Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

Отчет

По лабораторной работе №2

Логические классификаторы.

Решающие деревья.

Выполнил:

студент гр. ИП-712

Алексеев С.В.

Работу проверил: Ассистент кафедры  
Морозова К.И.

Новосибирск 2020 г.

**Теория.**

**Логические классификаторы**

**Теоретический базис:**

Пусть  — некоторый предикат, определённый на множестве объектов X. Предикат называют закономерностью, если он выделяет достаточно много объектов какого-то одного класса c, и практически не выделяет объекты других классов (более строгое определение будет дано ниже). Особую ценность представляют закономерности, которые описываются простой логической формулой. Их называют правилами (rules). Процесс поиска правил по выборке называют извлечением знаний из данных (knowledge discovery). К знаниям предъявляется особое требование — они должны быть интерпретируемы, то есть понятны людям. На практике логические закономерности часто ищут в виде конъюнкций небольшого числа элементарных высказываний.

***Решающие деревья***

Решающее дерево (decision tree, DT) — это ещё один логический алгоритм классификации, основанный на поиске конъюнктивных закономерностей. Но, в отличие от решающего списка, при синтезе дерева все конъюнкции строятся одновременно. Напомним некоторые понятия теории графов. Деревом называется конечный связный граф с множеством вершин  , не содержащий циклов и имеющий выделенную вершину  , в которую не входит ни одно ребро. Эта вершина называется корнем дерева. Вершина, не имеющая выходящих рёбер, называется терминальной или листом. Остальные вершины называются внутренними. Дерево называется бинарным, если из любой его внутренней вершины выходит ровно два ребра. Выходящие рёбра связывают каждую внутреннюю вершину  с левой дочерней вершиной  и с правой дочерней вершиной .

**Опр.** Бинарное решающее дерево — это алгоритм классификации, задающийся бинарным деревом, в котором каждой внутренней вершине  приписан предикат , каждой терминальной вершине приписано имя класса  . При классификации объекта  он проходит по дереву путь от корня до некоторого листа, в соответствии с Алгоритмом:

**function classify(x):**

**v=v0**

**if βv(x)=1**

**v:=Rv**

**else**

**v:=Lv**

**return yv**

Объект  доходит до вершины v тогда и только тогда, когда выполняется конъюнкция , составленная из всех предикатов, приписанных внутренним вершинам дерева на пути от корня  до вершины . Пусть  — множество всех терминальных вершин дерева. Множества объектов , выделяемых терминальными конъюнкциями , попарно не пересекаются, а их объединение совпадает со всем пространством  (это легко доказывается индукцией по числу вершин дерева). Отсюда следует, что решающее дерево никогда не отказывается от классификации, в отличие от решающего списка. Отсюда также следует, что алгоритм классификации  , реализуемый бинарным решающим деревом, можно представить в виде простого голосования конъюнкций:



причём для любого  одно и только одно слагаемое во всех этих суммах равно единице. Вместо суммирования можно было бы использовать и дизъюнкцию. Естественное требование максимизации информативности конъюнкций  означает, что каждая из них должна выделять как можно больше обучающих объектов, допуская при этом как можно меньше ошибок. Для повышения обобщающей способности решающего дерева число листьев должно быть как можно меньше, и они должны покрывать подвыборки примерно одинаковой мощности .

**Алгоритм построения решающего дерева ID3**

Или Induction of Decision Tree. Идея алгоритма заключается в последовательном дроблении выборки на две части до тех пор, пока в каждой части не окажутся объекты только одного класса.

**Трудоёмкость** алгоритма ID3 имеет порядок O(Bhℓ), где h — глубина дерева, B — среднее число предикатов.

**Код Программы.**

import csv

import math

import random

from collections import defaultdict

from collections import Counter

from typing import \*

import numpy as np

from sklearn import tree

from sklearn.impute import SimpleImputer

class DataItem:

def \_\_init\_\_(self, row: list):

self.attributes = [None] \* 13

for i in range(len(self.attributes)):

self.attributes[i] = None if row[i] == "?" else row[i]

self.clazz = row[len(row) - 1]

def \_\_str\_\_(self):

return str(self.attributes) + ", " + str(self.clazz)

class DecisionTree:

def \_\_init\_\_(self, train\_data: List[DataItem]):

train\_data = [random.choice(train\_data) for i in range(len(train\_data))]

n\_attributes = len(train\_data[0].attributes)

m = int(math.sqrt(n\_attributes))

self.attr\_subset\_indexes = random.sample(range(n\_attributes), m)

attribute\_subset = [subset\_by\_indexes(item.attributes, self.attr\_subset\_indexes) for item in train\_data]

classes = [item.clazz for item in train\_data]

self.sk\_dec\_tree = tree.DecisionTreeClassifier()

self.sk\_dec\_tree.fit(attribute\_subset, classes)

def predict(self, X) -> list:

X = [subset\_by\_indexes(attrs, self.attr\_subset\_indexes) for attrs in X]

return self.sk\_dec\_tree.predict(X)

class Forest:

def \_\_init\_\_(self, trees: List[DecisionTree]):

self.trees = trees

def predict(self, X) -> list:

vote\_table = zip(\*[dec\_tree.predict(X) for dec\_tree in self.trees])

return [one\_most\_common(votes) for votes in vote\_table]

def one\_most\_common(l):

return Counter(l).most\_common(1)[0][0]

def subset\_by\_indexes(l: list, indexes: List[int]) -> list:

return [l[i] for i in indexes]

def read\_csv(file: str) -> List[DataItem]:

with (open(file)) as csv\_file:

reader = csv.reader(csv\_file)

next(reader) # skip header

return [DataItem(row) for row in reader]

def group\_by\_class(data: List[DataItem]) -> dict:

result = defaultdict(list)

for entry in data:

result[entry.clazz].append(entry)

return result

def split\_data(data: List[DataItem], train\_data\_ratio: float) -> Tuple[List[DataItem], List[DataItem]]:

if train\_data\_ratio <= 0:

return [], data

if train\_data\_ratio >= 1:

return data, []

n = len(data)

n\_training = int(n \* train\_data\_ratio)

grouped\_by\_class = group\_by\_class(data)

classes = list(grouped\_by\_class.keys())

train\_data = []

test\_data = []

class\_i = 0

for i in range(n):

class\_i = (class\_i + 1) % len(classes)

clazz = classes[class\_i]

group = grouped\_by\_class[clazz]

if len(group) == 0:

classes.remove(clazz)

continue

group.pop

item = group.pop(random.randint(0, len(group) - 1))

target\_data = train\_data if i < n\_training else test\_data

target\_data.append(item)

return train\_data, test\_data

def fill\_missing\_values(data: List[DataItem]) -> List[DataItem]:

imp = SimpleImputer(missing\_values=np.nan, strategy='mean')

attributes = [item.attributes for item in data]

imp = imp.fit(attributes)

attributes\_imp = imp.transform(attributes)

return [DataItem([\*attributes\_imp[i], data[i].clazz]) for i in range(len(data))]

def count\_matches(forest: Forest, test\_data: List[DataItem]) -> float:

matches = 0

predicted: list = forest.predict([item.attributes for item in test\_data])

for i in range(len(predicted)):

if predicted[i] == test\_data[i].clazz:

matches += 1

return matches / len(test\_data)

def main():

all\_data = read\_csv('data.csv')

all\_data = fill\_missing\_values(all\_data)

(train\_data, test\_data) = split\_data(all\_data, 0.7)

trees = [DecisionTree(train\_data) for i in range(15)]

forest = Forest(trees)

train\_matches = '{:.4f}'.format(count\_matches(forest, train\_data))

test\_matches = '{:.4f}'.format(count\_matches(forest, test\_data))

print("Train: " + str(train\_matches))

print("Test: " + str(test\_matches))

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

main()

exit()

**Результат.**

Train: 0.9220

Test: 0.8636

Train: 0.9122

Test: 0.8523

Train: 0.9512

Test: 0.7955

Train: 0.9171

Test: 0.8750

Train: 0.8829

Test: 0.8750

Train: 0.9366

Test: 0.8864

Train: 0.9561

Test: 0.7386

Train: 0.9024

Test: 0.8409

Train: 0.9220

Test: 0.8636

Train: 0.9463

Test: 0.8636