Implementing Decision Trees

INF283 – Project 1

Sindre E. de Lange

Implementing the ID3 algorithm from scratch

Learn()

```python
def learn(X, y, imp_measure_alt='entropy', pruning=False, pruning_amount=0.30):
    """ Learn

    Learns a decision tree classifier from data.
    NOTE: Expects cleaned data, in particular categorical, discrete values (pd.factorize)

    Args:
        X: pandas dataframe
        y: pandas series
        imp_measure_alt: String. How to calculate the information gain for the datasets column.
            Either 'entropy' (standard) or 'gini'
        pruning: Boolean. To use pruning, or not to use pruning - that is the question
        pruning_amount: Float. Percentage distribution of the training dataset

    Returns:
        treelib.tree.Tree. Tree classifier learned from data
        """
```

As stated in the text it uses impurity measure = 'entropy' by default, but is possible to edit to 'gini' if the user wishes to do so.

When running this code (without pruning, seeing as this is one of the later tasks) it provides the user with a trained tree.

Example 1: Using the tennis dataset (as shown in class, and available in the delivered folder under 'cvs/tennis.csv'). Easier example to explain.

```
outlook
├── 0
│   └── humidity
│       ├── 0
│       │   └── 0
│       └── 1
│           └── 1
├── 1
│   └── 1
└── 2
    └── windy
        ├── 0
        │   └── 1
        └── 1
            └── 0
```

The names 'outlook', 'humidity' and 'windy' are names of the columns which the program deemed the most beneficial to split the branches on (based on the user-defined 'impurity measure' argument). While the numbers, except for the leaves, represents unique values in each of the columns. Lastly, the numbers in the leaves represents the prediction label for this specific path in the decision tree. Example: If the data row has value = 0 in the 'outlook' column, it would move to the 'humidity' node, then based on the value in this ('humidity') column, say 0, it would predict the data as to have label = 0.

Example 2: Using the mushroom dataset (available in the delivered folder under ‘cvs/mushrooms.csv’)

```
odor
├── 0
│   └── 0
├── 1
│   └── 1
├── 2
│   └── 1
├── 3
│   └── spore-print-color
│       ├── 0
│       │   └── 1
│       ├── 1
│       │   └── 1
│       ├── 4
│       │   └── 0
│       └── 5
│           └── cap-color
│               ├── 0
│               │   └── 1
│               ├── 1
│               │   └── 0
│               ├── 2
│               │   └── 0
│               ├── 3
│               │   └── 1
│               ├── 5
│               │   └── 1
│               └── 7
│                   └── 1
├── 4
│   └── 0
├── 5
│   └── 0
└── 6
    └── 0
```

This is a larger dataset (than example 1) with more columns, and more unique values in these columns, however the same ‘rules’ apply here, as described in example 1, with regards to traversing the tree.

Impurity Measure

I decided to separate out the ‘Impurity Measure’ method, to a .py file, because it made the Notebook cleaner. This can be accessed in the delivered folder under ‘classes/ImpurityMeasure.py’. I followed the example on https://nullpointerexception1.wordpress.com/2017/12/16/a-tutorial-to-understand-decision-tree-id3-learning-algorithm/, when implementing these algorithms, and had no time to refactor, so the code is pretty long, however seeing as it is implemented from the step by step tutorial, it should be a rather easy read, with the combination of comments and docstring.

The goal in using an impurity measure method is to calculate how much information one gets from knowing the value for each column for a data point. This is relevant for a decision tree, because an optimal decision tree is as shallow as possible, without compromising the accuracy, i.e. the fewer the levels, the better. This is because of overfitting and run time. So, we try to get the name of the column that provides the largest information gain:

```
def getLargestInformationGain(self, X, y):
    """  Get Largest Information Gain

    Gets the largest IG for any given dataset

    Args:
        X: pandas dataframe
        y: pandas series

    Returns:
        String. Name of column that gives the best/largest information gain
    """
```

Calculate the impurity of an entire dataset:

```
def calc_impurity_dataset(self, X, y):
    """ Calculate Entropy Dataset

    Calculates the entropy of an entire dataset

    Args:
        X: pandas dataframe
        y: pandas series

    Returns:
        Dictionary. Keys = column names, values = their entropy
    """
```

by:

1. Calculate the impurity of the target variable

```python
def calc_entropy_system(self, target_variable):
    """ Calculate the entropy for an entire system

    Calculates the entropy of the (entire) system, i.e. target variable

    Args:
        target_variable: pandas series

    Returns:
        purity: float
    """
```

2. Calculate the impurity of each unique value in each column

```python
def calc_impurity_feature(self, X_y_zip, tot_num_occurences):
    """ Calculate Entropy Feature

    Calculates the necessary numbers, to calculate the entropy - store it in a dictionary.

    Args:
        X_y_zip: Dictionary. Key = column names, values = tuples: (column name value, target variable value)
        tot_num_occurences: int. Number of data points (length of columns)

    Returns:
        Dictionary.
            Key = column names,
            value = dictionary:
                Key = Unique value in the outer dictionary column
                value = [total number of days, total number of occurences, total entropy for unique value]

    """
```

3. Calculate the total impurity of each column

```python
def calc_impurity_all_branches(self, feature_entr_dict, entropy_src):
    """ Calculate the entropy for multiple columns/features

    Calculates the entropy for all branches in a python dictionary

    Args:
        feature_entr_dict: Dictionary, on the format
                {'column feature':
                    {unique value: [
                        number of this value occured in the set
                        number of values in the set
                        entropy when this value occured in the set
                    ], ... }}
        entropy_src: float.

    Returns:
        Dictionary. Keys = column names, values = their entropy
    """
```

4. Calculate the total information gain

```python
def randomness_reduction(self, entropy_src, entropy_branch):
    """ Randomness Reduction

    Calculates the reduction in randomness, aka Information Gain.

    Args:
        entropy_src: float. entropy of the entire system (target variable)
        entropy_branch: float. entropy for a single branch
    Returns:
        Information Gain: float - restricted to 3 decimals.
    """
```

Note: This is related to task 2, but depending on the user wishes to use ‘entropy’ or ‘gini’ to calculate the impurity measure, the program uses one of these two:

‘entropy’

```python
def calc_entropy(self, p):
    """ Calculate Entropy

    Calculate the entropy for a given fraction

    Args:
        p: fraction (float)
    Returns:
        float
    """
```

$$Entropy: H(E) = -\sum_{j=1}^{c} p_j \log p_j$$

from https://datascience.stackexchange.com/questions/10228/gini-impurity-vs-entropy

Note: There are (a few) more functions in the 'ImpurityMeasure' class, however I felt like these painted the picture in a satisfying way (and they are, as mentioned, available in the folder).

Make Tree

The **learn**-method utilizes the method **make_tree()** to populate a tree:

```
def make_tree(X, y, tree, imp_measure, current_node=None):
    """ Make Tree

    Recursive method to make a tree of the type treelib.tree.Tree

    Args:
        X: pandas dataframe. Training features
        y: pandas series. Target variable
        tree: treelib.tree.Tree. Tree object to populate
        imp_measure: String. Name of impurity measure - either 'entropy' or 'gini'
        current_node: treelib.node.Node. Current node to build subtree from.

    Returns:
        treelib.tree.Tree. A (populated) decision tree based on inputed datasets X and y
    """
```

This method basically finds the column in the dataset that is best to split on (using the 'ImpurityMeasure' object's **getInformationGain()**), makes a node with this name and the related data, then calls on **make_children()**:

```
def make_children(X, y, tree, imp_measure, current_node):
    """ Make Children for a specific column/node in a tree

    Identifies the unique values, in a column, in a dataset, initialized node corresponding to that
    value, and appends them to current node, i.e. their parent.

    Args:
        X: pandas dataframe.
        y: pandas series.
        imp_measure: String. Name of impurity measure - either 'entropy' or 'gini'
        current_node: Reference to a specific node, in a tree, that one wishes to populate with children

    Returns:
        treelib.tree.Tree. Inputed tree, with appended children of current node
    """
```

Which, as stated in the docstring, identifies the unique values in the column, separates out a subset of the data relevant for each of these unique values, initializes a node, and connects it the 'column'-node.

Predict()

This is self-explanatory – one inputs the data row one wants to predict on (e.g. it's label), and the tree classifier that should be used for this.

```
def predict(x, tree):
    """Predict class label of some new data point x.

    Takes in a row of data, runs it through a tree classifier, and outputs the classifiers predicted label.

    Args:
        x: pandas series. Data row to predict on
        tree: treelib.tree.Tree. Tree classifier to predict the data row's label

    Returns:
        int. Assuming the dataset is factorized, otherwise it will be whatever the values are in the target variable series.
    """
```

Predict() uses **getClassificationLabel()** in order to traverse the inputted tree classifier, to get the (hopefully) correct label

```
def getClassificationLabel(x, tree, current_node):
    """ Get Classification Label

    Recursive method that uses the inputed data row 'x' to traverse the decision tree,
    find the leaf that corresponds to 'x's data, and return its label/data

    Args:
        x: pandas series. Data row to predict on
        tree: treelib.tree.Tree. Tree classifier to predict the data row's label
        current_node: treelib.node.Node. Current node to check if one of its children is the correct leaf

    Returns:
        int. Assuming the dataset is factorized, otherwise it will be whatever the values are in the target variable series.
    """
```

How the trees are traversed through are explained in Example 1, so I will not repeat that here.

Gini

Giving the user an alternative to calculate the impurity with the 'Gini Index'.

```
def calc_gini(self, p):
    """" Calculate Gini Inidex

    Calculate the Gini Index for a given fraction

    Args:
        p: fraction (float)
    Returns:
        float
    """
```

$$Gini: Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

from https://datascience.stackexchange.com/questions/10228/gini-impurity-vs-entropy

So, depending on the argument inputted into the aforementioned **learn()**, the system uses either the already showed 'entropy', or this 'gini' to calculate the information gain.

Pruning

Since I was not able to implement pruning I will show that I have understood the principle of pruning.

Pruning basically means to use an unseen dataset to remove seemingly unnecessary nodes from a decision tree. This can, as I understand it, be implemented in two ways:

1. Run the pruning set on the tree, and for each data point in the pruning set, store whether the leaf one traversed to have the correct label (of the data point). Then traverse back to its parent, calculate the majority label here, and store whether this is equal to the correct label (of the data point). Specifically, this means that one stores the error rate in the leaves, and their parent. After one has done this for the entire data set, one traverses the tree, bottom up, and checks if the leaves error rate >= their parents error rate. If so, delete the leaves, as one would end up with an equal, or better accuracy without them, and one reduces the size/dimensionality of the tree.
2. Make three copies of the tree:
    a. Original tree
    b. Original tree, but changing one leaves parent so that their label is 0 (this is for binary classification)
    c. Original tree, but changing one leaves parent so that their label is 1.

    Then one runs the pruning set on all of the three trees, and calculates their accuracy. If one of the new trees have an accuracy >= the original tree, this is the tree one continues with, i.e. it becomes 'the original tree'. This is done for all nodes in the tree, starting at it's leaves, until a complete traversing, i.e. visited all of the nodes, is complete and the original tree has not changed.

Even though nr. 2 is a far slower method than nr. 1, I tried to implement nr. 2. It seems that my implementation deletes every node that appears:

The tree before pruning can be seen in example 2, and this is my tree after pruning:

```
odor
├─ 0
│  └─ 0
├─ 1
│  └─ 1
├─ 2
│  └─ 1
├─ 3
│  └─ spore-print-color
├─ 4
├─ 5
└─ 6
```

It is worth mentioning that my code does not stop, so ones it hits this stage, it just stays at the same node. This is because of my 'stop-criteria' but I am not able to pin point the specific problem or where it is.

Classify Edible and Poisonous Mushrooms

Data splitting – train, test

Before one split a dataset one has to clean the data, for this I made a .py class **DataCleaning** which can be accessed in 'classes/DataCleaning.py'. This class consists of three methods, however I only use two of them, so I will let the last one be.

Factorize()

```python
def factorizeDf(self, dataframe):
    """ Factorizes inputed dataframe

    Factorized a dataframe so that its data is categorical, discrete values, only

    Args:
        dataframe: pandas dataframe

    Returns:
        pandas dataframe
    """
```

removeQMarksDF()

```python
def removeQmarksDf(self, dataframe):
    """ Remove Question Marks from Dataframe

    Removes rows containing '?' from a dataframe

    Args:
        dataframe: pandas dataframe

    Returns:
        pandas dataframe
    """
```

When the data is clean, one can split it. After consulting with the group leaders in the course I chose to use Scikit-learns **train_test_split()** to split the data into 'train' and 'test'.

```python
target_var = 'class'
X_no_qmarks_fact = data_shrooms_fact_no_qmarks.drop([target_var], axis=1)
y_no_qmarks_fact = data_shrooms_fact_no_qmarks[target_var]
```

```python
X_no_qmarks_fact_train, X_no_qmarks_fact_test, y_no_qmarks_fact_train, y_no_qmarks_fact_test
= train_test_split(X_no_qmarks_fact, y_no_qmarks_fact, test_size=0.3, random_state=42, stratify=y_no_qmarks_fact)
```

So, optimally one would use **RandomSearch()**, or **GridSearch()**, to get the optimal hyperparameters here, however, when having the amount of data (approximately 5500) we have test_size = 30% is a fair starting point. Normally I would've probably started at test_size = 40%, but seeing as we might divide the 'train' further, with pruning, my experience said that this is better.

Also, it is worth mentioning the 'stratify' hyperparameter which makes sure that the data is split in a reasonable fashion, i.e. if the dataset consists of 70% 0's and 30% 1's this makes sure that this is the distribution in the 'test' and 'train' set as well.

Questions

Pruning vs. no pruning: I have chosen not to add pruning because of the fact that I was not able to implement the pruning method (as described in “Pruning”). However, I have understood the pros and cons of using pruning, so I will try to give a brief description of these. As mentioned under “Pruning” this is a method to decrease a decision tree’s size, by eliminating seemingly unnecessary nodes. By doing this one will decrease the risk of overfitting, because the model will not contain “pure leaves” only, that is, leaves with one unique value in its target variable, and therefore label. This will make the model more general (definition of decreasing overfitting) and shortens the run time. However, one run the risks of wrongly classifying some data points. This can be linked to the “curse of dimensionality” https://ieeexplore.ieee.org/document/7358700/ .

Gini vs. Entropy: As found in https://www.unine.ch/files/live/sites/imi/files/shared/documents/papers/Gini_index_fulltext.pdf , the difference between the two metrics matters in an a relatively small percentages of use case, however the Entropy metric might be a little smaller to compute, because it uses the logarithm. Therefore, I would postulate that it does not play a significant role which metric one chooses.

Implementation Comparison

Results

My model is not optimal (to say the least):

```
mushroom_tree = learn(X_no_qmarks_fact_train, y_no_qmarks_fact_train)
print(acuraccy(X_no_qmarks_fact_test, y_no_qmarks_fact_test, mushroom_tree))
```

```
0.6576151121605667
```

```
from sklearn import tree
from sklearn.model_selection import train_test_split
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X_no_qmarks_fact_train, y_no_qmarks_fact_train)
print(clf.score(X_no_qmarks_fact_test, y_no_qmarks_fact_test))
```

```
1.0
```

Here is my accuracy function:

```
def acuraccy(X_prune, y_prune, this_tree):
    """ Acuraccy

    Calculates the acuraccy: Number of errors/total data length

    Args:
        X_prune: pandas dataframe
        y_prune: pandas series
        this_tree: treelib.tree.Tree

    Returns:
        float. corrected predicted labels / total number of labels
    """
```

Which should be the same calculations as **clf.score()**

`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.

From http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Note: ‘mushroom_tree’ has the same structure as the one shown in example 2.