INF264 Project 1

Implementing decision trees

Sophie Blum and Benjamin Friedl — 18.09.2020

About the document: The names behind every title point out, who initially wrote the code for this functionality. The largest amount of time however went into debugging and correcting the code, as well as re-evaluating the design choices and making the algorithm work. We did all this together, so that it is hard to point out, who did what exactly.

1 Implementation of the tree data structure (Sophie)

The decision tree is implemented as a node data structure in a semi-object-oriented way. A tree is viewed as a root with children connected to it. That means that every node can be the root of a tree (and every node is indeed the root of its subtree). As every node could be a tree, the decision tree is not encapsulated in a data structure itself and has to be given as a parameter for every function that uses it (see also: `learn()`) The nodes themselves are implemented in the node class, which contains variables for data and the majority label in the subtree denoted by this node. The data variable holds either the feature for the corresponding decision or a label, if the node is a leaf in the tree. The majority label is used for pruning. The node also has a reference to its parent and its children. The latter are organized in a list of tuples. Every tuple has a reference to the child node as well as the value that is used for the decision. Having both parent and children allows to traverse the data structure in both directions and is only done for convenience.

```
1     class node:
2         def __init__(self, data, parent = None,
    majority_label = None):
3             self.data = data
4             self.parent = parent
5             self.majority_label = majority_label
6             self.children = []
7         def __str__(self):
8             return f"{self.data} - {self.children}"
```

```
        def addChild(self, child):
            self.children.append(child)
```

The `printTree()` is an inorder traversal of a tree denoted by a root-node.

Remark: We designed the data structure before we knew we only had to compute binary trees, so this could have been implemented easier by just storing each child node separately and having only one variable to denote the decision value.

2 Implementation of learn (Sophie)

The learn function is splitted into two separate functions, as the integration of pruning is easier that way. In fact the `learn_rec()` has been `learn()` in the first place. With the addition of pruning and the integration of it in this function, `learn()` was splitted in two parts.

`Learn_rec()` is the powerhouse of the learn function and outputs the root node of the decision tree, that is learned on the input data.

It is important to note, that we considered a lot of different options for this function. We tried basically two different options concerning the decision tree:

1. On a given path in the tree one feature is used at most one time
2. A feature can be used multiple times to split on a given path in the tree

Both variants result in valid decision trees. Of course the decision tree that is return by the first one is a lot smaller, but it also has a significantly worse accuracy. We chose accuracy over performance and therefore decided to allow a label to be used more than once. We know that this solution is also more prone to overfitting, but the use of pruning should reduce this risk. For this reason there is some legacy code from the former implementation in the learn function. In particular we kept track of the already used labels in the `used_feature_list` array. We kept the code this way to show the process of our work in this project and it also allows us to switch back to the first option quicker if we wanted to.

The function first checks all base cases and outputs the majority label as a leaf.

```
    if same_value(y):
        return node(y[0])
    elif same_value(X):
        label = get_maj_label(y)
        return node(label)
```

When there is no base case present, a feature gets selected based on the selected impurity measure (separately implemented). The dataset then gets

splitted based on the mean of the selected feature. These separate datasets are then used for a recursive call of the same function. All the created nodes get assigned their respective parent and/or children. All this functionality is subdivided in different functions doing smaller tasks. These functions are summarized in the help function block. To display the implementation process, we also kept functions we implemented at some point but don't use anymore in a seperate block.

`Learn()` is just the packaging for the recursion and the pruning in one function. It splits the given dataset, trains a decision tree with learn and uses pruning afterwards. All of this according to the input, whether pruning is wanted or not.

```
    def learn(X, y, impurity_measure, pruning):
        seed = 130
        if pruning:
            X_train, X_prun, Y_train, Y_prun =
model_selection.train_test_split(X, y, test_size= 0.1,
shuffle=True, random_state = seed)
            root = learn_rec(np.array(X_train), np.array(
Y_train), [], impurity_measure)
            root = pruning_function(root, np.array(X_prun),
np.array(Y_prun))
        else:
            root = learn_rec(X, y, [], impurity_measure)
        return root
```

3 Implementation of the impurity measures (Benjamin)

Both, entropy as well as the Gini-impurity, need for its calculation only the probabilities of different possible values. The latter is computed by the function `probability_list` which takes a list of numbers and first counts the occurrence for all the different values. This is done through a for-loop over all elements in the input-list.

```
    for i in range(count_values):
        if (values[i] not in value_list):
            value_list.append(values[i])
            prob_list.append(1)
            different_val +=1
        else:
            for j in range(different_val):
                if(values[i]==value_list[j]): prob_list[j]+=1
```

If the value of the viewed element didn't occur before, it gets appended to the `value_list` while `prob_list` saves the value 1 at the same index. Else the algorithm searches the index in the `value_list` with the right value

and increments the value in the `prob_list` at the right index. To get the probabilities, every numbers of occurrence is divided by the total length of the input-list.

```
    for k in range(different_val):
        prob_list[k] = float(prob_list[k]) / count_values
```

In order to get the entropy and Gini-impurity, one can first compute this `probability_list` and implement the summation with a simple for-loop.

```
    def entropy(values):
        prob_list = probabiliy_list(values)
        different_values = len(prob_list)
        sum = 0
        for i in range(different_values):
            sum -= math.log(prob_list[i], 2) * prob_list[i]
        return sum

    def gini(values):
        prob_list = probabiliy_list(values)
        different_values = len(prob_list)
        sum = 0
        for i in range(different_values):
            sum += prob_list[i]*(1-prob_list[i])
        return sum
```

4 Implementation of the decrease in impurity through a split (Benjamin)

The value of decrease in impurity is computed by the function `information_gain`, which takes a feature-vector x, a label-vector y and a string describing the impurity-measure as input and is implemented in three steps. First, the mean value of all values in x is calculated. This is done by a simple for-loop over all values in x.

```
    mean = 0
    count = 0
    for i in range(len(x)):
        mean += x[i]
        count +=1
    mean /= count
```

Secondly, the split of the label-vector is realised in two lists `leq_mean` and `g_mean` containing all values from x less equal or greater than the mean. Additionally, the percentual occurrence of the values in `leq_mean` is computed in `prob_leq_mean`.

```
    leq_mean = []
    g_mean = []
    count_leq_mean = 0
```

```
    for i in range(len(x)):
        if(x[i] <= mean):
            count_leq_mean +=1
            leq_mean.append(y[i])
        else:
            g_mean.append(y[i])
    prob_leq_mean = count_leq_mean / count
```

Lastly the Impurities before and after the split are calculated using the already implemented functions `entropy()` or `gini()`.

```
    f(impurity_measure=="entropy"):
        impurity = entropy(y)
        conditional_impurity = prob_leq_mean * entropy(
leq_mean)+ (1-prob_leq_mean) * entropy(g_mean)
    elif(impurity_measure=="gini"):
        impurity = gini(y)
        conditional_impurity = prob_leq_mean * gini(leq_mean)
+ (1-prob_leq_mean) * gini(g_mean)
    else:
        print("Not known impurity measure")
```

Using that, the function returns the value of decrease in impurity through the split, which is the information gain in the case of entropy.

```
    return impurity - conditional_impurity
```

5 Implementation of predict (Benjamin)

This function gets as parameter a vector x and returns the predicted label. Generally, it is implemented as recursion starting with the root and then continuing with the child for which the value of x at the looked at feature fits. If the (sub-)tree is a leaf, that is when the (sub-)tree's children-list is empty, then the function returns the label memorized in node.data.

```
    if (not node.children):
        return node.data
```

Otherwise it will recursively call predict on the right child as well as the left child. For this the children list of the current node is used.

```
    if(x[node.data]<= node.children[0][1]):
        return predict(node.children[0][0],x)
    else:
        return predict(node.children[1][0],x)
```

Remark: It can't happen that a node has only one child because the binary split is always creating two children.

6 Implementation of pruning (Benjamin)

The pruning-function takes as input a node representing a (sub-)tree and the pruning data `X_prun` and `y_prun`. It returns a node representing the post-pruned input-tree. First, the pruning function is searching the leaves of the tree by recursively calling itself on its children using the split data set according to the feature stored in `node.data`.

```
    elif (len(node.children)!=0):
        X_right, X_left, y_right, y_left, val_avg =
split_dataset(X_prun, y_prun, node.data)
        pruning_function(node.children[0][0], np.array(X_left
), y_left)
        pruning_function(node.children[1][0], np.array(
X_right), y_right)
    else:
        #reached leaf
        return node
```

Afterwards, it goes back up the tree and checks for every node whether the accuracy of the (sub-)tree on the pruning-data is lower than the accuracy when predicting the majority label. In this case, we change the node into a leaf predicting the majority label.

```
    if (acc(node, X_prun, y_prun) <= majority_label_acc(node.
majority_label, y_prun)):
        node.data = node.majority_label
        node.children = []
    return node
```

The accuracies are implemented through a simple for-loop counting the right predictions.

```
    def majority_label_acc(majority_label, y_prun):
        x = 0
        for i in range(len(y_prun)):
            if(y_prun[i]==majority_label):
                x += 1
        return float(x) / len(y_prun)

    def acc(node, X, Y):
        x = 0
        for i in range(len(Y)):
            if(Y[i]==predict(node, X[i])):
                x += 1
        return float(x) / len(Y)
```

7 Evaluation (Benjamin, Sophie)

First, we split the Data into Training-, Evaluation and Test-data.

```
    seed = 5
    X_train, X_val_test, Y_train, Y_val_test =
   model_selection.train_test_split(X, y, test_size= 0.2,
   shuffle=True, random_state = seed)
    seed = 221
    X_val, X_test, Y_val, Y_test = model_selection.
   train_test_split(X_val_test, Y_val_test, test_size= 0.4,
   shuffle=True, random_state = seed)
```

We chose a ratio of 0.8, 0.12, 0.08 to have sufficient training-data whilst getting representative accuracies. The training dataset is also big enough to be splitted for pruning. Afterwards we implemented four trees, one for each impurity measure with and without pruning and computed the accuracy on the training- as well as validation-data. This example shows one the procedure for the tree using entropy and pruning.

```
    tree_ent_prun = learn(X_train, Y_train, impurity_measure=
   "entropy", pruning=True)
    val_acc_ent_prun = acc(tree_ent_prun, X_val, Y_val)
    train_acc_ent_prun = acc(tree_ent_prun, X_train, Y_train)
```

We got the following results:

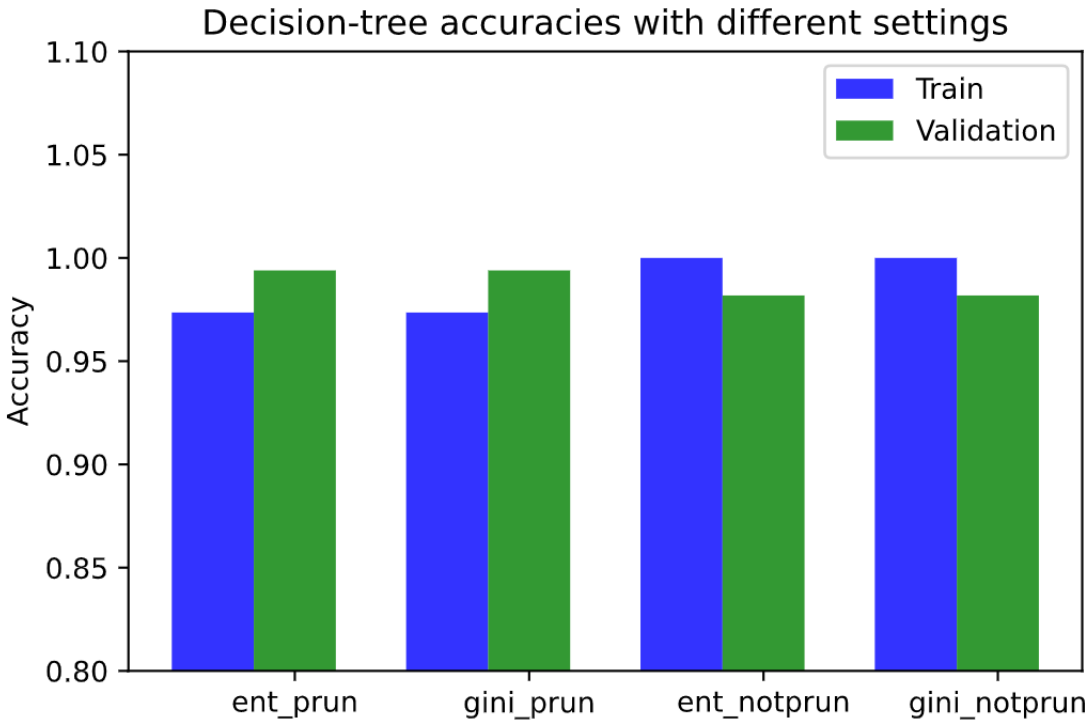

Figure 1: Results

The accuracies of the trees don't change for the different impurity measures. Moreover not using pruning, the training-accuracy is higher than the validation-accuracy, which was expected as it is susceptible to overfitting. As

the validation-accuracy is higher using pruning, we decided to use the tree trained by using entropy and pruning.

Remark: We played around with the seeds for the splitting and observed, that different seeds lead to very different results. The accuracies differ a lot between seeds, which suggests, that there is no clear pattern in the data points that can be learned. One example of very different accuracies for our trained models is displayed in figure 2. Here the accuracies for the pruning models do not match the expected results.

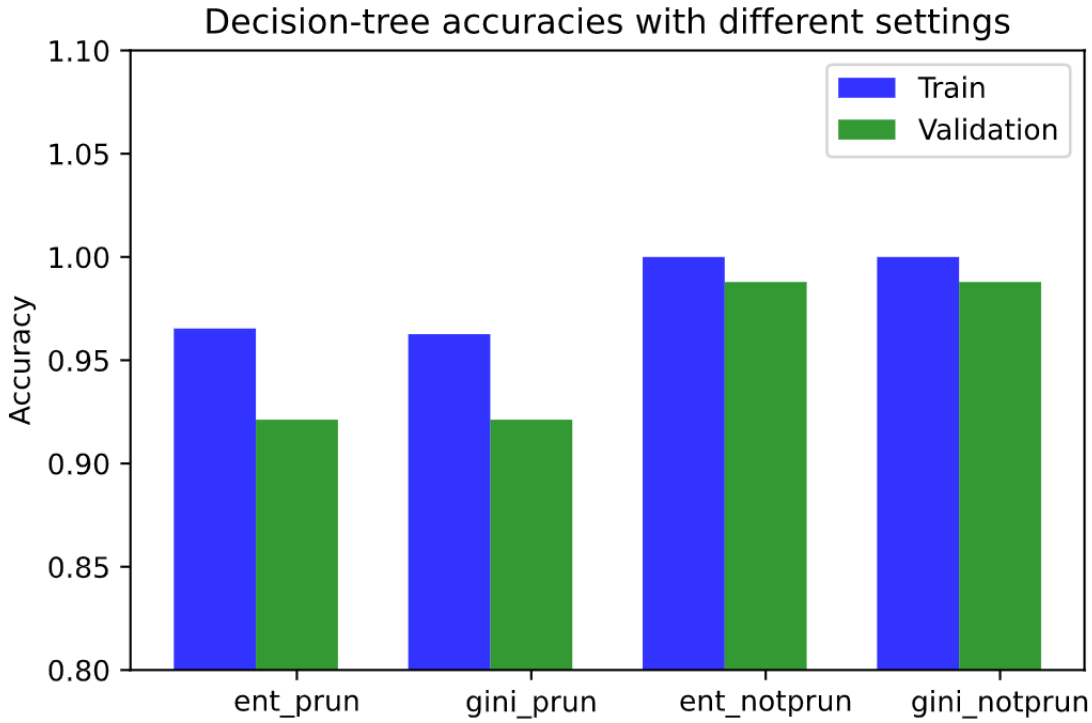

Figure 2: Different seed

8 Testing (Benjamin, Sophie)

We chose the model that uses entropy as the impurity measure and pruning, because the validation accuracies were better on the models using pruning. The impurity measures had no difference in their accuracies so we decided to use entropy arbitraryly. We now tested the chosen tree to get a representative estimate of the accuracy. For this we are using the testing data set, that has been unused up to this point, to get accurate testing results.

```
1    test_acc = acc(tree_ent_prun, X_test, Y_test)
```

The test-accuracy was with 0.9364 lower than the according training- and validation-accuracy. Nevertheless, the difference was less than 5% and a

	Learn time	Test time
Our decision tree	0.55307	0.00424
sklearn decision tree	0.01143	0.00084

Table 1: Time measurements for both implementations

lower test-accuracy is to be expected as we used the other accuracies to choose this tree. and the test data set was never seen by the model before.

9 Comparison to existing implementations (Sophie)

9.1 Time

To measure the time performance of both implementations, we measured the time before and after execution of the crucial parts and compared those. The same code is used for every measurement.

```
1     t1 = time.monotonic()
2     DecisionTree = DecisionTreeClassifier(criterion="entropy"
     )
3     DecisionTree.fit(X_train, Y_train)
4     t2 = time.monotonic()
5     sk_learn_time = t2 - t1
```

The time measurements (table 1) show, that the given implementation from the sklearn library is a lot faster. In learning and testing it is 10 times faster than our implementation. This is as expected for many reasons. The sklearn decision tree doesn't use pruning per default, which saves a lot of time. The library probably uses optimized data structures to build the tree as well as optimized algorithms. In the design of our decision tree we didn't consider time a critical factor and therefore didn't pay attention to time-saving code and functions.

Remark: The time measurements yield different values for every execution, so that the values are not exactly reproducable.

9.2 Accuracy

Figure 3 shows, that the classifier from the sklearn library perfoms better on test and training data. This is expected behaviour, as the sklearn classifier is most likely better optimized than our decision tree classifier. Our decision tree performs better on validation data, which can be explained by the fact, that it uses pruning. As already mentioned, different seeds for the splitting lead to very different results in the accuracies. This would also explain the different accuracies on validation data for the two different models.

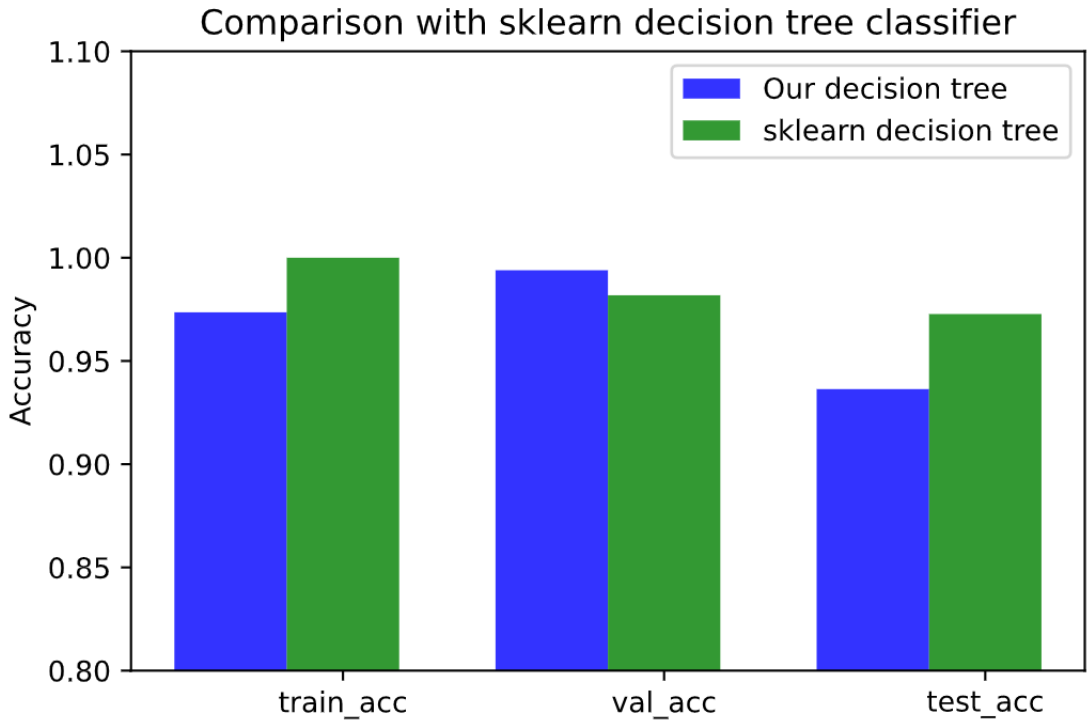

Figure 3: Comparison with sklearn