class FormulasHelper

{

string[][] Data {get;}

Column DecisionColumn {get;}

Dictionary<string, double> decisionRates;

Column column;

Column Column{get;set{

if(column.Index != Value.Index){

//recalculate cRates;

}

column = Value;

}}

Dictionary<string, Dictionary<string, double>> cRates;

FormulasHelper(string[][] data, Column DecisionColumn)

{

…

decisionRates = new();

//calculates decision rates of decision column.

}

Double GetEntropy(){};

Double GetEntropy(Column column, string value){};

//continuous

Double GetEntropy(Column column, string value,

bool greaterThanCurrentValue){};

Double GetInfoGain(Column column){};

//continuous

Double GetInfoGain(Column column, string value){};

Double GetSplitInfo(Column column){};

//continuous

Double GetSplitInfo(Column column, string value){};

Double GetGainRatio(Column column){};

//continuous

Double GetGainRatio(Column column, string value){};

}

class Column

{

String Name;

Int Index;

HashSet<string> Values;

Bool IsNominal;

Column(string name, int index, bool isNominal){

…

Values = new();

}

Column Clone(){}

Column DeepClone(){}

}

class Node

{

Dictionary<string, Node> childs;

Column column;

Func<string[], string> decideFunc;

Double? Threshold;

Node(){

childs = new Dic<>();

}

AddChild(string name, Node child){}

String Decide(string[] row) => Return decideFunc(row);

}

class NodeCreator

{

String[][] data;

Column[] columns;

Node node;

FormulasHelper formulas;

NodeCreator(String[][] data, Column[] columns)

:this(data, columns, new Node())

{}

NodeCreator(String[][] data, Column[] columns, Node node){

…

formulas =

new FormulasHelper(data,columns[columns.Length-1]);

}

}

//continue 1

class NodeCreator

{

Node GetNode()

{

If(node == null)

node = new Node();

ColumnWrapper cw = GetBranchingColumn();

node.Column = cw.Column;

node.Threshold = cw.Threshold;

if(node.Column != null){

if(node.Column.IsNominal){

node.decideFunc = (row) =>

return node

.Childs[row[node.Column.Index]]

.Decide(row);

}

Else node.decideFunc = (row) =>{

If(row[node.Column.Index].ToDouble()>

node.Threshold) {

return node

.Childs[true.ToString()]

.Decide(row);

}

Else return node

.Childs[false.ToString()]

.Decide(row);

}

}

Else node.decideFunc = (row) => mostAppearResult;

Return node;

}

Private class ColumnWrapper

{

Column Column;

Double? Threshold;

ColumnWrapper(Column column, Double? threshold)

:this(column)

{…}

ColumnWrapper(Column column){…}

}

}

//continue 2

class NodeCreator

{

ColumnWrapper GetBranchingColumn(){

If(Entropy == 0)

Return new ColumnWrapper(null);

Double gainRatio = minimum of double;

Double? threshold = null;

Column result = null;

//remove columns have 1 value

Columns newCols = columns.clone();

For(int j = 0;j<columns.length-1;j++){

Column col = columns[j];

If(col.Values.Length < 2){

newCols.remove(col);

}

//calculate gain ratio

Else If(col.IsNominal){

Double gR = calculate\_gain\_ratio;

if(gR > gainRatio){

gainRatio = gR;

threshold = null;

result = col;

}

}

Else {

For(int i = 0; i<col.Values.length-1;i++){

Double Th = col.Values[i];

Double gR = calculate\_gain\_ratio;

if(gR > gainRatio){

gainRatio = gR;

threshold = Th;

result = col;

}

}

}

}

Columns = newCols;

Return new ColumnWrapper(result, threshold);

}

func CreateTree(String[][] data)

{

Column[] columns = extracts\_from\_data();

//sort if column’s type is continuous

Stack<NodeCreator> stack = new ();

Node tree = new Node();

stack.Push(new NodeCreator(data, columns, tree));

while(!stack.Empty()){

NodeCreator nc = stack.Pop();

node = nc.GetNode();

if(node.Column == null)

continue;

if(node.Column.IsNominal){

foreach(string value in node.Column.Values)

{

Node child = new Node();

node.AddChild(value, child);

stack.Push(new NodeCreator(

filteredDataFrom\_nc,

filteredColumnsFrom\_nc,

child

));

}

}

Else{

//greater than threshold

Node child = new Node();

node.AddChild(true.ToString(), child);

stack.Push(new NodeCreator(

filteredDataFrom\_nc,

filteredColumnsFrom\_nc,

child

));

//lesser or equal to threshold

Node child = new Node();

node.AddChild(false.ToString(), child);

stack.Push(new NodeCreator(

filteredDataFrom\_nc,

filteredColumnsFrom\_nc,

child

));

}

//filteredColumnsFrom\_nc

//contain only unbranching columns

//update column’s values after filtered

//(deep clone column first, then update values)

}

return tree;

}