decision_trees

June 6, 2023

1 Introduction

- Decision trees are a type of model for predicting both continuous and categorical values.
- They classify data by partitioning the sample space efficiently.
- Decision trees are still one of the most powerful modeling tools in machine learning.
- They are highly interpretable and simple to explain. ## Entropy and Information Gain
- Decision trees can give different predictions based on the questions asked and their order.
- Selecting the right questions in the right order is crucial.
- Entropy and information gain are useful mechanisms for choosing the most promising questions in a decision tree.

1.1 From graphs to decision trees

- Decision trees are a type of classifier that partitions the sample space recursively.
- A decision tree is a directed acyclic graph with a root node and internal, leaf, and terminal nodes.
- Internal nodes have outgoing edges while terminal nodes have no outgoing edges.
- Directed Acyclic Graphs are collections of nodes and edges with specified traversal directions.
- Acyclic graphs are graphs where no node can be visited twice along any path from one node to another.
- DAGs are directed graphs with no cycles.
- DAGs have a topological ordering, which is a sequence of nodes with every edge directed from earlier to later in the sequence.

1.2 Partitioning the sample space

- Decision trees partition a sample space into sub-spaces based on attributes.
- Internal nodes check for a condition and perform a decision, while terminal nodes represent a class.
- Decision tree induction is related to rule induction.
- Each path from the root to a leaf can be transformed into a rule.

1.3 Definition

- Decision trees are a type of classifier where each node represents a choice and each leaf node represents a classification.

- Unknown instances are routed down the tree based on attribute values until they reach a leaf and are classified.
- Feature importance is crucial to decision trees as selecting the correct feature affects the classification process.
- Regression trees are represented similarly but predict continuous values instead of classifications.

1.4 Training process

- To train a decision tree for predicting target features:
- Present a dataset with features and a target
- Use feature selection and measures like information gain and Gini index to select predictors
- Grow the tree until a stopping criteria is met
- Use the trained tree to predict the class of new examples based on their features

1.5 Splitting criteria

- Decision trees are built using recursive binary splitting and a cost function to select the best split
- Two algorithms commonly used to build decision trees are CART and ID3
- CART uses the Gini Index as a metric while ID3 uses the entropy function and information gain as metrics.

1.6 Greedy search

- To classify data, we use decision trees with the best attribute at the root.
- We repeat the process to create further splits until all data is classified.
- The top-down, greedy search is used to find the best attribute.
- The information gain criteria helps identify the best attribute for ID3 classification trees.
- Decision trees always try to maximize information gain.
- The attribute with the highest information gain will be split on first.

1.7 Shannon's Entropy

- Entropy measures disorder or uncertainty.
- It is named after Claude Shannon, the "father of information theory".
- Information theory provides measures of uncertainty associated with random variables.
- The amount of uncertainty is measured in bits.
- The entropy of a variable is the "amount of information" contained in the variable.
- The amount of information is proportional to the amount of "surprise" its reading causes.
- Shannon's entropy quantifies the amount of information in a variable and provides a foundation for a theory around the notion of information.
- Entropy is an indicator of how messy data is.
- Higher entropy means less predictive power in data science.

1.8 Entropy and Decision Trees

- Decision trees are used to group data into classes based on a target variable.
- The goal is to maximize purity of the classes while creating clear leaf nodes.

- Data cannot always be fully classified, but can be made tidier through splits using different feature variables.
- Entropy is computed before and after each split to determine if it should be retained or stopped.

1.8.1 Calculating Entropy

- A dataset can contain both True and False values and be split into subsets according to their target value
- The ratio of Trues to Falses in the dataset can be calculated using p = n/N and q = m/N
- Entropy can be calculated using the equation E = -p . log_2(p) - q . log_2(q) and is a measure of the disorder or uncertainty in the dataset
- When the split between target classes is at 0.5, the entropy value is at its maximum, 1; when the split is at 0 or 1, the entropy value is 0
- The more one-sided the proportion of target classes, the less entropy; when the proportion is exactly equal, there is maximum entropy and perfect chaos
- Decision Trees can be used to split the contents of a dataset into subsets, creating more organized subsets based on common attributes.

[7]:
```
from math import log

# Write a function `entropy(pi)` to calculate total entropy in a given discrete␣
↪probability distribution `pi`
# The function should take in a probability distribution `pi` as a list of class␣
↪distributions. This should be a list of two integers, representing how many␣
↪items are in each class. For example: `[4, 4]` indicates that there are four␣
↪items in each class, `[10, 0]` indicates that there are 10 items in one class␣
↪and 0 in the other.
# Calculate and return entropy according to the formula: $$Entropy(p) = -\sum␣
↪(P_i . log_2(P_i))$$
# Make sure to avoid invalid operations like: $$log_2(0)$$

def entropy(pi):
    '''
    return the Entropy of a probability distribution:
        entropy(p) = - SUM (Pi * log(Pi) )
    '''
    total = 0
    for p in pi:
        p = p / sum(pi)
        if p != 0:
            total += p * log(p, 2)
    return -total

# Test the function

print(entropy([1, 1])) # Maximum Entropy e.g. a coin toss
```

```
print(entropy([0, 6])) # No entropy, ignore the -ve with zero , it's there due␣
 ↪to log function
print(entropy([2, 10])) # A random mix of classes

# 1.0
# -0.0
# 0.6500224216483541
```

```
1.0
-0.0
0.6500224216483541
```

1.8.2 Generalization of Entropy

- Entropy is a measure of uncertainty in a dataset
- It characterizes the amount of information contained within the dataset
- Equation to calculate entropy: H(S) = -sum(Pi*log2(Pi))
- When H(S) = 0, the dataset is perfectly classified
- We can easily calculate information gain for potential splits by knowing the amount of entropy in a subset.

$$H(S) = -\sum(P_i.log_2(P_i))$$

1.9 Information Gain

- Information gain is a criterion used by the ID3 algorithm to create decision trees.
- It is calculated by comparing entropy of the parent and child nodes after a split.
- A weighted average based on the number of samples in each class is used in the calculation.
- The attribute with the highest information gain is chosen for the split.
- The ID3 algorithm uses entropy to calculate information gain and pick the attribute to split on.

$$IG(A, S) = H(S) - \sum p(t)H(t)$$

Where:

- $H(S)$ is the entropy of set S
- t is a subset of the attributes contained in A (we represent all subsets t as T)
- $p(t)$ is the proportion of the number of elements in t to the number of elements in S
- $H(t)$ is the entropy of a given subset t

[8]:
```
# Write a function `IG(D,a)` to calculate the information gain
# As input, the function should take in `D` as a class distribution array for␣
 ↪target class, and `a` the class distribution of the attribute to be tested
# Using the `entropy()` function from above, calculate the information gain as:␣
 ↪$$gain(D,A) = Entropy(D) - \sum(\frac{|D_i|}{|D|}.Entropy(D_i))$$
# where $D_{i}$ represents distribution of each class in `a`.
```

```
def IG(D, a):
    '''
    return the information gain:
    gain(D, A) = entropy(D) SUM( |Di| / |D| * entropy(Di) )
    '''
    total = 0
    for i in a:
        total += sum(i) / sum(D) * entropy(i)
    return entropy(D) - total


# Test the function
# Set of example of the dataset - distribution of classes
test_dist = [6, 6] # Yes, No
# Attribute, number of members (feature)
test_attr = [ [4,0], [2,4], [0,2] ] # class1, class2, class3 of attr1 according␣
 ↪to YES/NO classes in test_dist

print(IG(test_dist, test_attr))

# 0.5408520829727552
```

```
0.5408520829727552
```

2 Decision Trees in Scikit-Learn

[9]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import OneHotEncoder
from sklearn import tree
```

[10]:
```
# Load the dataset
df = pd.read_csv('tennis.csv')

df.head()
```

[10]:

	outlook	temp	humidity	windy	play
0	sunny	hot	high	False	no
1	sunny	hot	high	True	no
2	overcast	hot	high	False	yes
3	rainy	mild	high	False	yes
4	rainy	cool	normal	False	yes

[11]:
```
X = df[['outlook', 'temp', 'humidity', 'windy']]
y = df[['play']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
↪random_state = 42)
```

[12]:
```
# One-hot encode the training data and show the resulting DataFrame with proper␣
↪column names
ohe = OneHotEncoder()

ohe.fit(X_train)
X_train_ohe = ohe.transform(X_train).toarray()

# Creating this DataFrame is not necessary its only to show the result of the ohe
ohe_df = pd.DataFrame(X_train_ohe, columns=ohe.get_feature_names(X_train.
↪columns))

ohe_df.head()
```

[12]:

	outlook_overcast	outlook_rainy	outlook_sunny	temp_cool	temp_hot	\
0	0.0	0.0	1.0	1.0	0.0	
1	1.0	0.0	0.0	0.0	1.0	
2	0.0	0.0	1.0	0.0	1.0	
3	0.0	1.0	0.0	0.0	0.0	
4	0.0	1.0	0.0	1.0	0.0	

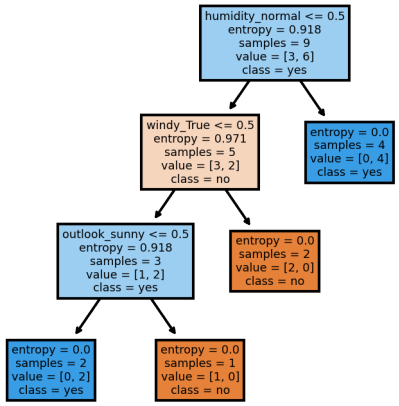
	temp_mild	humidity_high	humidity_normal	windy_False	windy_True
0	0.0	0.0	1.0	1.0	0.0
1	0.0	1.0	0.0	1.0	0.0
2	0.0	1.0	0.0	0.0	1.0
3	1.0	1.0	0.0	0.0	1.0
4	0.0	0.0	1.0	1.0	0.0

[13]:
```
# Create the classifier, fit it on the training data and make predictions on the␣
↪test set
clf = DecisionTreeClassifier(criterion='entropy')

clf.fit(X_train_ohe, y_train)
```

[13]:
```
DecisionTreeClassifier(criterion='entropy')
```

[14]:
```
fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (3,3), dpi=300)
tree.plot_tree(clf,
               feature_names = ohe_df.columns,
               class_names=np.unique(y).astype('str'),
               filled = True)
plt.show()
```



[15]:
```
X_test_ohe = ohe.transform(X_test)
y_preds = clf.predict(X_test_ohe)

print('Accuracy: ', accuracy_score(y_test, y_preds))
```

```
Accuracy:  0.6
```

3 Hyperparameter Tuning and Pruning in Decision Trees

4 Hyperparameter Optimization

- Hyperparameters are parameters set before the learning process in machine learning.
- Different model training algorithms require different hyperparameters.
- Lasso is an algorithm that adds a regularization hyperparameter to ordinary least squares regression.
- Hyperparameters affect the predictive performance and computational complexity of the model. Tree pruning
- We can optimize a decision tree classifier by tweaking parameters before learning takes place.
- Growing a decision tree beyond a certain level of complexity can lead to overfitting.
- Tree pruning can adjust the amount of overfitting or underfitting to optimize for accuracy, precision, and/or recall.
- Tree pruning can be done through adjusting maximum depth, minimum leaf sample size, maximum leaf nodes, and maximum features.

Let's look at a few hyperparameters and learn about their impact on classifier performance:

`max_depth`

- The max_depth parameter is the first one to tune for decision trees.
- A tree that is too deep may result in overfitting and difficulty generalizing on unseen data.
- A tree that is too shallow may result in underfitting and low accuracy for both training and test samples.

- The training accuracy keeps rising with greater depths, but the validation accuracy falls constantly.
- Finding the sweet spot, such as a depth of 4, is the first hyperparameter to tune.

`min_samples_split`

- The min_samples_split hyperparameter sets the minimum number of samples required to split an internal node.
- Increasing this parameter value makes the tree more constrained and considers more samples at each node.
- Training and test accuracy stabilize at a certain minimum sample split size, even if we increase the size of the split.
- Identifying the optimal sample size during training is imperative to avoid a complex model with similar accuracy to a much simpler model.
- Large values for min_samples_split and max_depth can create complex, dense, and long trees that are computationally expensive, especially for large datasets.

`min_samples_leaf`

- Min_samples_leaf is a hyperparameter used to set the minimum number of samples in a leaf node.
- It guarantees a minimum number of samples in a leaf, while min_samples_split can create arbitrary small leaves.
- Increasing the min_samples_leaf value after an optimal point reduces accuracy due to underfitting.
- An internal node will have further splits, while a leaf is a node without any children.
- If one of the leaves resulted will have less than the minimum number of samples required to be at a leaf node, the split won't be allowed.

4.1 Are there more hyperparameters?

- Scikit-learn has other hyperparameters for fine-tuning the learning process.
- Consult the official doc for more details.
- The hyperparameters mentioned here are related to complexity in decision trees.
- These hyperparameters are normally tuned when growing trees.

4.2 Additional resources:

overview of hyperparamter tuning

demystifying hyperparameter tuning

pruning decision trees

4.3 Maximum tree depth

[ ]:
```
# Identify the optimal tree depth for given data
# create an array for max_depth values ranging from 1 to 32
max_depths = np.linspace(1, 32, 32, endpoint=True)

# in a loop train the classifier for each depth value (32 runs)
```

```
# calculate the training and test AUC scores for each run
# plot a graph to show underfitting/overfitting and the optimal value
train_results = []
test_results = []
for max_depth in max_depths:
    dt = DecisionTreeClassifier(max_depth=max_depth, random_state=SEED)
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,␣
 ↪train_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(max_depths, train_results, 'b', label='Train AUC')
plt.plot(max_depths, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('Tree depth')
plt.legend()
plt.show()
```

4.4 minimum sample split

[ ]:
```
# Identify the optimal min-samples-split for given data
# Identify the optimal min-samples-split for given data
min_samples_splits = np.linspace(0.1, 1.0, 10, endpoint=True)
train_results = []
test_results = []
for min_samples_split in min_samples_splits:
    dt = DecisionTreeClassifier(criterion='entropy',␣
 ↪min_samples_split=min_samples_split, random_state=SEED)
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds =    roc_curve(y_train,␣
 ↪train_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)
```

```
plt.figure(figsize=(12,6))
plt.plot(min_samples_splits, train_results, 'b', label='Train AUC')
plt.plot(min_samples_splits, test_results, 'r', label='Test AUC')
plt.xlabel('Min. Sample splits')
plt.legend()
plt.show()
```

4.5 minimum sample leafs

[ ]:
```
# Calculate the optimal value for minimum sample leafs
# Calculate the optimal value for minimum sample leafs
min_samples_leafs = np.linspace(0.1, 0.5, 5, endpoint=True)
train_results = []
test_results = []
for min_samples_leaf in min_samples_leafs:
    dt = DecisionTreeClassifier(criterion='entropy',␣
 ↪min_samples_leaf=min_samples_leaf, random_state=SEED)
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,␣
 ↪train_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(min_samples_leafs, train_results, 'b', label='Train AUC')
plt.plot(min_samples_leafs, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('Min. Sample Leafs')
plt.legend()
plt.show()
```

4.6 Maximum features

[ ]:
```
# Find the best value for optimal maximum feature size
max_features = list(range(1, X_train.shape[1]))
train_results = []
test_results = []
for max_feature in max_features:
    dt = DecisionTreeClassifier(criterion='entropy', max_features=max_feature,␣
 ↪random_state=SEED)
```

```
    dt.fit(X_train, y_train)
    train_pred = dt.predict(X_train)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_train,␣
 ↪train_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    train_results.append(roc_auc)
    y_pred = dt.predict(X_test)
    false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_pred)
    roc_auc = auc(false_positive_rate, true_positive_rate)
    test_results.append(roc_auc)

plt.figure(figsize=(12,6))
plt.plot(max_features, train_results, 'b', label='Train AUC')
plt.plot(max_features, test_results, 'r', label='Test AUC')
plt.ylabel('AUC score')
plt.xlabel('max features')
plt.legend()
plt.show()
```

4.7 re-train with chosen values

[ ]:
```
# Train a classifier with optimal values identified above
dt = DecisionTreeClassifier(criterion='entropy',
                            max_features=6,
                            max_depth=3,
                            min_samples_split=0.7,
                            min_samples_leaf=0.25,
                            random_state=SEED)
dt.fit(X_train, y_train)
y_pred = dt.predict(X_test)
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
roc_auc
```

5 Regression with CART Trees

5.1 Recursive partitioning

- Linear regression is a global model that can be difficult and computationally expensive to assemble for data with complex features and nonlinear relations.
- Nonlinear regressions can be handled by partitioning the sample space into smaller regions using recursive partitioning.
- In regression trees, each leaf node represents a cell of the partition where a simple regression can be accurately fit to the data.
- The data is continuously subsetted down to smaller, more specific subsets until the simplest regression model can be built for the most specific subset.

5.2 Simple local models

- Simple regression models for each partition do not calculate the actual regression model, but instead use the sample mean of the dependent variable for that partition.
- This works well in practice, and has advantages including easier interpretation and faster inference.
- Decision trees have decision boundaries that are always horizontal or vertical because of the boolean logic used.
- The tree correctly represents the interaction between Horsepower and Wheelbase.
- All effort should go into finding a good partitioning of the data.

5.2.1 CART training algorithm

- The lab focuses on the CART algorithm for regression.
- The algorithm builds a binary tree where each non-leaf node has two children.
- The training examples are split into two subsets using a feature set and threshold.
- The algorithm selects the split that produces the smallest mean squared error at each node.
- The cost function used for selecting parameters is based on the number of samples on each subset and the MSE of each subset.
- The process is repeated until the maximum allowable depth or minimum number of samples is reached.
- New examples can be classified by navigating through the tree.

So at each step, the algorithm selects the parameters θ that minimizes the following cost function:

$$J(D, \theta) = \frac{n_{left}}{n_{total}} MSE_{left} + \frac{n_{right}}{n_{total}} MSE_{right} \tag{1}$$

- D: remaining training examples

- n_{total} : number of remaining training examples
- $\theta = (f, t_f)$: feature and feature threshold
- n_{left}/n_{right}: number of samples in the left/right subset
- MSE_{left}/MSE_{right}: MSE of the left/right subset

5.2.2 Mean Squared Error (MSE)

When performing regression with CART trees (i.e. the target values are continuous) we can evaluate a split using its MSE. The MSE of node m is computed as follows:

$$\hat{y}_m = \frac{1}{n_m} \sum_{i \in D_m} y_i \tag{2}$$

$$MSE_m = \frac{1}{n_m} \sum_{i \in D_m} (y_i - \hat{y}_m)^2 \tag{3}$$

- D_m: training examples in node m
- n_m : total number of training examples in node m
- y_i: target value of $i-$th example
- CART trees are used for regression where target values are continuous.

- Split evaluation is done using Mean Squared Error (MSE).
- MSE of a node is computed using the formula provided.
- D_m refers to training examples in node m and n_m refers to the total number of training examples in node m.
- y_i refers to the target value of the i-th example.

6 Key Takeaways

The key takeaways from this section include:

- Decision trees can be used for both categorization and regression tasks
- They are a powerful and interpretable technique for many machine learning problems (especially when combined with ensemble methods)
- Decision trees are a form of Directed Acyclic Graphs (DAGs) - you traverse them in a specified direction, and there are no "loops" in the graphs to go backward
- Algorithms for generating decision trees are designed to maximize the information gain from each split
- A popular algorithm for generating decision trees is ID3 - the Iterative Dichotomiser 3 algorithm
- There are several hyperparameters for decision trees to reduce overfitting - including maximum depth, minimum samples to split a node that is currently a leaf, minimum leaf sample size, maximum leaf nodes, and maximum features