Lesson 5: Decision Trees

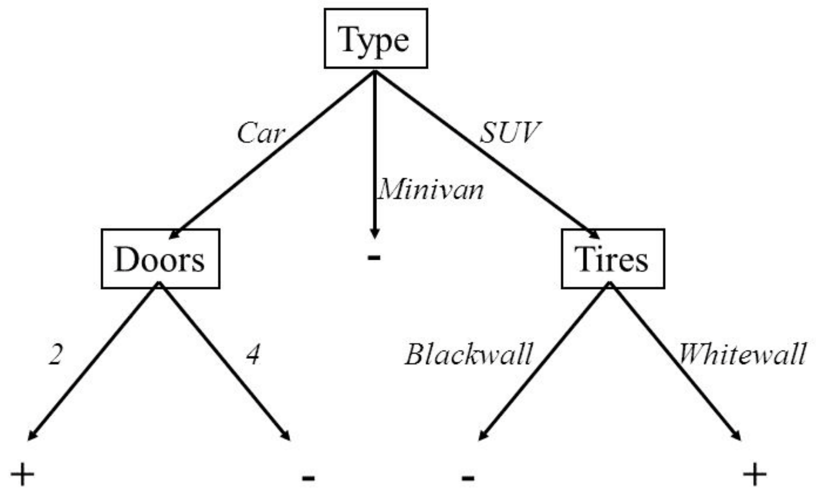
Decision Trees are one of the most popular tree-based models used in machine learning for both classification and regression tasks. Decision Trees are a powerful and interpretable method that are constructed by recursively partitioning the data into subsets based on the values of the input features.

One key concept in Decision Trees is the notion of information gain, which is used to determine which feature to split on at each node in the tree. Information gain is calculated by comparing the entropy (or impurity) of the parent node with the weighted average entropy of the child nodes. The feature that maximizes the information gain is chosen for splitting.

However, Decision Trees are prone to overfitting, especially when the tree is too large and complex. Pruning is a technique used to prevent overfitting by removing some of the branches in the tree. Two common methods of pruning are pre-pruning and post-pruning. Pre-pruning involves stopping the tree construction early, while post-pruning involves removing branches after the tree has been constructed.

Basic Concepts of Decision Trees

A decision tree is a tree-like structure where each internal node represents a test on a feature, each branch represents the outcome of the test, and each leaf node represents a class label or a numerical value. Decision trees are built by recursively partitioning the data into subsets based on the values of the features until the subsets become homogeneous with respect to the target variable.



The decision tree algorithm works by selecting the best feature to split the data at each node based on a criterion that maximizes the information gain. The information gain measures the amount of uncertainty removed about the target variable by splitting the data on a given feature.

Calculating Information Gain

Information gain is a measure used in decision trees to determine the relevance of a feature in making a classification decision. It measures how much the entropy (or impurity) of the data decreases when a feature is used for splitting. The formula for information gain is:

Information Gain = Entropy(Parent) - [Weighted Average] * Entropy(Child)

where "Entropy(Parent)" is the entropy of the parent node, and "Entropy(Child)" is the entropy of the child nodes resulting from the split. The "Weighted Average" is the proportion of samples in each child node.

The entropy is defined as:

Entropy = -p * log2(p) - (1-p) * log2(1-p)

where "p" is the proportion of samples in the node belonging to one class.

Here's an example code snippet using the iris dataset to calculate the information gain of each feature:

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np

# Load iris dataset
iris = load_iris()
X = iris.data
```

```python
y = iris.target
df = pd.DataFrame(X, columns=iris.feature_names)

# Calculate entropy of parent node
p = np.bincount(y) / len(y)
entropy_parent = -np.sum([p_i * np.log2(p_i) for p_i in p if p_i >
0])

# Calculate information gain of each feature
for feature in df.columns:
    df_temp = df[[feature]].copy()
    df_temp['target'] = y
    df_temp = df_temp.sort_values(by=[feature])

    thresholds = np.unique(df_temp[feature])
    for threshold in thresholds:
        left = df_temp['target'][df_temp[feature] <= threshold]
        right = df_temp['target'][df_temp[feature] > threshold]
        if len(left) == 0 or len(right) == 0:
            continue
        p_left = np.bincount(left) / len(left)
        p_right = np.bincount(right) / len(right)
        entropy_child = -(len(left) / len(y)) * np.sum([p_i *
np.log2(p_i) for p_i in p_left if p_i > 0])
        entropy_child -= (len(right) / len(y)) * np.sum([p_i *
np.log2(p_i) for p_i in p_right if p_i > 0])
        information_gain = entropy_parent - entropy_child
        print('Feature: {}, Threshold: {}, Information Gain:
{:.3f}'.format(feature, threshold, information_gain))
```

This code calculates the entropy of the parent node, and then loops through each feature and threshold to calculate the entropy of each child node and the resulting information gain. The results can be used to determine the best feature and threshold for splitting the data.

Pruning Techniques

Pruning is a technique used to prevent decision trees from overfitting to the training data. The basic idea is to remove parts of the tree that do not contribute much to its accuracy, thus reducing its complexity and improving its generalization performance.

There are two main types of pruning techniques: pre-pruning and post-pruning.

Pre-pruning involves setting a threshold on some measure of impurity, such as entropy or Gini index, and stopping the tree construction process when the impurity falls below the threshold. Pre-pruning can be effective in reducing the size of the tree and preventing overfitting, but it can also lead to underfitting if the threshold is set too high.

Post-pruning, also known as backward pruning, involves building the full decision tree and then removing nodes that do not improve its accuracy when replaced by their parent node. Post-pruning can be more effective than pre-pruning in reducing overfitting, but it can also be more computationally expensive.

In addition to pre-pruning and post-pruning, there are also other techniques for pruning decision trees, such as cost-complexity pruning and reduced error pruning.

Here is an example of post-pruning a decision tree using scikit-learn:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import graphviz


# load dataset
from sklearn.datasets import load_iris
iris = load_iris()


# split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.3, random_state=42)
```

```
# train decision tree classifier
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)


# make predictions on test set
y_pred = tree.predict(X_test)


# evaluate performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}".format(accuracy))


# visualize decision tree before pruning
dot_data = export_graphviz(tree, out_file=None,
feature_names=iris.feature_names, class_names=iris.target_names)
graph = graphviz.Source(dot_data)
graph.render("iris_tree_before_pruning")


# prune decision tree
tree.prune()


# make predictions on test set after pruning
y_pred_pruned = tree.predict(X_test)


# evaluate performance after pruning
accuracy_pruned = accuracy_score(y_test, y_pred_pruned)
print("Accuracy after pruning: {:.2f}".format(accuracy_pruned))


# visualize decision tree after pruning
dot_data_pruned = export_graphviz(tree, out_file=None,
feature_names=iris.feature_names, class_names=iris.target_names)
graph_pruned = graphviz.Source(dot_data_pruned)
```

```
graph_pruned.render("iris_tree_after_pruning")
```

Ensemble Methods

Ensemble methods are used to improve the performance and generalization ability of decision tree models by combining multiple models. Two commonly used ensemble methods for decision trees are bagging and boosting.

Bagging, short for bootstrap aggregating, involves training multiple decision tree models on different subsets of the training data and combining their predictions by averaging. By using different subsets of the training data, bagging reduces the variance of the model and helps to prevent overfitting.

Boosting involves training a sequence of decision tree models on weighted versions of the training data, with the weights updated based on the performance of the previous model. Boosting can help to improve the accuracy of the model by focusing on difficult examples that are misclassified by previous models.

Random forests are a specific type of decision tree ensemble that use bagging and random feature selection to improve the performance of decision trees. In a random forest, multiple decision trees are trained on different subsets of the training data and different subsets of the features, with the final prediction made by averaging the predictions of all the trees.

Here is an example of how to implement a random forest using Scikit-Learn:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()

# Split the data into training and test sets
```

```
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.2)

# Train a random forest classifier with 100 trees
rf = RandomForestClassifier(n_estimators=100)
rf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf.predict(X_test)

# Evaluate the performance of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}".format(accuracy))
```

Decision Tree Code Example

This code demonstrates the use of the decision tree classifier in the scikit-learn library. Here's what the code does:

1. The code imports the load_iris() function from the sklearn.datasets module, which loads the famous Iris dataset into the iris variable. This dataset is commonly used in machine learning for classification tasks.

2. The code then imports the DecisionTreeClassifier class from the sklearn.tree module, which is used to create a decision tree classifier.

3. The train_test_split() function from sklearn.model_selection module is used to split the Iris dataset into training and test sets.

4. A decision tree classifier object is created by calling DecisionTreeClassifier() with no arguments.

5. The decision tree classifier is trained on the training data using the fit() method of the classifier object.

6. The predict() method is used to make predictions on the test set.

7. The accuracy_score() function from sklearn.metrics module is used to calculate the accuracy of the classifier on the test set.

8. The code prints the accuracy of the decision tree classifier on the test set.

The code demonstrates how to train and test a basic decision tree classifier using the Iris dataset.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.2)

# Train a decision tree classifier
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Evaluate the performance of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}".format(accuracy))
```