ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)» (МАИ)

Направление подготовки: 27.03.05 «Инноватика»

**Лабораторная работа №2**

по дисциплине «Спецглавы высшей математики»

Выполнил:

Студенты гр. М3О-236Б-22

Жирков Е.В.

Ильина В.Н.

Преподаватель:

Александрова С.С.

Москва 2023

Оглавление

[Цель работы 3](#_Toc153571248)

[Ход работы 3](#_Toc153571249)

[Вывод 9](#_Toc153571250)

# Цель работы

Изучить статью приложенную к п.1, а также построить дерево решений и освоить алгоритмы его реализации на языке программирования Python.

# Ход работы

**Дерево решений** — тип контролируемого машинного обучения, который в основном используется в задачах классификации. Дерево решений само по себе — это в основном жадное, нисходящее, рекурсивное разбиение. «Жадное», потому что на каждом шагу выбирается лучшее разбиение. «Сверху вниз» — потому что мы начинаем с корневого узла, который содержит все записи, а затем делается разбиение.

**Корневой узел** — самый верхний узел в дереве решений называется корневой узел.

**Узел принятия решения** — подузел, который разделяется на дополнительные подузлы, известен как узел принятия решения.

**Лист/терминальный узел** — узел, который не разделяется на другие узлы, называется терминальный узел, или лист.

**Энтропия** — это мера случайности или неопределенности. Уровень энтропии колеблется от 0 до 1. Когда энтропия равна 0, это означает, что подмножество чистое, то есть в нем нет случайных элементов. Когда энтропия равна 1, это означает высокую степень случайности. Энтропия обозначается символами H(S).

**Энтропия вычисляется так:** -(p(0) \* log(P(0)) + p(1) \* log(P(1)))

**Задача 1**

**Код программы:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import tree

df=pd.read\_csv("Diabetes1.csv")

df.head()

x=df.iloc[:,:2]

y=df.iloc[:,2:]

x.head(3)

y.head(3)

model=tree.DecisionTreeClassifier(criterion="entropy")

model.fit(x,y)

model.score(x,y)

model.predict([[29,47]])

model.predict([[45,47]])

tree.plot\_tree(model)

**Задание 2**

**Код программы:**

class Node:

'''

Helper class which implements a single tree node.

'''

def \_\_init\_\_(self, feature=None, threshold=None, data\_left=None, data\_right=None, gain=None, value=None):

self.feature = feature

self.threshold = threshold

self.data\_left = data\_left

self.data\_right = data\_right

self.gain = gain

self.value = value

class DecisionTree:

'''

Class which implements a decision tree classifier algorithm.

'''

def \_\_init\_\_(self, min\_samples\_split=2, max\_depth=5):

self.min\_samples\_split = min\_samples\_split

self.max\_depth = max\_depth

self.root = None

@staticmethod

def \_entropy(s):

'''

Helper function, calculates entropy from an array of integer values.

:param s: list

:return: float, entropy value

'''

# Convert to integers to avoid runtime errors

counts = np.bincount(np.array(s, dtype=np.int64))

# Probabilities of each class label

percentages = counts / len(s)

# Caclulate entropy

entropy = 0

for pct in percentages:

if pct > 0:

entropy += pct \* np.log2(pct)

return -entropy

def \_information\_gain(self, parent, left\_child, right\_child):

'''

Helper function, calculates information gain from a parent and two child nodes.

:param parent: list, the parent node

:param left\_child: list, left child of a parent

:param right\_child: list, right child of a parent

:return: float, information gain

'''

num\_left = len(left\_child) / len(parent)

num\_right = len(right\_child) / len(parent)

# One-liner which implements the previously discussed formula

return self.\_entropy(parent) - (num\_left \* self.\_entropy(left\_child) + num\_right \* self.\_entropy(right\_child))

def \_best\_split(self, X, y):

'''

Helper function, calculates the best split for given features and target

:param X: np.array, features

:param y: np.array or list, target

:return: dict

'''

best\_split = {}

best\_info\_gain = -1

n\_rows, n\_cols = X.shape

# For every dataset feature

for f\_idx in range(n\_cols):

X\_curr = X[:, f\_idx]

# For every unique value of that feature

for threshold in np.unique(X\_curr):

# Construct a dataset and split it to the left and right parts

# Left part includes records lower or equal to the threshold

# Right part includes records higher than the threshold

df = np.concatenate((X, y.reshape(1, -1).T), axis=1)

df\_left = np.array([row for row in df if row[f\_idx] <= threshold])

df\_right = np.array([row for row in df if row[f\_idx] > threshold])

# Do the calculation only if there's data in both subsets

if len(df\_left) > 0 and len(df\_right) > 0:

# Obtain the value of the target variable for subsets

y = df[:, -1]

y\_left = df\_left[:, -1]

y\_right = df\_right[:, -1]

# Caclulate the information gain and save the split parameters

# if the current split if better then the previous best

gain = self.\_information\_gain(y, y\_left, y\_right)

if gain > best\_info\_gain:

best\_split = {

'feature\_index': f\_idx,

'threshold': threshold,

'df\_left': df\_left,

'df\_right': df\_right,

'gain': gain

}

best\_info\_gain = gain

return best\_split

def \_build(self, X, y, depth=0):

'''

Helper recursive function, used to build a decision tree from the input data.

:param X: np.array, features

:param y: np.array or list, target

:param depth: current depth of a tree, used as a stopping criteria

:return: Node

'''

n\_rows, n\_cols = X.shape

# Check to see if a node should be leaf node

if n\_rows >= self.min\_samples\_split and depth <= self.max\_depth:

# Get the best split

best = self.\_best\_split(X, y)

# If the split isn't pure

if best['gain'] > 0:

# Build a tree on the left

left = self.\_build(

X=best['df\_left'][:, :-1],

y=best['df\_left'][:, -1],

depth=depth + 1

)

right = self.\_build(

X=best['df\_right'][:, :-1],

y=best['df\_right'][:, -1],

depth=depth + 1

)

return Node(

feature=best['feature\_index'],

threshold=best['threshold'],

data\_left=left,

data\_right=right,

gain=best['gain']

)

# Leaf node - value is the most common target value

return Node(

value=Counter(y).most\_common(1)[0][0]

)

def fit(self, X, y):

'''

Function used to train a decision tree classifier model.

:param X: np.array, features

:param y: np.array or list, target

:return: None

'''

# Call a recursive function to build the tree

self.root = self.\_build(X, y)

def \_predict(self, x, tree):

'''

Helper recursive function, used to predict a single instance (tree traversal).

:param x: single observation

:param tree: built tree

:return: float, predicted class

'''

# Leaf node

if tree.value != None:

return tree.value

feature\_value = x[tree.feature]

# Go to the left

if feature\_value <= tree.threshold:

return self.\_predict(x=x, tree=tree.data\_left)

# Go to the right

if feature\_value > tree.threshold:

return self.\_predict(x=x, tree=tree.data\_right)

def predict(self, X):

'''

Function used to classify new instances.

:param X: np.array, features

:return: np.array, predicted classes

'''

# Call the \_predict() function for every observation

return [self.\_predict(x, self.root) for x in X]

import pandas as pd

import numpy as np

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

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

from sklearn.preprocessing import OneHotEncoder

df = pd.read\_csv('car.data', header=None)

df.head()

colnames = ['Buying-Price', 'Maintenance-Price', 'Doors', 'Capacity', 'Luggage-Boot', 'Safety', 'Class']

df.columns = colnames

df.head()

OneHotEncoder().fit\_transform(df)

X = df.drop(['Class'], axis=1)

y = df['Class']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.33, random\_state = 42)

model = DecisionTree()

X\_train = X\_train.to\_numpy()

y\_train = y\_train.to\_numpy()

model.fit(X\_train, y\_train)

preds = model.predict(X\_test)

sk\_model = DecisionTreeClassifier()

sk\_model.fit(X\_train, y\_train)

sk\_preds = sk\_model.predict(X\_test)

accuracy\_score(y\_test, sk\_preds)

# Вывод

Ознакомившись со статьей приложенной к п.1, мы построили дерево решений и освоили алгоритмы его реализации на языке программирования Python.