Project 1 - Decision trees

A project by

Brage Hamre Skjørestad and Martin Flo Øfstaas

Division of labor:

We worked together on the programming, often discussing the logic and how to do things. We worked around 50-50 on the project.

Data Analysis

This dataset contains 6 columns, with one binary classification column. All rows (or feature values) with a target 0 is a white wine, and with 1 a red wine. The columns are the features of a wine. Citric acid, representing the sourness of the wine,residual sugar representing the sweetness, and sulfates representing the saltiness. We also have pH values and alcohol levels. The dataset has not been modified in our code, only converted from pandas to numpy as the task did not mention it. It is worth noting that the dataset contains 50%-50% of both types of wine.

Implementation

2 programs:

- tree.py
- evaluate.py

Packages:

- pandas (only read_csv)
- numpy
- sklearn (for evaluating)
- time
- sys

In the beginning, we used both pandas and numpy. We started with pandas as the dataset is originally in a pandas dataset. However, while building the tree, we found that we wanted to use certain numpy methods, such as argwhere() and bincount(). Argwhere() made splitting the data with the threshold much easier. Then we would experience some errors while running our code, and eventually chose to stick to numpy and just convert the dataset in the very beginning. We use sklearn´s train_test_split() to split between train, prune, validation and test

data. We use their accuracy score to calculate correct predictions, and their own decision tree classifier to compare with ours. Finally, we use the time.time() method to see our model´s run time, and sys to be able to print with overwrite.

tree.py:

We decided early to create a class, instead of having many functions. Then we can create an object for different versions of our classifier. We also decided to use recursion in multiple methods, as it seemed the best way to traverse our tree.

Now for explaining the code. All methods are marked and explained below.

After creating an instance of the class, you call on the learn() method to train the model. Learn can be called 4 different ways (prune or not, gini or entropy). If you call with entropy, our attribute self.impurity_measure is set as the entropy function. And the same with gini.

Calling learn without pruning: Impurity measure is defined, and our second attribute (self.root_node) is set as the return of the build tree method. The return of the build tree method is a large and complex dictionary, representing our tree from the root node. Build tree is a recursive function.

Information_gain(): 3 parameters, feature column, label column (y) and threshold. It calculates “information gain” for a given set of a column and a label column. uses the attribute self.impurity_measure, which is either the **gini()** or the **entropy()** function. They are methods which use different formulas to calculate the amount of information in an input. We input the labels, and the methods returns a number between 0 and 1. Information_gain() splits the data with **split_graph()**, and we can then use the information gain formula to get the return value to represent information gained for the input column.

Build_tree(): It finds the best split feature for its feature values input, and splits the data based on the threshold for the best split column. The best split is found by looping through the feature columns and checking the information gain for every one, and creating a variable representing the column with the highest information gain. The split is done by the **split_graph()** method, which returns the indices of the rows over the threshold, and under (or equal) it. With these indices, we can split the data into 4 datasets, “left” (over threshold) and “right” (under or equal) for both the features and the labels. We return a “node” at each call, with information about that particular node. This node calls the function itself twice, once for each of the datasets. Build tree repeats this process for the new inputs, and continues to do so until it reaches a stop condition. Which is if all the feature values are identical or if all the data points have the same label. In our dataset, the feature values are never identical so only the one stopping condition is reached. However, when it reaches this stop condition, it returns a leaf node containing the label. Then we get ‘left (or right) subtree’: leaf node. This results in the final output of build_tree being a dictionary with a great number of nested dictionaries within itself. We can use this dictionary by recursively traversing it. Now we come

to our prediction functions. We first started by trying to be able to predict 1 datapoint, as we thought it was a good start.

predict_singe(): It first initializes the node to our root_node from build_tree(). The tree is now built, and we are ready to traverse it. Our dictionary is set so that we can access information about the top node at a certain point. In predict_single() we first need the feature index and threshold. The first split of the data set is on residual sugar. So the first step is to check if the residual sugar value of the input is above, equal or below the residual sugar threshold. If it's bigger, the function recursively calls itself, but this time with a node input representing the entire left subtree of the root node (right subtree if below or equal). Now it calls predict_single but with the left subtree, and the process is repeated. Now the best split feature may be another, since the rows have splitted. It continues this process until a subtree contains a leaf node. Then, when the function is called with this subtree as a node, it returns a label. Then the function stops.

predict_set(): When we were done with the predict_single(), we realized that the hard work was done. Now we create a loop to iterate through every data point in the set, and call the predict_single for that data point. We just needed to create an empty array we can append to store all labels.

prune_tree(): This function is only used if an object calls learn with prune = True (or not False to be precise), and with prune_data. Then, we first define prune_X and prune_y in learn to call the prune_tree() method. To use this function we first need to have a built tree, but we can use our attribute self.root_node as that is our entire tree. This function is also recursive, but unlike our other recursive function, our steps (the code) is implemented after the recursive call. That is because reduced error pruning is done bottom up. When we call recursively first, we can take advantage of recursion unwinding. We realized this quite late, but it made the code a lot smoother. With unwinding, it first completes the recursion for all the subtrees, and then starts from the bottom when running the code below. We can then prune according to the accuracy applied on the pruning data if we predict with our current node, or if we just use the most common label. We prune if the most common label accuracy is better or equal. To check how much of the tree that gets pruned, we added two extra attributes (just to check, they are not in the final program), self.count_nodes and self.count_pruned_nodes. Count_nodes was incremented in the recursion unwinding for every call. Count_pruned_nodes was only if the parent accuracy was equal or better. They got values 371 and 125, which means about 33% of the nodes were pruned.

Evaluate.py:

We created a separate program to evaluate our decision tree. This is a simple program, which starts by reading the wine dataset and creating variables X and y for the feature columns and the target column. From there, we split the data and find our best model by

calculating accuracy on the validation data. Finally, we compare our best model with the sklearn model.

Results

We split the data into 30% test data and 70% train data. The 30% test data is then split into 50% validation data. So, 15% test data, 15% validation data, and 70% train data.

seed = 5

```
model: {'impurity': 'gini', 'prune': False} - validation accuracy: 0.875
model: {'impurity': 'gini', 'prune': True} - validation accuracy: 0.9
model: {'impurity': 'entropy', 'prune': False} - validation accuracy: 0.879167
model: {'impurity': 'entropy', 'prune': True} - validation accuracy: 0.902083
best model: {'impurity': 'entropy', 'prune': True}
our classifier - score: 0.910417, time: 0.921559
sklearn classifier - score: 0.89375, time: 0.012525
```

seed = 123

```
model: {'impurity': 'gini', 'prune': False} - validation accuracy: 0.8875
model: {'impurity': 'gini', 'prune': True} - validation accuracy: 0.8875
model: {'impurity': 'entropy', 'prune': False} - validation accuracy: 0.895833
model: {'impurity': 'entropy', 'prune': True} - validation accuracy: 0.885417
best model: {'impurity': 'entropy', 'prune': False}
our classifier - score: 0.9, time: 0.304968
sklearn classifier - score: 0.902083, time: 0.005037
```

seed = 500

```
model: {'impurity': 'gini', 'prune': False} - validation accuracy: 0.879167
model: {'impurity': 'gini', 'prune': True} - validation accuracy: 0.895833
model: {'impurity': 'entropy', 'prune': False} - validation accuracy: 0.879167
model: {'impurity': 'entropy', 'prune': True} - validation accuracy: 0.89375
best model: {'impurity': 'gini', 'prune': True}
our classifier - score: 0.897917, time: 0.977806
sklearn classifier - score: 0.895833, time: 0.004794
```

seed = 800

```
model: {'impurity': 'gini', 'prune': False} - validation accuracy: 0.875
model: {'impurity': 'gini', 'prune': True} - validation accuracy: 0.891667
model: {'impurity': 'entropy', 'prune': False} - validation accuracy: 0.872917
model: {'impurity': 'entropy', 'prune': True} - validation accuracy: 0.885417
best model: {'impurity': 'gini', 'prune': True}
our classifier - score: 0.866667, time: 0.005302
sklearn classifier - score: 0.872917, time: 0.000383
```

Our best model varied from seed to seed, as they are very similar in their predicting. The accuracy varied also, but never going below 85% (from the seeds we tested). We tested a lot of seeds, and have included 4 of them here. Sometimes it performs better than sklearn, and sometimes not. Usually, our test data has a higher accuracy than validation data, but not always (seed = 800). Higher validation accuracy may indicate overfitting, but since it is not happening consistently, we would rather say it is random.

We find these results to be satisfying. Of course our model takes a little more time to run than the sklearn model, but we often get the same accuracy. The time we print in our screenshots is the time to build the entire tree and then predict. That is why the pruning models take more time, because you build the tree and then prune. When timing our predictions, the time with and without pruning was very similar, but pruning slightly faster.

```
model: {'impurity': 'gini', 'prune': False} - validation accuracy: 0.982038
model: {'impurity': 'gini', 'prune': True} - validation accuracy: 0.985337
model: {'impurity': 'entropy', 'prune': False} - validation accuracy: 0.981672
model: {'impurity': 'entropy', 'prune': True} - validation accuracy: 0.984971
best model: {'impurity': 'gini', 'prune': True}
our classifier - score: 0.984238, time: 0.025482
sklearn classifier - score: 0.983138, time: 0.003278
```

To check our code´s flexibility, we downloaded another dataset called “rice.csv”. In that dataset, there were 10 features and around 18000 rows. The results are in the screenshot above. Here, we timed only the predictions. Still a difference, but we see that our model does not take a very long time even for bigger data sets. It is building and pruning the tree which takes time, but that happens only once. When we just time predictions, we can see that the

difference is smaller than when we also timed the process of building and pruning. It depends on what happens with even bigger data sets, but since accuracy is usually very similar between ours and sklearn, their model is probably better than ours due to the run time (especially in the building of the tree).
Sklearn´s model is definitely more optimized, and possibly utilizes even more pruning and generally more efficient methods. We are however satisfied to have created a model that can keep up with sklearn.

Interestingly we can see that the model varies quite a bit in performance for the different seeds given. On the test data the accuracy score fluctuates between ~0,91 and ~0,86 for our model. A 5% difference between the models purely based on seeds. This indicates that the performance of the model is highly correlated to how the data is splitted in the dataset. This assumption gets reaffirmed when the sklearn model also has the same fluctuations on test data (~0,90 and ~0,87) and even more importantly, the accuracy score on validation is only 1% +- of our accuracy scores for each seed. This makes us believe that the dataset given in this task is simply too small to give our model a “generalized” accuracy score. We believe this is also the case for why the model is not consistent on choosing hyperparameters. To test our theory we checked our model again on the bigger “rice” dataset. Here we get a very consistent accuracy score on the test data and the best model chose ‘gini’ and Prune = True, 5/5 times. This reassures us that our model will eventually converge towards the best hyperparameters of the given dataset, provided the dataset allows for such training.

Running our code:

We upload two python files, tree.py and evaluate.py. After saving the two files in the same path, you should be able to simply run evaluate.py to get our output. If you want to check different random states, changing the seed variable will ensure the same random_state for all train_test_splits. Our solution allows for using pandas to read the data frame, but it requires the feature values and the target labels to be sent in as numpy arrays for efficiency. It will run for every dataset, as long as the target column is binary, and all feature columns are numerical.