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)
```