Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Ижевский государственный технический университет имени М.Т. Калашникова»

Кафедра «Программное обеспечение»

Отчет

по лабораторной работе №6 по теме

«Кластеризация образцов методом COB-WEB»

по дисциплине

«Математические основы искусственного интеллекта»

|  |  |
| --- | --- |
| Выполнил:  студенты гр. Б06-191-1 | Мусин Р.И.  Чапаева А.С. |
| Принял: | Коробейников А.В. |

Ижевск 2018

СОДЕРЖАНИЕ ЗАДАНИЯ

Разработать программу упрощённой кластеризации заданных образцов.

ХОД РАБОТЫ

Программа получает на вход файл, который генерируется программой Weka. Из этого файла используются следующие строки: “Title”, “@attribute” (имя атрибута), “@data” (после этой строки задаются образцы - каждый в новой строке)

Функция CobWeb.eval(node: Node, obj: Instance) выбирает лучший вариант дальнейших действий с новым образцом. Доступные действия: добавление нового образца к ветке с самой высокой полезностью, создание новой ветки и добавление туда нового образца.

Когда дерево будет полностью создано, выводятся промежуточные расчёты и отрисовывается само дерево.

РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

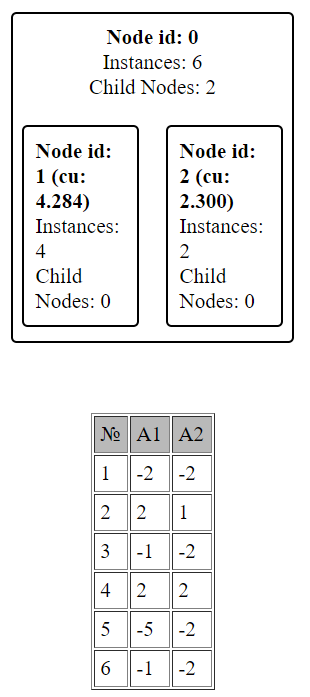


Рис. 1. Дерево кластеров

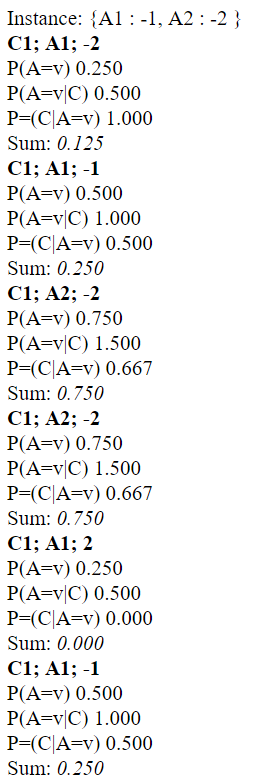


Рис. 2. Промежуточные результаты

ИСХОДНЫЙ КОД

export default class CobWeb {

static rootNode: Node;

static eval(node: Node, obj: Instance): Node {

if (isLeaf(node)) {

if (node.instances.length === 0) {

node.instances.push(obj);

} else { // добавление новых двух Node

const nodeLast = copyNode(node, false);

nodeLast.id = getLastNodeId(CobWeb.rootNode) + 1;

const nodeNew = new Node();

nodeNew.instances.push(obj);

nodeNew.id = nodeLast.id + 1;

node.child.push(nodeLast, nodeNew);

node.instances.push(obj);

}

} else if (isNode(node)) {

node.instances.push(obj);

const cloneNode = copyNode(node),

nodeChilds = cloneNode.child,

cu = new MethodOfCategory();

cu.bestCategory = CobWeb.S1(nodeChilds, obj);

cu.newCategory = CobWeb.S3(nodeChilds, obj, cloneNode.id);

cu.mergeCategory = CobWeb.S4(nodeChilds, obj);

cu.splitCategory = CobWeb.S5(nodeChilds, obj);

switch (bestMethodOfCat(cu)) {

case operations.currentCat: {

const selectedNodeId = cu.bestCategory.nodeId;

const bestNode = cloneNode.child.filter(c => c.id === selectedNodeId)[0];

bestNode.instances.push(obj);

bestNode.cu = cu.bestCategory.cu;

// CobWeb.eval(<Node>bestNode, obj);

break;

}

case operations.newCat: {

const newNode = new Node(),

lastId = getLastNodeId(node);

newNode.id = lastId + 1;

newNode.instances.push(obj);

newNode.cu = cu.newCategory.cu;

nodeChilds.push(newNode);

break;

}

case operations.mergeCat: {

const cloneChild2 = copyNode(cloneNode);

const newNode = new Node(),

fNode = <Node>cloneChild2.child.find(c => c.id === cu.mergeCategory.nodeId),

sNode = <Node>cloneChild2.child.find(c => c.id === cu.mergeCategory.nextNodeId);

cloneNode.child = cloneNode.child.filter(c => c.id !== fNode.id && c.id !== sNode.id);

++fNode.id;

++sNode.id;

newNode.id = cu.mergeCategory.nodeId;

newNode.child.push(<Node>fNode, <Node>sNode);

newNode.instances.push(...fNode.instances, ...sNode.instances);

const n = CobWeb.eval(newNode, obj);

cloneNode.child.push(newNode);

break;

}

case operations.splitCat: {

const cloneChild2 = copyNode(cloneNode);

const nodeForSplit = <Node>cloneNode.child.find(c => c.id === cu.splitCategory.nodeId);

cloneNode.child = cloneNode.child.filter(c => c.id !== cu.splitCategory.nodeId);

cloneNode.child.push(...nodeForSplit.child);

CobWeb.eval(cloneNode, obj);

break;

}

}

node = cloneNode;

}

return node;

}

static calcualteCU(selectedNode: Node, nodes: Array<Node | Leaf>): number {

let sum = 0;

for (let k = 0; k < nodes.length; k++) {

const nodeInstances = nodes[k].instances;

const attributes = nodeInstances[0].attributes;

for (let j = 0; j < attributes.length; j++) {

for (let i = 0; i < nodeInstances.length; i++) {

const vij = nodeInstances[i].attributes[j];

const c = calc(vij, nodes, selectedNode.id);

sum += c;

}

}

}

return sum;

}

static S1(nodeChilds: Array<Node | Leaf>, obj: Instance | null): BestWay {

const cloneChild = copyChild(nodeChilds),

cu = new Array<number>();

if (obj !== null) {

for (let i = 0; i < cloneChild.length; i++) { // добавление нового экземпляра для всех Node

cloneChild[i].instances.push(obj);

}

}

const copyObj = <Instance>obj;

for (let i = 0; i < cloneChild.length; i++) { // вычисление полезности веток

if (copyObj !== null) {

log.innerHTML += `<hr> Instance: ${

(copyObj.attributes.reduce(

(c, n, currentIndex, array) => {

return c += `${n.name} : ${n.value}${array.length - 1 === currentIndex ? '' : ','} `

}, '{') + '}')

}</br>`;

}

const res = CobWeb.calcualteCU(<Node>cloneChild[i], cloneChild);

log.innerHTML += `Result: ${res} </br>`;

cu.push(res);

}

const maxCu = Math.max(...cu),

nodeIdOfMaxCu = cloneChild[cu.indexOf(maxCu)].id;

return <BestWay>{

cu: maxCu,

nodeId: nodeIdOfMaxCu,

operation: operations.currentCat

};

}

static S3(nodeChilds: Array<Node | Leaf>, obj: Instance, nodeId: number): BestWay { // отнесение нового экземпляра к новой категории

const cloneChild = copyChild(nodeChilds),

newNode = new Node();

const id = Math.max(...cloneChild.map(c => c.id));

newNode.id = id + 1;

newNode.instances = copyInstances(new Array<Instance>(obj));

cloneChild.push(newNode);

const cu = CobWeb.S1(cloneChild, null);

if (cu.nodeId !== newNode.id) {

cu.cu = 0;

}

return <BestWay>{

cu: cu.cu,

nodeId: cu.nodeId,

operation: operations.newCat

};

}

function getLastNodeId(node: Node): number {

let lastNode = node.id;

node.child.forEach(c => {

if (isNode(<Node>c)) {

lastNode = getLastNodeId(<Node>c);

} else {

lastNode = c.id;

}

});

return lastNode;

}

function bestMethodOfCat(x: MethodOfCategory): operations {

const methods = new Array<any>();

methods.push([x.bestCategory.operation, x.bestCategory.cu]);

methods.push([x.newCategory.operation, x.newCategory.cu]);

methods.push([x.mergeCategory.operation, x.mergeCategory.cu]);

methods.push([x.splitCategory.operation, x.splitCategory.cu]);

const maxV = methods.sort((a, b) => {

return a[1] > b[1] ? 0 : 1;

});

return maxV[0][0];

}

function bestMethodOfCatCU(x: MethodOfCategory): number {

const methods = new Array<any>();

methods.push([x.bestCategory.operation, x.bestCategory.cu]);

methods.push([x.newCategory.operation, x.newCategory.cu]);

methods.push([x.mergeCategory.operation, x.mergeCategory.cu]);

methods.push([x.splitCategory.operation, x.splitCategory.cu]);

const maxV = methods.sort((a, b) => {

return a[1] > b[1] ? 0 : 1;

});

return maxV[0][1];

}

function calc(searchAttribute: Attribute, nodes: Array<Node | Leaf>, nodeId: number): number {

let pAV = 0,

pAVC = 0,

pCAV = 0;

let countAV1 = 0,

countAV2 = 0;

let countAVC1 = 0,

countAVC2 = nodes.filter((node: Node) => node.id === nodeId)[0].instances.length;

let countCVA1 = 0,

countCVA2 = 0;

for (let k = 0; k < nodes.length; ++k) {

const currentNode = <Node>nodes[k];

const nodeInstances = currentNode.instances;

countAV2 += nodeInstances.length;

for (let j = 0; j < nodeInstances.length; j++) {

for (let i = 0; i < nodeInstances[j].attributes.length; i++) {

const Attribute = nodeInstances[j].attributes[i];

if (Attribute.value === searchAttribute.value && Attribute.name === searchAttribute.name) {

if (currentNode.id === nodeId) {

countCVA1++;

}

countAVC1++;

countCVA2++;

countAV1++;

}

}

}

}

pAV = countAV1 / countAV2;

pAVC = countAVC1 / countAVC2;

pCAV = countCVA1 / countCVA2;

const result = +(pAV \* pAVC \* pCAV).toFixed(3);

log.innerHTML += `<b>C${nodeId}; ${searchAttribute.name}; ${searchAttribute.value}</b> </br>`;

log.innerHTML += `P(A=v) ${pAV.toFixed(3)} </br> P(A=v|C) ${pAVC.toFixed(3)} </br>

P=(C|A=v) ${pCAV.toFixed(3)} </br>`;

log.innerHTML += `Sum: <i>${result.toFixed(3)}</i> </br>`;

return result;

}

ВЫВОД

В ходе выполнения лабораторной работы была разработана программа, способная разделять на категории массив объектов. Данная программа не включается все варианты действий над категориями (разбиение и слияние), поэтому глубина дерева будет всегда равна единице.