Report

這次作業是要實作出一顆decision tree，和由這些樹形成的random forest，我用了六個data set來做實驗，分別是Iris, Wine, Glass, Ionosphere , Wdbc, Wpbc(Breast Cancer)，而我的實驗內容是分別用1, 3, 7, 15, 31, 51棵樹，和兩種train valid split方法(7:3和5:5)，所以每個dataset會有12種Accuracy結果，將以表格呈現。

Iris(共150筆資料)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.88 | 0.96 |
| 3 | 0.92 | 0.96 |
| 7 | 0.92 | 0.93 |
| 15 | 0.89 | 0.96 |
| 31 | 0.89 | 0.96 |
| 51 | 0.89 | 0.96 |

可以觀察到，7:3的Accuracy都比5:5要來的高，可能是資料量需要多一點才較能預測其他未知的部分，而隨著樹的數量成長，準確率似乎有比較好，但在此資料集不太明顯。

Wine(共178筆資料)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.94 | 0.89 |
| 3 | 0.97 | 0.85 |
| 7 | 0.96 | 0.94 |
| 15 | 0.93 | 0.93 |
| 31 | 0.96 | 0.94 |
| 51 | 0.97 | 0.96 |

由Wine的資料集可以觀察到，反而是5:5的split方式比較好，而Wine本身的資料筆數也比iris較多，但隨著樹的棵樹成長，準確率還是會有所提升。

Glass(共214筆資料)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.62 | 0.54 |
| 3 | 0.71 | 0.68 |
| 7 | 0.69 | 0.75 |
| 15 | 0.70 | 0.72 |
| 31 | 0.70 | 0.72 |
| 51 | 0.69 | 0.74 |

Glass的資料集不同於Iris和wine，總共有7個分類，這也是導致樹的數量太少時準確率較低的原因，而由觀察也可發現，樹的數量超過一定數目後，accuracy就會趨於穩定，且由原本的5:5較好變成7:3較好。

ionosphere(共351筆資料, 51棵樹的結果跑太久不放)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.898 | 0.906 |
| 3 | 0.892 | 0.887 |
| 7 | 0.926 | 0.906 |
| 15 | 0.9147 | 0.9151 |
| 31 | 0.932 | 0.925 |

結果也是符合預期，雖然split的方式結果看下來差不多，但樹的數量越多，確實會得到較好的結果，光是7棵樹就能夠得到不錯的結果。

Wdbc(共569筆資料)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.919 | 0.918 |
| 3 | 0.926 | 0.906 |
| 7 | 0.930 | 0.930 |
| 15 | 0.926 | 0.941 |
| 31 | 0.923 | 0.918 |

Wdbc的準確率從一開始的一棵樹就滿高的了，但隨著樹的數量提升，準確率還是有所提高，兩種split方法的結果也差不多，或許是因為他是二分類問題(B or M，良性或惡性腫瘤)，所以準確率會較高。

Wpbc(共198筆資料)

|  |  |  |
| --- | --- | --- |
| Tree number | Train: valid=5:5(Accuracy) | Train: valid=7:3(Accuracy) |
| 1 | 0.566 | 0.6 |
| 3 | 0.707 | 0.7 |
| 7 | 0.687 | 0.733 |
| 15 | 0.667 | 0.733 |
| 31 | 0.717 | 0.717 |

Wpbc和wdbc不同，資料量也較少，但也是二分類問題(R or N，是否會復發)，或許是因為資料量較少或資料特性，一開始的decision tree的準確率較低，但趨勢還是隨著樹的增加而提高準確率。

在decision tree的實作過程中，我是從train data裡面bagging(隨機取出)train data兩倍數量的sample來建立model，並從attribute中刪除大概1/3的feature，藉由此來達成每棵樹都不太一樣的random forest。

在實作時，我有把minimum number of samples per node和tree depth納入考慮，但我發現簡單的data set的tree depth本來就不高，其他data set在做這些調整後有些是變好有些是變壞，可能跟data本身的一些特性也有關。從這次的作業中，我也學到一些tree的實作技巧，還有如何整理data及做使用。另外因為建立tree的速度其實有點慢，大一點的data set要跑到51棵樹就花了一個多小時了(其實還好?)，反正我電腦就當掉了，希望之後還有機會整理更大的data set及跑更多的樹，並得到更高的準確率。

Code：

# Data wrangling

import pandas as pd

# Array math

import numpy as np

# Quick value count calculator

from collections import Counter

# just use to see confusion matrix

from sklearn.metrics import confusion\_matrix

# random to bagging

import random

class Node:

"""

Class for creating the nodes for a decision tree

"""

def \_\_init\_\_(

self,

Y: list,

X: pd.DataFrame,

min\_samples\_split=None,

max\_depth=None,

depth=None,

node\_type=None,

rule=None,

category\_num=None

):

# Saving the data to the node

self.Y = Y

self.X = X

# Saving the hyper parameters

self.min\_samples\_split = min\_samples\_split if min\_samples\_split else 2

self.max\_depth = max\_depth if max\_depth else 20

# Default current depth of node

self.depth = depth if depth else 0

# Extracting all the features

self.features = list(self.X.columns)

# Type of node

self.node\_type = node\_type if node\_type else 'root'

# Rule for spliting

self.rule = rule if rule else ""

# Category numbers

self.category\_num = category\_num if category\_num else 2

# Calculating the counts of Y in the node

self.counts = Counter(Y)

# Getting the GINI impurity based on the Y distribution

self.gini\_impurity = self.get\_GINI()

# Sorting the counts and saving the final prediction of the node

counts\_sorted = list(sorted(self.counts.items(), key=lambda item: item[1]))

# Getting the last item

yhat = None

if len(counts\_sorted) > 0:

yhat = counts\_sorted[-1][0]

# Saving to object attribute. This node will predict the class with the most frequent class

self.yhat = yhat

# Saving the number of observations in the node

self.n = len(Y)

# Initiating the left and right nodes as empty nodes

self.left = None

self.right = None

# Default values for splits

self.best\_feature = None

self.best\_value = None

# Define is or not leaf

#self.is\_leaf = True if len(self.counts) == 1 else False

#print(self.left, self.right)

@staticmethod

def GINI\_impurity(y\_count):

"""

Given the observations of a binary class calculate the GINI impurity

"""

n = 0

# Ensuring the correct types

for i in range(len(y\_count)):

if y\_count[i] is None:

y\_count[i] = 0

n += y\_count[i]

# Getting the total observations

#n = y1\_count + y2\_count

# If n is 0 then we return the lowest possible gini impurity

if n == 0:

return 0.0

# Getting the probability to see each of the classes

p = []

for i in range(len(y\_count)):

p.append(y\_count[i] / n)

# Calculating GINI

gini = 1

for i in range(len(p)):

gini = gini - (p[i] \*\* 2)

#print(gini)

# Returning the gini impurity

return gini

@staticmethod

def ma(x: np.array, window: int) -> np.array:

"""

Calculates the moving average of the given list.

"""

return np.convolve(x, np.ones(window), 'valid') / window

def get\_GINI(self):

"""

Function to calculate the GINI impurity of a node

"""

y\_count = []

# Getting the 0~n counts

for i in range(self.category\_num):

y\_count.append(self.counts.get(i, 0))

#y1\_count, y2\_count = self.counts.get(0, 0), self.counts.get(1, 0)

#print(y1\_count, y2\_count)

# Getting the GINI impurity

#print(y\_count)

#return self.GINI\_impurity(y1\_count, y2\_count)

return self.GINI\_impurity(y\_count)

def best\_split(self) -> tuple:

"""

Given the X features and Y targets calculates the best split

for a decision tree

"""

# Creating a dataset for spliting

df = self.X.copy()

df['Y'] = self.Y

# Getting the GINI impurity for the base input

GINI\_base = self.get\_GINI()

# Finding which split yields the best GINI gain

max\_gain = 0

# Default best feature and split

best\_feature = None

best\_value = None

for feature in self.features:

# Droping missing values

Xdf = df.dropna().sort\_values(feature)

# Sorting the values and getting the rolling average

xmeans = self.ma(Xdf[feature].unique(), 2)

for value in xmeans:

# Spliting the dataset

left\_counts = Counter(Xdf[Xdf[feature]<value]['Y'])

right\_counts = Counter(Xdf[Xdf[feature]>=value]['Y'])

# Getting the Y distribution from the dicts

y\_left = []

y\_right = []

for i in range(self.category\_num):

y\_left.append(left\_counts.get(i, 0))

y\_right.append(right\_counts.get(i, 0))

#y0\_left, y1\_left, y0\_right, y1\_right = left\_counts.get(0, 0), left\_counts.get(1, 0), right\_counts.get(0, 0), right\_counts.get(1, 0)

# Getting the left and right gini impurities

gini\_left = self.GINI\_impurity(y\_left)

gini\_right = self.GINI\_impurity(y\_right)

# Getting the obs count from the left and the right data splits

n\_left = np.sum(y\_left)

n\_right = np.sum(y\_right)

# Calculating the weights for each of the nodes

w\_left = n\_left / (n\_left + n\_right)

w\_right = n\_right / (n\_left + n\_right)

# Calculating the weighted GINI impurity

wGINI = w\_left \* gini\_left + w\_right \* gini\_right

# Calculating the GINI gain

GINIgain = GINI\_base - wGINI

# Checking if this is the best split so far

if GINIgain > max\_gain:

best\_feature = feature

best\_value = value

# Setting the best gain to the current one

max\_gain = GINIgain

return (best\_feature, best\_value)

def grow\_tree(self):

"""

Recursive method to create the decision tree

"""

# Making a df from the data

df = self.X.copy()

df['Y'] = self.Y

#print(self.depth, self.max\_depth)

#print(self.n, self.min\_samples\_split)

# If there is GINI to be gained, we split further

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

# Getting the best split

best\_feature, best\_value = self.best\_split()

#print(best\_feature, best\_value)

if best\_feature is not None:

# Saving the best split to the current node

self.best\_feature = best\_feature

self.best\_value = best\_value

# Getting the left and right nodes

left\_df, right\_df = df[df[best\_feature]<=best\_value].copy(), df[df[best\_feature]>best\_value].copy()

# Creating the left and right nodes

left = Node(

left\_df['Y'].values.tolist(),

left\_df[self.features],

depth=self.depth + 1,

max\_depth=self.max\_depth,

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

node\_type='left\_node',

rule=f"{best\_feature} <= {round(best\_value, 3)}",

category\_num=self.category\_num

)

self.left = left

self.left.grow\_tree()

right = Node(

right\_df['Y'].values.tolist(),

right\_df[self.features],

depth=self.depth + 1,

max\_depth=self.max\_depth,

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

node\_type='right\_node',

rule=f"{best\_feature} > {round(best\_value, 3)}",

category\_num=self.category\_num

)

self.right = right

self.right.grow\_tree()

def print\_info(self, width=4):

"""

Method to print the infromation about the tree

"""

# Defining the number of spaces

const = int(self.depth \* width \*\* 1.5)

spaces = "-" \* const

if self.node\_type == 'root':

print("Root")

else:

print(f"|{spaces} Split rule: {self.rule}")

print(f"{' ' \* const} | GINI impurity of the node: {round(self.gini\_impurity, 2)}")

print(f"{' ' \* const} | Class distribution in the node: {dict(self.counts)}")

print(f"{' ' \* const} | Predicted class: {self.yhat}")

def print\_tree(self):

"""

Prints the whole tree from the current node to the bottom

"""

self.print\_info()

if self.left is not None:

self.left.print\_tree()

if self.right is not None:

self.right.print\_tree()

def predict(self, X:pd.DataFrame):

"""

Batch prediction method

"""

predictions = []

for \_, x in X.iterrows():

values = {}

for feature in self.features:

values.update({feature: x[feature]})

predictions.append(self.predict\_obs(values))

return predictions

def predict\_obs(self, values: dict) -> int:

"""

Method to predict the class given a set of features

"""

cur\_node = self

#print(cur\_node.best\_feature)

while cur\_node.depth < cur\_node.max\_depth:

if cur\_node.left == None and cur\_node.right == None: break #is leaf

# Traversing the nodes all the way to the bottom

best\_feature = cur\_node.best\_feature

best\_value = cur\_node.best\_value

#print(cur\_node.best\_feature, cur\_node.best\_value, cur\_node.is\_leaf)

if cur\_node.n < cur\_node.min\_samples\_split:

break

if (values.get(best\_feature) < best\_value):

if self.left is not None:

cur\_node = cur\_node.left

else:

if self.right is not None:

cur\_node = cur\_node.right

return cur\_node.yhat

def plant\_a\_tree(d, cat\_col, d\_valid\_X):

d = d.sample(n=len(d)\*2,replace=True).reset\_index(drop=True)

category\_num = len(d[cat\_col].value\_counts())

X = d.drop(columns=[cat\_col])

#attribute bagging

for col in X.columns:

rand = random.randint(1, 3)

if rand == 1:

X = X.drop(columns=[col])

Y = d[cat\_col].values.tolist()

#print(Y)

# Initiating the Node

root = Node(Y, X, max\_depth=5, min\_samples\_split=10, category\_num=category\_num)

# Getting teh best split

root.grow\_tree()

d\_valid\_X\_subset1 = d\_valid\_X.copy()

d\_valid\_X\_subset1['yhat'] = root.predict(d\_valid\_X\_subset1)

return d\_valid\_X\_subset1['yhat']

def do\_train\_and\_votes(d, train\_valid\_size, tree\_num, category\_col):

print('train:valid = ', train\_valid\_size, ':', 1-train\_valid\_size, ', tree\_numbers:', tree\_num)

d\_valid = d[int(len(d)\*train\_valid\_size):len(d)] #valid data

d = d[0:int(len(d)\*train\_valid\_size)] #train data

d\_valid\_X = d\_valid.drop(columns=[category\_col]) #to predict, not need category

d\_valid\_Y = d\_valid[category\_col].values.tolist() #target

result = pd.DataFrame()

for i in range(tree\_num):

result = pd.concat([result, plant\_a\_tree(d, category\_col, d\_valid\_X)], axis = 1)

result = result.T

votes = []

#vote fot highest votes

for col in result.columns:

votes.append(result[col].mode()[0])

real\_Y = d\_valid\_Y

correct = 0

for i in range(len(votes)):

if votes[i] == real\_Y[i]: correct += 1

mat = confusion\_matrix(real\_Y, votes)

#confusion matrix

print(mat)

print('Acc: ', correct/len(votes))

print('-----------------------------------------------')

def wdbc(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/wdbc.data", header=None).dropna()

#replace some label to number

d[1].replace(['B'], 0,inplace = True )

d[1].replace(['M'], 1,inplace = True )

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

d = d.drop(columns=[0]) #useless feature

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, 1) #split

def wpbc(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/wpbc.data", header=None).dropna()

#replace some label to number

d[1].replace(['N'], 0,inplace = True )

d[1].replace(['R'], 1,inplace = True )

for col in d.columns:

d[col].replace(["?"], int(d[col].mode()[0]), inplace=True) #delete ? value

d[col] = d[col].astype('float')

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

d = d.drop(columns=[0]) #useless feature

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, 1) #split

def wine(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/wine.data").dropna()

#replace some label to number

d['category'].replace([1], 0,inplace = True )

d['category'].replace([2], 1,inplace = True )

d['category'].replace([3], 2,inplace = True )

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, 'category') #split

def iris(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/iris.data").dropna()

#replace some label to number

d['category'].replace(['Iris-setosa'], 0,inplace = True )

d['category'].replace(['Iris-versicolor'], 1,inplace = True )

d['category'].replace(['Iris-virginica'], 2,inplace = True )

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, 'category') #split

def glass(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/glass.data", header=None).dropna()

#replace some label to number

d[10].replace([1], 0,inplace = True )

d[10].replace([2], 1,inplace = True )

d[10].replace([3], 2,inplace = True )

d[10].replace([4], 3,inplace = True )

d[10].replace([5], 4,inplace = True )

d[10].replace([6], 5,inplace = True )

d[10].replace([7], 6,inplace = True )

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

d = d.drop(columns=[0]) #useless feature(ID)

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, 10) #split

def ionosphere(train\_valid\_size, tree\_num):

d = pd.read\_csv("../input/ai-hw-data/ionosphere.data", header=None).dropna()

#replace some label to number

d[34].replace(['b'], 0,inplace = True )

d[34].replace(['g'], 1,inplace = True )

d = d.sample(frac=1, random\_state=123).reset\_index(drop = True) #shuffle

do\_train\_and\_votes(d, train\_valid\_size, tree\_num, category\_col=34) #split

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

# Reading data

#d = pd.read\_csv("../data/iris.data")[['length1', 'length2', 'length3', 'length4', 'category']].dropna()

#d = pd.read\_csv("../data/wine.data")[['a', 'b','c','d','e','f','g','h','i','j','k','l','m', 'category']].dropna()

#wdbc(train\_valid\_size = 0.7, tree\_num = 1)

#wpbc(train\_valid\_size = 0.7, tree\_num = 1)

#wine(train\_valid\_size = 0.7, tree\_num = 1)

#iris(train\_valid\_size = 0.7, tree\_num = 1)

trees = [1, 3, 7, 15, 31]

print('iris')

for i in trees:

iris(train\_valid\_size = 0.5, tree\_num = i)

iris(train\_valid\_size = 0.7, tree\_num = i)

print('')

print('wine')

for i in trees:

wine(train\_valid\_size = 0.5, tree\_num = i)

wine(train\_valid\_size = 0.7, tree\_num = i)

print('')

print('glass')

for i in trees:

glass(train\_valid\_size = 0.5, tree\_num = i)

glass(train\_valid\_size = 0.7, tree\_num = i)

print('')

print('wdbc')

for i in trees:

wdbc(train\_valid\_size = 0.5, tree\_num = i)

wdbc(train\_valid\_size = 0.7, tree\_num = i)

print('')

print('wpbc')

for i in trees:

wpbc(train\_valid\_size = 0.5, tree\_num = i)

wpbc(train\_valid\_size = 0.7, tree\_num = i)

print('')

print('ionosphere')

for i in trees:

ionosphere(train\_valid\_size = 0.5, tree\_num = i)

ionosphere(train\_valid\_size = 0.7, tree\_num = i)