

Lab04: Decision Tree and Naive Bayes

- Student ID: 21127621
- Student name: Âu Dương Khang

```
md"""  
# Lab04: Decision Tree and Naive Bayes  
.  
- Student ID: 21127621  
- Student name: Âu Dương Khang  
"""
```

How to do your homework

You will work directly on this notebook; the word `TODO` indicate the parts you need to do.

You can discuss ideas with classmates as well as finding information from the internet, book, etc...; but *this homework must be your*.

How to submit your homework

- Before submitting, save this file as `<ID>.jl`. For example, if your ID is 123456, then your file will be `123456.jl`. And export to PDF with name `123456.pdf` then submit zipped source code and pdf into `123456.zip` onto Moodle.

Danger

Note that you will get 0 point for the wrong submit.

Contents:

- Decision Tree
- Naive Bayes

Import library

```
begin  
    using Distributions, Plots, LinearAlgebra, Random, Statistics  
end
```

MersenneTwister(2022)

- `Random.seed!(2022)`

Load Iris dataset

download_dataset (generic function with 2 methods)

- *# If you use Linux, use this function to download Iris dataset*
- `function download_dataset(save_path::String="data")`
- `# setup directory`
- `mkpath(joinpath(dirname(@__FILE__), save_path))`
- `data_dir = joinpath(dirname(@__FILE__), save_path)`
- `download("https://archive.ics.uci.edu/static/public/53/iris.zip", joinpath(data_dir, "iris.zip"))`
- `iris_file = joinpath(data_dir, "iris.zip")`
- `cd(data_dir)`
- `run(`unzip $iris_file -d $data_dir`)`
- `rm(iris_file)`
- `cd("..")`
- `end`

- *# If you have downloaded the dataset yet, please uncomment this line below and run this cell. Otherwise, keep it in uncomment state.*
- `# download_dataset()`

- *# If you don't use Linux, I have no solution for you. Please mkdir data, and goto <https://archive.ics.uci.edu/dataset/53/iris> for downloading*
- *# Then, you can extract data to this dir by yourself.*
- *#*
- *# Structure:*
- *# |— data*
- *# | |— bezdekIris.data*
- *# | |— Index*
- *# | |— iris.data*
- *# | |— iris.names*
- *# |— Lab4.jl*

iris_dataloader (generic function with 2 methods)

```
• function iris_dataloader(data_path::String="data/iris.data")
•     # Initialize empty arrays to store data
•     sepal_length = Float64[]
•     sepal_width = Float64[]
•     petal_length = Float64[]
•     petal_width = Float64[]
•     classes = Int64[]
•
•     # open and read the data file
•     open(data_path, "r") do file
•         # read data each line
•         for line in eachline(file)
•             if line != ""
•                 parts = split(line, ",")
•                 push!(sepal_length, parse(Float64, parts[1]))
•                 push!(sepal_width, parse(Float64, parts[2]))
•                 push!(petal_length, parse(Float64, parts[3]))
•                 push!(petal_width, parse(Float64, parts[4]))
•
•                 if parts[5] == "Iris-setosa"
•                     push!(classes, 0)
•                 elseif parts[5] == "Iris-versicolor"
•                     push!(classes, 1)
•                 else
•                     push!(classes, 2)
•                 end
•             end
•         end
•     end
•
•     # concat features
•     features = [sepal_length, sepal_width, petal_length, petal_width]
•     features = vcat(transpose.(features)...)
•     return features, classes
• end
```

```
• # function change_class_to_num(y)
• #   class = Dict("setosa"=> 0, "versicolor"=> 1, "virginica" => 2)
• #   classnums = [class[item] for item in y]
• #   return classnums
• # end
```

train_test_split (generic function with 2 methods)

```

• function train_test_split(X, y, test_ratio=0.33)
•     X = X'
•     n = size(X)[1]
•     idx = shuffle(1:n)
•     train_size = 1 - test_ratio
•     train_idx = view(idx, 1:floor(Int, train_size*n))
•     test_idx = view(idx, (floor(Int, train_size*n)+1):n)
•
•     X_train = X[train_idx,:]
•     X_test = X[test_idx,:]
•
•     y_train = y[train_idx]
•     y_test = y[test_idx]
•
•     return X_train, X_test, y_train, y_test
• end

```

```
((100, 4), (50, 4), (100), (50))
```

```

• begin
•     # Load features, and labels for Iris dataset
•     iris_features, iris_labels = iris_dataloader("data/iris.data")
•
•     #split dataset into training data and testing data
•     X_train, X_test, y_train, y_test = train_test_split(iris_features, iris_labels,
•         0.33)
•
•     size(X_train), size(X_test), size(y_train), size(y_test)
• end

```

1. Decision Tree: Iterative Dichotomiser 3 (ID3)

1.1 Information Gain

Expected value of the self-information (entropy):

$$Entropy = - \sum_i^n p_i \log_2(p_i)$$

The entropy function gets the smallest value if there is a value of p_i equal to 1, reaches the maximum value if all p_i are equal. These properties of the entropy function make it is an expression of the disorder, or randomness of a system, ...

entropy

Parameters:

- `counts` : shape (n_classes): list number of samples in each class
- `n_samples`: number of data samples

Returns

- `entropy`

```
• """  
• Parameters:  
• - `counts`: shape (n_classes): list number of samples in each class  
• - `n_samples`: number of data samples  
•  
• Returns  
• - entropy  
• """  
• function entropy(counts, n_samples)  
•     #TODO  
•     entropy = 0  
•     for count in counts  
•         p_i = count / n_samples  
•         entropy -= p_i * log2(p_i)  
•     end  
•  
•     return entropy  
•  
• end
```

entropy_of_one_division

Returns entropy of a divided group of data

Data may have multiple classes

```

• """
• Returns entropy of a divided group of data
•
• Data may have multiple classes
• """
• function entropy_of_one_division(division)
•
•     n_samples = size(division, 1)
•     n_classes = Set(division)
•
•     counts=[]
•
•     # count samples in each class then store it to list counts
•     #TODO:
•     counts = [count(x -> x == n_class, division) for n_class in n_classes]
•
•     return entropy(counts,n_samples),n_samples
•
• end

```

get_entropy

Returns entropy of a split

y_predict is the split decision by cutoff, True/Fasle

```

• """
• Returns entropy of a split
•
• y_predict is the split decision by cutoff, True/Fasle
• """
• function get_entropy(y_predict, y)
•     n = size(y,1)
•     # left hand side entropy
•     entropy_true, n_true = entropy_of_one_division(y[y_predict])
•
•     # right hand side entropy
•     entropy_false, n_false = entropy_of_one_division(y[.~y_predict])
•
•     # overall entropy
•     #TODO s=?
•     s = entropy_true * n_true / n + entropy_false * n_false / n
•     return s
• end

```

The information gain of classifying information set D by attribute A:

$$Gain(A) = Entropy(D) - Entropy_A(D)$$

At each node in ID3, an attribute is chosen if its information gain is highest compare to others.

All attributes of the Iris set are represented by continuous values. Therefore we need to represent them with discrete values. The simple way is to use a `cutoff` threshold to separate values of the data on each attribute into two part: `<cutoff` and `> = cutoff`.

To find the best `cutoff` for an attribute, we replace `cutoff` with its values then compute the entropy, best `cutoff` achieved when value of entropy is smallest ($\arg \min Entropy_A(D)$).

- `md"""`
- The information gain of classifying information set D by attribute A:
-
- $Gain(A) = Entropy(D) - Entropy_{\{A\}}(D)$
-
- At each node in ID3, an attribute is chosen if its information gain is highest compare to others.
-
- All attributes of the Iris set are represented by continuous values. Therefore we need to represent them with discrete values. The simple way is to use a `'cutoff'` threshold to separate values of the data on each attribute into two part: `'<cutoff'` and `'> = cutoff'`.
-
- To find the best `'cutoff'` for an attribute, we replace `'cutoff'` with its values then compute the entropy, best `'cutoff'` achieved when value of entropy is smallest $\left(\arg \min Entropy_{\{A\}}(D) \right)$.
- `"""`

1.2 Decision tree

dtfit

Parameters:

- X: training data
- y: label of training data

Returns

- node

node: each node represented by cutoff value and column index, value and children.

- cutoff value is threshold where you divide your attribute.
- column index is your data attribute index.
- value of node is mean value of label indexes, if a node is leaf all data samples will have same label.

Note that: we divide each attribute into 2 part => each node will have 2 children: left, right.

```

• """
• Parameters:
• - X: training data
• - y: label of training data
•
• Returns
• - node
•
• node: each node represented by cutoff value and column index, value and children.
• - cutoff value is threshold where you divide your attribute.
• - column index is your data attribute index.
• - value of node is mean value of label indexes, if a node is leaf all data samples
  will have same label.
•
• Note that: we divide each attribute into 2 part => each node will have 2 children:
  left, right.
• """
• function dtfit(X, y, node=Dict(), depth=0)
•     #Stop conditions
•
•     #if all value of y are the same
•     if all(y.==y[1])
•         return Dict("val"=>y[1])
•
•     else
•         # find one split given an information gain
•         col_idx, cutoff, entropy = find_best_split_of_all(X, y)
•
•         y_left = y[X[:,col_idx] .< cutoff]

```



```
• y_right = y[X[:,col_idx] .>= cutoff]
•
• node = Dict("index_col"=>col_idx,
•         "cutoff"=>cutoff,
•         "val"=> mean(y),
•         "left"=> Any,
•         "right"=> Any)
•
• left = dtfit(X[X[:,col_idx] .< cutoff, :], y_left, Dict(), depth+1)
• right= dtfit(X[X[:,col_idx] .>= cutoff, :], y_right, Dict(), depth+1)
•
• push!(node, "left" => left)
• push!(node, "right" => right)
•
• depth += 1
• end
• return node
end
```

find_best_split_of_all

Parameters:

- X: training data
- y: label of training data

Returns

- column index, cut-off value, and minimum entropy

```
• """
• Parameters:
• - X: training data
• - y: label of training data
•
• Returns
• - column index, cut-off value, and minimum entropy
• """
• function find_best_split_of_all(X, y)
•     col_idx = nothing
•     min_entropy = 1
•     cutoff = nothing
•
•     for i in 1:size(X,2)
•         col_data = X[:,i]
•         entropy, cur_cutoff = find_best_split(col_data, y)
•
•         # best entropy
•         if entropy == 0
•             return i, cur_cutoff, entropy
•         elseif entropy <= min_entropy
•             min_entropy = entropy
•             col_idx = i
•             cutoff = cur_cutoff
•         end
•     end
•     return col_idx, cutoff, min_entropy
• end
```

find_best_split

Parameters:

- col_data: data samples in column
- y: label of training data

Returns

- minimum entropy, and cut-off value

```

• """
• Parameters:
• - col_data: data samples in column
• - y: label of training data
•
• Returns
• - minimum entropy, and cut-off value
• """
• function find_best_split(col_data, y)
•     min_entropy = 10
•     cutoff = 0
•
•     #Loop through col_data find cutoff where entropy is minimum
•
•     for value in Set(col_data)
•         y_predict = col_data .< value
•         my_entropy = get_entropy(y_predict, y)
•
•         #TODO
•         #min entropy=?, cutoff=?
•         if my_entropy <= min_entropy
•             min_entropy = my_entropy
•             cutoff = value
•         end
•     end
•     return min_entropy, cutoff
• end

```

dtpredict (generic function with 1 method)

```

• function dtpredict(tree, data)
•     pred = []
•     n_sample = size(data, 1)
•     for i in 1:n_sample
•         push!(pred, _dtpredict(tree, data[i,:]))
•     end
•     return pred
• end

```

_dtpredict (generic function with 1 method)

```

• function _dtpredict(tree, row)
•     cur_layer = tree
•     while haskey(cur_layer, "cutoff")
•         if row[cur_layer["index_col"]] < cur_layer["cutoff"]
•             cur_layer = cur_layer["left"]
•         else
•             cur_layer = cur_layer["right"]
•         end
•     end
•     if !haskey(cur_layer, "cutoff")
•         return get(cur_layer, "val", false)
•     end
• end

```

1.3 Classification on Iris Dataset

tpfpntfn_cal (generic function with 2 methods)

```

• function tpfpntfn_cal(y_test, y_pred, positive_class=1)
•     true_positives = 0
•     false_positives = 0
•     true_negatives = 0
•     false_negatives = 0
•
•     # Calculate true positives, false positives, false negatives, and true negatives
•     for (true_label, predicted_label) in zip(y_test, y_pred)
•         if true_label == positive_class && predicted_label == positive_class
•             true_positives += 1
•         elseif true_label != positive_class && predicted_label == positive_class
•             false_positives += 1
•         elseif true_label == positive_class && predicted_label != positive_class
•             false_negatives += 1
•         elseif true_label != positive_class && predicted_label != positive_class
•             true_negatives += 1
•         end
•     end
•
•     return true_positives, false_positives, true_negatives, false_negatives
• end

```

tree =

Dict("left" => Dict("val" => 0), "cutoff" => 1.0, "right" => Dict("left" => Dict("left" =

• tree = dtfit(X_train, y_train)

```

• begin
•   pred = dtpredict(tree, X_test)
•
•   acc = 0
•   precision = 0
•   recall = 0
•   f1 = 0
•
•   for i ∈ [0, 1, 2]
•       # Calculate true positives, false positives, false negatives, and true
negatives
•       true_positives, false_positives, true_negatives, false_negatives =
tpfptnfn_cal(y_test, pred, i)
•
•       # Calculate precision, recall, and F1-score
•       acc += (true_positives + true_negatives) / (true_positives + false_positives
+ true_negatives + false_negatives)
•       precision += true_positives / (true_positives + false_positives)
•       recall += true_positives / (true_positives + false_negatives)
•   end
•
•   acc = acc / 3
•   precision = precision / 3
•   recall = recall / 3
•   f1 = 2 * precision * recall / (precision + recall)
•   print(" acc: $acc\n precision: $precision\n recall: $recall\n f1_score: $f1\n")
• end

```

```

acc: 0.96
precision: 0.9523809523809524
recall: 0.923076923076923
f1_score: 0.9375

```

2. Bayes Theorem

Bayes formulation $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

If B is our data \mathcal{D} , A and w are parameters we need to estimate:

$$\underbrace{P(w|\mathcal{D})}_{\text{Posterior}} = \frac{1}{\underbrace{P(\mathcal{D})}_{\text{Normalization}}} \underbrace{P(\mathcal{D}|w)}_{\text{Likelihood}} \underbrace{P(w)}_{\text{Prior}}$$

Naive Bayes

To make it simple, it is often assumed that the components of the \mathcal{D} random variable (or the features of the \mathcal{D} data) are independent with each other, if w is known. It mean:

$$P(\mathcal{D}|w) = \prod_{i=1}^d P(x_i|w)$$

- d: number of features

2.1. Probability Density Function

update (generic function with 1 method)

```

• #update histogram for new data
• function update(_hist, _mean, _std, data)
•     """
•     P(hypo/data)=P(data/hypo)*P(hypo)*(1/P(data))
•     """
•     hist = copy(_hist)
•     #P(hypo/data)=P(data/hypo)*P(hypo)*(1/P(data))
•
•     #Likelihood * Prior
•     #TODO
•     s = 0
•     for hypo in keys(hist)
•         hypo_likelihood = hist[hypo] * likelihood(_mean, _std, data, hypo)
•         hist[hypo] = hypo_likelihood
•         s+= hypo_likelihood
•     end
•     #Normalization
•
•     #TODO: s=P(data)
•     #s=?
•     for hypo in keys(hist)
•         hist[hypo] = hist[hypo]/s
•     end
•     return hist
• end

```

maxHypo (generic function with 1 method)

```

• function maxHypo(hist)
•     #find the hypothesis with maximum probability from hist
•     #TODO
•     max_value_index = argmax(collect(values(hist)))
•     max_keys = collect(keys(hist))
•     return max_keys[max_value_index]
• end

```

2.2 Classification on Iris Dataset

Gaussian Naive Bayes

- Naive Bayes can be extended to use on continuous data, most commonly by using a normal distribution (Gaussian distribution).
- This extension called Gaussian Naive Bayes. Other functions can be used to estimate data distribution, but Gauss (or the normal distribution) is the easiest to work with since we only need to estimate the mean and standard deviation from the training data.

Define Gauss function

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

Gauss (generic function with 1 method)

```

• function Gauss(std, mean, x)
•     #Compute the Gaussian probability distribution function for x
•     #TODO
•     return (1.0 / (std * sqrt(2 * pi))) * exp(-((x - mean)^2) / (2 * std^2))
• end

```

likelihood (generic function with 5 methods)

```

• function likelihood(_mean=nothing, _std=nothing, data=nothing, hypo=nothing)
•     """
•     Returns: res=P(data/hypo)
•     -----
•     Naive bayes:
•     Atributes are assumed to be conditionally independent given the class value.
•     """
•
•     std=_std[hypo]
•     mean=_mean[hypo]
•     res=1
•     #TODO
•     #res=res*P(x1/hypo)*P(x2/hypo)...
•     for i in 1:length(data)
•         res *= Gauss(std[i], mean[i], data[i])
•     end
•     return res
• end

```

gfit (generic function with 4 methods)

```

• function gfit(X, y, _std=nothing, _mean=nothing, _hist=nothing)
•     """Parameters:
•     X: training data
•     y: labels of training data
•     """
•     n=size(X,1)
•     #number of iris species
•     #TODO
•     #n_species=???
•     n_species = length(Set(y))
•
•     hist_dict = Dict()
•     mean_dict = Dict()
•     std_dict = Dict()
•
•     #separate dataset into rows by class
•     for hypo in Set(y)
•         #rows have hypo label
•         #TODO rows=
•         rows = findall(a->a==hypo,y)
•         #histogram for each hypo
•         #TODO probability=?
•         probability = length(rows) / n
•         hist_dict[hypo] = probability
•
•         #Each hypothesis represented by its mean and standard derivation
•         """mean and standard derivation should be calculated for each column (or each
attribute)"""
•         #TODO mean[hypo]=?, std[hypo]=?
•         mean_dict[hypo] = mean(X[rows, :], dims=1)
•         std_dict[hypo] = std(X[rows, :], dims=1)
•
•     end
•     _mean = mean_dict
•     _std = std_dict
•     _hist = hist_dict
•     return _hist, _mean, _std
• end

```


`_gpredict` (generic function with 2 methods)

```

• function _gpredict(_hist, _mean, _std, data, plot=true)
•     """
•     Predict label for only 1 data sample
•     -----
•     Parameters:
•     data: data sample
•     plot: True: draw histogram after update new record
•     -----
•     return: label of data
•     """
•     hist = update(_hist, _mean, _std, data)
•     if (plot == true)
•         plt = bar(collect(keys(hist)), collect(values(hist)))
•     end
•     return maxHypo(hist)
• end

```

`plot_pdf` (generic function with 1 method)

```

• function plot_pdf(_hist)
•     bar(collect(keys(_hist)), collect(values(_hist)))
• end

```

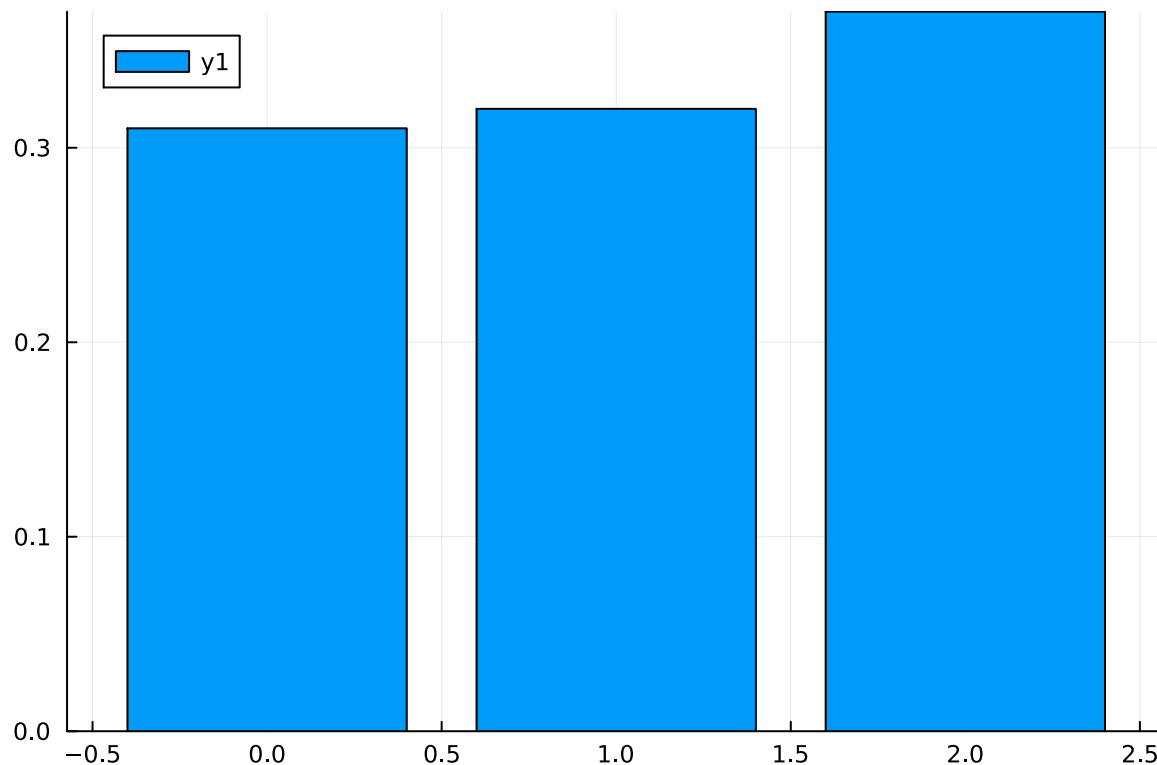
`gpredict` (generic function with 1 method)

```

• function gpredict(_hist, _mean, _std, data)
•     """Parameters:
•     Data: test data
•     -----
•     return labels of test data
•     """
•     pred=[]
•     n_sample = size(data, 1)
•     for i in 1:n_sample
•         push!(pred, _gpredict(_hist, _mean, _std, data[i,:]))
•     end
•     return pred
• end

```

Show histogram of training data



```

• begin
•   _hist, _mean, _std = gfit(X_train, y_train)
•   plt = plot_pdf(_hist)
• end

```

Test with 1 data record

```

• begin
•   #label of test_y[10]
•   print("Label of X_test[10]: ", y_test[20])
•
•   #update model and show histogram with X_test[10]:
•   print("\nOur histogram after update X_test[10]: ", _gpredict(_hist, _mean, _std,
•     X_test[20,:], true))
•
• end

```

```

Label of X_test[10]: 0
Our histogram after update X_test[10]: u

```

Evaluate your Gaussian Naive Bayes model

```

• begin
•   _pred = gpredict(_hist, _mean, _std, X_test)
•
•   _acc = 0
•   _p = 0
•   _r = 0
•   _f1 = 0
•
•   #TODO: Self-define and calculate accuracy, precision, recall, and f1-score
•   # Calculate accuracy, precision, recall, and f1-score
•   for i in 1:10
•       _true_positives, _false_positives, _true_negatives, _false_negatives =
         tpfpntfn_cal(y_test, _pred)
•
•       _acc += (_true_positives + _true_negatives) / (_true_positives +
         _false_positives + _true_negatives + _false_negatives)
•       _p += _true_positives / (_true_positives + _false_positives)
•       _r += _true_positives / (_true_positives + _false_negatives)
•   end
•   _acc /= 10
•   _p /= 10
•   _r /= 10
•   _f1 = 2 * _p * _r / (_p + _r)
•   print(" acc: $_acc\n precision: $_p\n recall: $_r\n f1_score: $_f1\n")
• end

```

```

acc: 0.92000000000000002
precision: 0.84999999999999999
recall: 0.94444444444444444
f1_score: 0.894736842105263

```

TODO: F1, Recall and Precision report