INF 264 Project 1: Decision tree

Ketil Fagerli Iversen

September 2020

Introduction

Remark: Some parts of the code might look very messy, i hope i make up for it with sufficient commenting. I began coding the project such that i wanted functions to accept a single data matrix as input. However i saw that the learn function should have two seperate data inputs, a feature matrix X and a label vector y. Therefore in the code i will often swap between handling data as one big matrix, and handling feature matrices and label vectors seperately.

Remark II: Usually when you run a decision tree and do not prune it, the accuracy on the training set should be 100%, however, after i have implemented the pruning function, this is not the case. This is simply because 25% of the original training data is withheld from the training process, as it is to be used for pruning. Thus when predicting the accuracy on the built tree using the training data, 25% of the datapoints are new to it and it can make mistakes. If one wants to achieve a 100% accuracy on the training data (for an unpruned tree), go down to page 16 under the learn function, and set **split_proportion** to 0.

The jupyter notebook contains everything in this PDF and more with test examples, which shows how the code should be executed and what they output.

Preprocessing data:

Before starting to develop the code i need to import the libraries i need, import the data, build a data splitter function and then split the data into training data, validation data and test data.

First i import the libraries i will need. The only things i need for now is numpy and random.

```
import numpy as np
import random
```

Then i load the dataset we will use later, the banknote authentication data:

```
data = np.loadtxt("data_banknote_authentication.txt", delimiter=",")
```

I then define a function which splits data into two parts based on a given proportion. I call it **data_split(data, split_proportion)**. This works by removing a random

sample of indices with size given by **split_proportion** from the original data, and then adding them to a subdataset. Both datasets are suffled and the random seed is fixed so that the results are repeatable.

```
def data_split(data, split_proportion):

    #Given some data, split it into a dataset and a subdataset. Both datasets have their rows shuffled.

    random.seed(0)
    np.random.seed(0)

    subdata_size    = round(split_proportion*len(data)) # Finds amount of data to put in the subdata-set
    indices         = list(np.arange(0,len(data),1))    # Makes a list from 1 to the number of samples(rows) in data
    subdata_indices = random.sample(population=indices,k=subdata_size) # Extracts a random sample of indices from this list

    subdata         = data[subdata_indices] # Subdata is the dataset with given indices
    dataset         = np.delete(data,subdata_indices, axis=0) # Dataset is the data but with rows removed
    np.random.shuffle(dataset) # Shuffle the dataset

    return dataset, subdata
```

In my case, i will split the training data, validation data and test data with the proportional sizes of 0.8, 0.16 and 0.04 respectively. Thus

```
training_data, val_test_data = data_split(data, split_proportion = 0.2)
val_data, test_data = data_split(val_test_data, split_proportion = 0.2)

print(training_data.shape)
print(val_data.shape)
print(test_data.shape)
```

```
(1098, 5)
(219, 5)
(55, 5)
```

Now, since the project wants the features and labels to be separated for the final **learn()** function, i will split these three matrices into their respective features and labels.

```
training_labels = training_data[:,-1]
val_labels = val_data[:,-1]
test_labels = test_data[:,-1]

training_features = training_data[:,:-1]
val_features = val_data[:,:-1]
test_features = test_data[:,:-1]
```

Task: 1.1-1.2

We want a **learn(X,y, impurity_measure)** algorithm which takes some feature matrix X and corresponding label vector y together with a impurity measure model, and thus returns a 'trained' decision tree object which we can use the **predict(x, tree)** algorithm on, to predict labels of unseen data, and also later on, determine its accuracy.

I start off with creating the functions that i need for the **learn()** algorithm:

Label uniformity: I need a function which can tell me wether or not the data i am looking at, has all of its labels being identical. I call this function **check_label_uniformity(data)**. This function takes a data matrix as input (both X and y). The function then only looks at the last column, which is the label column, and determines using **np.unique()**, how many unique labels there are. If the list of unique labels has a lenght of 1, then there is obviously only one unique label in the data and thus the data's labels are 'uniform'. If this is the case, return a True statement, otherwise False. Below i have attached a picture of the block of code.

```python
def check_label_uniformity(data):

    # Finds all the unique labels. We know the labels occupy the last column of the data
    unique_labels = np.unique(data[:,-1])

    # If there is only one unique label, the list will have one element, then return true, otherwise false
    if len(unique_labels) == 1:
        return True
    else:
        return False
```

Feature uniformity: I need a function which can tell me wether or not, for the data i am looking at, all of the feature values for each seperate column, are identical. I call this function **check_feature_uniformity(data)**. This function works similarly to the previous function. However this function will loop over each column and check for feature uniformity. If the number of uniform columns are the same as the number of feature columns, then the features are uniform. If this is the case, return True, otherwise False.

```python
def check_feature_uniformity(data):

    rows, columns = data.shape

    # Initialize the number of uniform columns
    num_uniform_feature_cols = 0

    # We skip over the last column as it is the label column
    for column in range(columns-1):

        # What are the unique values for the given column
        unique_features = np.unique(data[:,column])

        # If there is only one unique value, then the column is uniform
        if len(unique_features) == 1:
            num_uniform_feature_cols += 1

    # If the number of uniform columns are equal to the amount of feature columns, then the dataset
    # has a uniform featureset
    if num_uniform_feature_cols == (columns-1):
        return True

    else:
        return False
```

Classification: I need a function which can classify any given data matrix (both X and y). I call this function **classify_data(data)**. This function will look at the label column of the data, then by using the **np.unique()** function, i aquire a list of all the unique labels and also a count of how many there are for each. Then i find the label which has the highest count and i set this as the classification of the data, this will then be a 1×1 numpy array. However, sometimes the data is such that there is an equal amount of each label, thus giving a tie. In this case the **classification** variable will be a $1 \times n$ numpy array. If this is the case, i will select one of the labels at random as the classification.

```python
def classify_data(data):

    # We know that the labels occupy the last column of the data
    label_column = data[:,-1]

    # Finds all the unique labels and the amount for each
    unique_labels, label_counts = np.unique(label_column, return_counts = True)

    # Finds the label with the highest count, then finds the index of the most majority label in unique_labels.
    # Thus the majority label is the one with the highest count.
    majority_label_count = np.amax(label_counts)
    majority_label_index = np.where(label_counts == majority_label_count)
    majority_label = unique_labels[majority_label_index]

    classification = majority_label

    # HOWEVER, i ran into a problem in the code where classify_data() would occationally return 2 values.
    # This is because sometimes the data is such that theres an equal amount of each label.
    # Thus when such a conflict occurs, select one label at random. Also i need that the classification variable is
    # a 1x1 numpy array.

    np.random.seed(0)
    if len(classification) > 1:
        classification = np.array([np.random.choice(classification)])

    return classification
```

Find all the ways to split the feature data: I need a function which takes any given data (both X and y) and outputs a dictionary of all the possible ways of partitioning the data matrix into two parts. The reason why i want to use a dictionary here, is because every column does have an equal amount of features, however, some columns might have more repeat values than others, thus saving the possible partitions in a matrix would be difficult.

The way this code works is that it will take some data matrix as input, then loop through every column (except for the label column) and find every value which lies inbetween two consecutive feature values (the median of two consequtive values). It is important to note that the values are sorted from smallest to largest because the **np.unique()** function finds all the unique values for the column but also sorts them in a list. All of these values are then stored in a list called **splits** which is then added to the dictionary under the key **column**. This just means that the splits list can be accessed by the number of the column it belongs to.

```python
def find_possible_partitions(data):

    # I want to use a dictionary here because the data can have different amounts of splits per column
    possible_partitions = {}

    # I get the number of colums for the dataset by checking entry 2 in the .shape vector
    columns = data.shape[1]

    # We dont want to split by the label column so we skip the last column
    for column in range(columns - 1):

        # There might be repeat values in the column, so i will only look at unique values. This will also sort the column
        unique_rows = np.unique(data[:,column])
        splits = []

        # We skip the last unique row because we will partition by the middle of two points (N-1 splits for N rows)
        for row in range(len(unique_rows) - 1):

            value = unique_rows[row]
            next_value = unique_rows[row + 1]
            inbetween_value = (value + next_value) / 2 # Select partition inbetween two consecutive unique values

            # Append the median to the splits list
            splits.append(inbetween_value)

        # Appends the list of splits to the dictionary with the name(key) of the column number
        possible_partitions[column] = splits


    return possible_partitions
```

Partition data based on column and specified value: I need a function which takes a data matrix (both X and y) and some scalar column number and and some scalar value you wish to branch by, and returns two subdata sets. In other words,

given the data matrix and a column number and a branching value, split the data matrix into two parts, one where all its values in the given column lies above the branching value and the other where all its values in the given column are equal or below the branching value.

This function first gets the shape of the dataset by using the **.shape** attribute. Then it initializes a **lower_branch_data** and **upper_branch_data** as a $1 \times n$ numpy array filled with zeroes. This is done because otherwise the code wouldn't work. Then for each row in the data, check if the value occupying column number **branch_column** and row number **row** is less or equal to **branch_value**. If so, concatenate this row to the **lower_branch_data**, otherwise concatenate it to the **upper_branch_data**.

Then the function deletes the first row from both the upper and lower branch matrices. This is because we initialized the branches using a $1 \times n$ numpy array filled with zeroes, and this row is not needed anymore. After this return **lower_branch_data** and **upper_branch_data**.

```
def data_branches(data, branch_column, branch_value):

    column = data[:,branch_column] # Data in the given column
    rows, columns = data.shape # Number of rows and columns

    lower_branch_data = np.zeros([1,columns]) # Need to initialize empty arrays for the code to work
    upper_branch_data = np.zeros([1,columns]) # These are zero vectors with one row and 'columns' number of columns

    # If the value in the given row and column is less or equal to the branch_value, then we put it into the lower branch,
    # else if the value is greater, we put it into the upper branch.
    # We cycle through all the rows to partition all the datapoints.
    for row in range(rows):

        if (data[row,branch_column] <= branch_value):
            lower_branch_data = np.concatenate((lower_branch_data, [data[row]]), axis = 0)
        elif (data[row,branch_column] > branch_value):
            upper_branch_data = np.concatenate((upper_branch_data, [data[row]]), axis = 0)

    upper_branch_data = np.delete(upper_branch_data, 0, axis = 0)# Delete the first row as we initialized the upper
    lower_branch_data = np.delete(lower_branch_data, 0, axis = 0)# and lower branches with an empty row

    return lower_branch_data, upper_branch_data
```

I now need functions which can determine the best column and the best value to partition the data by. This will be based on the maximal information gain (lowest entropy for partition). This is done with three functions, **compute_entropy(data)**, **compute_branch_entropy(upper_branch, lower_branch)** and then **determine_optimal_branching(data, potential_splits)**.

compute_entropy(data): First i get a list of all the labels in the data. Then i find all the unique labels and the amount of each one. Then i loop over all unique labels and i use the definition for the entropy, to compute the entropy of the given data. The definition i use is;

$$H = -\sum_i P(x = x_i) log_2(P(x = x_i))$$

Here $P(x = x_i)$ is the probability that label x_i is to be randomly picked from the dataset.

```python
def compute_entropy(data):

    label_data = data[:,-1] # Collects a list of all the labels
    unique_labels, label_counts = np.unique(label_data, return_counts = True) # Gives all the unique labels and their count

    entropy = 0

    for i in range(len(unique_labels)):

        # Probability to pick the label unique_label[i] from the list label_data
        probability = label_counts[i] / len(label_data)

        # Use the definition of entropy to compute the entropy of data
        entropy -= probability*np.log2(probability)

    return entropy
```

compute_branch_entropy(upper_branch, lower_branch): Here i define a function which can compute the entropy of a given partition. It takes a lower and upper branch as inputs then outputs the branching entropy, i.e the entropy of the given partitioning. The definition i use is the definition of conditional entropy;

$$H(y|x) = \sum_i P(x = x_i)H(y|x = x_i)$$

```python
def compute_branch_entropy(upper_branch, lower_branch):

    # We want to compute the entropy of the given branching

    p_upper_branch = len(upper_branch) / (len(upper_branch) + len(lower_branch)) # Probability of upper branch
    p_lower_branch = len(lower_branch) / (len(upper_branch) + len(lower_branch)) # Probability of lower branch

    H_upper_branch = compute_entropy(upper_branch) # Computes entropy of dataset
    H_lower_branch = compute_entropy(lower_branch)

    # Use the definition of conditional entropy
    branch_entropy = p_upper_branch*H_upper_branch + p_lower_branch*H_lower_branch

    return branch_entropy
```

determine_optimal_branching(data, potential_splits): The following function takes some data and the '**potential_splits**' dictionary as inputs, and outputs the most optimal column and the most optimal value to partition the data. The function also outputs the lowest entropy, however this will only be used for an example computation in the jupyter notebook.

The most optimal partitioning is the one in which the labels in the upper and lower branches have the lowest overall entropy on average. Here i do not bother to compute the maximal amount of information gain, because it gives the same answer as finding the lowest overall entropy of partitioning.

```python
def determine_optimal_branching(data, potential_splits):

    rows, columns = data.shape

    i = 0

    for column in range(columns - 1): # Skip the last column as we will not partition on the label column
        split_values = potential_splits[column] # Branch values to check
        for split in range(len(split_values)):

            # Retrieves a lower branch and an upper branch by calling the data_branches() function with and
            # partitioning on a given column and split_value
            lower_branch, upper_branch = data_branches(data, branch_column = column, branch_value = split_values[split])

            # Computes the entropy of the partitioning
            entropy_of_branching = compute_branch_entropy(upper_branch, lower_branch)

            if i == 0: # Condition to ensure we have a startingpoint of comparison
                lowest_entropy = entropy_of_branching

                optimal_branch_column = column
                optimal_branch_value = split_values[split]

            # If the branch partitioning offers a lower entropy, record the new lowest entropy
            # Also record in what column and with what split_value this was achieved
            elif entropy_of_branching < lowest_entropy:
                lowest_entropy = entropy_of_branching

                optimal_branch_column = column
                optimal_branch_value = split_values[split]

            # Increment i by 1 so that we skip the startingpoint condition
            i += 1

    return optimal_branch_column, optimal_branch_value, lowest_entropy
```

We are ready to build a tree, but first i want to define similar functions for computing the minimal gini index. This is done similarly to the previous three functions, with the modification that the definition for gini index is slightly different from that of entropy:

$$G = \sum_i P(x = x_i) \cdot (1 - P(x = x_i))$$

$$G(y|x) = \sum_i P(x = x_i) \cdot G(y|x = x_i)$$

```python
def compute_gini_index(data):

    label_data = data[:,-1] # Collects a list of all the labels
    unique_labels, label_counts = np.unique(label_data, return_counts = True) # Gives all the unique labels and their count

    gini_idx = 0

    for i in range(len(unique_labels)):

        # Probability to pick the label unique_label[i] from the list label_data
        probability = label_counts[i] / len(label_data)

        # Use the definition of the gini index
        gini_idx += probability*(1-probability)

    return gini_idx
```

```python
def compute_branch_gini_index(lower_branch, upper_branch):

    # We want to compute the gini index of the given branching

    p_upper_branch = len(upper_branch) / (len(upper_branch) + len(lower_branch)) # Probability of upper branch
    p_lower_branch = len(lower_branch) / (len(upper_branch) + len(lower_branch)) # Probability of lower branch

    G_upper_branch = compute_gini_index(upper_branch) # Computes gini index of dataset
    G_lower_branch = compute_gini_index(lower_branch)

    # Compute the weighted sum of the gini indices
    branch_gini_idx = p_upper_branch*G_upper_branch + p_lower_branch*G_lower_branch

    return branch_gini_idx
```

```python
def determine_optimal_branching_gini(data, potential_splits):

    rows, columns = data.shape

    i = 0

    for column in range(columns - 1): # Skip the last column as we will not partition on the label column
        split_values = potential_splits[column] # Branch values to check
        for split in range(len(split_values)):

            # Retrieves a lower branch and an upper branch by calling the data_branches() function with and
            # partitioning on a given column and split_value
            lower_branch, upper_branch = data_branches(data, branch_column = column, branch_value = split_values[split])

            # Computes the entropy of the partitioning
            gini_of_branching = compute_branch_gini_index(upper_branch, lower_branch)

            if i == 0: # Condition to ensure we have a startingpoint of comparison
                lowest_gini = gini_of_branching

                optimal_branch_column = column
                optimal_branch_value = split_values[split]

            # If the branch partitioning offers a lower gini index, record the new lowest gini
            # Also record in what column and with what split_value this was achieved
            elif gini_of_branching < lowest_gini:
                lowest_gini = gini_of_branching

                optimal_branch_column = column
                optimal_branch_value = split_values[split]

            # Increment i by 1 so that we skip the startingpoint condition
            i += 1

    return optimal_branch_column, optimal_branch_value
```

Now finally we can begin building the decision tree. A decision tree consists of decision nodes and leaf nodes. The decision nodes should hold several different attributes. It should hold a **question** variable which is a 1×3 numpy array which consists of the branching column, the comparison operator and the branching value. For example will the question $[0, '<=', 1]$, ask the node "are the values at column $0 <= 1$"? If so the values go into the lower branch, if not it goes into the upper branch.

Another attribute is the subtree which lies along the lower branch, this i call the **lower_branch_node** similarly for the subtree which lies along the upper branch **upper_branch_node**. I also want to know all of the training data which is partitioned into the two subtrees **lower_branch_data** and **upper_branch_data**. Finally we want to store the majority label of the decision node which i store as **majority_label**.

```python
class Decision_node:

    def __init__(self, question, lower_branch_data, upper_branch_data, lower_branch_node, upper_branch_node, majority_label):

        self.question = question
        self.lower_branch_node = lower_branch_node
        self.upper_branch_node = upper_branch_node
        self.lower_branch_data = lower_branch_data
        self.upper_branch_data = upper_branch_data
        self.majority_label = majority_label
```

Another thing we need is the leaf node. They only need to store the label they are assigned.

```python
class Leaf:

    def __init__(self, label):

        self.predictions = label
```

The **build_tree(X,y, impurity_measure)** function takes training features and training labels as input and returns either a **Leaf** object or a **Decision_node** object. If all the labels in data are the same (only one unique label), it will return a leaf with that label. If all the features in each feature column are identical, it will return a leaf

with the majority label.

Otherwise, it will call upon the **find_possible_splits()** function to find all the possible ways to partition the data. Depending on the impurity measure being 'entropy' or 'gini', the function will call upon either the **determine_optimal_branching()** function or **determine_optimal_branching_gini()** function to find the optimal column to branch by and the optimal value to branch by. Then it defines the variable **question** as a list which contains the optimal branch column, a comparison operator and the optimal branch value. This will be used later for printing the tree and also for classifying new data.

Then the function gets the data which lands in the upper and lower branches with the **data_branches()** function. Finally the function recursively calls upon itself by calling on the **build_tree** function with the lower- and upper branches to create the subtrees **upper_branch_node** and **lower_branch_node**.

```
def build_tree(X_training, y_training, impurity_measure='entropy'):

    data = np.column_stack((X_training, y_training))
    majority_label = classify_data(data)

    # If all labels in given data is the same, return leaf. Store the majority label in this leaf.
    # In this case, the labels are all the same value, so the majority label is certain
    if check_label_uniformity(data) == True:
        return Leaf(majority_label)

    # If all features in given data is the same, return leaf. Store the majority label in this leaf.
    # In this case, the labels can have different values, and thus the majority label can have some uncertainty
    if check_feature_uniformity(data) == True:
        return Leaf(majority_label)

    # Finds all possible splits for each column in the given data. Stores it in a dictionary.
    potential_splits = find_possible_partitions(data)


    # If impurity_measure is given as 'entropy', then we find the optimal branch column and branch value by
    # minimizing the entropy.
    # If impurity_measure is given as 'gini', then we find the optimal branch column and branch value by
    # computing the gini index
    if impurity_measure == 'entropy':
        optimal_branch_column, optimal_branch_value, lowest_entropy = determine_optimal_branching(data, potential_splits)

    elif impurity_measure == 'gini':
        optimal_branch_column, optimal_branch_value = determine_optimal_branching_gini(data, potential_splits)


    # Question stores the column we split on and what value to split by. Will be used for printing the tree and prediction
    question = [optimal_branch_column, '<=', optimal_branch_value]

    # Values which lie below the given 'optimal_branch_value', goes to the lower branch,
    # the values which lie above go to the upper branch
    lower_branch_data, upper_branch_data = data_branches(data, optimal_branch_column, optimal_branch_value)


    # As our learn() function wants a data matrix X and a label vector y, we need to split the data matrices
    lower_branch_features = lower_branch_data[:,:-1]
    lower_branch_labels = lower_branch_data[:,-1]
    upper_branch_features = upper_branch_data[:,:-1]
    upper_branch_labels = upper_branch_data[:,-1]


    # Recursively build the lower branch.
    lower_branch_node = build_tree(lower_branch_features, lower_branch_labels, impurity_measure)

    # Recursively build the upper branch.
    upper_branch_node = build_tree(upper_branch_features, upper_branch_labels, impurity_measure)

    # Return a decision node. Store the question, the upper branch and lower branch in this node.
    return Decision_node(question, lower_branch_data, upper_branch_data,
                         lower_branch_node, upper_branch_node, majority_label)
```

The first learn() function:

I now create the **learn()** function that we wanted. This code might seem a bit redundant as it does not do anything but calling the **build_tree()** function, however it will become apparent why i create it like this once we get to implementing a pruning algorithm.

```
def learn(X, y, impurity_measure='entropy'):

    X_training = X
    y_training = y

    return build_tree(X_training, y_training, impurity_measure)
```

The way one uses this function to build a decision tree, is to execute the following:

tree = learn(X_training, y_training, impurity_measure)

Here **X_training** are the training features and **y_training** are the corresponding training labels. The impurity measure is up to the user to decide, and you can choose between either entropy or gini. **tree** is the built tree, and it can be given any arbitrary name.

I now define two unpruned decision trees, one with an impurity measure of entropy, and one with an impurity measure of gini;

```
my_tree = learn(training_features, training_labels)
```

```
my_2nd_tree = learn(training_features, training_labels, impurity_measure='gini')
```

Predictor functions and accuracy:

Before we can get to pruning the decision tree, we need some functions which can use the tree we have just built, to predict the labels of unseen data. This function will read a single row from a dataset and with the given tree, predict the label. If **print_progress** is set to True, then it will print the relevant questions and answers during the process as the algorithm is predicting the label.

```
def predict(data_row, node, print_progress=False):

    # If the remaining node is a leaf node, return the prediction of this leaf node
    if isinstance(node, Leaf):
        if print_progress == True:
            print('We have reached a leaf node, we predict the label is: ')
        return node.predictions

    # Printing relevant question for current node
    if print_progress == True:
        print('Is the value in column ' + str(node.question[0]) + ' ' + str(node.question[1])
              + ' ' + str(node.question[2]) + '?')

    # If the split value in the question is greater or equal to the value in the data_row at the given column
    # then go to the lower branch, otherwise go to the upper branch
    if node.question[2] >= data_row[node.question[0]]:
        if print_progress == True:
            print('True')
        return predict(data_row, node.lower_branch_node, print_progress)
    else:
        if print_progress == True:
            print('False')
        return predict(data_row, node.upper_branch_node, print_progress)
```

As an example, we can use the built tree **my_tree** to predict the label of the first row in the validation data. We can do that with the following code:

```
example = val_data[0]

print('Let us predict the label of the following example; \n')
print(str('[Column 0, Column 1, Column 2, Column 3, Label]'))
print(str(example) + '\n')
print(predict(example, my_tree, print_progress=True))

if predict(example, my_tree) == example[-1]:
    print('This prediction was correct!')
else:
    print('This prediction was false!')
```

```
Let us predict the label of the following example;

[Column 0, Column 1, Column 2, Column 3, Label]
[-3.8053  2.4273  0.6809 -1.0871  1.    ]

Is the value in column 0 <= 0.295535?
True
Is the value in column 1 <= 5.86535?
True
Is the value in column 2 <= 6.21865?
True
Is the value in column 1 <= 4.09405?
True
We have reached a leaf node, we predict the label is:
[1.]
This prediction was correct!
```

As we can see, the decision tree can correctly predict the label of this example. When the **print_progress** is set to True, the predict function will print the questions and answers that each node asks as you traverse down the tree.

I now make a function which can determine the prediction accuracy of a decision tree given a dataset (for example some validation or test data). I call this function **predict_data_accuracy(data_features, data_labels, node)**. This function takes some features and labels you want to test the tree with, aswell as the tree itself.

The way this function works is that it first gets the amount of rows in our feature set, then initializes **correct_predictions** and **incorrect_predictions** and **incorrect_rows**. Then for each row in the feature set, use the previous **predict()** function to predict a label given the features and the given tree. Then if the prediction is equal to the actual label, then we increment the amount of **correct_predictions** by 1, if not we increment the amount of **incorrect_predictions** by 1 and also append the row number into the **incorrect_rows** list. After the for loop is done, we compute the accuracy simply by computing the ratio of correct predictions over the total amount of predictions. Then print and return the relevant information.

```python
def predict_data_accuracy(data_features, data_labels, node):

    # Get the amount of rows in the feature data
    rows = data_features.shape[0]


    # Initialize amount of correct and incorrect predictions
    correct_predictions = 0
    incorrect_predictions = 0

    incorrect_rows = []

    for row in range(rows):

        # The classify function returns an np.array with a single element so we just extract the value
        prediction = predict(data_features[row], node)[0]

        # If the prediction is equal to the actual label, then we get a correct prediction
        if prediction == data_labels[row]:
            correct_predictions += 1

        # If not, we have an incorrect prediction
        else:
            incorrect_predictions += 1
            incorrect_rows.append(row)

    # The accuracy of our decision tree is then
    accuracy = correct_predictions / (correct_predictions + incorrect_predictions)

    print('Correct predictions: ' + str(correct_predictions))
    print('Incorrect predictions: ' + str(incorrect_predictions))
    print('The incorrect predictions happened for datapoints: ' + str(incorrect_rows))

    return accuracy
```

The following code gives the accuracy of the current **my_tree** decision tree (the one using entropy):

```python
predict_data_accuracy(val_features,val_labels, my_tree)
```

```
Correct predictions: 215
Incorrect predictions: 4
The incorrect predictions happened for datapoints: [14, 46, 124, 214]

0.9817351598173516
```

And here is the accuracy of the other tree using the gini index:

```python
predict_data_accuracy(val_features,val_labels, my_2nd_tree)
```

```
Correct predictions: 214
Incorrect predictions: 5
The incorrect predictions happened for datapoints: [46, 82, 124, 129, 214]

0.9771689497716894
```

As we can see, the code will give us the amount of correct predictions, the amount of incorrect predictions, and also for which row the incorrect predictions occur in the validation data and finally the accuracy. We also notice that in our case, the accuracy for the gini tree is slightly worse.

Task 1.3 Reduced error pruning:

I now want to modify the **build_tree()** function. It should build a tree using training data, then use pruning data to replace each subtree in the tree if the accuracy of the pruning data is not adversly affected by replacing the subtree by a leaf node containing the majority label of the subtree.

I first define a function which can compute wether or not replacing a given subtree by a leaf with the majority label is benefitial for the pruning accuracy or not. I call this function **prune_subtree_benefit()**. This function will only be called when **both** the lower and upper branch nodes are leaf nodes. Thus both the upper and lower branches contain the **.predictions** attribute.

The way this code works is by taking the pruning data **prune_features** and **prune_labels**, and stacks them together into one matrix. Then it initializes the prediction of the pruned subtree **prediction_leaf** as **majority_label**. Then initializes the amount of correct and incorrect predictions for the pruned subtree and the amount of correct and incorrect predictions for the unpruned tree.

For each row in the pruning data, check if the value of the prune data is above or below the **branch_value** contained in **question**[**2**], then make the prediction on either the upper or lower branch. If the prediction is correct, increment **correct** by 1, if not increment **incorrect** by 1 if the prediction by pruning the subtree (**prediction_leaf**) is correct, increment **correct_pruned** by 1 if not increment **incorrect_pruned** by 1. Then compute the corresponding accuracies. If the pruned accuracy is greater or equal to the unpruned accuracy on the pruning data, then return True, otherwise return False.

```python
def prune_subtree_benefit(prune_features, prune_labels, question, majority_label, lower_branch_node, upper_branch_node):

    # I want the pruning data to be one large matrix with the first columns being feature columns and the last being
    # a label column
    prune_data = np.column_stack((prune_features,prune_labels))

    # The prediction of the current node is the majority label here
    prediction_leaf = majority_label

    # Initialize amount of correct and incorrect predictions for pruned subtree
    correct_pruned = 0
    incorrect_pruned = 0

    # Initialize amount of correct and incorrect predictions for unpruned subtree
    correct = 0
    incorrect = 0

    # For each row in the pruning data, check if the value of the prune data is above or below the branch_value
    # contained in question[2], then make the prediction on either the upper or lower branch. If the prediction
    # is correct, increment correct by 1, if not increment incorrect by 1
    # if the prediction by pruning the subtree (prediction_leaf) is correct, increment correct_pruned by 1
    # if not increment incorrect_pruned by 1.
    for row in range(len(prune_labels)):

        if question[2] >= prune_data[row,question[0]]:
            prediction = lower_branch_node.predictions
        else:
            prediction = upper_branch_node.predictions

        # Increment amount of correct / incorrect predictions for original subtree
        if prediction == prune_labels[row]:
            correct += 1
        else:
            incorrect += 1

        # Increment amount of correct / incorrect predictions for pruned subtree
        if prediction_leaf == prune_labels[row]:
            correct_pruned += 1
        else:
            incorrect_pruned += 1

    # Compute accuracy
    prune_accuracy_pruned = correct_pruned / (correct_pruned + incorrect_pruned)
    prune_accuracy = correct / (correct + incorrect)

    # If the accuracy of the pruned tree is the same or better, return True, if not, False
    if prune_accuracy_pruned >= prune_accuracy:
        return True
    else:
        return False
```

Now we can build the final tree builder funtion. I will call it **build_tree_prune(X_training, y_training, X_prune, y_prune, impurity_measure='entropy', prune=False)**

This function works exactly the same as the previous tree builder function, however it accepts aditional arguments; **X_prune** and **y_prune**. They are the pruning features and the pruning labels. Also it accepts the argument **prune** which is at default set to False.

The way this function prunes the tree, is that it simultaneously builds the tree aswell as pruning it. This is done by first asking if **prune** is set to True, if so check if both the lower branch node and the upper branch node (both subtrees) at the current node, are Leaf nodes. If so, this means we are at one of the deepest layers of the tree. Thus we can possibly prune at the current node. We also need a fourth condition, this one asks if we have any pruning data that has reached this node. Otherwise we dont have any pruning data to use for comparison. If all four conditions are met, it will call upon the **prune_subtree_benefit()** function. If the **prune_subtree_benefit()** function returns True, then **build_tree_prune()** will return a leaf node with the majority label. If not, it will contionue just as the previous tree builder function by recursively building nodes.

```
def build_tree_prune(X_training,y_training, X_prune, y_prune, impurity_measure='entropy', prune=False):

    data = np.column_stack((X_training, y_training))
    prune_data = np.column_stack((X_prune, y_prune))

    majority_label = classify_data(data)

    # If all labels in given data is the same, return leaf. Store the data in this leaf.
    if check_label_uniformity(data) == True:
        return Leaf(majority_label)

    # If all features in given data is the same, return leaf. Store the majority label in this leaf.
    # In this case, the labels can have different values, and thus the majority label can have some uncertainty
    if check_feature_uniformity(data) == True:
        return Leaf(majority_label)

    # Finds all possible splits for each column in the given data. Stores it in a dictionary.
    potential_splits = find_possible_partitions(data)


    # If impurity_measure is given as 'entropy', then we find the optimal branch column and branch value by
    # minimizing the entropy.
    # If impurity_measure is given as 'gini', then we find the optimal branch column and branch value by
    # computing the gini index
    if impurity_measure == 'entropy':
        optimal_branch_column, optimal_branch_value, lowest_entropy = determine_optimal_branching(data, potential_splits)

    elif impurity_measure == 'gini':
        optimal_branch_column, optimal_branch_value = determine_optimal_branching_gini(data, potential_splits)



    # Question stores the column we split on and what value to split by. Will be used for printing the tree and prediction
    question = [optimal_branch_column, '<=', optimal_branch_value]

    # Values which lie below the given 'optimal_branch_pruneue', goes to the lower branch,
    # the values which lie above go to the upper branch
    lower_branch_data, upper_branch_data = data_branches(data, optimal_branch_column, optimal_branch_value)
    lower_branch_prune, upper_branch_prune = data_branches(prune_data, optimal_branch_column, optimal_branch_value)


    # As our learn() function wants a data matrix X and a label vector y, we need to split the data matrices
    lower_branch_features = lower_branch_data[:,:-1]
    lower_branch_labels = lower_branch_data[:,-1]
    upper_branch_features = upper_branch_data[:,:-1]
    upper_branch_labels = upper_branch_data[:,-1]


    lower_branch_prune_features = lower_branch_prune[:,:-1]
    lower_branch_prune_labels = lower_branch_prune[:,-1]
    upper_branch_prune_features = upper_branch_prune[:,:-1]
    upper_branch_prune_labels = upper_branch_prune[:,-1]

    # Recursively build the lower branch.
    lower_branch_node = build_tree_prune(lower_branch_features, lower_branch_labels,
                                         lower_branch_prune_features, lower_branch_prune_labels, impurity_measure, prune)

    # Recursively build the upper branch.
    upper_branch_node = build_tree_prune(upper_branch_features, upper_branch_labels,
                                         upper_branch_prune_features, upper_branch_prune_labels, impurity_measure, prune)

    # If both the lower and upper nodes are leaf nodes, and if we have any pruning data that has reached this point,
    # then compute the pruning benefit. If pruning has a benefit on the accuracy of the pruning data, then
    # make the current node into a leaf node with the majority label of the training data at this point.
    if (prune == True) and isinstance(lower_branch_node, Leaf) and isinstance(upper_branch_node, Leaf)
    and (len(prune_data[:,-1]) > 0):
        pruning_benefit = prune_subtree_benefit(prune_data[:,:-1], prune_data[:,-1], question,
                                                majority_label, lower_branch_node, upper_branch_node)
        if pruning_benefit == True:
            return Leaf(majority_label)


    # Return a decision node. Store the question, the upper branch and lower branch in this node.
    return Decision_node(question, lower_branch_data, upper_branch_data,
                         lower_branch_node, upper_branch_node, majority_label)
```

Now we can finalize the **learn()** function that we want. This will be a simple function that only takes **X**, **y**, **impurity_measure** and **prune** as inputs, and returns a fully built decision tree by using the **build_tree_prune()** function. In this specific case, i have set **split_proportion** as 0.25, this means that 25% of the training data will be removed from the training dataset and be used as pruning data. The data will be split using the **data_split()** function i defined at the start.

```
def learn(X, y, impurity_measure='entropy', prune=False):

    # The ratio of prune data by amount of total training data
    split_proportion = 0.25

    # I want to combine the features and labels before splitting
    data = np.column_stack((X, y))

    # Split the data into training data and purning data
    training_data, prune_data = data_split(data, split_proportion)

    # Split the matrices back into feature matrices and label vectors
    X_training = training_data[:,:-1]
    y_training = training_data[:,-1]

    X_prune = prune_data[:,:-1]
    y_prune = prune_data[:,-1]

    # Build the tree
    return build_tree_prune(X_training, y_training, X_prune, y_prune, impurity_measure, prune)
```

Visualizing the tree:

We want to be able to visualize the tree for the next section. This is because we want to be able to visually compare pruned and unpruned trees. I will call this function **tree_printer(node, indent="")**. This function takes a decision tree or subtree (node) and an indent as inputs. The indent variable is here so that each time the tree funciton is called, the indent grows by another space. That has the effect of adding an extra indentation to each layer of the tree, making it easier to read and interpret. If the node we are looking at is a leaf node, print the prediction of the label. Otherwise we are at a decision node, so thus print the question at this node and then recursively call the **tree_printer()** function on the **lower_branch_node** and the **upper_branch_node** with an extra space added to the **indent** variable.

```
def tree_printer(node, indent=""):

    # The indent variable is here so that each time the tree funciton is called, the indent
    # grows by another space. That has the effect of adding an extra indentation
    # to each layer of the tree, making it easier to read and interpret.

    # If the node we are looking at is a leaf node, print the prediction for the label.
    if isinstance(node, Leaf):
        print (indent + "Predict", node.predictions)
        return

    # Otherwise print the question at the decision node
    print(indent + 'Is value at column: ' + str(node.question[0]) + str(node.question[1]) + str(node.question[2]) + '?')

    # Recursively decend the tree down the lower branch
    print(indent + '___ Value lies below:')
    tree_printer(node.lower_branch_node, indent + "  ")

    # Recursively decend the tree down the upper branch
    print(indent + '^^^ Value lies above:')
    tree_printer(node.upper_branch_node, indent + "  ")
```

Task 1.4 Evaluation:

We now want to evaluate the decision tree using validation data and vary the hyperparameters and then select the most suitable model. We will train the decision tree using training data **X_training** and **y_training**. Recall that inside the **learn()** function, 25% of this training data is removed from the training data and instead used as pruning data. Then, once the tree is built, we will call the function **predict_data_accuracy()** with **val_features val_labels** and the decision tree, to compute the trees accuracy on the validation data. I will vary the hyperparameters **impurity_measure** and **prune** and check the accuracy for all the four combinations

Pruned vs Unpruned & Entropy vs Gini Trees:

First off i compare the unpruned tree versus the prune tree on the validation data when **impurity_measure = 'entropy'**. Here are the results:

```
my_tree = learn(training_features, training_labels, impurity_measure='entropy', prune=False)
```

```
my_tree_pruned = learn(training_features, training_labels, impurity_measure='entropy', prune=True)
```

```
print('Correct and incorrect predictions for pruned tree: ')
accuracy_pruned = predict_data_accuracy(val_features, val_labels, my_tree_pruned)
print('\n Correct and incorrect predictions for non-pruned tree: ')
accuracy_nonpruned = predict_data_accuracy(val_features, val_labels, my_tree)

difference = accuracy_pruned - accuracy_nonpruned

print('\n Accuracy pruned tree: ')
print(accuracy_pruned)
print('\n Accuracy non-pruned tree: ')
print(accuracy_nonpruned)
print('\n Accuracy difference: ')
print(difference)
```

```
Correct and incorrect predictions for pruned tree:
Correct predictions: 215
Incorrect predictions: 4
The incorrect predictions happened for datapoints: [46, 82, 124, 214]

 Correct and incorrect predictions for non-pruned tree:
Correct predictions: 213
Incorrect predictions: 6
The incorrect predictions happened for datapoints: [46, 82, 95, 103, 124, 214]

 Accuracy pruned tree:
0.9817351598173516

 Accuracy non-pruned tree:
0.9726027397260274

 Accuracy difference:
0.0091324200913242
```

Now i do the same for when **impurity_measure = 'gini'**:

```
my_tree_gini = learn(training_features, training_labels, impurity_measure='gini', prune=False)
```

```
my_tree_pruned_gini = learn(training_features, training_labels, impurity_measure='gini', prune=True)
```

```
print('Correct and incorrect predictions for pruned tree: ')
accuracy_pruned = predict_data_accuracy(val_features, val_labels, my_tree_pruned_gini)
print('\n Correct and incorrect predictions for non-pruned tree: ')
accuracy_nonpruned = predict_data_accuracy(val_features, val_labels, my_tree_gini)

difference = accuracy_pruned - accuracy_nonpruned

print('\n Accuracy pruned tree: ')
print(accuracy_pruned)
print('\n Accuracy non-pruned tree: ')
print(accuracy_nonpruned)
print('\n Accuracy difference: ')
print(difference)
```

```
Correct and incorrect predictions for pruned tree:
Correct predictions: 215
Incorrect predictions: 4
The incorrect predictions happened for datapoints: [46, 82, 124, 214]

 Correct and incorrect predictions for non-pruned tree:
Correct predictions: 214
Incorrect predictions: 5
The incorrect predictions happened for datapoints: [46, 82, 124, 129, 214]

 Accuracy pruned tree:
0.9817351598173516

 Accuracy non-pruned tree:
0.9771689497716894

 Accuracy difference:
0.004566210045662156
```

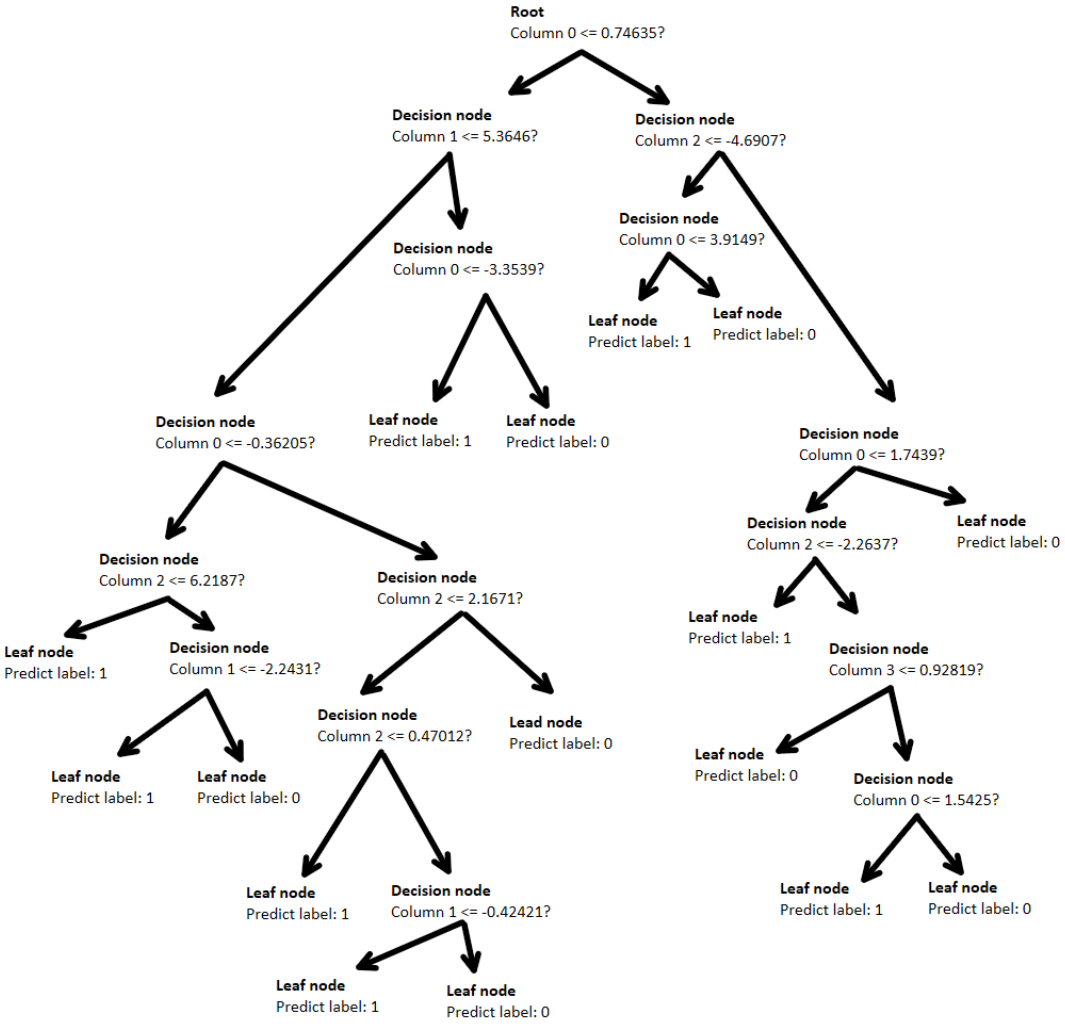
We can summarize the results in the following table. I will limit the accuracies to 5 significant figures.

Accuracy	Pruned	Unpruned	Accuracy gained from pruning
Entropy	0.98174	0.97260	+0.00913
Gini	0.98174	0.97717	+0.00457

We can see that the decision trees that performed the best on our validation data were both the pruned decision tree with **impurity_measure = 'entropy'** and the pruned decision tree with **impurity_measure = 'gini'**. They both only had four incorrect predictions out of 219. However, all decision trees had a remarkably high accuracy, only differing by a single percent from eachother. This could be a bit suspect. Especially since two of the decision trees have the exact same accuracy and they also fail to predict the exact same validation datapoints. If you play around with different **split_proportions** when defining the training, validation and test data, you will get different results, sometimes zero or even a negative benefit for pruning. For now however, i will keep to these results.

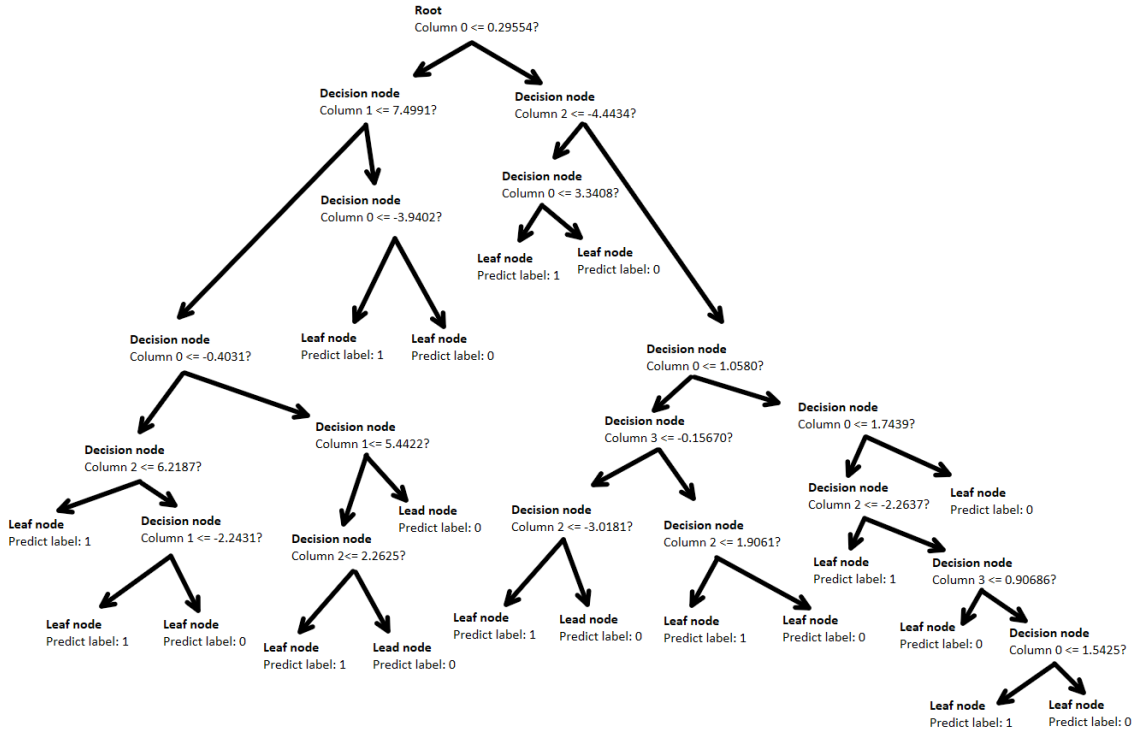
As we want the simplest tree possible, with the highest possible validation accuracy, we can either choose a **pruned** decision tree with **impurity_measure='entropy'** or **impurity_measure='gini'**. These two decision trees have the exact same accuracy. However, we might have a difference in the complexity of the two trees. I will plot the two trees to check which one is the simpler tree. I use the **tree_printer()** function above, to print the trees. However their output is very long and unweildy to interpret at a glance. Thus i have done the work of drawing the trees from this information and will show that instead. If you want to inspect the raw data given by the **tree_printer()** function, have a look at the section called **Pruned vs Unpruned Entropy vs Gini Trees** in the jupyter notebook.

The way the trees are plotted, each node asks a question about if the value in the given column is less than a certain value. If yes then go left, if not go right. First, i have plotted the pruned decision tree using entropy as its impurity measure:



This tree has 15 decision nodes and 16 leaf nodes, totalling 31 nodes. It also has a maximum depth of 6.

Now for the pruned decision tree using the gini index as its impurity measure:



This tree has 18 decision nodes and 19 leaf nodes, totalling 37 nodes. It also has a maximum depth of 7. Thus we can see that the gini tree is actually slightly more complicated than the entropy tree, with essentially the same accuracy. Hence we should choose the pruned entropy tree as our model:

my_tree_pruned =
learn(training_features, training_labels, impurity_measure='entropy', prune=True)

Thus we can check the generalized accuracy of this selected model by using the test data. We simply run the **predict_data_accuracy()** function on our tree and the test data

```
accuracy = predict_data_accuracy(test_features, test_labels, my_tree_pruned)
print('The accuracy of our selected model is: ' + str(accuracy))
```

```
Correct predictions: 54
Incorrect predictions: 1
The incorrect predictions happened for datapoints: [16]
The accuracy of our selected model is: 0.9818181818181818
```

Thus, our tree has a really good performance of 98.18% on the test data.

Task 1.5 Compare to an existing implementation:

I will use Sklearn's decision tree classifier. This will not require a lot of code, and it will be quite self-explanatory. First I import the things i need

```
# Import what we need
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
```

Now we define a decision tree using the **DecisionTreeClassifier()** function. One with entropy and one using the gini index. I want to set the **random_state = 0** to get repeatable results. Then I fit the classifiers using the same training data as the one used in my own implementation

```
# Define a decision tree with criterion 'entropy' and one with criterion 'gini'
tree_sklearn = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)
tree_sklearn_gini = DecisionTreeClassifier(criterion = 'gini', random_state = 0)

# Fit the two decision trees with the training data
tree_sklearn.fit(training_features, training_labels)
tree_sklearn_gini.fit(training_features, training_labels)
```

Now we check the accuracies of the two decision trees using the **.score()** attribute with the same test data as before. Then we print the accuracies.

```
# Check the accuracy of the two decision trees
accuracy_sklearn = tree_sklearn.score(val_features, val_labels)
accuracy_sklearn_gini = tree_sklearn_gini.score(val_features, val_labels)
```

```
# Print the accuracies
print('The accuracy of the sklearn tree was: ' + str(accuracy_sklearn))
print('The accuracy of the sklearn tree (gini) was: ' + str(accuracy_sklearn_gini))
```

```
The accuracy of the sklearn tree was: 0.9817351598173516
The accuracy of the sklearn tree (gini) was: 0.9817351598173516
```

Here, we see that my implementation is just as accurate as the sklearn implementation on the validation data. However, a big problem with my implementation is that it takes around 30 seconds to train and create a decision tree. The Sklearn implementation will fit its classifier within a fraction of a second. The most time consuming part of my code is the **determine_optimal_branching()** function. This function needs to compute the entropy or the gini index for every single possible split between each unique value for up to a thousand rows over 4 columns (atleast for the root node). The sklearn implementation probably has a much more clever and elegant way of doing this which leads to a more efficient code. Thus, if we had to choose, we should choose the sklearn implementation. In this specific case the accuracies of the **tree_sklearn** and **tree_sklearn_gini** are the same. I will just pick the first of the two trees as my final selected model, the **tree_sklearn** tree.

Now i check the accuracy of my final selected model on the test data:

```
accuracy_final_model = tree_sklearn.score(test_features, test_labels)
print(accuracy_final_model)
```

```
0.9818181818181818
```

We can see the model has a great performance of 98.18% on the test data.