×10 DataFrame

Row	x1	x2	x3	x4	x5	x6	x7	x8	x9	y
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	0.867347	-1.22672	0.183976	-1.87215	-0.205782	0.295222	0.183203	0.358659	-0.833507	-2.20367
2	-0.901744	-0.541716	-1.27635	-0.668331	-1.22338	-0.338215	0.975083	0.488578	0.0827196	-3.31337
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

49 | -1.00978 -1.66323 0.797165 -0.644069 0.127747 0.742054 0.196089 -1.05575 0.771387 -2.364
50 | -0.543805 -0.521229 0.103145 -1.37931 0.143105 -0.0665944 -2.34887 -1.37548 0.590872 -4.76852

46 rows omitted

julia> model = lm(formula, df)

StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64, Matrix{Float64}}}}, Matrix{Float64}}

y ~ 1 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9

Coefficients:

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	1.01641	0.140685	7.22	<1e-08	0.732078	1.30075
x1	0.879802	0.141336	6.22	<1e-06	0.594151	1.16545
x2	1.14157	0.160897	7.10	<1e-07	0.81639	1.46676

Calculate Mean Square Error of the model:

julia> deviance(model) / nrow(df)

0.6689313350544223

julia> mse(model, df) = sum(x->x^2, predict(model, df) - df.y) / nrow(df);

julia> mse(model, df)

0.6689313350544223

Doing cross validation:

julia> df.fold = shuffle!((1:nrow(df)) .% 10)

50-element Vector{Int64}:

```
 4
 6
 5
 7
 ⋮
 8
 6
 7

get_fold_data(df, fold) =
    (train=view(df, df.fold .!= fold, :),
     test=view(df, df.fold .== fold, :))

julia> mean(0:9) do fold
           train, test = get_fold_data(df, fold)
           model_cv = lm(formula, train)
           return mse(model_cv, test)
       end
0.9500293257502437
```

Test the Procedure using Simulation:

mse_whole: calculated on training data set;
mse_cv: calculated using cross validation;
mse_t: expected prediction squared error.

```
using DataFrames
using GLM
using Random
mse(model, df) = sum(x->x^2, predict(model, df) - df.y) / nrow(df);
mset(model) = 1 + sum(x -> (1 - x) ^ 2, coef(model));
get_fold_data(df, fold) =
    (train=view(df, df.fold .!= fold, :),
     test=view(df, df.fold .== fold, :))
function runtest(id)
```

```
    df = DataFrame(randn(50, 9), :auto)
    df.y = sum(eachcol(df)[1:5]) .+ 1.0 .+ randn(50)
    formulas = [Term(:y) ~ sum([Term(Symbol(:x, i)) for i in 1:n]) for n in 1:9]
    models = [lm(f, df) for f in formulas]
    mse_wholes = [mse(m, df) for m in models]

    mse_ts = [mset(m) for m in models]

    df.fold = shuffle!((1:nrow(df)) .% 10)

    mse_cvs = map(formulas) do f
        return mean(0:9) do fold
            train, test = get_fold_data(df, fold)
            model_cv = lm(f, train)
            return mse(model_cv, test)
        end
    end

    return DataFrame(id=id, vars=1:9, mse_whole=mse_wholes, mse_cv=mse_cvs,
mse_t=mse_ts)
end
```

Output:

```
julia> Random.seed!(12);

julia> res = DataFrame([runtest() for _ in 1:10_000])
10000×3 DataFrame
  Row │ mse_whole  mse_cv    mse_t
      │ Float64    Float64   Float64
──────┼──────────────────────────────
    1 │  0.767821  1.20796   1.43191
    2 │  0.595431  1.11465   1.2233
    3 │  0.801025  1.41942   1.25682
  ⋮   │    ⋮         ⋮         ⋮
 9998 │  0.589998  0.930069  1.24527
 9999 │  0.444815  0.685732  1.36184
```

```
 10000 │  1.05747   1.68496   1.23118
                9994 rows omitted

julia> describe(res, :all)
3×13 DataFrame
 Row │ variable   mean     std      min      q25      median   q75      max      nunique  nmissing  first    last     eltype
     │ Symbol     Float64  Float64  Float64  Float64  Float64  Float64  Float64  Nothing  Int64     Float64  Float64  DataType
─────┼──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
   1 │ mse_whole  0.799776  0.177712  0.286417  0.672772  0.785994  0.912926  1.71692  0         0.767821  1.05747  Float64
   2 │ mse_cv     1.29295   0.305371  0.419517  1.07337   1.26767   1.48104   3.0517   0         1.20796   1.68496  Float64
   3 │ mse_t      1.25542   0.132023  1.01737   1.16157   1.22982   1.32126   2.07322  0         1.43191   1.23118  Float64

julia> cor(Matrix(res))
3×3 Matrix{Float64}:
  1.0         0.947       -0.00555373
  0.947       1.0         -0.00844714
 -0.00555373 -0.00844714   1.0
```

K Nearest Neighbors:

Scenario

Let's assume that we wanted to build a system where given size of a orange the system should determine if the orange should be sent to luxury hotels, or if it needs to be sent to retail as edible fruit or if it needs to sent for juice factories.

Let's assume you are in the orange sorting facility and are measuring fruit sizes in bins that are labeled one of these. Since this is not real now, we have written a function to simulate it, where given orange size it will label it.

```
function category(orange_size)
    if orange_size > 6
        return "Luxury Hotels"
    end

    if orange_size > 4
        return "Edible Fruit"
    end

    "Juice"
end
```

Output:

```
category (generic function with 1 method)
category(1.5)
"Juice"
category(4.5)
"Edible Fruit"
category(6.2)
"Luxury Hotels"
```

Basic Functions:

let's write a function to count number occourances of things in an array.

```
function counter(array_of_elements)

    counts = Dict()



    for element in array_of_elements

        counts[element] = get(counts, element, 0) + 1

    end



    counts

end
```

Output:

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

In the above function we pass an array `array_of_elements` to it. We have a dictionary called counts:

```
function counter(array_of_elements)

    counts = Dict()

end
```

Next, for every `element` in `array_of_elements`:

```
function counter(array_of_elements)

    counts = Dict()
```

```
    for element in array_of_elements

    end

end

function counter(array_of_elements)

    counts = Dict()



    for element in array_of_elements

        counts[element] = get(counts, element, 0) + 1

    end

end
```

Finally we return the `counts` dictionary:

```
function counter(array_of_elements)

    counts = Dict()



    for element in array_of_elements

        counts[element] = get(counts, element, 0) + 1

    end



    counts

end
```

Let's test our function:

```
counts = counter(["Juice", "Edible Fruit", "Juice", "Luxury Hotels"])
```

Output:

```
Dict{Any, Any} with 3 entries:
  "Edible Fruit"  => 1
  "Juice"         => 2
  "Luxury Hotels" => 1
```

Next we should see which count is the highest. For that we write a function `highest_vote` as shown below:

```
function highest_vote(counts)
    max_val = 0
    max_vote = ""

    for (key, value) in counts
        if value > max_val
            max_val = value
            max_vote = key
        end
    end

    max_vote
end
```

Output:

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

```
function highest_vote(counts)
    max_val = 0
    max_vote = ""
end
```

Next for each and every key value pair in `counts`, which should be a `Dict`:

```
function highest_vote(counts)
    max_val = 0
    max_vote = ""

    for (key, value) in counts
    end
end
function highest_vote(counts)
    max_val = 0
    max_vote = ""

    for (key, value) in counts
        if value > max_val
            max_val = value
            max_vote = key
        end
```

```
    end


    max_vote
end
```

Now let's test this function with `counts` which contains the following dictionary:

```
{
  "Edible Fruit"  => 1
  "Juice"         => 2
  "Luxury Hotels" => 1
}
highest_vote(counts)
```

Output:

```
"Juice"
```

Plotting our Sample Data:

```
key_values = Dict(
    "Luxury Hotels" => [],
    "Juice" => [],
    "Edible Fruit" => []
)
```

```
for (size, sell) in orange_sizes
    push!(key_values[sell], size)
end


key_values
```

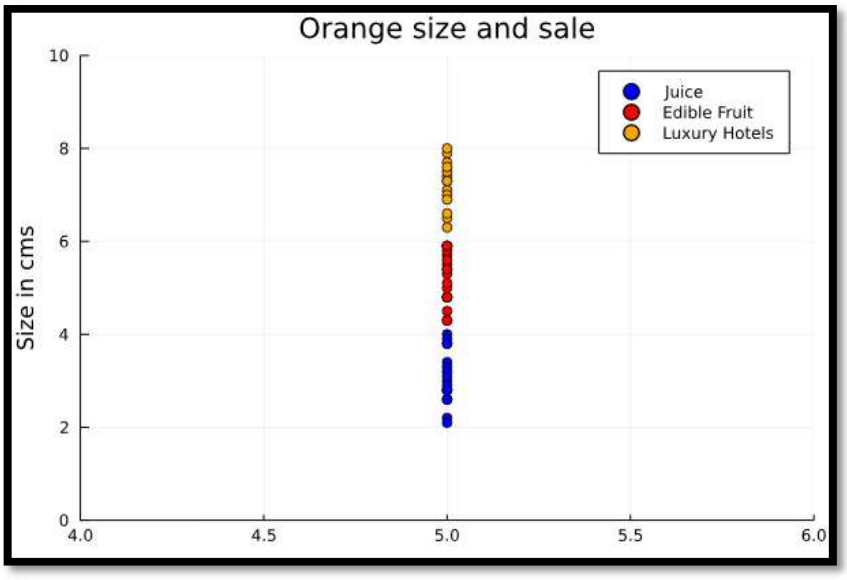
Output:

```
Dict{String, Vector{Any}} with 3 entries:
  "Edible Fruit"  => [5.7, 5.9, 5.4, 5.3, 4.3, 4.5, 4.8, 5.9, 5.8, 5.5, 4.3, 5....
  "Juice"         => [3.4, 2.6, 3.0, 4.0, 3.8, 2.8, 3.3, 3.9, 2.9, 3.8, 2.2, 2....
  "Luxury Hotels" => [7.9, 7.4, 7.3, 6.5, 8.0, 7.1, 7.0, 6.6, 6.3, 7.7, 6.9, 7....
```

```
using Plots


label = "Juice"
y = key_values[label]
x = fill(5, length(y))
p = scatter!(x, y, xlims=(4, 6), ylims=(0, 10), label = label,
    title = "Orange size and sale", ylabel = "Size in cms",
    color = "blue"
)
```

```
label = "Edible Fruit"

y = key_values[label]

x = fill(5, length(y))

scatter!(p, x, y, label = label, color = "red")



label = "Luxury Hotels"

y = key_values[label]

x = fill(5, length(y))

scatter!(p, x, y, label = label, color = "Orange")
```

Output:




```
errors_and_sells = []



for (size, sell) in orange_sizes

    push!(errors_and_sells, (Δ(orange_size, size), sell))

end
```

Le's inspect the `errors_and_sells`:

```
errors_and_sells
```

Output:

```
50-element Vector{Any}:

 (1.6, "Juice")

 (2.9000000000000004, "Luxury Hotels")

 (0.7000000000000002, "Edible Fruit")

 (2.4000000000000004, "Luxury Hotels")
```

(2.3, "Luxury Hotels")

(0.9000000000000004, "Edible Fruit")

(0.40000000000000036, "Edible Fruit")

(1.5, "Luxury Hotels")

(3.0, "Luxury Hotels")

(0.2999999999999998, "Edible Fruit")

(2.4, "Juice")

(0.7000000000000002, "Edible Fruit")

(2.0, "Juice")

⋮

(2.2, "Juice")

(1.9000000000000004, "Luxury Hotels")

(2.3, "Luxury Hotels")

(2.9, "Juice")

(0.20000000000000018, "Edible Fruit")

(0.0, "Edible Fruit")

(2.5, "Luxury Hotels")

(0.5999999999999996, "Edible Fruit")

(0.09999999999999964, "Edible Fruit")

(0.9000000000000004, "Edible Fruit")

(2.5999999999999996, "Luxury Hotels")

(2.2, "Juice")

Now let's define out K value, we take it to be 20 here, we sort `errors_and_sells` and take first K values as shown below:

```
k = 20

nearest_errors_and_sells = sort(errors_and_sells)[1:k]
```

Output:

```
20-element Vector{Any}:
 (0.0, "Edible Fruit")
 (0.09999999999999964, "Edible Fruit")
 (0.20000000000000018, "Edible Fruit")
 (0.20000000000000018, "Edible Fruit")
 (0.20000000000000018, "Edible Fruit")
 (0.2999999999999998, "Edible Fruit")
 (0.40000000000000036, "Edible Fruit")
 (0.40000000000000036, "Edible Fruit")
 (0.5, "Edible Fruit")
 (0.5, "Edible Fruit")
 (0.5999999999999996, "Edible Fruit")
 (0.7000000000000002, "Edible Fruit")
 (0.7000000000000002, "Edible Fruit")
 (0.7000000000000002, "Edible Fruit")
 (0.7999999999999998, "Edible Fruit")
 (0.9000000000000004, "Edible Fruit")
```

```
(0.9000000000000004, "Edible Fruit")

(0.9000000000000004, "Edible Fruit")

(0.9000000000000004, "Edible Fruit")

(1.0, "Juice")
```

Note that its mostly occupied by the label Edible Fruit, as a human we know the answer, for the computer to pick it we must count the sell's or it's labels. So we collect all the labels in a variable named `nearest_sells`:

```
nearest_sells = [sell for (_error, sell) in nearest_errors_and_sells]
```

Output:

```
20-element Vector{String}:

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"

 "Edible Fruit"
```

```
"Edible Fruit"

"Edible Fruit"

"Edible Fruit"

"Edible Fruit"

"Edible Fruit"

"Edible Fruit"

"Edible Fruit"

"Juice"
```

We count it:

```
counted_nearest = counter(nearest_sells)
```

Output:

```
Dict{Any, Any} with 2 entries:

  "Edible Fruit" => 19

  "Juice"        => 1
```

And we take the highest value:

```
highest_vote(counted_nearest)
```

Output:

```
"Edible Fruit"
```

Week 8:

Working with Classification and Regression Trees

Packages Available via:

AutoMLPipeline.jl - create complex ML pipeline structures using simple expressions

CombineML.jl - a heterogeneous ensemble learning package

MLJ.jl - a machine learning framework for Julia

ScikitLearn.jl - Julia implementation of the scikit-learn API

Classification

pre-pruning (max depth, min leaf size)

post-pruning (pessimistic pruning)

multi-threaded bagging (random forests)

adaptive boosting (decision stumps)

cross validation (n-fold)

support for ordered features (encoded as Reals or Strings)

Regression

pre-pruning (max depth, min leaf size)

multi-threaded bagging (random forests)

cross validation (n-fold)

support for numerical features

Classification Example:

```
using DecisionTree

features, labels = load_data("iris")    # also see "adult" and "digits" datasets
```

```
# the data loaded are of type Array{Any}
# cast them to concrete types for better performance
features = float.(features)
labels   = string.(labels)
Pruned Tree Classifier:
# train depth-truncated classifier
model = DecisionTreeClassifier(max_depth=2)
fit!(model, features, labels)
# pretty print of the tree, to a depth of 5 nodes (optional)
print_tree(model, 5)
# apply learned model
predict(model, [5.9,3.0,5.1,1.9])
# get the probability of each label
predict_proba(model, [5.9,3.0,5.1,1.9])
println(get_classes(model)) # returns the ordering of the columns in predict_proba's output
# run n-fold cross validation over 3 CV folds
# See ScikitLearn.jl for installation instructions
using ScikitLearn.CrossValidation: cross_val_score
accuracy = cross_val_score(model, features, labels, cv=3)
```

Decision Tree Classifier

```
# train full-tree classifier
model = build_tree(labels, features)
# prune tree: merge leaves having >= 90% combined purity (default: 100%)
model = prune_tree(model, 0.9)
```

```
# pretty print of the tree, to a depth of 5 nodes (optional)
print_tree(model, 5)
# apply learned model
apply_tree(model, [5.9,3.0,5.1,1.9])
# apply model to all the sames
preds = apply_tree(model, features)
# generate confusion matrix, along with accuracy and kappa scores
confusion_matrix(labels, preds)
# get the probability of each label
apply_tree_proba(model, [5.9,3.0,5.1,1.9], ["Iris-setosa", "Iris-versicolor", "Iris-virginica"])
# run 3-fold cross validation of pruned tree,
n_folds=3
accuracy = nfoldCV_tree(labels, features, n_folds)


# set of classification parameters and respective default values
# pruning_purity: purity threshold used for post-pruning (default: 1.0, no pruning)
# max_depth: maximum depth of the decision tree (default: -1, no maximum)
# min_samples_leaf: the minimum number of samples each leaf needs to have (default: 1)
# min_samples_split: the minimum number of samples in needed for a split (default: 2)
# min_purity_increase: minimum purity needed for a split (default: 0.0)
# n_subfeatures: number of features to select at random (default: 0, keep all)
# keyword rng: the random number generator or seed to use (default Random.GLOBAL_RNG)
n_subfeatures=0; max_depth=-1; min_samples_leaf=1; min_samples_split=2
min_purity_increase=0.0; pruning_purity = 1.0; seed=3
```

```
model    =   build_tree(labels, features,
                        n_subfeatures,
                        max_depth,
                        min_samples_leaf,
                        min_samples_split,
                        min_purity_increase;
                        rng = seed)

accuracy = nfoldCV_tree(labels, features,
                        n_folds,
                        pruning_purity,
                        max_depth,
                        min_samples_leaf,
                        min_samples_split,
                        min_purity_increase;
                        verbose = true,
                        rng = seed)
```

Regression Example

```
n, m = 10^3, 5

features = randn(n, m)

weights = rand(-2:2, m)

labels = features * weights
```

Regression Tree

```
# train regression tree

model = build_tree(labels, features)
```

```
# apply learned model
apply_tree(model, [-0.9,3.0,5.1,1.9,0.0])
# run 3-fold cross validation, returns array of coefficients of determination (R^2)
n_folds = 3
r2 = nfoldCV_tree(labels, features, n_folds)

# set of regression parameters and respective default values
# pruning_purity: purity threshold used for post-pruning (default: 1.0, no pruning)
# max_depth: maximum depth of the decision tree (default: -1, no maximum)
# min_samples_leaf: the minimum number of samples each leaf needs to have (default: 5)
# min_samples_split: the minimum number of samples in needed for a split (default: 2)
# min_purity_increase: minimum purity needed for a split (default: 0.0)
# n_subfeatures: number of features to select at random (default: 0, keep all)
# keyword rng: the random number generator or seed to use (default Random.GLOBAL_RNG)
n_subfeatures = 0; max_depth = -1; min_samples_leaf = 5
min_samples_split = 2; min_purity_increase = 0.0; pruning_purity = 1.0 ; seed=3

model = build_tree(labels, features,
                   n_subfeatures,
                   max_depth,
                   min_samples_leaf,
                   min_samples_split,
                   min_purity_increase;
                   rng = seed)
```

```
r2 =  nfoldCV_tree(labels, features,
                   n_folds,
                   pruning_purity,
                   max_depth,
                   min_samples_leaf,
                   min_samples_split,
                   min_purity_increase;
                   verbose = true,
                   rng = seed)
```

Saving Models

Models can be saved to disk and loaded back with the use of the JLD2.jl package.

```
using JLD2
@save "model_file.jld2" model
```

Week 9:

Working with Random Forests and Gradient Boosting algorithms

The Iris flower dataset

The Iris flower dataset is commonly used for beginner machine learning problems. The full dataset can be found on Kaggle at www.kaggle.com/arshid/iris-flower-dataset. It consists of 150 entries for 3 types of iris plants, and 4 features: sepal length and width, and petal length and width.

Based on these, a simple baseline model can be developed:

1. If PetalLength < 2.5cm, class is Setosa.
2. Else determine scores score1 and score2 as follows:

   score1: add 1 for each of the following that is true:

   2.5cm < PetalLength ≤ 5.0cm

   1.0cm≤ PetalWidth ≤ 1.8cm

   score2: add 1 for each of the following that is true:

   7.0cm≤ SepalLength

   3.5cm≤ SepalWidth

   5.0cm≤ PetalLength

   1.7cm< PetalWidth

3. If score1 > score2, classify as Veriscolor. If score1 < score2, classify as Virginica. If score1 = score2, leave unknown, or classify at random.

This simple strategy guarantees that 140 samples, which is 93.3% of the samples, will be correctly classified.

module TreeEnsemble

export AbstractClassifier, predict, score, fit!, perm_feature_importance,

```
        # binary tree
        BinaryTree, add_node!, set_left_child!, set_right_child!, get_children,
        is_leaf, nleaves, find_depths, get_max_depths,
        # Decision Tree Classifier
        DecisionTreeClassifier, predict_row, predict_batch, predict_prob,
        feature_importance_impurity, print_tree, node_to_string,
        # Random Forest Classifier
        RandomForestClassifier,
        # utilities
        check_random_state, split_data, confusion_matrix, calc_f1_score

include("RandomForest.jl")

end

RANDOM FOREST CLASSIFIER:
using Random
using CSV, DataFrames, Printf

include("DecisionTree.jl")

mutable struct RandomForestClassifier{T}  <: AbstractClassifier
    #internal variables
    n_features::Union{Int, Nothing}
    n_classes::Union{Int, Nothing}
```

```
    features::Vector{String}
    trees::Vector{DecisionTreeClassifier}
    feature_importances::Union{Vector{Float64}, Nothing}

    # external parameters
    n_trees::Int
    max_depth::Union{Int, Nothing}
    max_features::Union{Int, Nothing} # sets n_features_split
    min_samples_leaf::Int
    random_state::Union{AbstractRNG, Int}
    bootstrap::Bool
    oob_score::Bool
    oob_score_::Union{Float64, Nothing}

    RandomForestClassifier{T}(;
            n_trees=100,
            max_depth=nothing,
            max_features=nothing,
            min_samples_leaf=1,
            random_state=Random.GLOBAL_RNG,
            bootstrap=true,
            oob_score=false
        ) where T = new(
            nothing, nothing, [], [], nothing, n_trees,
            max_depth,
```

```
                    max_features,
                    min_samples_leaf,
                    check_random_state(random_state),
                    bootstrap,
                    oob_score,
                    nothing
        )
end

function fit!(forest::RandomForestClassifier, X::DataFrame, Y::DataFrame)
    @assert size(Y, 2) == 1 "Output Y must be an m x 1 DataFrame"

    # set internal variables
    forest.n_features = size(X, 2)
    forest.n_classes = size(unique(Y), 1)
    forest.features = names(X)
    forest.trees = []

    # create decision trees
    rng_states = typeof(forest.random_state)[]  # save the random states to regenerate the random indices for the oob_score
    for i in 1:forest.n_trees
        push!(rng_states, copy(forest.random_state))
        push!(forest.trees, create_tree(forest, X, Y))
    end
```

```
    # set attributes
    forest.feature_importances = feature_importance_impurity(forest)
    if forest.oob_score
        if !forest.bootstrap
            println("Warning: out-of-bag score will not be calculated because
bootstrap=false")
        else
            forest.oob_score_ = calculate_oob_score(forest, X, Y, rng_states)
        end
    end

    return
end

function create_tree(forest::RandomForestClassifier, X::DataFrame, Y::DataFrame)
    n_samples = nrow(X)

    if forest.bootstrap # sample with replacement
        idxs = [rand(forest.random_state, 1:n_samples) for i in 1:n_samples]
        X_ = X[idxs, :]
        Y_ = Y[idxs, :]
    else
        X_ = copy(X)
        Y_ = copy(Y)
    end
```

```
        T = typeof(forest).parameters[1] # use the same type T used for the forest
    new_tree = DecisionTreeClassifier{T}(
            max_depth = forest.max_depth,
            max_features = forest.max_features,
            min_samples_leaf = forest.min_samples_leaf,
            random_state = forest.random_state
    )
    fit!(new_tree, X_, Y_)

    return new_tree
end
```

The prediction of the forest is done through majority voting. In particular, a 'soft' vote is done, where each tree's vote is weighted by its probability prediction per class. The final prediction is therefore equivalent to the class with the maximum sum of probabilities.

```
function predict_prob(forest::RandomForestClassifier, X::DataFrame)
    if length(forest.trees) == 0
        throw(NotFittedError(:forest))
    end
    probs = zeros(nrow(X), forest.n_classes)
    for tree in forest.trees
        probs .+= predict_prob(tree, X)
    end
    return probs
end
```

```
function predict(forest::RandomForestClassifier, X::DataFrame)
    probs = predict_prob(forest, X)
    return mapslices(argmax, probs, dims=2)[:, 1]
end


function calculate_oob_score(
    forest::RandomForestClassifier, X::DataFrame, Y::DataFrame,
    rng_states::Vector{T}) where T <: AbstractRNG
    n_samples = nrow(X)
    oob_prob  = zeros(n_samples, forest.n_classes)
    oob_count = zeros( n_samples)
    for (i, rng) in enumerate(rng_states)
        idxs = Set([rand(forest.random_state, 1:n_samples) for i in 1:n_samples])
        # note: expected proportion of out-of-bag is 1-exp(-1) = 0.632...
        # so length(row_oob)/n_samples ≈ 0.63
        row_oob =  filter(idx -> !(idx in idxs), 1:n_samples)
        oob_prob[row_oob, :] .+= predict_prob(forest.trees[i], X[row_oob, :])
        oob_count[row_oob] .+= 1.0
    end
    # remove missing values
    valid = oob_count .> 0.0
    oob_prob = oob_prob[valid, :]
    oob_count = oob_count[valid]
    y_test = Y[valid, 1]
    # predict out-of-bag score
```

```
    y_pred = mapslices(argmax, oob_prob./oob_count, dims=2)[:, 1]
    return mean(y_pred .==  y_test)
end
```

The final function in RandomForestClassifier calculates the impurity based feature importance. It does so by finding the mean of the feature importances in each tree. The detail behind these will be delayed to the next section.

```
function feature_importance_impurity(forest::RandomForestClassifier)
    if length(forest.trees) == 0
        throw(NotFittedError(:forest))
    end
    feature_importances = zeros(forest.n_trees, forest.n_features)
    for (i, tree) in enumerate(forest.trees)
        feature_importances[i, :] = tree.feature_importances
    end
    return mean(feature_importances, dims=1)[1, :]
end
```

Gradient Boosting

```
using GradientBoost.ML
using RDatasets

# Obtain iris dataset
iris = dataset("datasets", "iris")
instances = array(iris[:, 1:end-1])
```

```
labels = [species == "setosa" ? 1.0 : 0.0 for species in array(iris[:, end])]

# Obtain training and test set (20% test)
num_instances = size(instances, 1)
train_ind, test_ind = GradientBoost.Util.holdout(num_instances, 0.2)

# Build GBLearner
gbdt = GBDT(;
  loss_function = BinomialDeviance(),
  sampling_rate = 0.6,
  learning_rate = 0.1,
  num_iterations = 100
)
gbl = GBLearner(
  gbdt,  # Gradient boosting algorithm
  :class # Output (:class, :class_prob, :regression)
)
# Train
ML.fit!(gbl, instances[train_ind, :], labels[train_ind])

# Predict
predictions = ML.predict!(gbl, instances[test_ind, :])
# Obtain accuracy
accuracy = mean(predictions .== labels[test_ind]) * 100.0
println("GBDT accuracy: $(accuracy)")
```

```
gbdt = GBDT(;
  loss_function = BinomialDeviance(), # Loss function
  sampling_rate = 0.6,               # Sampling rate
  learning_rate = 0.1,               # Learning rate
  num_iterations = 100,              # Number of iterations
  tree_options = {                   # Tree options (DecisionTree.jl regressor)
    :maxlabels => 5,
    :nsubfeatures => 0
  }
)

import GLM: fit, predict, LinearModel

# Extend functions
function ML.learner_fit(lf::LossFunction,
  learner::Type{LinearModel}, instances, labels)

  model = fit(learner, instances, labels)
end
function ML.learner_predict(lf::LossFunction,
  learner::Type{LinearModel}, model, instances)

  predict(model, instances)
end
```

```
gbl = GBBL(
  LinearModel;                  # Base Learner
  loss_function = LeastSquares(), # Loss function
  sampling_rate = 0.8,           # Sampling rate
  learning_rate = 0.1,           # Learning rate
  num_iterations = 100           # Number of iterations
)
gbl = GBLearner(gbl, :regression)


import GradientBoost.GB
import GradientBoost.LossFunctions: LossFunction


# Must subtype from GBAlgorithm defined in GB module.
type ExampleGB <: GB.GBAlgorithm
  loss_function::LossFunction
  sampling_rate::FloatingPoint
  learning_rate::FloatingPoint
  num_iterations::Int
end


# Model training and co-efficient optimization should be done here.
function GB.build_base_func(
  gb::ExampleGB, instances, labels, prev_func_pred, psuedo)


  model_const = 0.5
```

```
  model_pred = (instances) -> Float64[
    sum(instances[i,:]) for i = 1:size(instances, 1)
  ]

  return (instances) -> model_const .* model_pred(instances)
end
```