上海理工大学光电信息与计算机工程学院

**《人工智能原理与应用》实验报告**

****

**专　　业 智能科学与技术**

**姓 名　　 高浩琦**

**学　 号 2035060413**

**年　　级 2020级**

**指导教师 杨晶东**

**成 绩：**

**教师签字：**

目录

[**实验一 线性回归机器学习** 1](#_Toc74657770)

[**实验二 Logistic回归** 10](#_Toc74657771)

[**实验三 SVM** 19](#_Toc74657772)

[**实验四 神经网络学习** 29](#_Toc74657773)

**实验一 子句集9步法冲突消解**

一、实验目的和要求

理解和掌握消解原理、谓词公式化为子句集的九个步骤，实现任意谓词公式化成子句集。

二、实验内容

利用九步消解法，将谓词公式化为子句集形式。

三、实验要求

（1）采用C++、Python等语言编写出谓词公式消解演示程序。

（2）界面中可以通过实例按钮，由程序指定具体的实例，给出原始谓词公式；

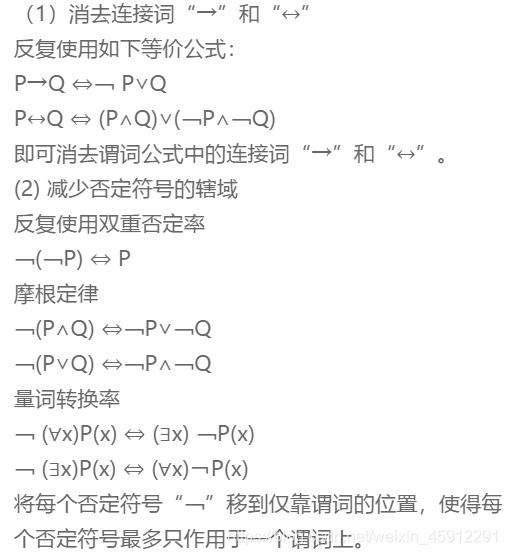
（3）设计九个步骤的按钮，每按一步按钮，给出这一步消解的结果。

四、实验过程

在谓词逻辑中，任何一个谓词公式都可以通过应用等价关系及推理规则化成相应的子句集。

用python实现以下6个步骤

其化简步骤如下:



(3)对变元标准化

在一个量词的辖域内，把谓词公式中受该量词约束的变元全部用另外一个没有出现过的任意变元代替，使不同量词约束的变元有不同的名字。

(4)化为前束范式

化为前束范式的方法:把所有量词都移到公式的左边，并且在移动时不能改变其相对顺序。

(5)消去存在量词

(6)化为Skolem标准形

对上述前束范式的母式应用以下等价关系

IMG_256

(7)消去全称量词

(8)消去合取词

在母式中消去所有合取词，把母式用子句集的形式表示出来。其中，子句集中的每一个元素都是一个子句。

(9)更换变量名称

对子句集中的某些变量重新命名，使任意两个子句中不出现相同的变量名。

实验环境：Dev—C++

#include <iostream>

#include <set>

#include <string.h>

#include <vector>

using namespace std;

typedef string Formula;

typedef vector<string> SubsentenceSet;

typedef string::iterator FormulaIter;

typedef string::reverse\_iterator FormulaRevIter;

// 公式符号定义

const char EQ = '#'; // 存在量词符号

const char UQ = '@'; // 全称量词符号

const char IMPLICATION = '>'; // 蕴含符号

const char NEGATION = '~'; // 否定符号

const char CONJUNCTION = '&'; // 合取符号

const char DISJUNCTION = '|'; // 析取符号

const char CONSTANT\_ALPHA[] = {'a', 'b', 'c', 'd', 'e', 'i', 'j', 'k'};

const char FUNC\_ALPHA[] = {'f', 'g', 'h', 'l', 'm', 'n', 'o'};

struct transform //一组置换

{

string t\_f1;

string t\_f2;

};

bool same(const string f1, const string f2);

transform dif(const string f1, const string f2);

string change(string f, transform q);

string change2(string f, transform q);

bool syncretism(const string f1, const string f2, vector<transform> &);

int legal(transform &);

bool var(const string s);

string varData(string s);

FormulaIter find(FormulaIter begin, FormulaIter end, const char c); //在正向字符串迭代器之间寻找字符c

FormulaRevIter findRev(FormulaRevIter begin, FormulaRevIter end, const char c); //在反向字符串迭代器之间寻找字符c

FormulaRevIter GetBeginOfFormula(FormulaRevIter begin, FormulaRevIter end); // 查找蕴含前件

FormulaIter FindFormula(FormulaIter begin, FormulaIter end); // 寻找左括号

FormulaIter FindPairChar(FormulaIter begin, FormulaIter end, char a, char b); //寻找与这个左括号匹配的右括号

bool IsConnector(char c); //判断c是不是连接词

FormulaIter FindQuantifier(FormulaIter begin, FormulaIter end); //寻找量词

char FindNewLowerAlpha(set<char> checkedAlpha); //寻找新的小写字母

char FindFuncAlpha(set<char> checkedAlpha); //寻找函数字母

void replace(FormulaIter begin, FormulaIter end, char varname, char newname); //新名字与旧名字替换

char GetNewConstantAlha(Formula f); //获取新的常量字母

void copy\_if(FormulaIter begin, FormulaIter end, set<char> &checkedAlpha); //防止字母重复的记录

Formula ReplaceAlphaWithString(Formula &f, char oldName, Formula funcFormula); //字符与字符串替换

FormulaRevIter leftfind(FormulaRevIter begin, FormulaRevIter end); //左寻括号

FormulaIter rightfind(FormulaIter begin, FormulaIter end); //右寻括号

Formula RemoveOuterBracket(Formula &f, FormulaIter begin); //去除外括号

Formula &RemoveImplication(Formula &f); //消去蕴含连接词

Formula &MoveNegation(Formula &f); //将否定符号移到紧靠谓词的位置

Formula &StandardizeValues(Formula &f); //适当改名使量词间不含同名指导变元，对变元标准化。

Formula &TransformToPNF(Formula &f); //化为前束范式

FormulaRevIter leftleft(FormulaRevIter begin, FormulaRevIter end); //查找存在量词的外括号

Formula &RemoveEQ(Formula &f); //消去存在量词。

Formula &RemoveUQ(Formula &f); //消去全称量词。

Formula &TransformToSkolem(Formula &f); //化为Skolem标准型。

void ExtractSubsentence(SubsentenceSet &subset, Formula &f); //消去合取词，以子句为元素组成一个集合S。

string pdrev(string s, bool &rev); //判断取反

vector<string>::reverse\_iterator findreviter(SubsentenceSet &subset, string \_goal, bool rev); //找到一个“反”式子

void count(SubsentenceSet subset); //输出子集合

void dosyncretism(string s, string ss, string &str); //合一

bool resolution(SubsentenceSet &subset); //归结

int main()

{

SubsentenceSet subset;

Formula f;

cout << "Input:" << endl;

cin >> f;

RemoveImplication(f); //消去蕴含

cout << "Elimination of implied symbols:" << endl;

cout << f << endl;

cout << "" << endl;

MoveNegation(f); //消去非

cout << "Move the negation sign to the front of each predicate:" << endl;

cout << f << endl;

cout << "" << endl;

StandardizeValues(f); //标准化

cout << "Standardization of variables:" << endl;

cout << f << endl;

cout << "" << endl;

RemoveEQ(f);

cout << "Eliminate existential quantifiers:" << endl;

cout << f << endl;

cout << "" << endl;

TransformToPNF(f); //化为前束

cout << "Convert to a front bundle:" << endl;

cout << f << endl;

cout << "" << endl;

RemoveUQ(f);

cout << "Omit universal quantifiers:" << endl;

cout << f << endl;

cout << "" << endl;

RemoveOuterBracket(f, f.begin());

TransformToSkolem(f);

cout << "Standardization of Skolem:" << endl;

cout << f << endl;

cout << "" << endl;

ExtractSubsentence(subset, f);

cout << "Eliminate the conjunction:" << endl;

for (vector<string>::iterator begin = subset.begin(); begin != subset.end(); begin++)

{ cout << \*begin << endl;}

cout << "" << endl;

if (resolution(subset))

{ cout << "Success" << endl;}

else

{cout << "Failure" << endl;}

}

bool syncretism(const string tf1, const string tf2, vector<transform> &mgu) //合一方法，判断是否可进行合一

{

string f1 = tf1, f2 = tf2;

while (!same(f1, f2)) //f1与f2中的符号不完全相同时才进入while循环

{ transform t = dif(f1, f2); //得到f1和f2的一个差异集，并把它赋给t

int flag = legal(t);

if (flag == 0)

return false;

else

{ mgu.push\_back(t);

f1 = change(f1, mgu.back());

f2 = change(f2, mgu.back());

cout << "after change:" << endl;

cout << "f1:" << f1 << endl;

cout << "f2:" << f2 << endl;

if (same(f1, f2))

return true; //f1和f2相同时就停止循环

}

}

return false;

}

bool same(const string f1, const string f2) //判断两个谓词f1和f2是否相同

{

if (f1.length() == f2.length())

{ int i = 0;

while (i < f1.length() && f1.at(i) == f2.at(i))

i++;

if (i == f1.length()) return true;

else

{return false;}

}

else return false;

}

transform dif(const string f1, const string f2) //求解f1和f2的差异集

{

int i = 0;

transform t;

while (f1.at(i) == f2.at(i)) //第i个相等时就转向比较i+1，直到遇到不相等时就跳出while循环

i++;

int j1 = i;

while (j1 < f1.length() - 1 && f1.at(j1) != ',') //从不相等的部分开始，直到遇到‘，’或到达结尾时跳出while循环

j1++;

if (j1 - i == 0) return t;

t.t\_f1 = f1.substr(i, j1 - i); //将f1中的不相同的项截取出来放入变量t.t\_f1中

int j2 = i;

while (j2 < f2.length() - 1 && f2.at(j2) != ',')

j2++;

if (j2 - i == 0) return t;

t.t\_f2 = f2.substr(i, j2 - i); //将f2中的不相同的项截取出来放入变量t.t\_f2中

while (t.t\_f1[j1 - i - 1] == t.t\_f2[j2 - i - 1]) //去除相同的部分

{ t.t\_f1.erase(j1 - 1 - i);

t.t\_f2.erase(j2 - i - 1);

j1--;

j2--;

}

return t;

}

int legal(transform &t) //判断置换t是否合法

{

if (t.t\_f1.length() == 0 || t.t\_f2.length() == 0) return 0;

if (var(t.t\_f2))

{ if (var(t.t\_f1) && (varData(t.t\_f1) == varData(t.t\_f2))) //不能代换合一

return 0;

else return 2;

}

if (!var(t.t\_f1)) //若t\_f1和t\_f2都不是变量，也不能合一

return 0;

string temp;

temp = t.t\_f1;

t.t\_f1 = t.t\_f2;

t.t\_f2 = temp; //在t\_f1是变量而t\_f2不是变量的情况下，交换t\_f1和t\_f2

return 1;

}

string varData(string s) //该函数是剥去外层括号

{

if (s.length() == 1 || s.length() == 0)

return s;

if (s.length() > 1)

{

int i = 0;

while (i < s.length() && s.at(i) != '(')

i++;

int j = i;

while (j < s.length() && s.at(j) != ')')

j++;

string ss = s.substr(i + 1, j - i - 1);

return varData(ss);

}

}

bool var(const string s)

{

if (s.length() == 0)

return false;

if (s.length() == 1 && s[0] >= 'A' && s[0] <= 'Z')

return false;

if (s.length() > 1)

{

int i = 0;

while (i < s.length() && s.at(i) != '(') //略过不是'('的字符

i++;

int j = i;

while (j < s.length() && s.at(j) != ')') //略过')'前的字符

j++;

string ss = s.substr(i + 1, j - i - 1); //取出'('和')'之间的串

return var(ss); //递归调用var

}

else

{

return true;

}

}

string change(string f, transform q) //该函数查找t\_f2在f中的位置并用t\_f1替代f中相应的t\_f2

{

int i = f.find(q.t\_f2);

while (i < f.length())

{

i = f.find(q.t\_f2);

if (i < f.length())

f = f.replace(i, q.t\_f2.length(), q.t\_f1);

}

return f;

}

FormulaIter find(FormulaIter begin, FormulaIter end, const char c) //在正向字符串迭代器之间寻找字符c

{

FormulaIter t = begin;

while (t < end)

{

if (\*t == c)

{

return t;

}

t++;

}

return end;

}

FormulaRevIter findRev(FormulaRevIter begin, FormulaRevIter end, const char c) //在反向字符串迭代器之间寻找字符c

{

FormulaRevIter t = begin;

while (t < end)

{

if (\*t == c)

{

return t;

}

t++;

}

return end;

}

FormulaRevIter GetBeginOfFormula(FormulaRevIter begin, FormulaRevIter end) // 查找蕴含前件

{

FormulaRevIter t = begin;

int num = 0;

while (t < end)

{

if (\*t == ')')

{

num--;

}

else if (\*t == '(')

{

num++;

}

if (num == 1)

{

return t;

}

t++;

}

return end;

}

FormulaIter FindFormula(FormulaIter begin, FormulaIter end) // 寻找左括号

{

FormulaIter t = begin;

while (t < end)

{

if (\*t == '(')

{

return t;

}

t++;

}

return end;

}

FormulaIter FindPairChar(FormulaIter begin, FormulaIter end, char a, char b) //寻找与这个左括号匹配的右括号

{

int num = 0;

FormulaIter t = begin;

while (t < end)

{

if (\*t == a)

{

num++;

}

else if (\*t == b)

{

num--;

}

if (num == 0)

{

return t;

}

t++;

}

return end;

}

bool IsConnector(char c) //判断c是不是连接词

{

if (c == CONJUNCTION || c == DISJUNCTION)

return true;

return false;

}

FormulaIter FindQuantifier(FormulaIter begin, FormulaIter end) //寻找量词

{

FormulaIter t = begin;

while (t < end)

{

if (\*t == '@' || \*t == '#')

return t;

t++;

}

return end;

}

char FindNewLowerAlpha(set<char> checkedAlpha) //寻找新的小写字母

{

char t = 'x';

int i = 0;

while (1)

{

if (checkedAlpha.find(char(t + i)) == checkedAlpha.end())

{

return 'x' + (i >= 3 ? (i - 10) : i);

}

i++;

}

}

char FindFuncAlpha(set<char> checkedAlpha) //寻找函数字母

{

for (int i = 0; i < 7; i++)

{

if (checkedAlpha.find(FUNC\_ALPHA[i]) == checkedAlpha.end())

{

return FUNC\_ALPHA[i];

}

}

return FUNC\_ALPHA[0];

}

void replace(FormulaIter begin, FormulaIter end, char varname, char newname) //新名字与旧名字替换

{

FormulaIter t = begin;

while (t < end)

{

if (\*t == varname)

{

\*t = newname;

}

t++;

}

}

char GetNewConstantAlha(Formula f) //获取新的常量字母

{

for (int i = 0; i < 8; i++)

{

if (f.find(CONSTANT\_ALPHA[i]))

{

}

else

{

return CONSTANT\_ALPHA[i];

}

}

return CONSTANT\_ALPHA[0];

}

void copy\_if(FormulaIter begin, FormulaIter end, set<char> &checkedAlpha) //防止字母重复的记录

{

FormulaIter t = begin;

while (t < end)

{

if (checkedAlpha.find(\*t) == checkedAlpha.end())

{

checkedAlpha.insert(\*t);

}

t++;

}

}

Formula ReplaceAlphaWithString(Formula &f, char oldName, Formula funcFormula) //字符与字符串替换

{

FormulaIter begin = f.begin();

FormulaIter end = f.end();

for (int i = 0; begin < end; begin++, i++)

{

if (\*begin == oldName)

{

f.replace(i, 1, funcFormula);

}

}

return f;

}

FormulaRevIter leftfind(FormulaRevIter begin, FormulaRevIter end) //左寻括号

{

int num = 0;

FormulaRevIter t = begin;

while (t < end)

{

if (\*t == ')')

{

num--;

}

else if (\*t == '(')

{

num++;

}

else if (\*t == '|')

{

return end;

}

if (num == 1)

{

return t;

}

t++;

}

return end;

}

FormulaIter rightfind(FormulaIter begin, FormulaIter end) //右寻括号

{

int num = 0;

FormulaIter t = begin;

while (t < end)

{

if (\*t == '(')

{

num--;

}

else if (\*t == ')')

{

num++;

}

else if (\*t == '|')

{

return end;

}

if (num == 1)

{

return t;

}

t++;

}

return end;

}

Formula RemoveOuterBracket(Formula &f, FormulaIter begin) //去除外括号

{

FormulaIter con = find(begin, f.end(), CONJUNCTION);

if (con != f.end())

{

FormulaRevIter left = leftfind(FormulaRevIter(con), f.rend());

FormulaIter right = rightfind(con, f.end());

FormulaIter leftt = left.base() - 1;

if (left != f.rend() && right != f.end())

{

f.erase(leftt);

f.erase(right - 1);

}

return RemoveOuterBracket(f, con + 1);

}

else

{return f;}

}

//消去蕴含连接词

Formula &RemoveImplication(Formula &f)

{

FormulaIter iter;

while ((iter = find(f.begin(), f.end(), IMPLICATION)) != f.end()) //寻找蕴含符号

{

\*iter = DISJUNCTION; // 将蕴含符号替换为析取符号

FormulaRevIter revIter(iter);

revIter = GetBeginOfFormula(revIter, f.rend()); // 查找蕴含前件

iter = revIter.base(); // 反向迭代器到正向迭代器

f.insert(iter, NEGATION); // 在前件前面插入否定

}

return f;

}

//将否定符号移到紧靠谓词的位置

Formula &MoveNegation(Formula &f)

{

FormulaIter iter = find(f.begin(), f.end(), NEGATION);

while (iter != f.end())

{

if (\*(iter + 1) == '(')

{ // 否定不是直接修饰谓词公式，需要内移

// 否定符号修饰着带量词的谓词公式

if (\*(iter + 2) == EQ || \*(iter + 2) == UQ)

{

// 量词取反

\*(iter + 2) == EQ ? \*(iter + 2) = UQ : \*(iter + 2) = EQ;

string leftDonePart(f.begin(), iter + 5);

// 移除否定符号

leftDonePart.erase(find(leftDonePart.begin(), leftDonePart.end(), NEGATION));

string rightPart(iter + 5, f.end());

// 否定内移

rightPart.insert(rightPart.begin(), NEGATION);

// 递归处理右部分

MoveNegation(rightPart);

string(leftDonePart + rightPart).swap(f);

return f;

}

else

{ // 修饰着多个公式，形如~(P(x)|Q(x))

iter = f.insert(iter + 2, NEGATION); // 内移否定符号

while (1)

{

iter = FindFormula(iter, f.end());

//assert(iter != f.end() && "No Predicate Formula!");

FormulaIter iter2 = FindPairChar(

iter, f.end(), '(', ')');

++iter2;

if (IsConnector(\*iter2))

{

\*iter2 == DISJUNCTION ? \*iter2 = CONJUNCTION

: \*iter2 = DISJUNCTION;

iter = f.insert(iter2 + 1, NEGATION);

}

else

break;

}

f.erase(find(f.begin(), f.end(),

NEGATION)); // 清除原否定符号

return MoveNegation(f);

}

}

else if (\*(iter + 1) == NEGATION)

{ // 两个否定，直接相消

f.erase(iter, iter + 2);

return MoveNegation(f); // 重新处理

}

else

{

string leftDonePart(f.begin(), iter + 1);

string rightPart(iter + 1, f.end());

MoveNegation(rightPart);

string(leftDonePart + rightPart).swap(f);

return f;

//iter = find(iter + 1, f.end(), NEGATION);

}

}

return f;

}

//适当改名使量词间不含同名指导变元，对变元标准化。

Formula &StandardizeValues(Formula &f)

{

set<char> checkedAlpha;

FormulaIter iter = FindQuantifier(f.begin(), f.end());

while (iter != f.end())

{

char varName = \*++iter; // 获取变量名

if (checkedAlpha.find(varName) == checkedAlpha.end())

{

checkedAlpha.insert(varName);

}

else

{ // 变量名冲突了，需要改名

// 获取新名子

char newName = FindNewLowerAlpha(checkedAlpha);

checkedAlpha.insert(newName);

// 查找替换右边界

FormulaIter rightBorder = FindPairChar(

iter + 2, f.end(), '(', ')');

// 将冲突变量名替换为新的名子

\*iter = newName;

replace(iter, rightBorder, varName, newName);

iter = rightBorder; // 移动到新的开始

}

iter = FindQuantifier(iter, f.end());

}

return f;

}

//化为前束范式。

Formula &TransformToPNF(Formula &f)

{

FormulaIter iter = FindQuantifier(f.begin(), f.end());

if (iter == f.end())

return f;

else if (iter - 1 == f.begin())

{ // 量词已经在最前面

iter += 3;

string leftPart(f.begin(), iter);

string rightPart(iter, f.end());

TransformToPNF(rightPart); // 递归处理右部分

(leftPart + rightPart).swap(f);

}

else

{ // 量词在内部，需要提到前面

string quantf(iter - 1, iter + 3); // 保存量词

f.erase(iter - 1, iter + 3); // 移除量词

f.insert(f.begin(), quantf.begin(), quantf.end());

return TransformToPNF(f); // 继续处理

}

return f;

}

FormulaRevIter leftleft(FormulaRevIter begin, FormulaRevIter end) //查找存在量词的外括号

{

int num = 0;

FormulaRevIter t = begin + 2;

while (t < end)

{

if (\*t == ')')

{num--;}

else if (\*t == '(')

{num++;}

if (num == 1)

{return t;}

t++;

}

return end;

}

//消去存在量词。

Formula &RemoveEQ(Formula &f)

{

set<char> checkedAlpha;

FormulaIter eqIter = find(f.begin(), f.end(), EQ);

if (eqIter == f.end())

return f;

FormulaRevIter left = leftleft(FormulaRevIter(eqIter), f.rend());

FormulaRevIter uqIter = findRev(left, f.rend(), UQ);

if (uqIter == f.rend())

{ // 该存在量词前没有任意量词

char varName = \*(eqIter + 1);

char newName = GetNewConstantAlha(f);

auto rightBound = FindPairChar(eqIter + 3, f.end(), '(', ')');

//assert(rightBound != f.end());

replace(eqIter + 3, rightBound, varName, newName); // 常量化

f.erase(eqIter - 1, eqIter + 3); // 移除存在量词

}

else

{

// 记录公式中已经存在的字母

copy\_if(f.begin(), f.end(), checkedAlpha);

const char oldName = \*(eqIter + 1);

// 准备任意量词的函数来替换该存在量词

const char funcName = FindFuncAlpha(checkedAlpha);

string funcFormula;

funcFormula = funcFormula + funcName + '(' + \*(uqIter - 1) + ')';

f.erase(eqIter - 1, eqIter + 3); // 移除存在量词

ReplaceAlphaWithString(f, oldName, funcFormula);

}

//RemoveOuterBracket(f, f.begin());

return RemoveEQ(f); // 递归处理

}

//消去全称量词。

Formula &RemoveUQ(Formula &f)

{

FormulaIter uqIter = find(f.begin(), f.end(), UQ);

while (uqIter != f.end())

{

uqIter = f.erase(uqIter - 1, uqIter + 3); // 直接移除全称量词

uqIter = find(uqIter, f.end(), UQ); // 继续扫描

}

//RemoveOuterBracket(f, f.begin());

return f;

}

//化为Skolem标准型。

Formula &TransformToSkolem(Formula &f)

{

FormulaIter dis = find(f.begin(), f.end(), DISJUNCTION);

FormulaRevIter left = leftfind((FormulaRevIter)dis, f.rend());

FormulaIter leftt = left.base() - 1;

FormulaIter right = rightfind(dis + 1, f.end());

FormulaRevIter leftcon = findRev((FormulaRevIter)dis, left, CONJUNCTION);

FormulaIter lefttcon = leftcon.base() - 1;

FormulaIter rightcon = find(dis, right, CONJUNCTION);

if (leftt != lefttcon && rightcon == right)

{

cout << string(dis + 1, right) << endl;

cout << string(leftt + 1, lefttcon) << endl;

cout << string(lefttcon + 1, dis) << endl;

string temp = "(" + string(dis + 1, right) + "|" + string(leftt + 1, lefttcon) + ")&(" + string(dis + 1, right) + "|" + string(lefttcon + 1, dis - 1) + ")";

// cout << temp << endl;

f.replace(leftt, right, temp);

}

else if (leftt == lefttcon && right != rightcon)

{

// cout << string(leftt + 1, dis) << endl;

// cout << string(dis + 1, rightcon) << endl;

// cout << string(rightcon + 1, right - 1) << endl;

string temp = "(" + string(leftt + 1, dis) + "|" + string(dis + 1, rightcon) + ")&(" + string(leftt + 1, dis) + "|" + string(rightcon + 1, right - 1) + ")";

// cout << temp << endl;

f.replace(leftt, right, temp);

}

else if (leftt != lefttcon && right != rightcon)

{

string str1 = string(leftt + 1, lefttcon);

string str2 = string(lefttcon + 1, dis);

string str3 = string(dis + 1, rightcon);

string str4 = string(rightcon + 1, right);

cout << str1 << endl;

cout << str2 << endl;

cout << str3 << endl;

cout << str4 << endl;

string temp;

if (str1 == str3)

{

temp = str1 + "&(" + str2 + "|" + str4 + ")";

}

else if (str1 == str4)

{

temp = str1 + "&(" + str2 + "|" + str3 + ")";

}

else if (str2 == str3)

{

temp = str2 + "&(" + str1 + "|" + str4 + ")";

}

else if (str2 == str4)

{

temp = str2 + "&(" + str1 + "|" + str3 + ")";

}

f.replace(leftt, right + 1, temp);

}

return f;

}

//消去合取词，以子句为元素组成一个集合S。

void ExtractSubsentence(SubsentenceSet &subset, Formula &f)

{

FormulaIter leftIter = f.begin();

FormulaIter middleIter = find(f.begin(), f.end(), CONJUNCTION);

while (middleIter != f.end())

{

if (\*leftIter == '(' && \*(middleIter - 1) == ')')

subset.push\_back(string(leftIter + 1, middleIter - 1));

else

subset.push\_back(string(leftIter, middleIter));

leftIter = middleIter + 1;

middleIter = find(middleIter + 1, f.end(), CONJUNCTION);

}

if (\*leftIter == '(' && \*(middleIter - 1) == ')')

subset.push\_back(string(leftIter + 1, middleIter - 1));

else

subset.push\_back(string(leftIter, middleIter));

}

string pdrev(string s, bool &rev) //判断取反

{

if (s[0] == '~')

{

rev = true;

return s.erase(0, 1);

}

else

{

rev = false;

return "~" + s;

}

}

vector<string>::reverse\_iterator findreviter(SubsentenceSet &subset, string \_goal, bool rev) //找到一个“反”式子

{

char c = rev ? \_goal[0] : \_goal[1];

for (vector<string>::reverse\_iterator rbegin = subset.rbegin(); rbegin != subset.rend(); rbegin++)

{

string temp = (\*rbegin);

for (int i = 0; i < temp.length(); i++)

{

FormulaIter iter = find(temp.begin(), temp.end(), c);

if (iter != temp.end())

{

if (rev)

{

if (iter == temp.begin())

return rbegin;

else if (\*(--iter) != NEGATION)

{return rbegin;}

}

else

{

if (\*(--iter) == NEGATION)

{return rbegin;}

}

}

}

}

return subset.rend();

}

void count(SubsentenceSet subset) //输出子集合

{

for (vector<string>::iterator begin = subset.begin(); begin != subset.end(); begin++)

{cout << \*begin << endl;}

}

void dosyncretism(string s, string ss, string &str) //合一

{

vector<transform> mgu;

if (syncretism(s, ss, mgu)) //存在最一般合一，并输出结果

{

cout << "mgu={ ";

int i = 0;

for (i = 0; i < mgu.size() - 1; i++)

cout << mgu[i].t\_f1 << "/" << mgu[i].t\_f2 << ", ";

cout << mgu[i].t\_f1 << "/" << mgu[i].t\_f2 << " }" << endl;

for (i = 0; i < mgu.size(); i++)

{

int j = 0;

for (FormulaIter begin = str.begin(); begin < str.end(); begin++, j++)

{

// if (mgu[i].t\_f2[0] == \*begin) //这里应该是对字符串的判断、删除、插入，待修改

// {

// \*begin = mgu[i].t\_f1[0];

// }

if (mgu[i].t\_f2[0] == \*begin)

{

str.erase(begin, begin + (mgu[i].t\_f2).length());

str.insert(j, mgu[i].t\_f1);

}

}

}

}

else //不存在最一般合一

{cout << "cannot be syncretized" << endl;}

}

bool resolution(SubsentenceSet &subset) //归结

{

for (vector<string>::reverse\_iterator rbegin = subset.rbegin(); rbegin != subset.rend();) //从尾部，即目标公式开始归结

{

bool f = false;

bool rev = false; //是否取反

string s = \*rbegin;

string ss;

FormulaIter left = s.begin();

FormulaIter middle = find(left, s.end(), DISJUNCTION);

string goal = string(left, middle); //归结元素

string \_goal = pdrev(goal, rev); //归结元素的逆

vector<string>::reverse\_iterator iter;

while (middle != s.end()) //寻找归结元素的逆，找到末尾结束

{

iter = findreviter(subset, \_goal, rev);

if (iter == subset.rend()) //找不到继续循环

{

left = middle + 1;

middle = find(left, s.end(), DISJUNCTION);

goal = string(left, middle);

\_goal = pdrev(goal, rev);

}

else

{

f = true; //找到标记

break;

}

}

if (!f) //判断是否找到，前面最后一个left和middle没有处理，这里继续处理。

{

iter = findreviter(subset, \_goal, rev);

if (iter == subset.rend())

{ rbegin++;

continue;}

}

ss = (\*iter);

char cc = goal[0];

int len = goal.length();

if (rev)

{ cc = goal[1];

len -= 1;

}

dosyncretism(s.substr(s.find(cc), len), ss.substr(ss.find(cc), len), \*iter); //合一

//擦除归结了的式子

(\*rbegin).erase((\*rbegin).find(goal), goal.length());

string c = "";

if (rev)

{c += \_goal[0];}

else

{ c += \_goal[0] + \_goal[1];}

(\*iter).erase((\*iter).find(c), \_goal.length());

//合并剩余式子

string newstr = (\*rbegin) + "|" + (\*iter);

//擦除多余析取符

for (int i = 0; i < newstr.length(); i++)

{

if ((i == 0 || i == newstr.length() - 1) && newstr[i] == DISJUNCTION)

{newstr.erase(i, 1);}

else if (newstr[i] == DISJUNCTION && newstr[i + 1] == DISJUNCTION)

{ newstr.erase(i + 1, 1);}

}

cout << endl;

//删除旧的式子，添加新的式子。

subset.erase(subset.rbegin().base());

subset.erase(iter.base() - 1);

subset.push\_back(newstr);

rbegin = subset.rbegin();

count(subset);

//归结出空，则结论为真，归结结束。

if (newstr == "")

{return true;}

}

return false;

}

五、实验结果

Input:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Elimination of implied symbols:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Move the negation sign to the front of each predicate:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Standardization of variables:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Eliminate existential quantifiers:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Convert to a front bundle:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Omit universal quantifiers:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Standardization of Skolem:

(~P(x)|Q(x))&(~P(y)|R(y))&S(a)&(~S(z)|~R(z))&P(a)

Eliminate the conjunction:

~P(x)|Q(x)

~P(y)|R(y)

S(a)

~S(z)|~R(z)

P(a)

after change:

f1:P(a)

f2:P(a)

mgu={ a/y }

~P(x)|Q(x)

S(a)

~S(z)|~R(z)

R(a)

after change:

f1:R(a)

f2:R(a)

mgu={ a/z }

~P(x)|Q(x)

S(a)

~S(a)

cannot be syncretized

~P(x)|Q(x)

Success

**实验二 基于A\*算法实现野人和修道士过河**

一、实验目的

掌握启发式搜索A\*算法及其可采纳性，求解在代价最小的情况下将MC问题的初始状态转为另终止状态的路径。

二、实验内容

在河的左岸有三个修道士，三个野人和一条船，修道士想用这条船把所有人都运到河对岸,但受到以下条件限制：修道士、野人都会划船，但船一次只能装运两个人。在任何岸边野人数不能超过修道士，否则修道士会被野人吃掉。

三、实验要求

（1）采用Python、C++等算法实现MC问题求解，用三元组表示渡河过程中的状态，并用箭头表明这些状态间的迁移。

（2）如问题有解，则输出一个最佳方案。若无解，则给出“渡河失败”信息。

四、实验过程

1、A算法的基本原理分析：

在或图的一般搜索算法中，如果在搜索过程的步骤⑦利用估价函数f(n)=g(n)+h(n)对open表中的节点进行排序，则该搜索算法为A算法。

g(n)：从初始节点到n的实际代价。

因为n为当前节点，搜索已达到n点，所以g(n)可计算出。

h(n)：启发函数，从n到目标节点的最佳路径的估计代价。

因为尚未找到解路径，所以h(n)仅仅是估计值。

对A算法中的g(n)和h(n)做出限制：

g(n) >= g(n)（g\*(n)为S0到n的最小费用）

-h(n) <= h\*(n)（h\*(n)为n到Sg的实际最小费用）

则算法被称为A\*算法。

2、传教士—野人过河问题的知识表示方法分析：

在这个问题中，需要考虑：

1、两岸的传教士人数和野人人数 2、船在左岸还是在右岸

已知：传教士和野人数：N（两者默认相同），船的最大容量：K

定义：M：左岸传教士人数 C：左岸野人人数 B：左岸船个数

可用一个三元组来表示左岸状态，即S=(M, C, B)。

约束条件：M>=0,C>=0,B=1或0

已知左岸状态，右岸的状态为：

右岸传教士人数：M’=N-M

右岸野人人数：C’=N-C

右岸船数：B’=1-B

满足同样的约束条件

实验算法：

def GJ(this,k):#估价函数计算 h(n) = M + C - K \* B

return this[0] + this[1] - k \* this[2]

def creat(array,M,C,B,N):#判断生成节点是否符合规则、判断是否重复

P = array[:]

if M == N :#左岸传教士数量等于总数

if C >=0 and C <= N :

P.insert(0,[M,C,1-B])

for i in open:

if P[0] == i[0]:

return False

for i in closed:

if P[0] == i[0]:

return False

open.append(P)

return True

else:

return False

elif M > 0 :#左岸传教士数量基于0到N之间时

if C >=0 and M >= C and M <= N and C <= N and N-M >= N-C:

P.insert(0,[M,C,1-B])

for i in open:

if P[0] == i[0]:

return False

for i in closed:

if P[0] == i[0]:

return False

open.append(P)

return True

else:

return False

elif M == 0:#左岸传教士为0

if C >= 0 and C <= N:

P.insert(0, [M, C, 1 - B])

for i in open:

if P[0] == i[0]:

return False

for i in closed:

if P[0] == i[0]:

return False

open.append(P)

return True

else:

return False

else:

return False

if \_\_name\_\_ == '\_\_main\_\_':

N = int(input("传教士和野人的人数（默认相同）："))

K =int(input("船的最大容量："))

open = [] #创建open表

closed = [] #创建closed表

sample = [N,N,1] #初始状态

goal = [0,0,0]#目标状态

open.append([sample])

creatpoint = searchpoint = 0

while(1):

if sample == goal:

print("初始状态为目标状态！")

break

if len(open) == 0:

print("未搜索到解！")

break

else:

this = open.pop(0)

closed.append(this)

if this[0] == goal:

print("搜索成功！")

print('共生成节点数：{}，共搜索节点数:{}'.format(creatpoint,searchpoint + 1))

print('过河方案如下：')

print(' [M, C, B]')

for i in this[::-1]:

print('---->',i)

exit()

#扩展节点

searchpoint += 1

if this[0][2] == 1 :#船在左岸时

for i in range(1,K+1):#只

if creat(this,this[0][0]-i,this[0][1],this[0][2],N):

creatpoint += 1

for i in range(1,K+1):

if creat(this,this[0][0],this[0][1]-i,this[0][2],N):

creatpoint += 1

for i in range(1,K):

for r in range(1,K-i+1):

if creat(this,this[0][0] - i,this[0][1] - r, this[0][2],N):

creatpoint += 1

else:#船在右岸时

for i in range(1,K+1):

if creat(this,this[0][0]+i,this[0][1],this[0][2],N):

creatpoint += 1

for i in range(1,K+1):

if creat(this,this[0][0],this[0][1]+ i,this[0][2],N):

creatpoint += 1

for i in range(1,K):

for r in range(1,K-i+1):

if creat(this,this[0][0] + i,this[0][1] + r, this[0][2],N):

creatpoint += 1

#计算估计函数h(n) = M + C - K \* B 重排open表

for x in range(0,len(open)-1):

m = x

for y in range(x+1,len(open)):

if GJ(open[x][0],K) > GJ(open[y][0],K):

m = y

if m != x:

open[x],open[m] = open[m],open[x]

**实验三 基于机器学习算法中医声诊样本分类**

一、实验目的

采用机器学习算法，对中医临床声诊样本分析，实现疑似冠心病患者的声诊样本三分类。(Knn,Svm, Bayes, RF,Xgboost,LR)

二、实验内容

通过对各种机器学习算法的函数调用，实现疑似冠心病患者（高、中、低）危险等级自主分类。

三、实验要求

（1）熟练掌握Python Sklearn安装包、各种机器学习算法的调用方法

（2）输出测试集精度(Precision, Recall,F1score,AUC(ROC面积),mAP(PR面积)), loss, ROC曲线(Recall,1-特异度),PR曲线(Precision, Recall)。

四、实验过程

常用的分类：线性、决策树、SVM、KNN，朴素贝叶斯；

import pandas as pd

# %%

high = pd.read\_excel("声诊练习.xlsx",sheet\_name= '汇总   高危   声 ')

medi = pd.read\_excel("声诊练习.xlsx",sheet\_name= '汇总  中危组  声  ')

low = pd.read\_excel("声诊练习.xlsx",sheet\_name= '低危  声  总表  ')

# %%

data = pd.concat([high,medi,low],axis= 0)

# %%

data\_y = data.iloc[:,-1]

data\_x = data.iloc[:,:-1]

# %%

from sklearn.model\_selection import train\_test\_split

# %%

xtrain,xtest,ytrain,ytest = train\_test\_split(data\_x,data\_y,test\_size = 0.2 ,random\_state = 1 )

KNN

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()

clf = knn.fit(xtrain,ytrain)

clf.score(xtest,ytest)

SVM

from sklearn.svm import SVC

svc = SVC(probability= True)

clf = svc.fit(xtrain,ytrain)

clf.score(xtest,ytest)

Bayes

from sklearn.naive\_bayes import GaussianNB

gnb = GaussianNB()

clf = gnb.fit(xtrain,ytrain)

clf.score(xtest,ytest)

决策树

from sklearn.tree import DecisionTreeClassifier

dtc = DecisionTreeClassifier()

clf = dtc.fit(xtrain,ytrain)

clf.score(xtest,ytest)

# %%

from sklearn.metrics import confusion\_matrix as CM

# 查看预测结果的概率\

y\_pred = clf.predict(xtest)

proba = clf.predict\_proba(xtest)

print(proba.shape) # 返回的是每个样本，对应每个分类的概率

print(proba.sum(1)) # 每一行的概率之和都是1

# 查看混淆矩阵

print(CM(ytest, y\_pred))

# %%

from sklearn import metrics  # 模型评价

y\_prob = clf.predict\_proba(xtest)[:,1]

confusion\_matrix=metrics.confusion\_matrix(ytest,y\_pred)

print(confusion\_matrix)

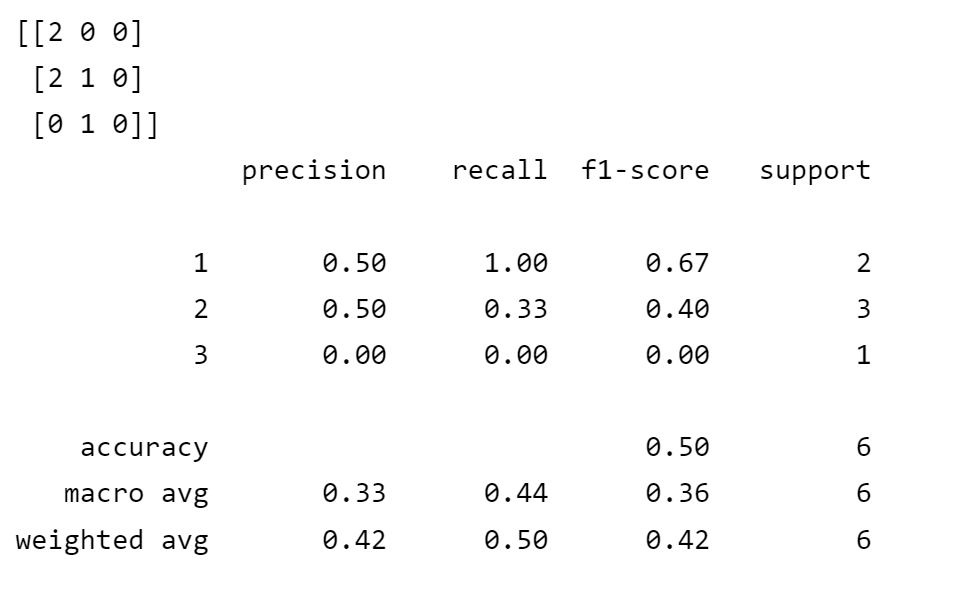
auc\_roc=metrics.classification\_report(ytest,y\_pred)

print(auc\_roc)

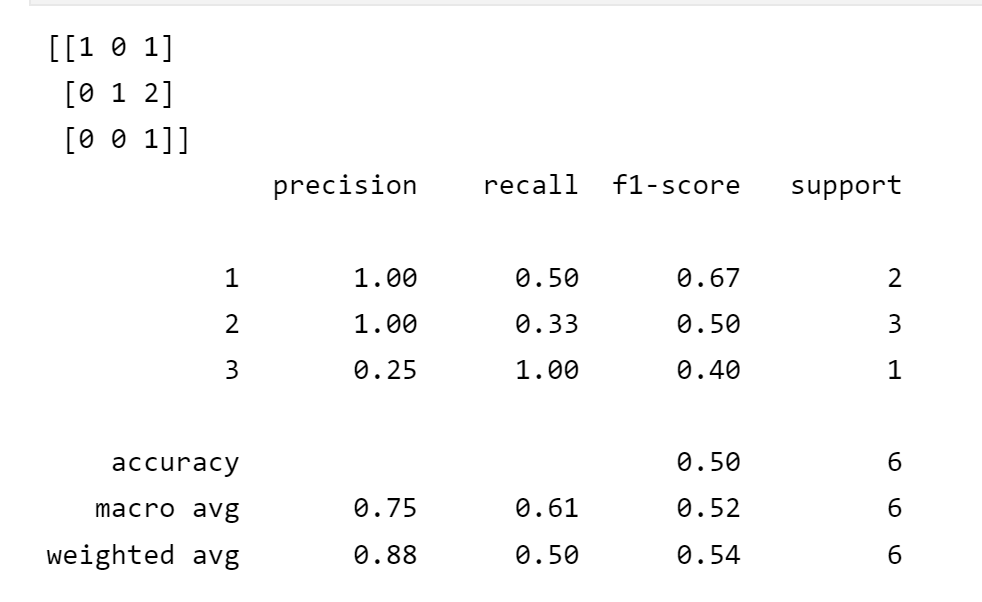
五、实验结果

混淆矩阵、测试集精度

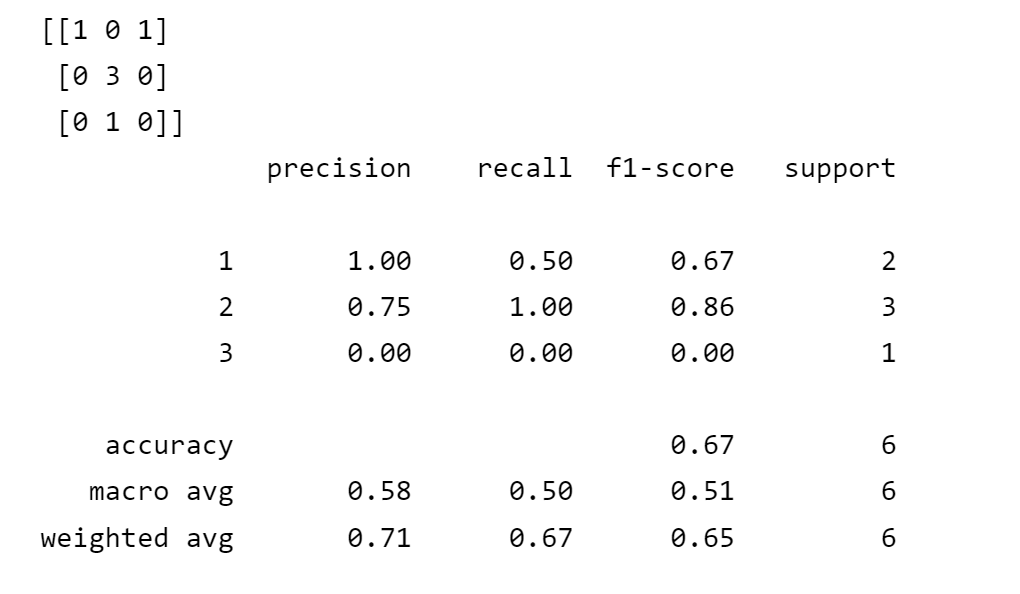
KNN（0.5）



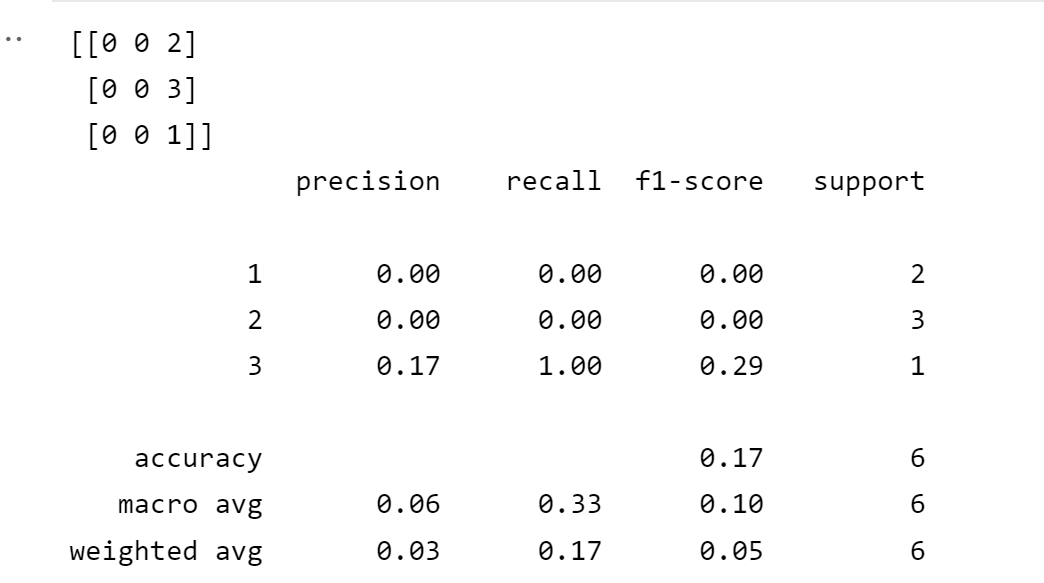
决策树（0.5）



Bayes（0.66666）



SVM（0.16666）



**实验四 基于深度学习算法的轴承故障自主分类**

**一、实验目的**

熟练掌握深度学习算法及应用，深入理解监督学习过程。

**二、实验内容**

采用基于CNN、RNN等最新深度学习算法，实现对具有序列特性的轴承故障样本的自主分类。

**三、实验要求**

（1）利用Python sklearn安装包，调用CNN、RNN算法，对轴承故障样本实现自主分类。

（2）输出测试集精度(Precision,Recall,F1score,AUC(ROC面积),mAP(PR面积)), loss, ROC曲线(Recall,1-特异度),PR曲线(Precision, Recall)

**四、实验过程**

import keras

from scipy.io import loadmat

import matplotlib.pyplot as plt

import glob

import numpy as np

import pandas as pd

import math

import os

from keras.layers import \*

from keras.models import \*

from keras.optimizers import \*

import numpy as np

MANIFEST\_DIR = "Bear\_data/train.csv"

Batch\_size = 20

Long = 792

Lens = 640

#把标签转成oneHot

def convert2oneHot(index,Lens):

    hot = np.zeros((Lens,))

    hot[int(index)] = 1

    return(hot)

def xs\_gen(path=MANIFEST\_DIR,batch\_size = Batch\_size,train=True,Lens=Lens):

    img\_list = pd.read\_csv(path)

    if train:

        img\_list = np.array(img\_list)[:Lens]

        print("Found %s train items."%len(img\_list))

        print("list 1 is",img\_list[0,-1])

        steps = math.ceil(len(img\_list) / batch\_size)    # 确定每轮有多少个batch

    else:

        img\_list = np.array(img\_list)[Lens:]

        print("Found %s test items."%len(img\_list))

        print("list 1 is",img\_list[0,-1])

        steps = math.ceil(len(img\_list) / batch\_size)    # 确定每轮有多少个batch

    while True:

        for i in range(steps):

            batch\_list = img\_list[i \* batch\_size : i \* batch\_size + batch\_size]

            np.random.shuffle(batch\_list)

            batch\_x = np.array([file for file in batch\_list[:,1:-1]])

            batch\_y = np.array([convert2oneHot(label,10) for label in batch\_list[:,-1]])

            yield batch\_x, batch\_y

TEST\_MANIFEST\_DIR = "Bear\_data/test\_data.csv"

def ts\_gen(path=TEST\_MANIFEST\_DIR,batch\_size = Batch\_size):

    img\_list = pd.read\_csv(path)

    img\_list = np.array(img\_list)[:Lens]

    print("Found %s train items."%len(img\_list))

    print("list 1 is",img\_list[0,-1])

    steps = math.ceil(len(img\_list) / batch\_size)    # 确定每轮有多少个batch

    while True:

        for i in range(steps):

            batch\_list = img\_list[i \* batch\_size : i \* batch\_size + batch\_size]

            #np.random.shuffle(batch\_list)

            batch\_x = np.array([file for file in batch\_list[:,1:]])

            #batch\_y = np.array([convert2oneHot(label,10) for label in batch\_list[:,-1]])

            yield batch\_x

**模型搭建**

TIME\_PERIODS = 6000

def build\_model(input\_shape=(TIME\_PERIODS,),num\_classes=10):

    model = Sequential()

    model.add(Reshape((TIME\_PERIODS, 1), input\_shape=input\_shape))

    model.add(Conv1D(16, 8,strides=2, activation='relu',input\_shape=(TIME\_PERIODS,1)))

    model.add(Conv1D(16, 8,strides=2, activation='relu',padding="same"))

    model.add(MaxPooling1D(2))

    model.add(Conv1D(64, 4,strides=2, activation='relu',padding="same"))

    model.add(Conv1D(64, 4,strides=2, activation='relu',padding="same"))

    model.add(MaxPooling1D(2))

    model.add(Conv1D(256, 4,strides=2, activation='relu',padding="same"))

    model.add(Conv1D(256, 4,strides=2, activation='relu',padding="same"))

    model.add(MaxPooling1D(2))

    model.add(Conv1D(512, 2,strides=1, activation='relu',padding="same"))

    model.add(Conv1D(512, 2,strides=1, activation='relu',padding="same"))

    model.add(MaxPooling1D(2))

    model.add(GlobalAveragePooling1D())

    model.add(Dropout(0.3))

    model.add(Dense(num\_classes, activation='softmax'))

    return(model)

Show\_one = True

Train = True

if \_\_name\_\_ == "\_\_main\_\_":

    if Show\_one == True:

        show\_iter = xs\_gen()

        for x,y in show\_iter:

            x1 = x[0]

            y1 = y[0]

            break

        print(y)

        print(x1.shape)

        plt.plot(x1)

        plt.show()

**五、实验结果**

