Homework 5

Due: **October 29th, 5pm** (late submission until November 1st, 5pm -- no submission possible afterwards)

Written assignment: 5 points

Coding assignment: 25 points

Project report: 10 points

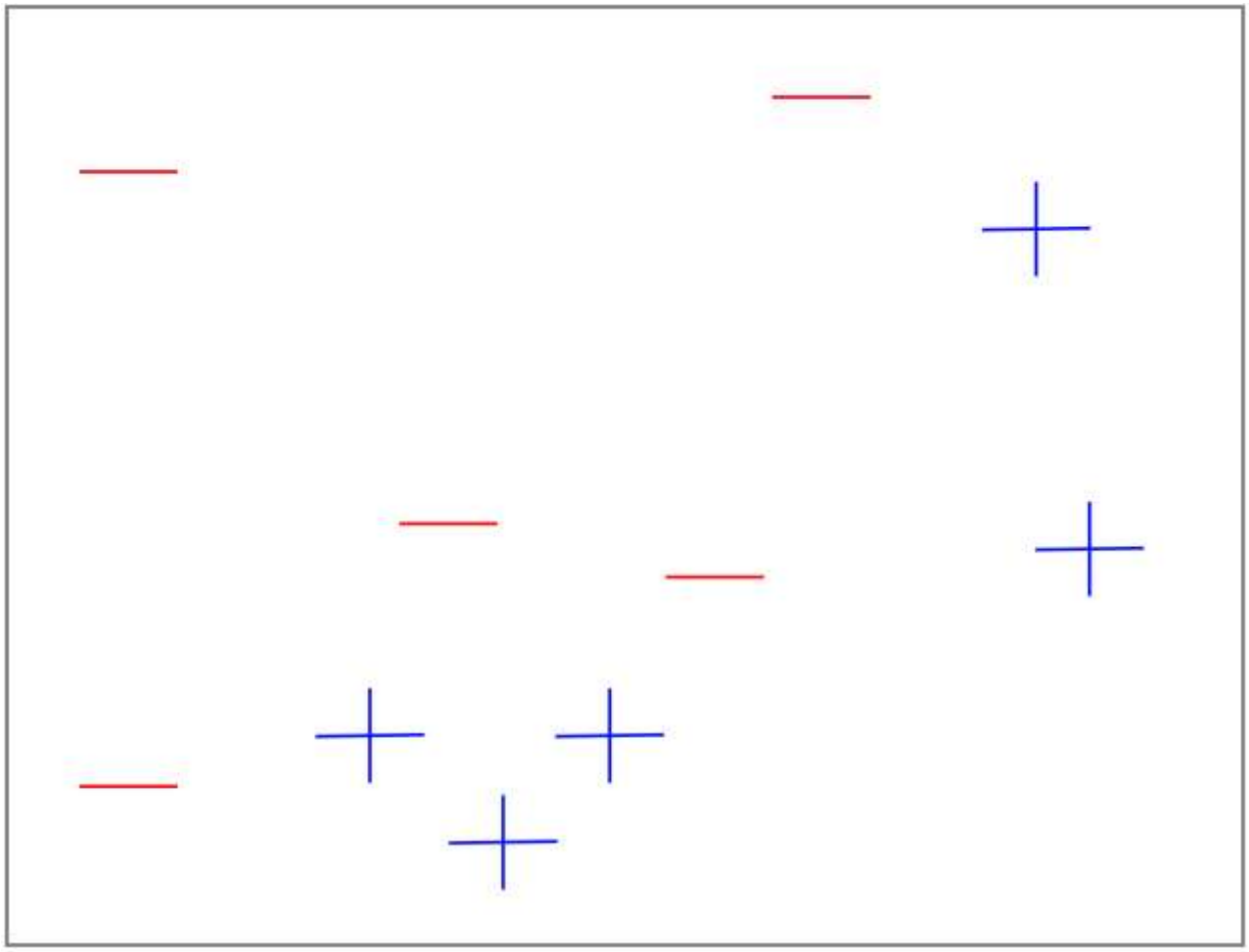
Name: [Yawen Tan]

Link to the github repo: [https://github.com/IsabellaTan/Brown-DATA2060-HW5#]

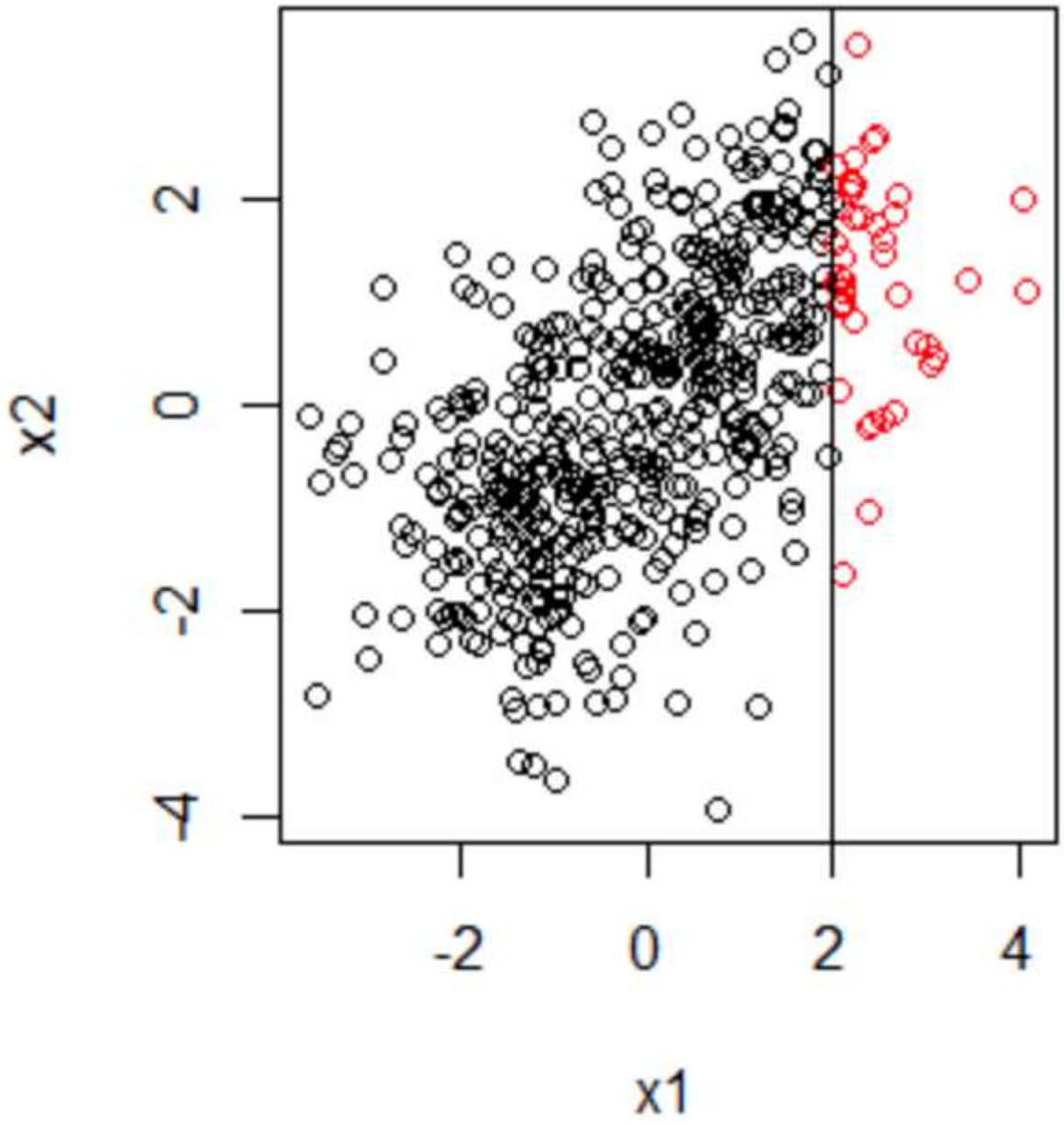
Written Assignment (5 points)

AdaBoost and Decision Stumps

Below is a graph of points, which are classified as `+` or `-`. The features are the x and y coordinates.



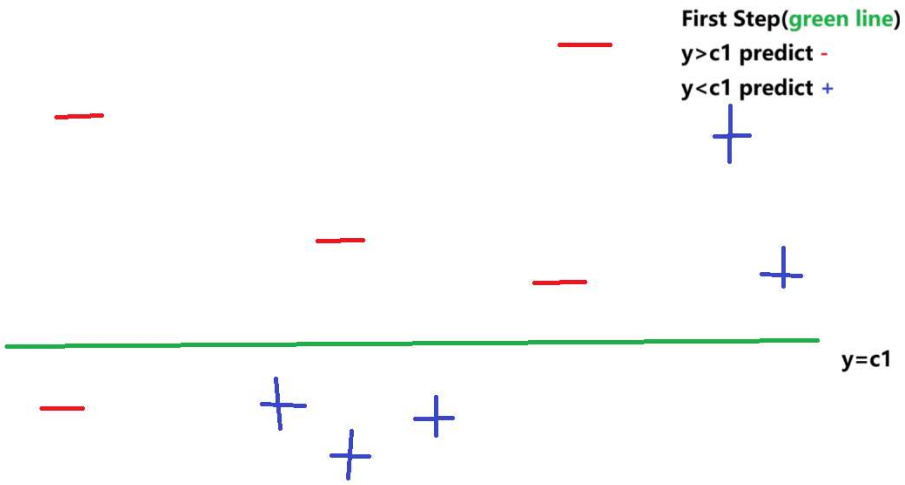
Note: Decision stumps are 1D decision trees. For continuous features (in our case, x and y values) a threshold value is chosen, and there is a chosen classification for inputs falling above or below that threshold. For example, one decision stump may be the vertical line represented `x > 2`, with values satisfying this (i.e., to the right of the line) being classified one way, and values not satisfying this (i.e., to the left of the line) being classified separately. Note that decision stumps are **not** equivalent to halfspaces. See example below.

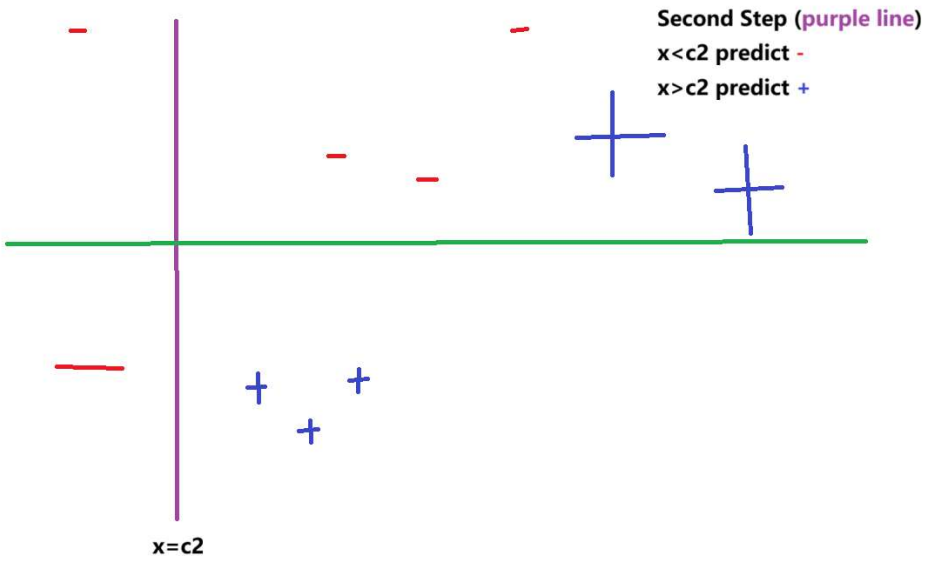


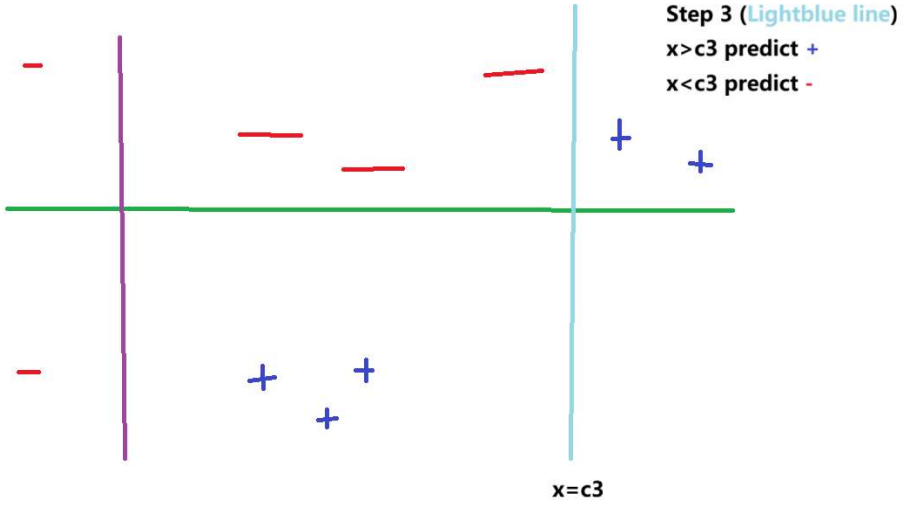
Question 1

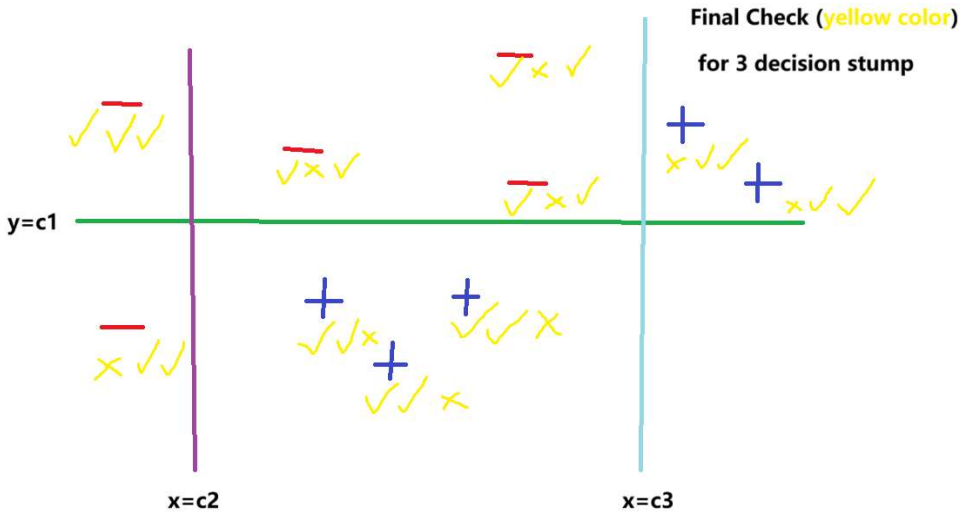
Using decision stumps, produce images representing a simulation of the Adaboost algorithm on this dataset. Represent the changing relative weight of each datapoint by enlarging or shrinking the symbol (+/-). Each image should represent one step of the algorithm, and the last image should be the final classification of the ensemble. Recall that decision stumps can only be vertical or horizontal.

Solution:




Second Step (purple line)
x<c2 predict -
x>c2 predict +
x=c2


Step 3 (Lightblue line)
x>c3 predict +
x<c3 predict -
x=c3



In Step 1, the first decision stump (green horizontal line, y = c1) separates the data roughly by the y-coordinate: samples above the line are predicted as "−" and those below as "+". But several points near the boundary are misclassified.

In Step 2, AdaBoost increases the weights(enlarge) of those misclassified samples so that the next weak classifier focuses more on them. A second decision stump (purple vertical line, x = c2) is added. This stump divides the plane vertically: samples to the left of the line are predicted "−", and those to the right are predicted "+". And it still remains incorrectly classified.

In Step 3, the algorithm again emphasizes(enlarge) the remaining misclassified points. A third decision stump (light-blue vertical line, x = c3) is introduced on the right side to correct these errors. Now the combination of three stumps—two vertical and one horizontal—successfully separates all the red "−" and blue "+" samples.

Finally, the Final Check panel confirms that with these three decision stumps, all points are correctly classified (as shown by the yellow check marks).

Programming Assignment (25 points)

Introduction

In this assignment, you will be implementing Decision Trees to solve binary classification problems. By the end of this assignment, you will have a classifier that you will use to predict the result of chess matches and classify emails as spam. We recommend that you start this assignment early, as the logic you have to implement may be complicated.

Stencil Code

We have provided the following stencil code within this file:

- `Models` contains the `DecisionTree` class, which will contain the bulk of the code you write.
- `Check Model` contains a series of tests to ensure you are coding your model properly.
- `Main` is the entry point of your program, which will read in the data, run the classifiers and print the results. You will make small modifications to this file.
- `Get Data` contains the data loading and processing. You do **not** need to change this file.

You should *not* modify any code in `Get Data`. All the functions you need to fill in reside in `Main` and `Models`, marked by `TODO`s.

Data

Spambase Dataset

You will be testing your Decision Trees on a real world dataset, the Spambase dataset. Our goal is to train a model that can classify whether an email is spam or not. The dataset features attributes such as the frequency of certain words and the amount of capital letters in a given message. You can find more details on the dataset **here**. We will only be using a subset of the full dataset.

Chess Dataset

Each row of the `chess.csv` dataset contains 36 features, which represent the current state of the chess board. Given this representation, the task is to use the Decision Trees to classify whether or not it is possible for white to win the game. For more information on the dataset, see **here**).

We have taken care of all the data preprocessing required so that you can focus on implementing the machine-learning algorithms! We hope that from these two examples, you can understand the versatility and power of your abstract decision tree

classifier.

The Assignment

Decision Trees

Part I: Generic Decision Trees in Python

In this part, you will be implementing a generic decision tree for binary classification given binary features. Your decision tree will take training data $S = ((\mathbf{x}_1, y_1) \ldots (\mathbf{x}_m, y_m))$---where $\mathbf{x}_i \in \{0,1\}^d$ represents the binary feature vectors and $y \in \{0,1\}$ are the class labels---and attempt to find a tree that minimizes training error. Recall that the training error for a hypothesis h is defined as the *average* $0-1$ loss

$$L_S(h) = \frac{1}{m} \sum_{(\mathbf{x},y) \in S} (y \neq h(\mathbf{x})).$$

The primary methods of the `DecisionTree` class are as follows:

- **Functionality:**
  - `DecisionTree(data, validation_data=None, gain_function=node_score_entropy, max_depth=40)` creates a `DecisionTree` that greedily minimizes training error on the given dataset. The depth of the tree should not be greater than `max_depth`. If **`validation\_data`** is passed as an argument, the validation data should be used to prune the tree after it has been constructed.
  - `predict(features)` predicts a label $y \in \{0,1\}$ given $\text{features} \in \{0,1\}^d$. Note that in our implementation features are represented as Python `bool` types (`True`, `False`) and class labels are Python `int`s (`0, 1`).
  - `accuracy(data)` computes accuracy, defined as $1 -$ **`loss(self, data)`**.
  - `loss(data)` computes the training error, or the *average* loss, $L_{\texttt{data}}(h)$.

- **Helper functions:** This is where most of the algorithmic work will take place. Each helper function begins with an underscore: `_predict_recurs, _prune_recurs, _is_terminal, _split_recurs, _calc_gain`. We already implemented `_predict_recurs` for you, so please do not modify that function. You should implement the other helper functions without changing the function signatures. **Note that when splitting on the i-th feature, the left child will have data points with i-th feature 0 and the right child will have data points with i-th feature 1.**

- **Debugging:** We have given you two functions to help you visualize your decision tree for debugging purposes. You are free to use (or not use) them. We will not be grading you on whether you use or modify this code.

  - `print_tree()` prints the tree to the command line. We have provided a working implementation, which you are free to improve. The current tree visualization works best for very shallow trees.

  - `loss_plot_vec(data)` returns a vector of loss values where the i-th element corresponds to the loss of the tree with i nodes. The result can be plotted with `matplotlib.pyplot` to visualize the loss as your tree expands.

  To use these debugging functions, you must store the maximum gain at each node and the number of data points that have made it to that node when training using _set_info().

Your task is to ensure that the `DecisionTree` class is fully implemented. If you are unsure where to begin, we have provided `TODO` comments in the stencil code to help get you started! We recommend testing your code incrementally. It would be easiest to program `_is_terminal, _calc_gain` and one of the gain functions first as they are all needed in `_split_recurs`. You should start working on pruning at the last step when you are sure that other functions work. You are free to write your own tests for any of the provided functions to ensure that they are working correctly.

Part II: Measures of Gain

As mentioned in lecture, there are multiple measures of gain that an algorithm can use when determining on which feature to split the current node. In this assignment, you will be implementing and comparing the results of three measures of gain: decrease in training error, information gain (entropy) and Gini index. We recommend reviewing the lecture slides or textbook if these terms sound unfamiliar.

The `DecisionTree` class takes an optional `gain_function` parameter. This function will be one of the three functions left for you to implement: `node_score_error`, `node_score_entropy` and `node_score_gini`. All of these gain functions should return a float.

Part III: Chess Predictions & Spam Classification

Once you have implemented the `DecisionTree` class, you are ready to explore the chess and spam datasets! You should now write code in `main` that will print the following loss values:

- For each dataset (`chess.csv`, `spam.csv`)
  - For each gain function (Training error, Entropy, Gini)
    - Print training loss without pruning
    - Print test loss without pruning
    - Print training loss with pruning
    - Print test loss with pruning

Your final program should print **exactly** 24 lines of output. Each line may contain text, but should end with the loss values defined above.

In [1]:
```
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
```

```
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.11"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.11"):
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'matplotlib': "3.10.5", 'numpy': "2.3.2",'sklearn': "1.7.1",
                'pandas': "2.3.2", 'pytest': "8.4.1", 'torch':"2.7.1"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.11

[ OK ] matplotlib version 3.10.5 is installed.
[ OK ] numpy version 2.3.2 is installed.
[ OK ] sklearn version 1.7.1 is installed.
[ OK ] pandas version 2.3.2 is installed.
[ OK ] pytest version 8.4.1 is installed.
[ OK ] torch version 2.7.1 is installed.
```

Get Data

In [2]:

```python
import random
import csv
import numpy as np

#####################################################################################################################
# Data Processing Section
# Helper function for preparing data for a decision tree classifiction problem. Parsing the data such
# that for each feature, the property can only either be True or False. Label can only be 1 or 0.
# For the chess.csv dataset won=1, nowin=0
# In more detail:
# Dataset with n instances, for each instance, there are m attributes. For the i-th attribute,
# the property should be chosen from a set with size of m_i to represent the information.
# Input: array with size of n*(m+1), the first column is the label
# Output: array with size of n*(m_1 + m_2 + ... + m_m + 1), the first column is 1 or 0 corresponding to label
#####################################################################################################################

def get_data(filename, class_name):
    data = read_data(filename)
    data = convert_to_binary_features(data, class_name)
    return np.array(split_data(data), dtype=object)

def read_data(filename):
    data = []
    with open(filename) as f:
        reader = csv.reader(f)
        for row in reader:
            data.append(row)
    return data
```

```
def convert_to_binary_features(data, class_name):
    features = []
    for feature_index in range(0, len(data[0])-1):
        feature_values = list(set([obs[feature_index] for obs in data]))
        feature_values.sort()
        if len(feature_values) > 2: features.append(feature_values[:-1])
        else: features.append([feature_values[0]])
    new_data = []
    for obs in data:
        new_obs = [1 if obs[-1] == class_name else 0] # label = 1 if label in the dataset is won
        for feature_index in range(0, len(data[0]) - 1):
            current_feature_value = obs[feature_index]
            for possible_feature_value in features[feature_index]:
                new_obs.append(current_feature_value == possible_feature_value)
        new_data.append(new_obs)

    return new_data

def split_data(data, num_training=1000, num_validation=1000):
    random.shuffle(data)
    # casting to a numpy array
    data = np.array(data)
    return data[0:num_training], data[num_training:num_training + num_validation], data[num_training + num_validation:len(data
```

Model

In [ ]:

```
import copy
import math

def node_score_error(prob):
    '''
        TODO:
        Calculate the node score using the train error of the subdataset and return it.
        For a dataset with two classes, C(p) = min{p, 1-p}
    '''
    return min(prob, 1 - prob)

def node_score_entropy(prob):
```

```
    '''
        TODO:
        Calculate the node score using the entropy of the subdataset and return it.
        For a dataset with 2 classes, C(p) = -p * log(p) - (1-p) * log(1-p)
        For the purposes of this calculation, assume 0*log0 = 0.
        HINT: remember to consider the range of values that p can take!
    '''
    # HINT: If p < 0 or p > 1 then entropy = 0
    if prob <= 0 or prob >= 1:
        return 0
    return -prob * math.log(prob) - (1 - prob) * math.log(1 - prob)


def node_score_gini(prob):
    '''
        TODO:
        Calculate the node score using the gini index of the subdataset and return it.
        For dataset with 2 classes, C(p) = 2 * p * (1-p)
    '''
    return 2 * prob * (1 - prob)

class Node:
    '''
    Helper to construct the tree structure.
    '''
    def __init__(self, left=None, right=None, depth=0, index_split_on=0, isleaf=False, label=1):
        self.left = left
        self.right = right
        self.depth = depth
        self.index_split_on = index_split_on
        self.isleaf = isleaf
        self.label = label
        self.info = {} # used for visualization


    def _set_info(self, gain, num_samples):
        '''
        Helper function to add to info attribute.
        You do not need to modify this.
        '''
```

```
        self.info['gain'] = gain
        self.info['num_samples'] = num_samples


class DecisionTree:

    def __init__(self, data, validation_data=None, gain_function=node_score_entropy, max_depth=40):

        # TODO: find majority class; set to class 0 if exactly balanced. This is the default label of your root node.
        # Make sure to assign it to an instance variable `majority_class`.

        # Find prediction y first
        y = [row[0] for row in data]
        # Find number of class 1 and class 0
        num_class_1 = y.count(1)
        num_class_0 = y.count(0)

        if num_class_1 > num_class_0:
            self.majority_class = 1
        else:
            self.majority_class = 0

        self.max_depth = max_depth
        self.root = Node(label = self.majority_class)
        self.gain_function = gain_function

        indices = list(range(1, len(data[0])))

        self._split_recurs(self.root, data, indices)

        # Pruning
        if validation_data is not None:
            self._prune_recurs(self.root, validation_data)


    def predict(self, features):
        '''
        Helper function to predict the label given a row of features.
        You do not need to modify this.
        '''
        return self._predict_recurs(self.root, features)
```

```
def accuracy(self, data):
    '''
    Helper function to calculate the accuracy on the given data.
    You do not need to modify this.
    '''
    return 1 - self.loss(data)


def loss(self, data):
    '''
    Helper function to calculate the loss on the given data.
    You do not need to modify this.
    '''
    cnt = 0.0
    test_Y = [row[0] for row in data]
    for i in range(len(data)):
        prediction = self.predict(data[i])
        if (prediction != test_Y[i]):
            cnt += 1.0
    return cnt/len(data)


def _predict_recurs(self, node, row):
    '''
    Helper function to predict the label given a row of features.
    Traverse the tree until leaves to get the label.
    You do not need to modify this.
    '''
    if node.isleaf or node.index_split_on == 0:
        return node.label
    split_index = node.index_split_on
    if not row[split_index]:
        return self._predict_recurs(node.left, row)
    else:
        return self._predict_recurs(node.right, row)


def _prune_recurs(self, node, validation_data):
    '''
```

```
        TODO:
        Prune the tree bottom up recursively. Nothing needs to be returned.
        Do not prune if the node is a leaf.
        Do not prune if the node is non-leaf and has at least one non-leaf child.
        Prune if deleting the node could reduce loss on the validation data.
        NOTE:
        This might be slightly different from the pruning described in lecture.
        Here we won't consider pruning a node's parent if we don't prune the node
        itself (i.e. we will only prune nodes that have two leaves as children.)
        HINT: Think about what variables need to be set when pruning a node!
        '''
        # Checks if node is not a Leaf
        if not node.isleaf:
            # determine whether to prune left and right subtrees first
            if node.left is not None:
                # TODO: Prune node.left
                self._prune_recurs(node.left, validation_data)
            if node.right is not None:
                # TODO: Prune node.right
                self._prune_recurs(node.right, validation_data)
            # if both children are leaves, consider pruning this node
            if (node.left.isleaf) and (node.right.isleaf):
                # TODO: Prune node only if loss is reduced
                # Compute loss before pruning
                loss_before = self.loss(validation_data)
                # Save old values
                old_isleaf = node.isleaf
                old_left = node.left
                old_right = node.right
                old_index = node.index_split_on
                # Prune and change node to leaf
                node.isleaf = True
                node.left = None
                node.right = None
                node.index_split_on = 0
                # Compute loss after pruning
                loss_after = self.loss(validation_data)
                # If loss increases, revert the changes
                if loss_after > loss_before:
                    node.isleaf = old_isleaf
                    node.left = old_left
```

```
                node.right = old_right
                node.index_split_on = old_index


    def _is_terminal(self, node, data, indices):
        '''
        TODO:
        Helper function to determine whether the node should stop splitting.
        Stop the recursion if:
            1. The dataset (as passed to parameter data) is empty.
            2. There are no more indices to split on.
            3. All the instances in this dataset belong to the same class
            4. The depth of the node reaches the maximum depth.
        Set the node label to be the majority label of the training dataset if:
            1. The number of class 1 points is equal to the number of class 0 points.
            2. The dataset is empty.
        Return:
            - A boolean, True indicating the current node should be a leaf and
              False if the node is not a leaf.
            - A label, indicating the label of the leaf (or the label the node would
              be if we were to terminate at that node). If there is no data left, you
              must return the majority class of the training set.
        '''
        y = [row[0] for row in data]

        # TODO: Check Cases if the node should stop splitting
        # 1. The dataset (as passed to parameter data) is empty.
        if len(y) == 0:
            return True, self.majority_class
        # 2. There are no more indices to split on.
        if len(indices) == 0:
            num_class_1 = y.count(1)
            num_class_0 = y.count(0)
            if num_class_1 > num_class_0:
                return True, 1
            elif num_class_0 > num_class_1:
                return True, 0
            else:
                return True, self.majority_class
        # 3. All the instances in this dataset belong to the same class
        if all(label == 1 for label in y):
```

```
            return True, 1
        if all(label == 0 for label in y):
            return True, 0
        # 4. The depth of the node reaches the maximum depth.
        if node.depth >= self.max_depth:
            num_class_1 = y.count(1)
            num_class_0 = y.count(0)
            if num_class_1 > num_class_0:
                return True, 1
            elif num_class_0 > num_class_1:
                return True, 0
            else:
                return True, self.majority_class


        # TODO: Check cases if the node should be set to the majority label of the training dataset
        num_class_1 = sum(y)
        num_class_0 = len(y) - num_class_1
        if num_class_1 > num_class_0:
            return False, 1
        elif num_class_0 > num_class_1:
            return False, 0
        else:
            return False, self.majority_class


    def _split_recurs(self, node, data, indices):
        '''
        TODO:
        Recursively split the node based on the rows and indices given.
        Nothing needs to be returned.

        First use _is_terminal() to check if the node needs to be split.
        If so, select the column that has the maximum infomation gain to split on.
        Store the label predicted for this node, the split column, and use _set_info()
        to keep track of the gain and the number of datapoints at the split.
        Then, split the data based on its value in the selected column.
        The data should be recursively passed to the children.
        '''
        # TODO: Check if node needs to be split
        should_stop, label = self._is_terminal(node, data, indices)
```

```
        node.label = label # Set node label
        # if should stop, set node as leaf and return
        if should_stop:
            node.isleaf = True
            node.index_split_on = 0  # When isleaf, index_split_on = 0
            node._set_info(gain=0.0, num_samples=len(data))
            return

        if not node.isleaf:
            # TODO: Initialize and Calculate Gain
            best_gain = float('-inf')
            best_index = None
            for idx in indices:
                gain = self._calc_gain(data, idx, self.gain_function)
                if gain > best_gain:
                    best_gain = gain
                    best_index = idx
            if best_index is None:
                node.isleaf = True
                node.index_split_on = 0
                node._set_info(gain=0.0, num_samples=len(data))
                return

            # TODO: Split the column and use _set_info()
            # set node info
            node.isleaf = False
            node.index_split_on = best_index
            node._set_info(gain=best_gain, num_samples=len(data))

            # TODO: Split the Data and Pass it recursively to the  children
            # Split data: Left: feature is False; Right: feature is True
            left_data = [row for row in data if not row[best_index]]
            right_data = [row for row in data if row[best_index]]
            # Find the indices for child nodes
            child_indices = [i for i in indices if i != best_index]
            # Create left and right child nodes
            node.left = Node()
            node.right = Node()
            # Intialize child and set depth and label
            node.left.depth = node.depth + 1
            node.left.label = label
```

```
            node.right.depth = node.depth + 1
            node.right.label = label
            # Recursively split left and right children
            self._split_recurs(node.left, left_data, child_indices)
            self._split_recurs(node.right, right_data, child_indices)


    def _calc_gain(self, data, split_index, gain_function):
        '''
        TODO:
        Calculate the gain of the proposed splitting and return it.
        Gain = C(P[y=1]) - P[x_i=True] * C(P[y=1|x_i=True]) - P[x_i=False] * C(P[y=0|x_i=False])
        Here the C(p) is the gain_function. For example, if C(p) = min(p, 1-p), this would be
        considering training error gain. Other alternatives are entropy and gini functions.
        '''
        y = [row[0] for row in data]
        xi = [row[split_index] for row in data]

        if len(y) != 0 and len(xi) != 0:
            # TODO: Calculate Gain
            # Calculate P[y=1]
            p_y1 = sum(y) / len(y)
            # Separate y
            true_y = [y[i] for i in range(len(y)) if xi[i] == True]
            false_y = [y[i] for i in range(len(y)) if xi[i] == False]
            # Calculate P[x_i=True] and P[x_i=False]
            p_true = len(true_y) / len(y)
            p_false = len(false_y) / len(y)
            # Calculate P[y=1 | x_i=True] and P[y=1 | x_i=False]
            if len(true_y) > 0:
                p_y1_true = sum(true_y) / len(true_y)
            else:
                p_y1_true = 0  # if empty, set to 0
            if len(false_y) > 0:
                p_y1_false = sum(false_y) / len(false_y)
            else:
                p_y1_false = 0

            # -Calculate Gain
            gain = gain_function(p_y1) - p_true * gain_function(p_y1_true) - p_false * gain_function(p_y1_false)
```

```
        else:
            gain = 0
        return gain


    def print_tree(self):
        '''
        Helper function for tree_visualization.
        Only effective with very shallow trees.
        You do not need to modify this.
        '''
        print('---START PRINT TREE---')
        def print_subtree(node, indent=''):
            if node is None:
                return str("None")
            if node.isleaf:
                return str(node.label)
            else:
                decision = 'split attribute = {:d}; gain = {:f}; ' \
                'number of samples = {:d}'.format(
                    node.index_split_on, node.info['gain'], node.info['num_samples'])
            left = indent + '0 -> '+ print_subtree(node.left, indent + '\t\t')
            right = indent + '1 -> '+ print_subtree(node.right, indent + '\t\t')
            return (decision + '\n' + left + '\n' + right)

        print(print_subtree(self.root))
        print('----END PRINT TREE---')


    def loss_plot_vec(self, data):
        '''
        Helper function to visualize the loss when the tree expands.
        You do not need to modify this.
        '''
        self._loss_plot_recurs(self.root, data, 0)
        loss_vec = []
        q = [self.root]
        num_correct = 0
        while len(q) > 0:
            node = q.pop(0)
            num_correct = num_correct + node.info['curr_num_correct']
```

```
            loss_vec.append(num_correct)
            if node.left != None:
                q.append(node.left)
            if node.right != None:
                q.append(node.right)

        return 1 - np.array(loss_vec)/len(data)


    def _loss_plot_recurs(self, node, rows, prev_num_correct):
        '''
        Helper function to visualize the loss when the tree expands.
        You do not need to modify this.
        '''
        labels = [row[0] for row in rows]
        curr_num_correct = labels.count(node.label) - prev_num_correct
        node.info['curr_num_correct'] = curr_num_correct

        if not node.isleaf:
            left_data, right_data = [], []
            left_num_correct, right_num_correct = 0, 0
            for row in rows:
                if not row[node.index_split_on]:
                    left_data.append(row)
                else:
                    right_data.append(row)

            left_labels = [row[0] for row in left_data]
            left_num_correct = left_labels.count(node.label)
            right_labels = [row[0] for row in right_data]
            right_num_correct = right_labels.count(node.label)

            if node.left != None:
                self._loss_plot_recurs(node.left, left_data, left_num_correct)
            if node.right != None:
                self._loss_plot_recurs(node.right, right_data, right_num_correct)
```

Check Model

In [7]:

```
import pytest
np.random.seed(0)
random.seed(0)

# Tests for node_score_error
assert node_score_error(.3) == .3
assert node_score_error(.6) == .4

# Tests for node_score_entropy
assert node_score_entropy(.5) == pytest.approx(.69, .01)
assert node_score_entropy(0) == node_score_entropy(1) == 0
assert node_score_entropy(.7) == pytest.approx(.61,.01)

# Tests for node_score_gini
assert node_score_gini(1) == node_score_gini(0) == 0
assert node_score_gini(.4) == .48

### Finish TODO's in DecisionTree then run tests below

# Creates Test Model and Dummy Data
x = np.array([[0,1,0,0],[1,0,1,1],[1,1,0,1],[0,0,1,0],[0,1,1,1],[0,0,0,0]])
test_model = DecisionTree(x)

# Test for majority_class
assert test_model.majority_class == 0
# test checking that the first column is used to calculate the majority class
x_majority_class_test = np.array([[0,1,1,1],[1,0,1,1],[1,1,0,1],[0,0,1,0],[0,1,1,1],[0,0,0,0]])
majority_class_test_model = DecisionTree(x_majority_class_test)
assert majority_class_test_model.majority_class == 0

# Tests for _is_terminal
node1 = Node(left=None, right=None, depth=0, index_split_on=3, isleaf=False, label=0)
x_filtered_node2 = np.array([row for row in x if row[3] == 1])
node2 = Node(left=None, right=None, depth=1, index_split_on=1, isleaf=False, label=1)
x_filtered_node3 = np.array([row for row in x_filtered_node2 if row[1] == 1])
node3 = Node(left=None, right=None, depth=2, index_split_on=2, isleaf=False, label=0)
x_filtered_node4 = np.array([row for row in x_filtered_node3 if row[2] == 1])
node4 = Node(left=None, right=None, depth=3, index_split_on=None, isleaf=True, label=0)
```

```
assert test_model._is_terminal(node=node1, data=x, indices=[1, 2, 3]) == (False, 0)
assert test_model._is_terminal(node=node2, data=x_filtered_node2, indices=[1, 2]) == (False, 1)
assert test_model._is_terminal(node=node3, data=x_filtered_node3, indices=[2]) == (False, 0)
assert test_model._is_terminal(node=node4, data=x_filtered_node4, indices=[]) == (True, 0)


# Tests _calc_gain
# Testing gain for index 3
assert test_model._calc_gain(x, 3, node_score_error) == pytest.approx(0.166, .01)
assert test_model._calc_gain(x, 3, node_score_entropy) == pytest.approx(0.318, .01)
assert test_model._calc_gain(x, 3, node_score_gini) == pytest.approx(0.222, .01)

# Testing gain for index 1
assert test_model._calc_gain(x_filtered_node2, 1, node_score_error) == pytest.approx(5.551115123125783e-17, abs=1e-18)
assert test_model._calc_gain(x_filtered_node2, 1, node_score_entropy) == pytest.approx(0.174, .01)
assert test_model._calc_gain(x_filtered_node2, 1, node_score_gini) == pytest.approx(0.111, .01)

# Testing gain for index 2
assert test_model._calc_gain(x_filtered_node3, 2, node_score_error) == pytest.approx(0.5, .01)
assert test_model._calc_gain(x_filtered_node3, 2, node_score_entropy) == pytest.approx(0.693, .01)
assert test_model._calc_gain(x_filtered_node3, 2, node_score_gini) == pytest.approx(0.5, .01)

# testing gain for case when - P[x_i=False] * C(P[y=0|x_i=False] is nonzero
new_dummy_data = [[0,1,0,],[1,0,1],[1,1,0],[0,0,1],[0,1,1],[0,0,0]]
# testing gain for index 1
assert test_model._calc_gain(new_dummy_data, 1, node_score_error) == pytest.approx(0.0, .01)
assert test_model._calc_gain(new_dummy_data, 1, node_score_entropy) == pytest.approx(0.0, .01)
assert test_model._calc_gain(new_dummy_data, 1, node_score_gini) == pytest.approx(0.0, .01)
# testing gain for index 2
assert test_model._calc_gain(new_dummy_data, 1, node_score_error) == pytest.approx(0.0, .01)
assert test_model._calc_gain(new_dummy_data, 2, node_score_entropy) == pytest.approx(0.0, .01)
assert test_model._calc_gain(new_dummy_data, 2, node_score_gini) == pytest.approx(0.0, .01)


# Check Tree is created Properly, Compare with text below
test_model.print_tree()

# Tests _prune_recurs
# Pruned tree should be smaller
# with higher training loss and lower validation loss
x_val = np.array([[1,1,1,1],[1,0,0,1]])
```

```
print('training loss not pruned:', test_model.loss(x))
print('validation loss not pruned:', test_model.loss(x_val), '\n')

test_model_pruned = DecisionTree(x,validation_data=x_val)
test_model_pruned.print_tree()
print('training loss pruned:', test_model_pruned.loss(x))
print('validation loss pruned:', test_model_pruned.loss(x_val))

# checking that the pruned tree root is not a leaf and has the correct label
assert test_model_pruned.root.label == 0
assert not test_model_pruned.root.isleaf
# the pruned tree should have leaves with correct labels (tests to confirm the visual printout of the tree)
assert test_model_pruned.root.right.label == 1
assert test_model_pruned.root.left.label == 0
assert test_model_pruned.root.right.isleaf
assert test_model_pruned.root.left.isleaf

from datetime import date
#[TODO] Print your name with the date, using today function from date
print()
print('----'*25)
print('Yawen Tan')
print(date.today())
```

```
---START PRINT TREE---
split attribute = 3; gain = 0.318257; number of samples = 6
0 -> 0
1 -> split attribute = 1; gain = 0.174416; number of samples = 3
                0 -> 1
                1 -> split attribute = 2; gain = 0.693147; number of samples = 2
                                0 -> 1
                                1 -> 0
----END PRINT TREE---
training loss not pruned: 0.0
validation loss not pruned: 0.5

---START PRINT TREE---
split attribute = 3; gain = 0.318257; number of samples = 6
0 -> 0
1 -> 1
----END PRINT TREE---
training loss pruned: 0.16666666666666666
validation loss pruned: 0.0

-----------------------------------------------------------------------------------------------
Yawen Tan
2025-10-29
```

In [ ]:

```
'''
Decision Trees should look similar to below (the second one is the pruned tree)

---START PRINT TREE---
split attribute = 3; gain = 0.318257; number of samples = 6
0 -> False
1 -> split attribute = 1; gain = 0.174416; number of samples = 3
                0 -> True
                1 -> split attribute = 2; gain = 0.693147; number of samples = 2
                                0 -> True
                                1 -> False
----END PRINT TREE---


---START PRINT TREE---
split attribute = 3; gain = 0.318257; number of samples = 6
0 -> False
```

```
1 -> True
----END PRINT TREE---
training loss pruned: 0.16666666666666666
validation loss pruned: 0.0
'''
```

Check Model 2

In [8]:

```
import pandas as pd
toy_train = pd.read_csv("./data/toy_train.csv", header=None).to_numpy()
toy_val = pd.read_csv("./data/toy_val.csv", header=None).to_numpy()

test_model = DecisionTree(toy_train, max_depth=4)
test_model.print_tree()
print('training loss not pruned:', test_model.loss(toy_train))
print('validation loss not pruned:', test_model.loss(toy_val), '\n')

test_model_pruned = DecisionTree(toy_train, validation_data=toy_val, max_depth=4)
test_model_pruned.print_tree()
print('training loss pruned:', test_model_pruned.loss(toy_train))
print('validation loss pruned:', test_model_pruned.loss(toy_val))
```

```
---START PRINT TREE---
split attribute = 7; gain = 0.074266; number of samples = 50
0 -> 1
1 -> split attribute = 2; gain = 0.028476; number of samples = 46
                0 -> split attribute = 3; gain = 0.323642; number of samples = 8
                                0 -> split attribute = 5; gain = 0.219512; number of samples = 6
                                                        0 -> 1
                                                        1 -> 0
                                1 -> 0
                1 -> split attribute = 8; gain = 0.020641; number of samples = 38
                                0 -> 0
                                1 -> split attribute = 5; gain = 0.003781; number of samples = 36
                                                        0 -> 0
                                                        1 -> 0
----END PRINT TREE---
training loss not pruned: 0.26
validation loss not pruned: 0.22

---START PRINT TREE---
split attribute = 7; gain = 0.074266; number of samples = 50
0 -> 1
1 -> split attribute = 2; gain = 0.028476; number of samples = 46
                0 -> split attribute = 3; gain = 0.323642; number of samples = 8
                                0 -> 1
                                1 -> 0
                1 -> 0
----END PRINT TREE---
training loss pruned: 0.26
validation loss pruned: 0.16
```

In [ ]:

```
'''---START PRINT TREE---
Decision Trees should look similar to below (the second one is the pruned tree)

split attribute = 7; gain = 0.074266; number of samples = 50
0 -> 1
1 -> split attribute = 2; gain = 0.028476; number of samples = 46
                 0 -> split attribute = 3; gain = 0.323642; number of samples = 8
                                 0 -> split attribute = 5; gain = 0.219512; number of samples = 6
                                                         0 -> 1
                                                         1 -> 0
                                 1 -> 0
```

```
                1 -> split attribute = 8; gain = 0.020641; number of samples = 38
                                0 -> 0
                                1 -> split attribute = 5; gain = 0.003781; number of samples = 36
                                                0 -> 0
                                                1 -> 0
----END PRINT TREE---
training loss not pruned: 0.26
validation loss not pruned: 0.22

---START PRINT TREE---
split attribute = 7; gain = 0.074266; number of samples = 50
0 -> 1
1 -> split attribute = 2; gain = 0.028476; number of samples = 46
                0 -> split attribute = 3; gain = 0.323642; number of samples = 8
                                0 -> 1
                                1 -> 0
                1 -> 0
----END PRINT TREE---
training loss pruned: 0.26
validation loss pruned: 0.16'''
```

Main

In [10]:

```python
import matplotlib.pyplot as plt

def loss_plot(ax, title, tree, pruned_tree, train_data, test_data):
    '''
        Example plotting code. This plots four curves: the training and testing
        average loss using tree and pruned tree.
        You do not need to change this code!
        Arguments:
            - ax: A matplotlib Axes instance.
            - title: A title for the graph (string)
            - tree: An unpruned DecisionTree instance
            - pruned_tree: A pruned DecisionTree instance
            - train_data: Training dataset returned from get_data
            - test_data: Test dataset returned from get_data
    '''
    fontsize=8
```

```
    ax.plot(tree.loss_plot_vec(train_data), label='train non-pruned')
    ax.plot(tree.loss_plot_vec(test_data), label='test non-pruned')
    ax.plot(pruned_tree.loss_plot_vec(train_data), label='train pruned')
    ax.plot(pruned_tree.loss_plot_vec(test_data), label='test pruned')


    ax.locator_params(nbins=3)
    ax.set_xlabel('number of nodes', fontsize=fontsize)
    ax.set_ylabel('loss', fontsize=fontsize)
    ax.set_title(title, fontsize=fontsize)
    legend = ax.legend(loc='upper center', shadow=True, fontsize=fontsize-2)

def explore_dataset(filename, class_name):
    train_data, validation_data, test_data = get_data(filename, class_name)

    # TODO: Print 12 loss values associated with the dataset.
    # For each measure of gain (training error, entropy, gini):
    #      (a) Print average training loss (not-pruned)
    #      (b) Print average test loss (not-pruned)
    #      (c) Print average training loss (pruned)
    #      (d) Print average test loss (pruned)
    gain_functions = [("error", node_score_error),
                      ("entropy", node_score_entropy),
                      ("gini", node_score_gini)]
    # Loop through each gain function
    for item in gain_functions:
        name = item[0]  # name of gain function
        gain_fn = item[1]  # gain function

        # Not pruned decision tree
        tree = DecisionTree(train_data, gain_function=gain_fn, max_depth=40)
        # Calculate unpruned training loss
        train_loss_nonpruned = tree.loss(train_data)
        # Compute unpruned testing loss
        test_loss_nonpruned = tree.loss(test_data)

        # Pruned decision tree
        pruned_tree = DecisionTree(train_data, validation_data=validation_data,
                                   gain_function=gain_fn, max_depth=40)
        # Calculate pruned training loss
        train_loss_pruned = pruned_tree.loss(train_data)
```

```
        # Compute pruned testing loss
        test_loss_pruned = pruned_tree.loss(test_data)

        # Print the results
        print(f"[{filename}] | Gain: {name} | Train (no prune): {train_loss_nonpruned:.4f}")
        print(f"[{filename}] | Gain: {name} | Test  (no prune): {test_loss_nonpruned:.4f}")
        print(f"[{filename}] | Gain: {name} | Train (pruned):   {train_loss_pruned:.4f}")
        print(f"[{filename}] | Gain: {name} | Test  (pruned):   {test_loss_pruned:.4f}")

    # TODO: Feel free to print or plot anything you like here. Just comment
    # make sure to comment it out, or put it in a function that isn't called
    # by default when you hand in your code!

    # import matplotlib.pyplot as plt
    # fig, ax = plt.subplots(1, 1)
    # title = "Loss vs Nodes: " + filename
    # loss_plot(ax, title, tree, pruned_tree, train_data, test_data)
    # plt.show()


########### PLEASE DO NOT CHANGE THESE LINES OF CODE! ###################
random.seed(1)
np.random.seed(1)
#########################################################################
explore_dataset('data/chess.csv', 'won')
explore_dataset('data/spam.csv', '1')
```

```
[data/chess.csv] | Gain: error | Train (no prune): 0.0000
[data/chess.csv] | Gain: error | Test  (no prune): 0.0518
[data/chess.csv] | Gain: error | Train (pruned):   0.0170
[data/chess.csv] | Gain: error | Test  (pruned):   0.0360
[data/chess.csv] | Gain: entropy | Train (no prune): 0.0000
[data/chess.csv] | Gain: entropy | Test  (no prune): 0.0142
[data/chess.csv] | Gain: entropy | Train (pruned):   0.0040
[data/chess.csv] | Gain: entropy | Test  (pruned):   0.0134
[data/chess.csv] | Gain: gini | Train (no prune): 0.0000
[data/chess.csv] | Gain: gini | Test  (no prune): 0.0142
[data/chess.csv] | Gain: gini | Train (pruned):   0.0040
[data/chess.csv] | Gain: gini | Test  (pruned):   0.0134
[data/spam.csv] | Gain: error | Train (no prune): 0.0140
[data/spam.csv] | Gain: error | Test  (no prune): 0.1519
[data/spam.csv] | Gain: error | Train (pruned):   0.0610
[data/spam.csv] | Gain: error | Test  (pruned):   0.1142
[data/spam.csv] | Gain: entropy | Train (no prune): 0.0140
[data/spam.csv] | Gain: entropy | Test  (no prune): 0.1184
[data/spam.csv] | Gain: entropy | Train (pruned):   0.0410
[data/spam.csv] | Gain: entropy | Test  (pruned):   0.1119
[data/spam.csv] | Gain: gini | Train (no prune): 0.0140
[data/spam.csv] | Gain: gini | Test  (no prune): 0.1250
[data/spam.csv] | Gain: gini | Train (pruned):   0.0430
[data/spam.csv] | Gain: gini | Test  (pruned):   0.1180
```

Project Report (10 points)

Question 1

Comment on the results of your final program. Discuss the differences in training and test error of pruned and non-pruned trees. Which measure of gain most effectively reduced training error? Was pruning effective? Use tables or graphs to demonstrate your findings.

Solution:

The following table summarizes the average training and test losses for both datasets (`chess.csv` and `spam.csv`) under three gain measures — error, entropy, and gini — with and without pruning.

Dataset	Gain	Train (no prune)	Test (no prune)	Train (pruned)	Test (pruned)
chess.csv	error	0.0000	0.0518	0.0170	0.0360
chess.csv	entropy	0.0000	0.0142	0.0040	0.0134
chess.csv	gini	0.0000	0.0142	0.0040	0.0134
spam.csv	error	0.0140	0.1519	0.0610	0.1142
spam.csv	entropy	0.0140	0.1184	0.0410	0.1119
spam.csv	gini	0.0140	0.1250	0.0430	0.1180

For both datasets, all gain measures achieved nearly perfect training accuracy before pruning. However, the entropy and gini gains consistently yielded lower test losses than the error-based gain, particularly in the `chess.csv` dataset (test loss ≈ 0.013 vs. 0.036 for error). In `spam.csv`, the entropy gain produced the lowest training and test losses (train: 0.014 → 0.041; test: 0.118 -> 0.112 after pruning), showing that it captures class information most effectively.

Across all settings, pruning slightly increased training loss (e.g., 0.000 → 0.004 for `chess.csv`, 0.014 → 0.041 for `spam.csv`) but consistently reduced test loss (e.g., 0.0142 -> 0.0134 for `chess.csv`, 0.1519 → 0.1119 for `spam.csv`). This indicates that pruning successfully reduced overfitting and improved generalization.

The non-pruned trees, especially for `spam.csv`, had very low training loss but relatively high test loss, confirming classic overfitting behavior in unpruned decision trees.

In summary, entropy was the most effective gain measure, minimizing both training and test errors. Pruning was effective, providing a better trade-off between bias and variance.

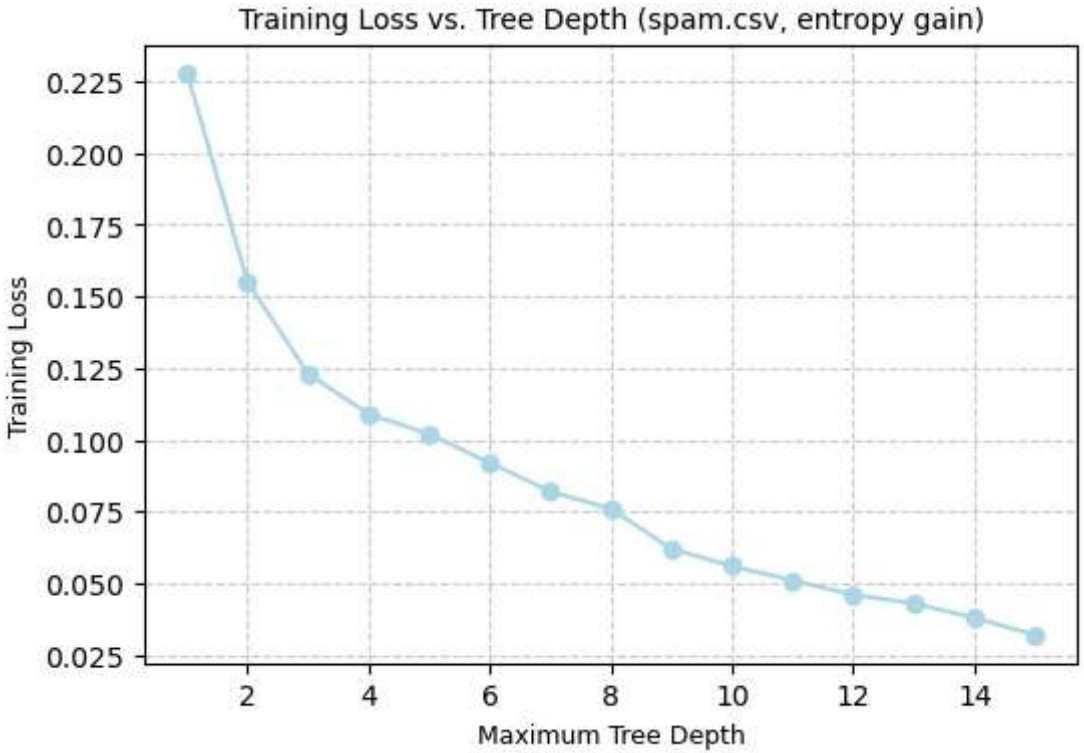
Question 2

Using the `spam.csv` dataset, plot the loss of your decision tree on the *training* set for trees with maximum depth set to each value between 1 to 15. For these plots, the trees should not be pruned and you can use the entropy gain function. Discuss any trends you find and attempt to explain them in three to five sentences. (10 points)

Note: Your graph should be (or close to) monotonically decreasing.

In [13]:

```python
# Load library
import matplotlib.pyplot as plt
# Get span.csv and only use training set, no be pruned
train_data, validation_data, test_data = get_data("data/spam.csv", "1")
# Define the range of depth
depths = list(range(1, 16))  # [1, 2, 3, ..., 15]
train_losses = []  # to save the traning loss for every depth
# Loop every depth and train a tree
for d in depths:
    # Create a decision tree, no be pruned
    tree = DecisionTree(train_data, gain_function=node_score_entropy, max_depth=d)
    # Calculate the 0-1 loss
    loss = tree.loss(train_data)
    # Save the loss
    train_losses.append(loss)
# Plot the graph
plt.figure(figsize=(6, 4))
plt.plot(depths, train_losses, marker='o', color='lightblue', linewidth=1.5)
plt.title("Training Loss vs. Tree Depth (spam.csv, entropy gain)", fontsize=10)
plt.xlabel("Maximum Tree Depth", fontsize=9)
plt.ylabel("Training Loss", fontsize=9)
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```



Solution:

The training loss decreased monotonically as the maximum tree depth increased from 1 to 15. This is expected because deeper trees have more capacity to fit the training data. The loss dropped sharply between depths 1 and 6, then decreased slowly after depth 10. Beyond depth 10, the loss approached nearly zero, indicating overfitting to the training set. Overall, the trend demonstrates that increasing model complexity reduces training error but may harm generalization performance on unseen data.