DECISION TREE report

Gaia Casotto
Fabio Gaiba

23th September 2022

1 Design decisions made:

- the decision tree is always *binary*, no matter the dataset it is trained on. However, it is built so that it can deal with datasets that have more than just two types of labels
- class **Tree**, within which all functions pertaining to the building and pruning, the accuracy calculations and the predictions are defined. An instance of **Tree** has:
  1. a reference to the root node of the tree
  2. a measure of its' accuracy
- The root of the tree is an instance of the class **Node**. An instance of **Node** has:
  1. a reference to parent node (if root, then parent = None)
  2. a label (if not leaf, then label is equal to the majority label of the subdataset associated with the node)
  3. a reference to the subdataset that results from the split that creates the node itself
  4. references to a lower child node (which refers to those lines of the dataset that have a lower nth_feature value than that of the threshold) and a greater child (which refers to those lines of the dataset that have a greater nth_feature value than that of the threshold)
- the *threshold* used for splitting datasets is the average of all values of a particular feature in the dataset: this makes the code more computationally efficient, although splitting the dataset for each value of a feature column would return a better information gain for that column, and so a more efficient split of the dataset
- class **Dataset**: an dataset is compromised of
  1. a feature matrix X and a label vector y
  2. a label_table in which the pairs (label, count) represent the umber of occurrences of a certain label within the label vector

3. an impurity measure, calculated either with 'gini' or 'entropy'

Any feature matrix X and label vector y passed to the learn(...) function will be used to create an instance of class **Dataset**

- **decisionTree.py** in which class **Tree** is implemented.
- **utilities.py** contains service functions, like
  1. functions for calculating the impurity measures and the split value for a column of a dataset
  2. class **Dataset**
  3. get_data() (with which the dataset *magic04.data* is loaded)
  4. split_dataset(X, y, percent=0.8), a method to split data into two different sets of label vectors and feature matrices. This is used to distinguish the data points that will be used for training/validation/testing.
- **datastructs.py** contains the two data structures used within class **Tree**:
  1. class **DecisionData**, which is used to store the characteristics of the sub-dataset created by a split, and so:
     - the majority label, which is the label with most occurrences within the dataset
     - the "question" a.k.a. the threshold value that determines how the dataset is split at a certain node
     - the nth_feature, which is the index of the feature value that has to be compared to the question in order to split the dataset
     - a feature matrix X and a label vector y
  2. class **Node**
- **sklearnTrees.py**, in which the sklearn.tree is tested.
- **modelSelection.py**, in which the best model is chosen and then tested.

2 Training the tree:

- The **chose_feature(...)** method decides on which feature to split the data set on. For each column, it calculates the average value of that feature, and then splits the data according to whether the value of that feature is greater or lower than the average. It then calculates the information gain, and checks whether this newly calculated information gain is greater than the previously calculated one or not. If it is, then it saves the index of the feature, the two resulting data sets and the average, which it then returns if no greater information gain is found.
  Its' parameters are: a dataset and an impurity measure.

```
def choose_feature(self, data, impurity_measure):
best_info_gain = None
index = -1
split_val = 0
best_split = []
for nth_feature in range(len(data.X[0])):
    average = ut.average(data.X, nth_feature)
    data1, data2 = ut.split_average(data, average, nth_feature, impurity_
    info_gain = ut.info_gain(data, data1, data2)
    if best_info_gain == None or info_gain > best_info_gain:
        best_split = [data1, data2]
        best_info_gain = info_gain
        split_val = average
        index = nth_feature

return index, split_val, best_split
```

- **learn(...)** is the function that trains the tree, and if specified, also does the reduced-error pruning.
  This function returns a node (the root of the tree).
  Its' parameters are :
  - a matrix of data points
  - a vector of labels
  - an impurity measure
  - a boolean to specify if the pruning is to be done or not

  1. `train(...)` is a recursive function that creates the nodes, dividing the datasets until the y column of a dataset associated to a node is completely pure (it presents only one type of label) or if all data points of said dataset have all the same feature values. In these cases, it returns a leaf node with an appropriate label.
     This function returns a node, and its' parameters are
     - a dataset
     - an impurity measure
     - a node to pass as the parent node to the children in training
  2. `prune_all_subtree(...)` is the recursive method through which the `prune(...)` method is called on the entire tree. It is in reality an implementation of a post order traversal of a tree, where each child is visited (in this case, pruned) before its' parent.
  3. `prune(...)` calculates the accuracy of the tree, then temporarily places a leaf with the appropriate majority label instead of the node that the function is

being called on, and calculates the accuracy of the tree again. If this new accuracy is less than the previous accuracy, then the function inverts the temporary switch made and the tree is left unchanged. Else, the temporary switch becomes permanent. Its' parameters are: {a node, a feature matrix and a label vector}

- `accuracy(....)` saves all predictions of a certain feature matrix X inside a vector, and then counts how many predicted labels are correct.

```
good_predicitons = 0
for i in range(len(X)):
res = self.predict(X[i])
if res == y[i]:
    good_predicitons += 1
```

Its' required parameters are a feature matrix X and a label vector y.
It returns the division between the number of good predictions made and the total amount of predictions made.

3 Impurity measures

Both gini and entropy calculation methods are found in the **utilities.py** file.

- **entropy**

```
def entropy(label_table,y):
    entropy = 0
    for label_count in label_table.values():
        entropy -= (label_count/len(y) math.(log(label_count/len(y), 2) )
    return entropy
```

where label_table is one of the instance variables of the class Dataset, and y is a label vector.
The formula for entropy is

$$H(X) = -\sum_{i=1}^{n}(pi * \log_2 pi)$$

- **gini index**

```
def gini(label_table,y):
    gini = 0
    for label_count in label_table.values():
```

```
            gini += (label_count/len(y) * (1-(label_count/len(y))))

        return gini
```

The formula for gini index impurity is

$$G(x) = \sum_{i=1}^{n} (pi) * (1 - pi)$$

4 Predicting

`predict(self, x, node="default")` travels down the branches of the tree until it reaches a leaf. To chose which child to travel down to next, the method compares the split value (saved in the node) with the value of the nth element in the feature vector of which it has to predict a label (n = index saved within the node)

```
...
index = node.decData.nth_feature
question = node.decData.split_val
...
```

It returns a the label of that leaf.
It's parameters are

- a vector of data features
- a node that is set to `None` by default.

```
def predict(self, x, node=None):
    if node == None:
        node = self.root
    while not node.is_leaf():
        index = node.decData.nth_feature
        question = node.decData.split_val
        if x[index] > question:
            node = node.greater
        else:
            node = node.lower
    if node.is_leaf():
        return node.decData.majority_label
    else:
        raise "Leaf is not leaf" #error check
```

5 Evaluation of algorithm

following are the performance measures obtained for *one* possible split of train data and test data.

- model 1: **'entropy'**
  number of nodes: 2575
  training accuracy: 1.0 validation accuracy: 0.805

- model 2: **'gini'**
  number of nodes: 2586
  training accuracy: 1.0 validation accuracy: 0.799

- model 3: **'entropy' with pruning**
  number of nodes: 381
  pruning time: 30.79
  training accuracy: 0.887 validation accuracy: 0.845

- model 4: **'gini' with pruning**
  number of nodes: 356
  pruning time: 30.67
  training accuracy: 0.883 validation accuracy: 0.834

from this example we can see that the best model to chose would be model 4 (setting: 'entropy', prune = True):

- pruning decreases the size of the tree by about 2200 nodes.
- validation accuracy is highest amongst all models
- accuracy on validation data increases by about 4%
- it does not have the lowest amount of nodes (settings: 'gini', prune = True) has less nodes

This was expected: pruning is done so that the decision tree, which would usually tend to overfit, generalizes better. This is confirmed by the fact that the two non pruned trees have 100% accuracy in training data, and then a much much lower accuracy on validation data. Training accuracy in the pruned models, when confronted with the validation accuracy, shows that pruning does in fact reduce overfitting, as the two measures are very close.

- testing accuracy of best model: 0.842

6 Comparison with sklearn.tree

The sklearn decision tree is implemented so that the default impurity measure is the gini index. By default, calling fit(X,y) does not prune the tree. Pruning for the sklearn tree is done with cost complexity pruning, which is a more labour-intensive pruning algorithm than reduced error pruning.
Two non-pruned examples of the sklearn tree are created, one with gini index as the impurity measure and the other with entropy.
The accuracy of the trees is calculated using the `sklearn.metrics` accuracy function.
In this implementation, validation accuracy score tends to be higher when using entropy, which also has less nodes and a very close test accuracy score.

- 2907
- training accuracy -> 0.1
- validation accuracy -> 0.836
- testing accuracy -> 0.828

-> Comparing it with one of the non pruned models from above (in this case, since it performed slightly better, model 1).

- both models **overfit** (training accuracy » validation accuracy)

These two instances of decision trees perform approximately in the same way.

Now for the comparison between pruned trees: reduced-error pruning is in general faster than cost complexity pruning, since ccp first has to find the best value to prune with, and this takes a long time. For this reason, the sklearn tree is pruned with a random value from within the ccp_alpha vector created by the function **cost_complexity_pruning_path(trainX, trainY)**.
For a certain alpha the resulting tree performs this way:
validation accuracy score with gini, prune 0.867
test accuracy score with gini, prune 0.846
number of nodes = 691

This is probably not the best result that can be obtained from a sklearn when pruned, but the alpha has been chosen randomly.

```
bestAcc = 0
for _ in range(100):
    ccp_alphas, impurities = path.ccp_alphas, path.impurities
    index = random.randint(0, len(ccp_alphas))
    alfa = ccp_alphas[index]
#print(alfa)
    decTree3 = tree.DecisionTreeClassifier(random_state =0, ccp_alpha = alfa)
    decTree3.fit(trainX, trainY)
```

```
tr_pred3     = decTree3.predict(trainx)
tr_score3    = accuracy_score(trainY, pred3)
pred3        = decTree3.predict(val_x)
val_score3   = accuracy_score(val_y, pred3)

if val_score3 > bestAcc:
    bestAcc = val_score3
if tr_score3 > tr_bestAcc:
    tr_bestAcc = tr_score3
```

These results are slightly higher, but very close to the values obtained from the best model selected earlier. Neither pruned model seems to be overfitting.

7 Division of labour

- design choices made together
- learning, optimization, performance measuring: Fabio Gaiba
- predict and pruning, sklearn comparison, assignment report: Gaia Casotto