

CS 178: Machine Learning & Data Mining

Homework 3: Due Friday, 8 March 2024 (11:59 PM)

Instructions

This homework (and many subsequent ones) will involve data analysis and reporting on methods and results using Python code. You will submit a **single PDF file** that contains everything to Gradescope. This includes any text you wish to include to describe your results, the complete code snippets of how you attempted each problem, any figures that were generated, and scans of any work on paper that you wish to include. It is important that you include enough detail that we know how you solved the problem, since otherwise we will be unable to grade it.

Your homeworks will be given to you as Jupyter notebooks containing the problem descriptions and some template code that will help you get started. You are encouraged to modify these starter Jupyter notebooks to complete your assignment and to write your report. You may add additional cells (containing either code or text) as needed. This will help you not only ensure that all of the code for the solutions is included, but also will provide an easy way to export your results to a PDF file (for example, doing ***print preview and printing to pdf***). Before submitting, ensure that your submission is complete, all text and code is legible (i.e. not cut off), and the pdf includes page breaks (i.e. not one very long page). We recommend liberal use of Markdown cells to create headers for each problem and sub-problem, explaining your implementation/answers, and including any mathematical equations. For parts of the homework you do on paper, scan it in such that it is legible (there are a number of free Android/iOS scanning apps, if you do not have access to a scanner), and include it as an image in the Jupyter notebook.

If you have any questions/concerns about using Jupyter notebooks, ask us on Piazza. There you can also find additional instructions on how to convert to a .pdf.

Summary of Assignment: 100 total points

- Problem 1: Drawing a Decision Tree (15 points)
- Problem 2: Computing the Gini Index (15 points)
- Problem 3: Implementing Decision Trees (50 points)
 - Problem 3.1: `class_prob_vector` (5 points)
 - Problem 3.2: `leaf_condition` (10 points)
 - Problem 3.3: `gini_score` (10 points)
 - Problem 3.4: `find_best_split` (15 points)
 - Problem 3.5: `build_tree` (10 points)
- Problem 4: Experimenting with Sklearn (15 points)
 - Problem 4.1: Training a small DT (5 points)
 - Problem 4.2: Varying depth (5 points)
 - Problem 4.3: Varying min_leaf (5 points)
- Statement of Collaboration (5 points)

Before we get started, let's import some libraries that you will make use of in this assignment. Make sure that you run the code cell below in order to import these libraries.

Important: In the code block below, we set `seed=1234` . This is to ensure your code has reproducible results and is important for grading. Do not change this. If you are not using the provided Jupyter notebook, make sure to also set the random seed as below.

Important: Do not change any codes we give you below, except for those waiting for you to complete. This is to ensure your code has reproducible results and is important for grading.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification, load_breast_cancer, load_wine
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import warnings
warnings.filterwarnings('ignore')

# Fix the random seed for reproducibility
# !! Important !! : do not change this
seed = 1234
np.random.seed(seed)
```

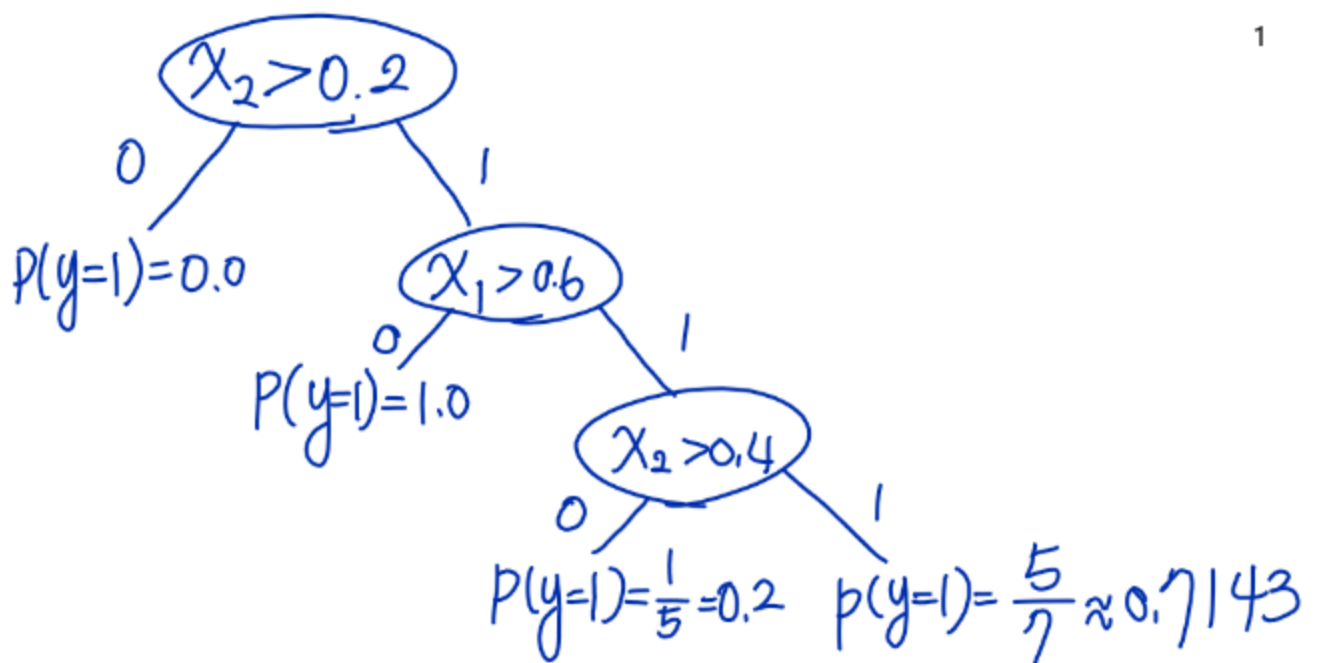
Problem 1 (15 points): Draw a Decision Tree

In the image below, you are given some data for a binary classification problem with features X_1 and X_2 and labels $y = 0$ or $y = 1$. In addition, the dashed black lines depict the splits of a particular decision tree trained on this data.

- Draw the decision tree corresponding to the splits in the image. For every leaf node, you should also include $p(y = 1|\text{path})$.
- Write your answer on paper and include a picture of your answer in this notebook. In order to include an image in Jupyter notebook, save the image in the same directory as the .ipynb file and then write `! [caption](image.png)` . Alternatively, you may go to Edit --> Insert Image at the top menu to insert an image into a Markdown cell. Double check that your image is visible in your PDF submission.



```
In [1]: from IPython.display import Image, display
display(Image(filename="p1.png"))
```



Problem 2 (15 points): Computing the Gini Index

In the table below, you are given some data for a classification problem with 3 features and 3 classes. The feature X_1 is a binary feature, and the features X_2 and X_3 are real-valued features. The three classes are labeled $y = 0, 1, 2$.

In this problem, you will use the Gini index to decide which feature to split on. In other words, you will be creating the root of a decision tree based on this data. You should do this problem by hand, and include an image (or LaTeX) with your solution in your notebook. See Problem 1 for details on how to include an image in your notebook.

- What is the Gini index of the data before doing any splitting?
- For each feature, compute the Gini index of splitting on that feature. Use a threshold of $t = 0.4$ for X_2 and $t = 2.0$ for X_3 . (Why don't we need to specify a threshold for X_1 ? You don't have to answer this, but you should think about it.)
- Based on your answer to the previous question, which feature should we split on for the root of our decision tree?

X_1	X_2	X_3	y
0	0.1	3.4	0
1	0.3	3.0	0
0	-0.2	2.9	0
0	1.3	0.1	1
1	2.2	-0.5	1
0	4.0	0.3	1
1	0.5	1.2	2
1	3.0	0.75	2

X1	X2	X3	y
1	2.2	0.1	2
0	0.25	0.1	2

In [2]: `display(Image(filename="p2-1.jpg"))`
`display(Image(filename="p2.jpg"))`

X1	X2	X3	y
0	0.1	3.4	0
1	0.3	3.0	0
0	-0.2	2.9	0
0	1.3	0.1	1
1	2.2	-0.5	1
0	4.0	0.3	1
1	0.5	1.2	2
1	3.0	0.75	2
1	2.2	0.1	2
0	0.25	0.1	2

$X_1 = 0 \Rightarrow 5 \text{ data } P_0 = \frac{1}{2}$

$$P(y=0) = \frac{2}{5}$$

$$P(y=1) = \frac{2}{5}$$

$$P(y=2) = \frac{1}{5}$$

$$\begin{aligned} \text{gini} &= 1 - \left(\frac{2^2}{5} + \frac{2^2}{5} + \frac{1^2}{5} \right) \\ &= 1 - \frac{9}{25} = \frac{16}{25} \end{aligned}$$

$X_1 = 1 \rightarrow 5 \text{ data } P_1 = \frac{1}{2}$

$$P(y=0) = \frac{1}{5}$$

$$P(y=1) = \frac{1}{5}$$

$$P(y=2) = \frac{3}{5}$$

$$\begin{aligned} \text{gini} &= 1 - \left(\frac{1^2}{5} + \frac{1^2}{5} + \frac{3^2}{5} \right) \\ &= 1 - \frac{11}{25} = \frac{14}{25} \end{aligned}$$

$$\begin{aligned} \text{gini}(t) &= P_0 G_0 + P_1 G_1 \\ &= \frac{1}{2} \left(\frac{16}{25} \right) + \frac{1}{2} \left(\frac{14}{25} \right) = \frac{15}{25} = \frac{3}{5} = \underline{\underline{0.6}} \end{aligned}$$

$$P(y=0)=0.3 \quad P(y=1)=0.3 \quad P(y=2)=0.4$$

1. Before splitting

$$\begin{aligned} \text{Gini Index} &= 1 - \sum_k (p_k)^2 \\ &= 1 - (0.3^2 + 0.3^2 + 0.4^2) = 0.66. \end{aligned}$$

2. Split $t=0.4$ for X_2

When $X_2 < 0.4 \Rightarrow 4$ points (total) $P_L = 0.4$

$$P(y=0) = \frac{3}{4} \quad P(y=1) = 0 \quad P(y=2) = \frac{1}{4}$$

$$\text{Gini Index } G_L = 1 - \left(\left(\frac{3}{4} \right)^2 + \left(\frac{1}{4} \right)^2 \right) = 1 - (0.75^2 + 0.25^2) = 0.375$$

When $X_2 \geq 0.4 \Rightarrow 6$ points (total) $P_R = 0.6$

$$P(y=0) = 0 \quad P(y=1) = \frac{3}{6} = 0.5 \quad P(y=2) = \frac{3}{6} = 0.5$$

$$G_R = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$\text{Gini}(t) = P_L G_L + P_R G_R = 0.4 * 0.375 + 0.6 * 0.5 = \boxed{0.45}$$

When $X_3 < 2.0 \Rightarrow 7$ points $P_L = 0.7$

$$P(y=0) = 0 \quad P(y=1) = \frac{3}{7} \quad P(y=2) = \frac{4}{7}$$

$$G_L = 1 - \left(\left(\frac{3}{7} \right)^2 + \left(\frac{4}{7} \right)^2 \right) = 0.4898$$

When $X_3 \geq 2.0 \Rightarrow 3$ points $P_R = 0.3$

$$P(y=0) = 1 \quad P(y=1) = 0 \quad P(y=2) = 0$$

$$G_R = 1 - 1 = 0$$

$$\text{Gini}(t) = 0.7 * 0.4898 + 0 = \boxed{0.3429}$$

3.

We should split $t=2.0$ for X_3 since we want to minimize $\text{Gini}(t)$

Problem 3: Implementing a Decision Tree

You will now implement an algorithm for learning a decision tree from data. You are given some starter code below that you will need to complete. To keep things simple, your implementation will only work for binary classification.

The class `Node` represents a single node in a decision tree. This class is already completed for you, and contains several useful attributes. In addition, the class `DecisionTree` is partially implemented for you. Before attempting this problem, it is important that you read and understand both of these classes.

```
In [4]: class Node:
        """ A class representing a node in a decision tree.
        """

        def __init__(self, depth):
            self.depth = depth          # What level of the tree this node is at; depth=0 is the root

            self.split_feature = None    # The index of the feature that this node splits, if any
            self.threshold = None        # The threshold used to split the feature

            self.left_child = None       # A node object (or None) representing the left-hand child of
            self.right_child = None      # A node object (or None) representing the right-hand child of

            self.probs = None            # A numpy array of length 2 representing [p(y=0), p(y=1)] at

        def __repr__(self):
            # Gives a nice looking representation if you call print on a node

            return f'DT Node: \n -| Depth: {self.depth}' \
                   f'\n -| Split feature: {self.split_feature}' \
                   f'\n -| Threshold: {self.threshold}' \
                   f'\n -| Probs: {self.probs}'
```

```
In [10]: class DecisionTree:
        """ A class representing a decision tree.
        """

        def __init__(self, max_depth=3):
            self.root = None            # A Node object which is the root of our tree
            self.max_depth = max_depth  # An integer representing the maximum allowed depth of the t

        def class_prob_vector(self, y):
            """ Given an array of labels y, compute p(y=0) and p(y=1).

            returns: a numpy array containing [p(y=0), p(y=1)].
            """
            ### YOUR CODE STARTS HERE ###
            y_size = len(y)
            y0 = np.sum(y==0)
            y1 = np.sum(y==1)
            p_y0 = y0/y_size
            p_y1 = y1/y_size
            probs = np.array([p_y0, p_y1])
            return probs
            ### YOUR CODE ENDS HERE ###

        def leaf_condition(self, node):
            """ Given a Node object, returns True if this is a leaf node and False otherwise.

            A Node is considered a leaf node if all labels at the node belong to the same class,
            or if the node is at the maximum allowed depth of the tree.
            """
            ### YOUR CODE STARTS HERE ###
            if node.probs[0]==1 or node.probs[1]==1 or node.depth == self.max_depth:
                return True
            else:
                return False
            ### YOUR CODE ENDS HERE ###
```

```

def gini_score(self, X, y, i, threshold):
    """ Given features X and labels y, computes the Gini index of splitting
        the i-th feature at the given threshold.
    """
    ### YOUR CODE STARTS HERE ###
    #combine X and y first
    total_point = len(y)
    X_y = np.column_stack((X, y))
    X_y_L = X_y[X_y[:,i]<=threshold]
    num_point_L = len(X_y_L)
    P_L = num_point_L/total_point
    X_y_R = X_y[X_y[:,i]>threshold]
    num_point_R = len(X_y_R)
    P_R = num_point_R/total_point

    X_L = X_y_L[:, :-1] # All rows, all columns except the last one
    y_L = X_y_L[:, -1]
    X_R = X_y_R[:, :-1] # All rows, all columns except the last one
    y_R = X_y_R[:, -1]
    probs_L = self.class_prob_vector(y_L)
    G_L = 1 - np.sum(probs_L**2)
    probs_R = self.class_prob_vector(y_R)
    G_R = 1 - np.sum(probs_R**2)
    return G_L*P_L+G_R*P_R
    ### YOUR CODE ENDS HERE ###

def find_best_split(self, X, y):
    """ Given features X and labels y, finds the best split based on the Gini index.

    returns: an index corresponding to which feature we are splitting,
            as well as the threshold we are splitting the feature at.
    """
    ### YOUR CODE STARTS HERE ###
    scores=[]
    #sort the first feature
    for f in range(X.shape[1]):
        gini_score=[]
        sorted_indices = np.argsort(X[:, f])
        X_sorted = X[sorted_indices]
        for i in range(len(X_sorted)-1):
            threshold = (X_sorted[i][f]+X_sorted[i+1][f])/2
            score = self.gini_score(X, y, f, threshold)
            gini_score.append(score)
        scores.append(gini_score)
    scores = np.array(scores)
    # Find the index of the minimum value
    min_gini = np.argmin(scores)
    # Convert the flattened index to row and column indices
    feature_index, row_index= np.unravel_index(min_gini, scores.shape)
    sorted_indices = np.argsort(X[:, feature_index])
    X_sorted = X[sorted_indices]
    threshold = (X_sorted[row_index][feature_index]+X_sorted[row_index+1][feature_index])/2

    return feature_index, threshold
    ### YOUR CODE ENDS HERE ###

def fit(self, X, y):
    """ Fits the decision tree given features X and labels y.
    """
    assert isinstance(X, np.ndarray), 'X must be a numpy array'
    assert isinstance(y, np.ndarray), 'y must be a numpy array'

    self.build_tree(X, y, 0)

```

```

def build_tree(self, X, y, depth):
    """ Recursively builds the decision tree.
    """

    # Create a new node
    node = Node(depth)
    if depth == 0:
        self.root = node

    # Get the class probabilities for this node
    node.probs = self.class_prob_vector(y)

    # Check if this new node is a leaf node; otherwise, split it
    if self.leaf_condition(node):
        return node
    else:
        # Find which feature to split on and the splitting threshold
        split_idx, split_threshold = self.find_best_split(X, y)
        X_y = np.column_stack((X, y))
        # Create left/right splits of data based on split_idx, split_threshold
        X_y_L = X_y[X_y[:, split_idx] <= split_threshold]
        X_y_R = X_y[X_y[:, split_idx] > split_threshold]
        X_L, y_L = X_y_L[:, :-1], X_y_L[:, -1]
        X_R, y_R = X_y_R[:, :-1], X_y_R[:, -1]

        # Recursively create the left/right nodes
        depth = depth + 1
        node_L = self.build_tree(X_L, y_L, depth)
        node_R = self.build_tree(X_R, y_R, depth)

        # Fill in node information
        node.split_feature = split_idx
        node.threshold = split_threshold
        node.left_child = node_L
        node.right_child = node_R
        return node

def predict(self, X):
    """ After fitting the decision tree, this function can be called to make predictions
    for every data point in the feature array X.
    """
    y_hat = []

    for x in X:
        # Make a prediction for every row in X
        y_hat.append(self._predict(x))

    y_hat = np.array(y_hat)
    return y_hat

def _predict(self, x):
    """ Makes predictions on individual datapoints x.
    """
    current_node = self.root

    while True:
        if self.leaf_condition(current_node):
            # If we're at a leaf node, make a prediction based on the probabilities
            probs = current_node.probs
            y_hat = np.argmax(probs)
            return y_hat
        else:
            # Otherwise, traverse the tree based on the splits

```



```

        go_left = x[current_node.split_feature] <= current_node.threshold
        if go_left:
            current_node = current_node.left_child
        else:
            current_node = current_node.right_child

def __repr__(self):
    # Pretty printing if we call print on our DecisionTree
    return f'Decision Tree \n -| Max Depth: {self.max_depth}'

```

Problem 3.1 (5 points):

- Complete the function `class_prob_vector`. This function takes in array of labels `y` and returns a numpy array containing $p(y = 0)$ and $p(y = 1)$.
- Run the code block given below to test your implementation. If your code is correct, all tests should pass.

In [13]: *# Use this code block to test your implementation in Problem 3.1*
Don't change anything here -- just run it

```

dt = DecisionTree(max_depth=3)

y = np.array([0, 0, 1, 1, 1])
expected = np.array([0.4, 0.6])
out = dt.class_prob_vector(y)

print(f'Test 1 passed: {np.array_equal(expected, out)}')

y = np.array([1, 1, 1])
expected = np.array([0., 1.])
out = dt.class_prob_vector(y)

print(f'Test 2 passed: {np.array_equal(expected, out)}')

```

Test 1 passed: True
 Test 2 passed: True

Problem 3.2 (10 points):

In our decision tree implementation, we will consider a node to be a leaf node if either (a) all labels at the node belong to the same class, or (b) the node is at depth `max_depth`, where `max_depth` is an attribute of our `DecisionTree` that we can specify.

- Complete the function `leaf_condition`. This function should take in a `Node` object and return True if this node is a leaf node (according to the above criteria) and False otherwise. You should be able to determine this based on the attributes already defined in the `Node` class.
- Run the code block given below to test your implementation. If your code is correct, all tests should pass.

In [14]: *# Use this code block to test your implementation in Problem 3.2*
Don't change anything here -- just run it

```

dt = DecisionTree(max_depth=3)

node = Node(depth=2)
node.probs = np.array([0.5, 0.5])
expected = False

```

```

out = dt.leaf_condition(node)
print(f'Test 1 passed: {np.array_equal(expected, out)}')

node = Node(depth=3)
node.probs = np.array([0.5, 0.5])
expected = True
out = dt.leaf_condition(node)
print(f'Test 2 passed: {np.array_equal(expected, out)}')

node = Node(depth=1)
node.probs = np.array([1., 0.])
expected = True
out = dt.leaf_condition(node)
print(f'Test 3 passed: {np.array_equal(expected, out)}')

```

Test 1 passed: True
Test 2 passed: True
Test 3 passed: True

Problem 3.3 (10 points):

Your next task is to implement a function that computes the Gini index.

- Complete the function `gini_score`. This function takes in features `X` and labels `y`, as well as a feature index `i` and a scalar `threshold`. Given these inputs, the function `gini_score` should return the Gini index (i.e. a single number) obtained by splitting the `i` th feature in `X` at the specified threshold.
- Run the code block given below to test your implementation. If your code is correct, all tests should pass. Note that the numerical tests here are the same as the examples in Lectures 18 and 19, which may be helpful in debugging your code.

```

In [15]: # Use this code block to test your implementation in Problem 3.3
# Don't change anything here -- just run it

dt = DecisionTree(max_depth=3)

# Note: these are the same values as the example in Lectures 18 and 19

X = np.array([[1,1], [2,8], [4,9], [6,7], [7,4], [8,11], [3,3], [5,5], [9,5], [10,8], [11, 6], [
y = np.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

expected = 5/11
out = dt.gini_score(X, y, 0, 1.5)
print(f'Test 1 passed: {np.isclose(expected, out)}')

expected = 2/5
out = dt.gini_score(X, y, 0, 2.5)
print(f'Test 2 passed: {np.isclose(expected, out)}')

expected = 5/11
out = dt.gini_score(X, y, 1, 2)
print(f'Test 3 passed: {np.isclose(expected, out)}')

expected = 4/9
out = dt.gini_score(X, y, 1, 6.5)
print(f'Test 4 passed: {np.isclose(expected, out)}')

```

Test 1 passed: True
Test 2 passed: True
Test 3 passed: True
Test 4 passed: True

Problem 3.4 (15 points):

Now, you will use your function `gini_score` to compute the best splits. To do this, you will need to complete the function `find_best_split`. This function takes in features `X` and labels `y`, and returns the feature index and threshold corresponding to the best split as determined by `gini_score`. That is, for every feature and every threshold, you should compute the Gini index of splitting that feature at that threshold, and you should return the index of the feature and the threshold that results in the lowest Gini index.

To determine the thresholds, we will use the **midpoint** strategy. That is, given an array of feature values, we will consider all thresholds given by the midpoints between consecutive feature values. Here's an example. Suppose we are given a feature matrix X with four datapoints and two features, given by

$$X = \begin{bmatrix} 1 & 2 \\ 1.5 & 2.5 \\ 0.75 & -1.0 \\ 3.0 & 0.5 \end{bmatrix}.$$

The thresholds to consider when splitting the second feature would then be $[-0.25, 1.25, 2.25]$.

- Complete the function `find_best_split` as detailed above.
- Run the code block given below to test your implementation. If your code is correct, all tests should pass. Note that the numerical tests here are the same as the examples in Lectures 18 and 19, which may be helpful in debugging your code.

```
In [16]: # Use this code block to test your implementation in Problem 3.4
# Don't change anything here -- just run it

dt = DecisionTree(max_depth=3)

# Note: these are the same values as the example in Lectures 18 and 19

X = np.array([[1,1], [2,8], [4,9], [6,7], [7,4], [8,11], [3,3], [5,5], [9,5], [10,8], [11, 6], [
y = np.array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

expected = (0, 8.5)
out = dt.find_best_split(X, y)
print(f'Test 1 passed: {expected == out}')

X_L, y_L = X[X[:, 0] <= 8.5], y[X[:, 0] <= 8.5]
expected = (1, 6)
out = dt.find_best_split(X_L, y_L)
print(f'Test 2 passed: {expected == out}')
```

Test 1 passed: True

Test 2 passed: True

Problem 3.5 (10 points):

For the last step in implementing `DecisionTree`, you will need to complete the function `build_tree`. This function uses the functions you implemented in Problems 3.1-3.4 in order to recursively build your decision tree.

- Complete the function `build_tree`. Fill in the lines of code marked `#TODO`.

- Run the code block given below to test your implementation. If your code is correct, all tests should pass. Note that the *first test* here is the same as the examples in Lectures 18 and 19, which may be helpful in debugging your code.

```
In [17]: # Use this code block to test your implementation in Problem 3.5
# Don't change anything here -- just run it

# Note: these are the same values as the example in Lectures 18 and 19
X = np.array([[1,1], [2,8], [4,9], [6,7], [7,4], [8,11], [3,3], [5,5], [9,5], [10,8], [11, 6], [
y = np.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

expected = np.array([0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1])
dt = DecisionTree(max_depth=3)
dt.fit(X, y)
out = dt.predict(X)
print(f'Test 1 passed: {np.array_equal(expected, out)}')

X, y = make_classification(n_samples = 250, n_features=5, n_informative=3, random_state=seed)
sklearn_dt = DecisionTreeClassifier(max_depth=3, criterion='gini', random_state=seed)
sklearn_dt.fit(X, y)
expected = sklearn_dt.predict(X)
dt = DecisionTree(max_depth=3)
dt.fit(X, y)
out = dt.predict(X)
print(f'Test 2 passed: {np.array_equal(expected, out)}')

Test 1 passed: True
Test 2 passed: True
```

Problem 4: Experimenting with Decision Trees

In the final problem of this assignment, you will experiment with the scikit-learn implementation of decision trees on the Wine dataset. This dataset consists of 178 datapoints with 13 real-valued features and 3 possible labels. The features correspond to a chemical analysis of various wines, and the labels correspond to which type of wine we are considering. See the documentation [here](#) for some more information on this dataset.

Before attempting this problem, you should read and understand the documentation for the `DecisionTreeClassifier`, available [here](#).

Important: for every decision tree classifier you train in this problem, make sure to set `random_state=seed` for reproducibility.

We will first load in this dataset and create a train/test split.

```
In [18]: X, y = load_wine(return_X_y=True)
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.3, random_state=seed)
```

```
In [19]: X_tr.shape
```

```
Out[19]: (124, 13)
```

```
In [20]: X_te.shape
```

Problem 4.1 (5 points):

First, you will see how to fit and visualize a decision tree classifier in sklearn.

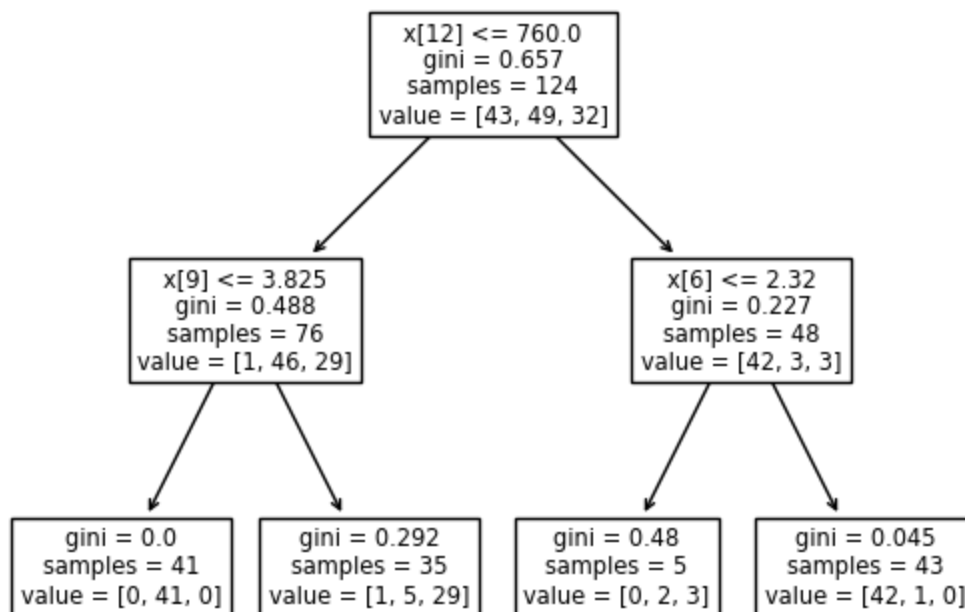
- Using the class `DecisionTreeClassifier` from scikit-learn, train a decision tree on the training data. Use `max_depth=2`, and leave all other settings as their defaults. Note that, by default, `DecisionTreeClassifier` uses the Gini index to split nodes.
- What is the training set error and testing set error of your classifier?
- Use the function `plot_tree` to visualize your decision tree. This is already imported for you at the top of this notebook. See [here](#) for the corresponding documentation page.
- For the first split of the data, what feature and what threshold is being used in your classifier? You don't need to find the name of this feature, just its index, i.e. which column of `X`.

```
In [25]: clf = DecisionTreeClassifier(random_state=seed, max_depth=2).fit(X_tr, y_tr)
y_trpred = clf.predict(X_tr)
traccuracy = accuracy_score(y_tr, y_trpred)
trerror = 1-traccuracy
print(f'Training set error: {trerror}')
y_teprde = clf.predict(X_te)
teaccuracy = accuracy_score(y_te, y_teprde)
testerror = 1-teaccuracy
print(f'Testing set error: {testerror}')
plot_tree(clf)
print('For the first split of the data, index feature 12 and threshold 760.0 is being used in th
```

Training set error: 0.07258064516129037

Testing set error: 0.18518518518518523

For the first split of the data, index feature 12 and threshold 760.0 is being used in the classifier



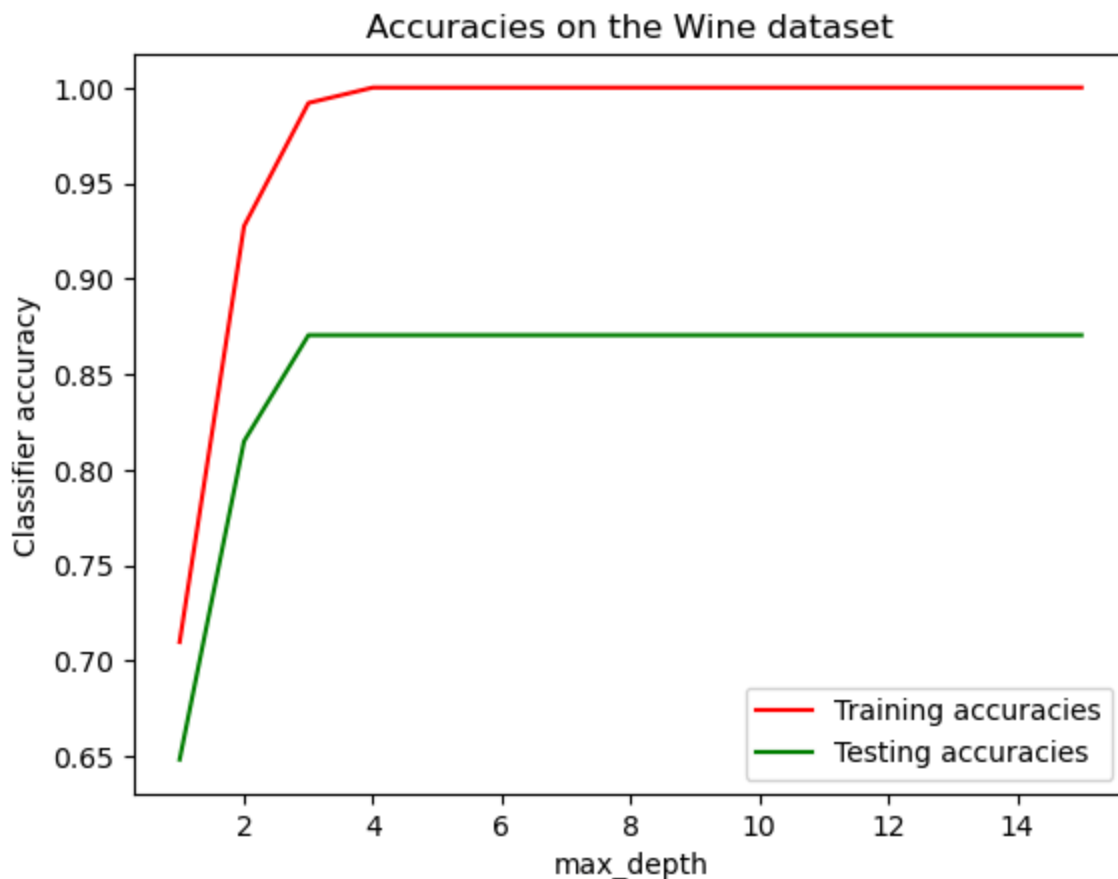
Problem 4.2 (5 points):

You will now vary the maximum depth allowed in your decision tree and see what effect this has on the error rate.

- Train a decision tree for every value of `max_depth` in `[1, 2, ..., 15]`. Use the default settings (other than `max_depth`).
- Plot the resulting training and testing set accuracies as a function of depth. Be sure to include an x-label, a y-label, and a legend in your plot.
- Describe what you see happen as you increase the depth.

```
In [29]: traccuracies=[]
teaccuracies=[]
x_values = range(1, 16)
for md in x_values:
    clf = DecisionTreeClassifier(random_state=seed, max_depth=md).fit(X_tr, y_tr)
    y_trpred = clf.predict(X_tr)
    traccuracy = accuracy_score(y_tr, y_trpred)
    traccuracies.append(traccuracy)
    y_tepred = clf.predict(X_te)
    teaccuracy = accuracy_score(y_te, y_tepred)
    teaccuracies.append(teaccuracy)
figure, axes = plt.subplots(1)
axes.plot(x_values, traccuracies, label= 'Training accuracies', color= 'red')
axes.plot(x_values, teaccuracies, label= 'Testing accuracies', color= 'green')
axes.set_xlabel('max_depth')
axes.set_ylabel('Classifier accuracy')
axes.set_title('Accuracies on the Wine dataset')
axes.legend()
```

Out[29]: <matplotlib.legend.Legend at 0x21bceaba3d0>



Both training and testing accuracies increase as the depth increases. However, after a certain depth, both accuracies, implying that this dataset does not require further splitting beyond that depth. Additionally, after

a depth of 4, all labels at each node belong to the same class.

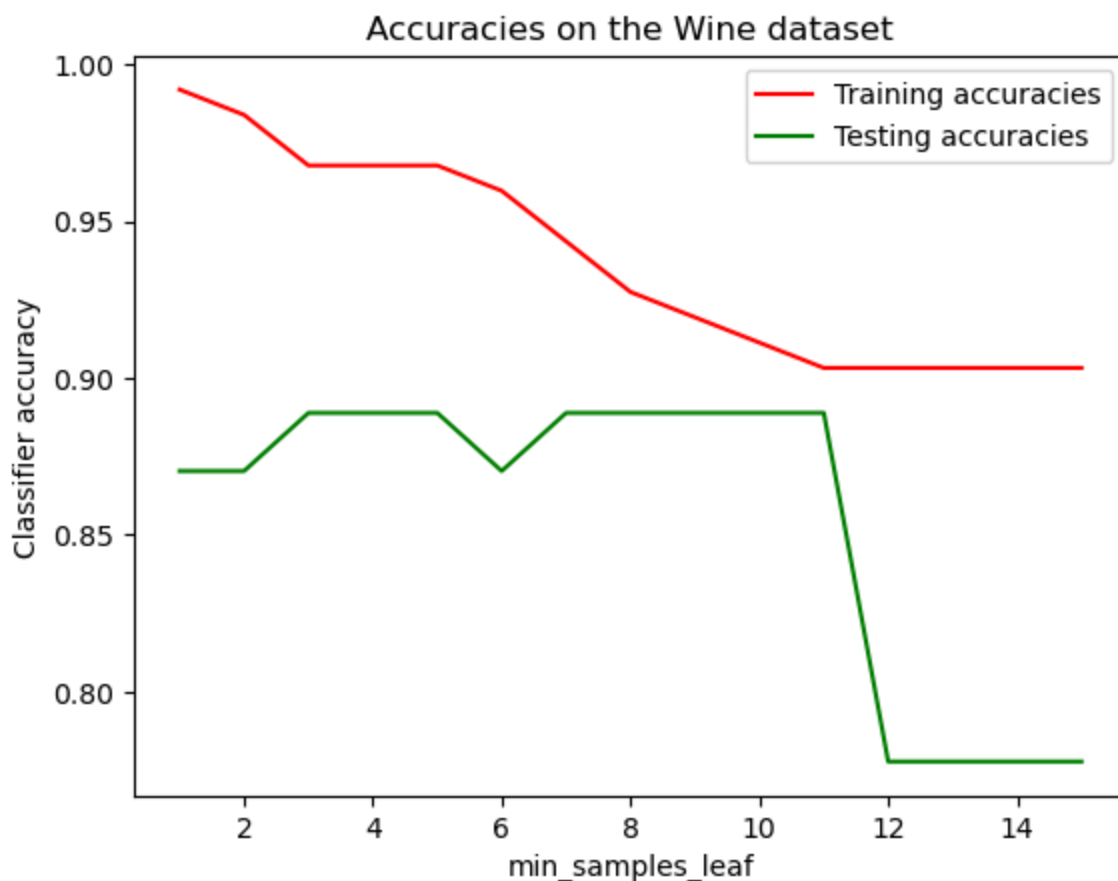
Problem 4.3 (5 points):

Lastly, you will vary the minimum number of datapoints allowed in a leaf node in your decision tree and see what effect this has on the error rate.

- Train a decision tree for every value of `min_samples_leaf` in `[1, 2, ..., 15]`. Use `max_depth=3` and use the default settings for all other parameters.
- Plot the resulting training and testing set accuracies as a function of the minimum leaf samples. Be sure to include an x-label, a y-label, and a legend in your plot.
- You should see that the training error increases as we increase the minimum number of leaf samples. Give an explanation for why this might happen.

```
In [30]: traccuracies=[]
teaccuracies=[]
x_values = range(1, 16)
for msl in x_values:
    clf = DecisionTreeClassifier(random_state=seed, max_depth=3,min_samples_leaf=msl).fit(X_tr,
y_trpred = clf.predict(X_tr)
traccuracy = accuracy_score(y_tr, y_trpred)
traccuracies.append(traccuracy)
y_tepred = clf.predict(X_te)
teaccuracy = accuracy_score(y_te, y_tepred)
teaccuracies.append(teaccuracy)
figure, axes = plt.subplots(1)
axes.plot(x_values, traccuracies ,label= 'Training accuracies', color= 'red')
axes.plot(x_values, teaccuracies ,label= 'Testing accuracies', color= 'green')
axes.set_xlabel('min_samples_leaf')
axes.set_ylabel('Classifier accuracy')
axes.set_title('Accuracies on the Wine dataset')
axes.legend()
```

```
Out[30]: <matplotlib.legend.Legend at 0x21bceb7fad0>
```



The training error increases as we increase the minimum number of leaf samples because increasing this minimum can decrease overfitting, thereby reducing model complexity and decreasing the depth of the tree. Therefore, the training error will increase.

Statement of Collaboration (5 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed. If you did not collaborate you can simply write N/A.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Piazza) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to Piazza, etc.). Especially after you have started working on the assignment, try to restrict the discussion to Piazza as much as possible, so that there is no doubt as to the extent of your collaboration.

N/A