# 《人工智能》实验三报告

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 孙潇桐 | 学号 | 2021117405 |
| 实验地点 | 院楼321机房 | 实验日期 | 2023-11-1 |

**一、实验内容**

Implement ID3 in a language of your choice and run it on the credit history example from  
the text。(自选一种语言实现ID3算法，数据库采用课本上预测个人财务风险的数据集)

**数据集如下：**





1. **实验原理**

我最近在学 rust，这次的实验使用rust完成。

将数据按照某个属性分开，选择能够让分类之后信息熵最小的分类方式，下面是我的实现代码：

在求出信息增益之后进行排序

for attr\_num in mask.iter() {

        entropy\_arr.push((entropy\_attr(attr\_num, attributes, &data), \*attr\_num));

    }

    // println!("{:?}",entropy\_arr);

    entropy\_arr.sort\_by(|(a, \_), (b, \_)| a.partial\_cmp(b).unwrap());

这里是求出信息熵的代码

fn entropy\_attr(attr\_num: &usize, attributes: &Vec<Attribute>, data: &Vec<DataRow>) -> f64 {

    let mut e = 0f64;

    let data\_num = data.len();

    match &attributes[\*attr\_num].values {

        AttrType::Strings(attrs) => {

            for attr\_val in attrs.iter() {

                let mut div: Vec<DataRow> = vec![];

                for row in data.iter() {

                    if let AttrVal::String(now) = &row[\*attr\_num] {

                        if attr\_val.eq(now) {

                            div.push(row.clone());

                        }

                    }

                }

                if div.is\_empty() {

                    continue;

                }

                e += (div.len() as f64 / data\_num as f64) \* entropy(attributes, &div);

            }

        }

        \_ => todo!("add continue value support"),

    };

    e

}

最后对树进行前序输出就好了：

fn show\_tree(node: &Node, layer: usize) {

    let spaces: String = std::iter::repeat('\t').take(layer).collect();

    match &node.node\_type {

        NodeType::Leaf(data) => {

            println!("{spaces}Leaf: {}", node.name);

            println!("{spaces}Data contains:");

            for row in data.iter() {

                println!("{spaces}{:?}", row);

            }

        }

        NodeType::Node(sons) => {

            println!("{spaces}Node: {}", node.name);

            for son in sons {

                show\_tree(son, layer + 1);

            }

        }

    }

}

在下面我给出完整代码：

use std::collections::BTreeSet;

use std::{

    fs::File,

    io::{BufRead, BufReader},

};

type DataRow = Vec<AttrVal>;

static THRESHOLD: f64 = 0.0;

enum NodeType {

    Node(Vec<Node>),

    Leaf(Vec<DataRow>),

}

struct Node {

    name: String,

    node\_type: NodeType,

}

fn entropy(attributes: &Vec<Attribute>, data: &Vec<DataRow>) -> f64 {

    let mut e = 0f64;

    let data\_num = data.len();

    match &attributes.last().unwrap().values {

        AttrType::Strings(attrs) => {

            let type\_num = attrs.len();

            let mut cnt: Vec<i32> = vec![0; type\_num];

            for row in data.iter() {

                if let AttrVal::String(row\_class) = row.last().unwrap() {

                    for (idx, now) in attrs.iter().enumerate() {

                        if now.eq(row\_class) {

                            cnt[idx] += 1;

                        }

                        break;

                    }

                }

            }

            for i in cnt.iter() {

                if \*i == 0 {

                    continue;

                }

                let pr = \*i as f64 / data\_num as f64;

                let lg = pr.log2();

                // e -= now \* now.log2();

                e -= pr \* lg;

            }

        }

        \_ => {

            todo!("add continue value support");

        }

    };

    e

}

fn entropy\_attr(attr\_num: &usize, attributes: &Vec<Attribute>, data: &Vec<DataRow>) -> f64 {

    let mut e = 0f64;

    let data\_num = data.len();

    match &attributes[\*attr\_num].values {

        AttrType::Strings(attrs) => {

            for attr\_val in attrs.iter() {

                let mut div: Vec<DataRow> = vec![];

                for row in data.iter() {

                    if let AttrVal::String(now) = &row[\*attr\_num] {

                        if attr\_val.eq(now) {

                            div.push(row.clone());

                        }

                    }

                }

                if div.is\_empty() {

                    continue;

                }

                e += (div.len() as f64 / data\_num as f64) \* entropy(attributes, &div);

            }

        }

        \_ => todo!("add continue value support"),

    };

    e

}

fn decision\_tree(

    data: Vec<DataRow>,

    attributes: &Vec<Attribute>,

    mut mask: BTreeSet<usize>,

    node: &mut Node,

) {

    let p\_0 = entropy(attributes, &data);

    if mask.is\_empty() || p\_0 < 1e-12 {

        if let NodeType::Leaf(arr) = &mut node.node\_type {

            \*arr = data;

        }

        return;

    }

    let mut entropy\_arr: Vec<(f64, usize)> = vec![];

    for attr\_num in mask.iter() {

        entropy\_arr.push((entropy\_attr(attr\_num, attributes, &data), \*attr\_num));

    }

    // println!("{:?}",entropy\_arr);

    entropy\_arr.sort\_by(|(a, \_), (b, \_)| a.partial\_cmp(b).unwrap());

    let (p\_g, best\_attr) = entropy\_arr.first().unwrap().clone();

    if p\_0 - p\_g < THRESHOLD {

        if let NodeType::Leaf(arr) = &mut node.node\_type {

            \*arr = data;

        }

        return;

    }

    mask.remove(&best\_attr);

    let attr\_name = &attributes[best\_attr].name;

    node.node\_type = NodeType::Node(vec![]);

    match &attributes[best\_attr].values {

        AttrType::Strings(attrs) => {

            for attr\_val in attrs {

                let mut distro\_data: Vec<DataRow> = vec![];

                for row in data.iter() {

                    if let AttrVal::String(row\_val) = &row[best\_attr] {

                        if attr\_val.eq(row\_val) {

                            distro\_data.push(row.clone());

                        }

                    }

                }

                if !distro\_data.is\_empty() {

                    let mut new\_node = Node {

                        name: format!("{attr\_name}=={attr\_val}").to\_string(),

                        node\_type: NodeType::Leaf(vec![]),

                    };

                    if let NodeType::Node(sons) = &mut node.node\_type {

                        decision\_tree(distro\_data, attributes, mask.clone(), &mut new\_node);

                        sons.push(new\_node);

                    }

                }

            }

        }

        \_ => {}

    }

}

fn show\_tree(node: &Node, layer: usize) {

    let spaces: String = std::iter::repeat('\t').take(layer).collect();

    match &node.node\_type {

        NodeType::Leaf(data) => {

            println!("{spaces}Leaf: {}", node.name);

            println!("{spaces}Data contains:");

            for row in data.iter() {

                println!("{spaces}{:?}", row);

            }

        }

        NodeType::Node(sons) => {

            println!("{spaces}Node: {}", node.name);

            for son in sons {

                show\_tree(son, layer + 1);

            }

        }

    }

}

fn main() {

    let (attributes, data) = read\_data("data.arff");

    let mask = show\_info(&attributes);

    let mut tree: Node = Node {

        name: String::from("All"),

        node\_type: NodeType::Leaf(vec![]),

    };

    decision\_tree(data, &attributes, mask, &mut tree);

    show\_tree(&tree, 0);

}

fn show\_info(attributes: &Vec<Attribute>) -> BTreeSet<usize> {

    println!("There are {} attributes:", attributes.len());

    for (idx, attr) in attributes.iter().enumerate() {

        println!("Attr{} :{}", idx, attr.to\_string());

    }

    println!("Please input the attribute index you want to remove, split by space(can be empty)");

    let mut buff = String::new();

    let \_ = std::io::stdin().read\_line(&mut buff);

    let mut mask: BTreeSet<usize> = (0..attributes.len() - 1).collect();

    if !buff.is\_empty() {

        buff.split\_whitespace().into\_iter().for\_each(|x| {

            mask.remove(&(x.parse().unwrap()));

        })

    }

    mask

}

fn read\_data(path: &str) -> (Vec<Attribute>, Vec<DataRow>) {

    let mut attributes: Vec<Attribute> = vec![];

    let mut data: Vec<DataRow> = vec![];

    let file = File::open(path).unwrap();

    let reader = BufReader::new(file);

    let mut reading\_data = false;

    for (idx, line) in reader.lines().enumerate() {

        let l = line.unwrap();

        if reading\_data {

            let vals = l.split(',');

            let mut row: DataRow = vec![];

            row.reserve(attributes.len());

            for (idx, str) in vals.enumerate() {

                match attributes[idx].values {

                    AttrType::Numbers => row.push(AttrVal::Number(str.parse().unwrap\_or(0))),

                    AttrType::Strings(\_) => row.push(AttrVal::String(str.to\_owned())),

                }

            }

            if row.len() != attributes.len() {

                panic!("Data error at line {}", idx);

            }

            data.push(row);

        } else {

            let strs: Vec<&str> = l.split(' ').collect();

            if strs[0] == "@attribute" {

                attributes.push(Attribute {

                    name: strs[1].to\_owned(),

                    values: if strs[2] == "numeric" {

                        AttrType::Numbers

                    } else {

                        AttrType::Strings(

                            strs[2][1..strs[2].len() - 1]

                                .split(',')

                                .map(|x| x.to\_owned())

                                .collect(),

                        )

                    },

                })

            } else if strs[0] == "@data" {

                reading\_data = true;

            }

        }

    }

    return (attributes, data);

}

#[derive(Debug)]

enum AttrVal {

    Number(i32),

    String(String),

}

impl Clone for AttrVal {

    fn clone(&self) -> Self {

        match self {

            AttrVal::Number(x) => AttrVal::Number(x.clone()),

            AttrVal::String(str) => AttrVal::String(str.clone()),

        }

    }

}

#[derive(Debug)]

enum AttrType {

    Numbers,

    Strings(Vec<String>),

}

#[derive(Debug)]

struct Attribute {

    name: String,

    values: AttrType,

}

impl ToString for Attribute {

    fn to\_string(&self) -> String {

        let vals = match &self.values {

            AttrType::Numbers => "number".to\_string(),

            AttrType::Strings(strs) => format!("{{{}}}", strs.join(",")),

        };

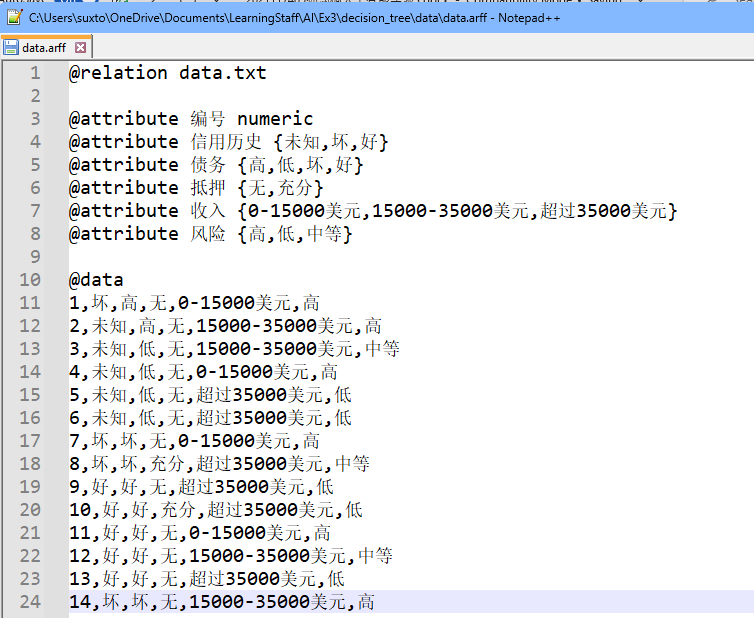
        format!("Name: {}, Vals: {}", self.name, vals)

    }

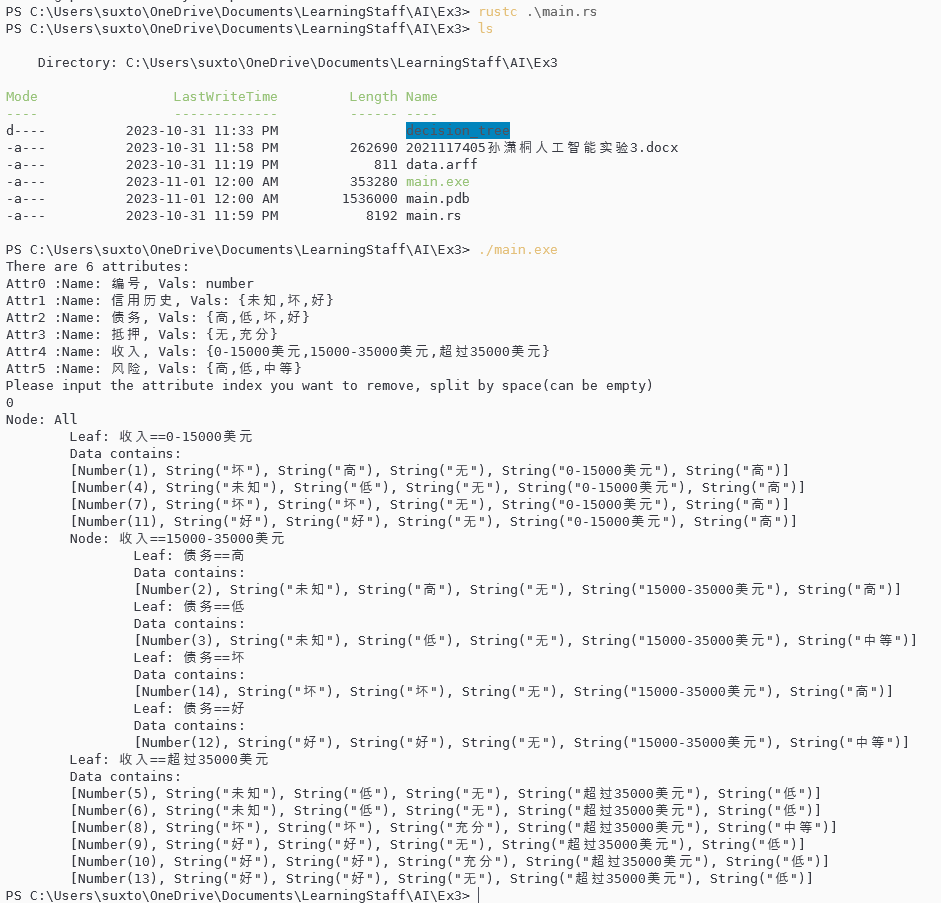
}

1. **实验过程以及结果分析**

**首先对数据转化成我最常进行分析的arff文件：**

****

**然后编译运行就好了，这样就得到ID3的结果**

****

1. **实验总结**

经过这个实验，我对ID3有了更加深刻的理解。之前我做项目用过ID3，但都是调用其他人写的库。这次我完全理解了决策树的运行方式，对人工智能有了更进一步的理解。