Learning Objectives

At the end of the experiment, you will be able to

- know the concept of decision tree algorithm
- know how to visualize a decision tree and make illustrations from it
- understand the classification and regression tree (CART) algorithm
- know about the basic mathematics behind the predictions and classifications made by decision trees
- know what is hyperparameter regularization

Decision Trees

Decision Trees are supervised Machine Learning algorithms that can perform both classification and regression tasks and even multioutput tasks. They can handle complex datasets. As the name shows, it uses a tree-like model to make decisions in order to classify or predict according to the problem. It is an ML algorithm that progressively divides datasets into smaller data groups based on a descriptive feature until it reaches sets that are small enough to be described by some label.

The importance of decision tree algorithm is that it has many applications in the real world. For example:

1. In the Healthcare sector: To develop Clinical Decision Analysis tools which allow decision-makers to apply for evidence-based medicine and make objective clinical decisions when faced with complex situations.
2. Virtual Assistants (Chatbots): To develop chatbots that provide information and assistance to customers in any required domain.
3. Retail and Marketing: Sentiment analysis detects the pulse of customer feedback and emotions and allows organizations to learn about customer choices and drives decisions.

How can an algorithm be represented as a tree?

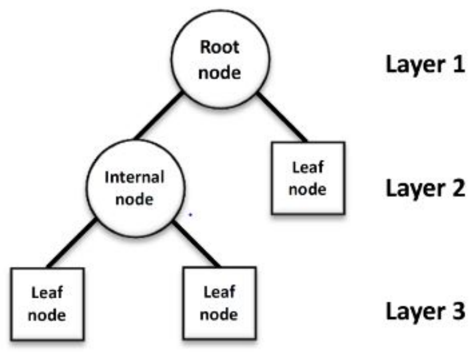


Figure 1: Basic Structure of Decision Tree

For this, let us see the basic example of the titanic dataset which predicts whether a passenger survives or not. The below tree uses 3 attributes from the dataset, namely sex, age, and sibsp (Number of Siblings/Spouses Aboard).

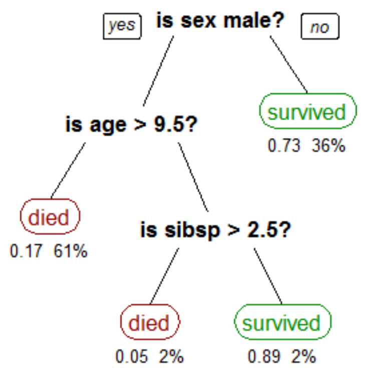


Figure 2: Decision tree using example

A decision tree is drawn upside down with its root at the top. In the image on the left (is age $>$ 9.5), the bold text in black represents a condition/internal node, based on which the tree splits into branches/ edges. The end of the branch that doesn't split anymore is the decision/leaf, in this case, whether the passenger died or survived, represented as red and green text respectively.

To know more about, decision trees, click here.

Notebook Overview

The notebook contains the following topics:

- Different methods of visualization of decision tree: Classification and Regression
- Training of decision trees
- The mathematics of decision tree for both classification and regression problems
- Overview of CART algorithm
- Regularization of Hyperparameters
- Visualization of decision boundaries (hyper-plane) with default hyperparameters and after regularization of hyperparameters
- Limitations of decision tree algorithm

Setup Steps:

Please enter your registration id to start:

Id: " Insert text here "

Show code

Please enter your password (your registered phone number) to continue:

password: " Insert text here "

Show code

Run this cell to complete the setup for this Notebook

Show code

Import required packages

```
# Install dtreeviz library
!pip -qq install dtreeviz --upgrade
!pip install graphviz
!apt-get -qq install -y graphviz
```

```
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 91.8/91.8 kB 2.6 MB/s eta 0:00:00
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-packages (0.20.3)
```

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap                                # to import color map
from sklearn import datasets
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor          # to import DT classifier and Regressor
import graphviz                                                             # to import graphviz
from dtreeviz.trees import *                                          # to import dtreeviz
%matplotlib inline
```

Training and Visualizing a Decision Tree

The decision trees can be divided, with respect to the target values, into:

- Classification trees: used to classify samples, assign to a limited set of values - classes. In Scikit-Learn it is `DecisionTreeClassifier`.
- Regression trees: used to assign samples into numerical values within the range. In Scikit-Learn it is `DecisionTreeRegressor`.

To understand Decision Trees, let's just build one and take a look at how it makes predictions. The following code trains a DecisionTreeClassifier on the iris dataset:

```
# Prepare the data into X (predictor) and Y (target), considering sepal length and width
iris = datasets.load_iris()
X = iris.data[:, 2:]
y = iris.target


# Fit the DT classifier with max_depth = 2
clf = DecisionTreeClassifier(max_depth = 2, random_state=1234)
model = clf.fit(X, y)
```

Text Representation

Visualizing the Decision tree in text using `text.export_text`. This is important when we want to obtain the model information in a text file.

```
text_representation = tree.export_text(clf)
# Display result
print(text_representation)
```

```
|--- feature_0 <= 2.45
|   |--- class: 0
|--- feature_0 >  2.45
|   |--- feature_1 <= 1.75
|   |   |--- class: 1
|   |--- feature_1 >  1.75
|   |   |--- class: 2
```

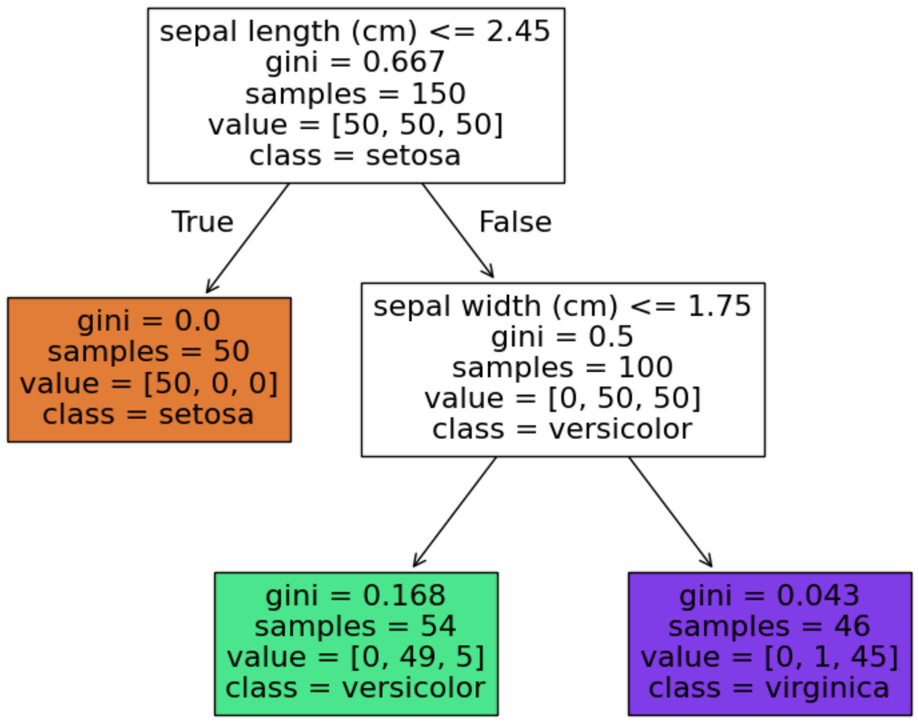
If you want to save it to the file, it can be done with the following code:

```
# Save in figure
with open("decision_tree.log", "w") as f_out:
    f_out.write(text_representation)
```

Visualize Tree with `plot_tree`

This feature requires matplotlib library to be installed. It allows us to easily visualize the tree into a figure (without intermediate exporting to graphviz). You can also refer the docs to learn more about `plot_tree`.

```
# Visualize tree
fig = plt.figure(figsize=(10,8))
_ = tree.plot_tree(clf,
                   feature_names=iris.feature_names,
                   class_names=iris.target_names,
                   filled=True)                       # filled = True uses color coding for majority of classes
```



```
# Save the figure to the .png file
fig.savefig("decision_tree.png")
```

How Decision Tree Makes Prediction

In the above example, we see how the Decision tree makes predictions. In this, we need to classify a flower. At depth 0, at the top (root node), this node asks the condition whether the flower's sepal length $\leq$ 2.45 cm. If it is, then we move down to the root's left child node (depth 1, left). Here, it's a leaf node (means it does not have any child nodes), so it does not ask any questions: you can simply look at the predicted class for that node and the Decision Tree predicts that the flower is an Iris-Setosa (class=setosa).

Now, in another case, when the sepal length of a flower is $\geq$ 2.45cm. We then move down to right the child node (depth 1, right), which is not a leaf node, and ask if sepal width is $\leq$ 1.75cm. If it is, then the flower is Iris-Versicolor (depth 2, left). If not, it is surely, Iris-Virginica (depth 2, right).

On each node, the `samples` represent the number of training instances. For example, 54 samples have sepal length $\geq$ 2.45cm and sepal width $\leq$ 1.75cm (depth 2, left). The node's `values` show how many training instances of each class this node applies. For example, at depth 2, right, applies 0 to Iris-Setosa, 1 to Iris-Versicolor, and 45 to Iris-Virginica.

At last, the `gini-impurity` measures the node's impurity, if a node is `pure` (gini = 0), means that all the instances belong to a single class. Example, depth 1, left node.

Gini Impurity and Entropy

In a decision tree, by default `gini_impurity` is used. But, we can use `entropy` by changing the `criterion` hyperparameter to "`entropy`". The gini impurity is one of the methods used in decision tree algorithms to decide the optimal split from a root node and subsequent splits.

The entropy came from thermodynamics, where if entropy approaches 0 that means the molecules are still and well-ordered. In ML, it is used to measure the impurity: a set's entropy 0 means that all the instances belong to the same class.

Equation for calculating the Gini-impurity:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

The above equation shows how the training algorithm computes the gini-score G_i of the i^{th} node.

For example, the *depth-2 left node* has a gini score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Equation for calculating the Entropy:

$$H_i = -\sum_{i=1}^{n} p_{i,k} \log_2(p_{i,k}), \quad where, p_{i,k} \neq 0$$

The above equation shows the definition of the entropy of the i^{th} node.

For example, given above, the *depth-2 left node* has an entropy equal to $-\frac{49}{54}\log_2(\frac{49}{54}) - \frac{5}{54}\log_2(\frac{5}{54}) \approx 0.445$

Where, k is class of the problem. In this example, there are three classes, Iris-versicolor, Iris-Setosa, and Iris-Virginica.

Estimating Class Probabilities

The instance probability denoted by $p_{i,k}$ denotes the probability of an instance belongs to a class k.

For example, we have a flower whose sepal length is 5cm and sepal width is 1.5cm (depth 2, left node). The following probabilities: 0% for Iris-Setosa $(\frac{0}{54})$, 90.74% for Iris-Versicolor $(\frac{49}{54})$, and 9.25% for Iris-Verginica $(\frac{5}{54})$. And this will predict the class as Iris-Versicolor.

```
# Predicting probability of a class
clf.predict_proba([[5, 1.5]])
```

```
array([[0.        , 0.90740741, 0.09259259]])
```

```
# Predicting a class
clf.predict([[5, 1.5]])
```

```
array([1])
```

The CART Training Algorithm

Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (also called "growing" trees). CART constructs binary trees using the feature and threshold that yield the largest information gain or minimum gini-impurity at each node.

Cart Cost Function For Classification

$$J_{k,t_k} = \frac{m_{left}}{m}G_{left} + \frac{m_{right}}{m}G_{right}$$

$$Where \begin{cases} G_{left/right} & \text{measures the impurity of the left/right subset,} \\ m_{left/right} & \text{is the number of instances in the left/right subset} \end{cases}$$

Equation shows that the algorithm first the splits the training subsets using a single feature k and a limiting value or condition t_k. The algorithm works as follows:

- It creates different pairs of (k, t_k). It searches for the pair that produces the purest subset (i.e.minimum gini-impurity) weighted by their size.

Once it splits the training set in two, it splits the subsets using the same algorithm and this iteration will be done until it reaches the maximum depth (i.e., `max_depth` hyperparameter). A few other hyperparameters control additional stopping conditions (`min_samples_split, min_samples_leaf, min_weight_fraction_leaf, and max_leaf_nodes`).

Regularization Hyperparameters

Decision tree does not make any prior assumptions of training data (unlike linear models, which assume the data as linear). If left unconstrained, the tree will closely fit itself according to the training instances, and most likely lead to overfitting (means the model will perform poorly on the test dataset). So, to avoid overfitting the training data, the hyperparameters need to be restricted and passed during the modeling. This process is what is known as Regularization Hyperparameters.

In Scikit-Learn, this is controlled by the `max_depth` hyperparameter (the `default value` is `None`, which means unlimited). Reducing max_depth will regularize the model and thus reduce the risk of overfitting. The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree: `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `min_weight_fraction_leaf` (same as min_samples_leaf but expressed as a fraction of the total number of weighted instances), `max_leaf_nodes` (maximum number of leaf nodes), and `max_features` (maximum number of features that are evaluated for splitting at each node). Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

In the example below, we will visualize the decision tree, one by regularizing hyperparameter and one without any restrictions.

```
# Load iris dataset and define variables with sepal length and width
iris = datasets.load_iris()
X = iris.data[:, 2:]
y = iris.target

# First tree without restrictions
tree_clf = DecisionTreeClassifier(random_state=42)
```

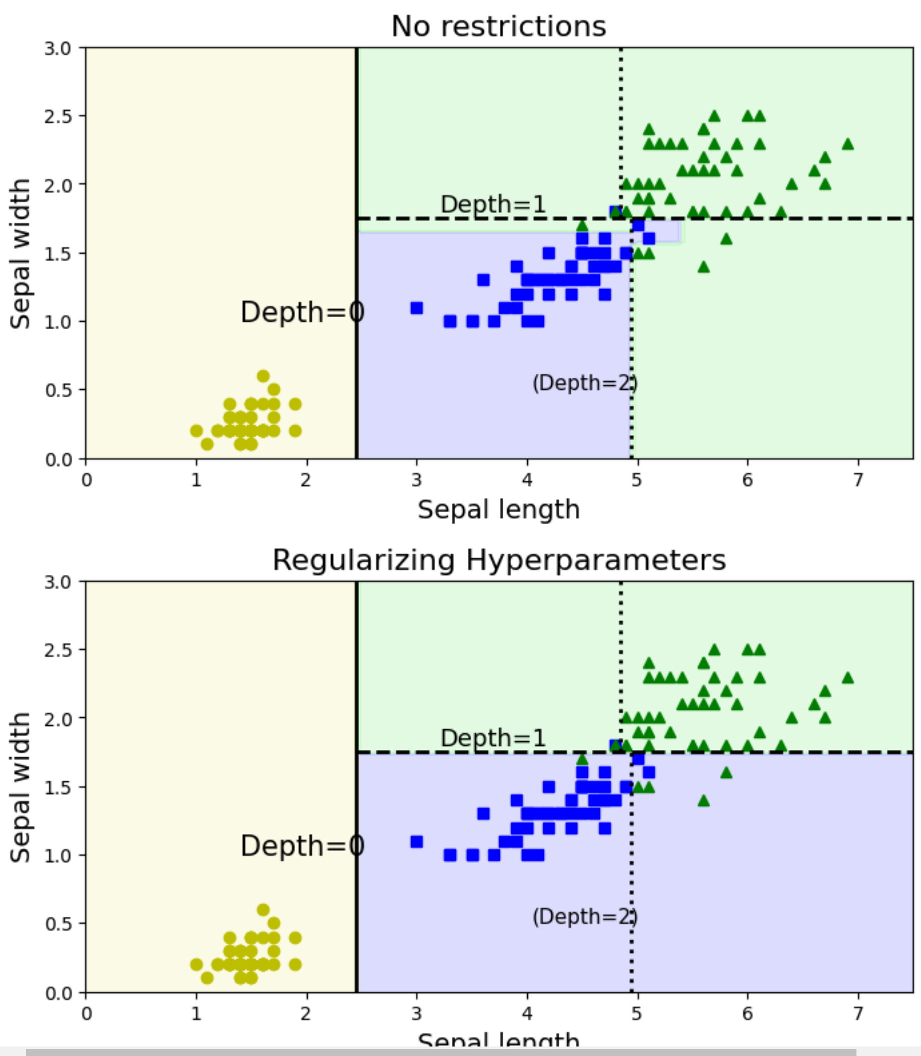
```
tree_clf.fit(X, y)

# Second tree with hyperparameters
tree_clf2 = DecisionTreeClassifier(max_depth =2, min_samples_leaf =1, min_samples_split = 2, random_state=2)
tree_clf2.fit(X, y)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=2, random_state=2)
```

```
# Define a function for plotting decision boundary
def plot_decision_boundary(clf, X, y, axes=[0, 7.5, 0, 3], iris = True, legend=False, plot_training=True):

    # define array for x1 and x2 axes
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)

    # make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids
    x1, x2 = np.meshgrid(x1s, x2s)

    # the numpy.ravel() functions returns contiguous flattened array(1D array with all the input-array elements and with the same type a
    X_new = np.c_[x1.ravel(), x2.ravel()]
    # predict and reshape the y_pred according to x
    y_pred = clf.predict(X_new).reshape(x1.shape)

    # module is used for mapping numbers to colors or color specification conversion in a 1-D array of colors also known as colormap
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)

    if plot_training:
        # plot Setosa in yellow
        plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris setosa")
        # plot Versicolor in blue
        plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris versicolor")
        # plot Virginica in green
        plt.plot(X[:, 0][y==2], X[:, 1][y==2], "g^", label="Iris virginica")
        plt.axis(axes)

    if iris:
        # define x_axes label
        plt.xlabel("Sepal length", fontsize=14)
        # define y_axes label
        plt.ylabel("Sepal width", fontsize=14)

    if legend:
        plt.legend(loc="lower right", fontsize=14)
```

To know more about Listedcolormap and meshgrid in the above code, click here.

```
# Plot both the decision tree
plt.figure(figsize=(8, 4))

# call the plot_decision_boundary function for tree_clf
plot_decision_boundary(tree_clf, X, y)
plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2)
plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)
plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)
plt.title("No restrictions", fontsize=16)
plt.text(1.40, 1.0, "Depth=0", fontsize=15)
plt.text(3.2, 1.80, "Depth=1", fontsize=13)
plt.text(4.05, 0.5, "(Depth=2)", fontsize=11)

plt.figure(figsize=(8, 4))

# call the plot_decision_boundary function for tree_clf2
plot_decision_boundary(tree_clf2, X, y)
plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2)
plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)
plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)
plt.title("Regularizing Hyperparameters", fontsize=16)
plt.text(1.40, 1.0, "Depth=0", fontsize=15)
plt.text(3.2, 1.80, "Depth=1", fontsize=13)
plt.text(4.05, 0.5, "(Depth=2)", fontsize=11)

plt.show()
```



From the above figure, we can see that, when there is no restriction of hyperparameters in decision tree, it can adjust itself according to the training dataset (overfitting problem). While, on the other hand, where there is the regularization of hyperparameters, the model will probably generalize better.

Decision Tree: Regression

Decision Trees are also capable of performing regression tasks. Below, present the `DecisionTreeRegressor` visualization methods from Scikit-Learn package. Here, in this section, we use the Boston dataset to create a decision tree regressor.

```
# Prepare the data
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
X = housing.data
y = housing.target
housing.feature_names
```

```
['MedInc',
 'HouseAge',
 'AveRooms',
 'AveBedrms',
 'Population',
 'AveOccup',
 'Latitude',
 'Longitude']
```

To keep the size of the tree small, set max_depth = 3.

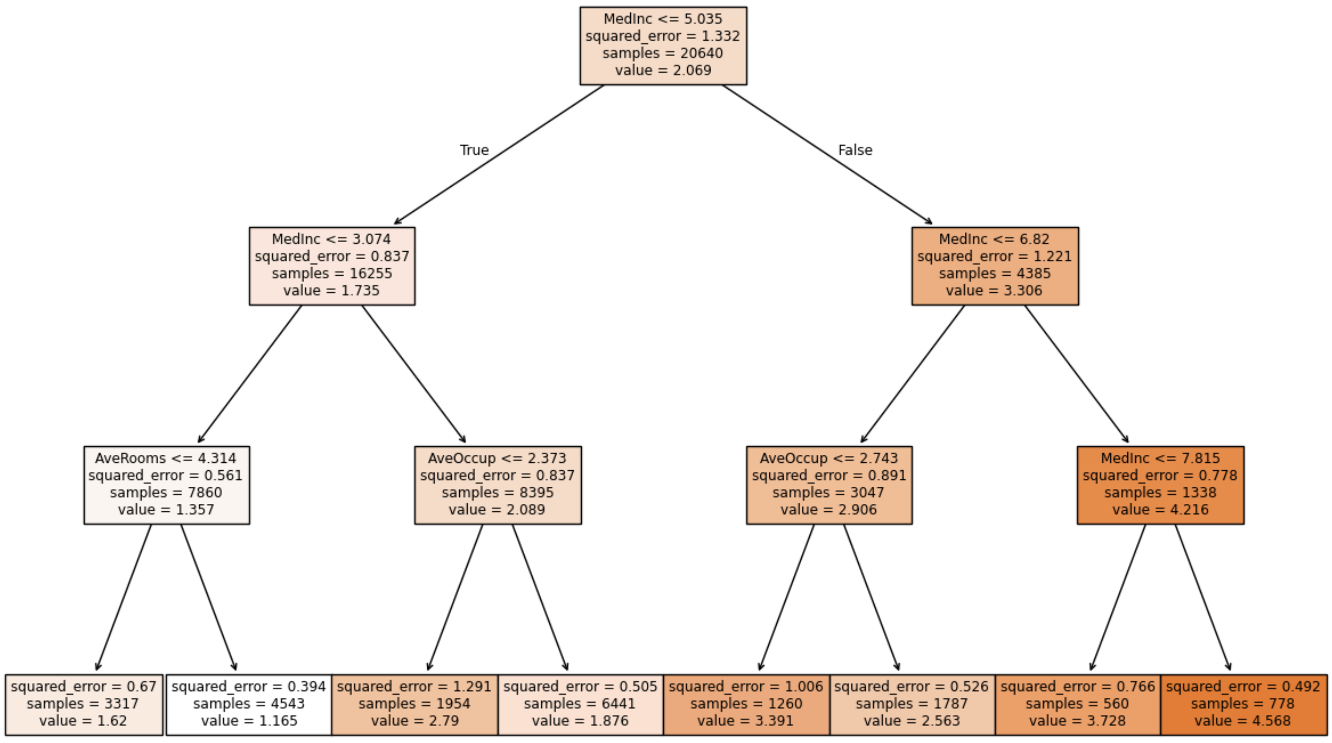
```
# Fit the regressor, set max_depth = 3
dt = DecisionTreeRegressor(max_depth=3, random_state=1234)
model = dt.fit(X, y)
```

Visualizing Decision Tree Regression same as classifier in previous topic.

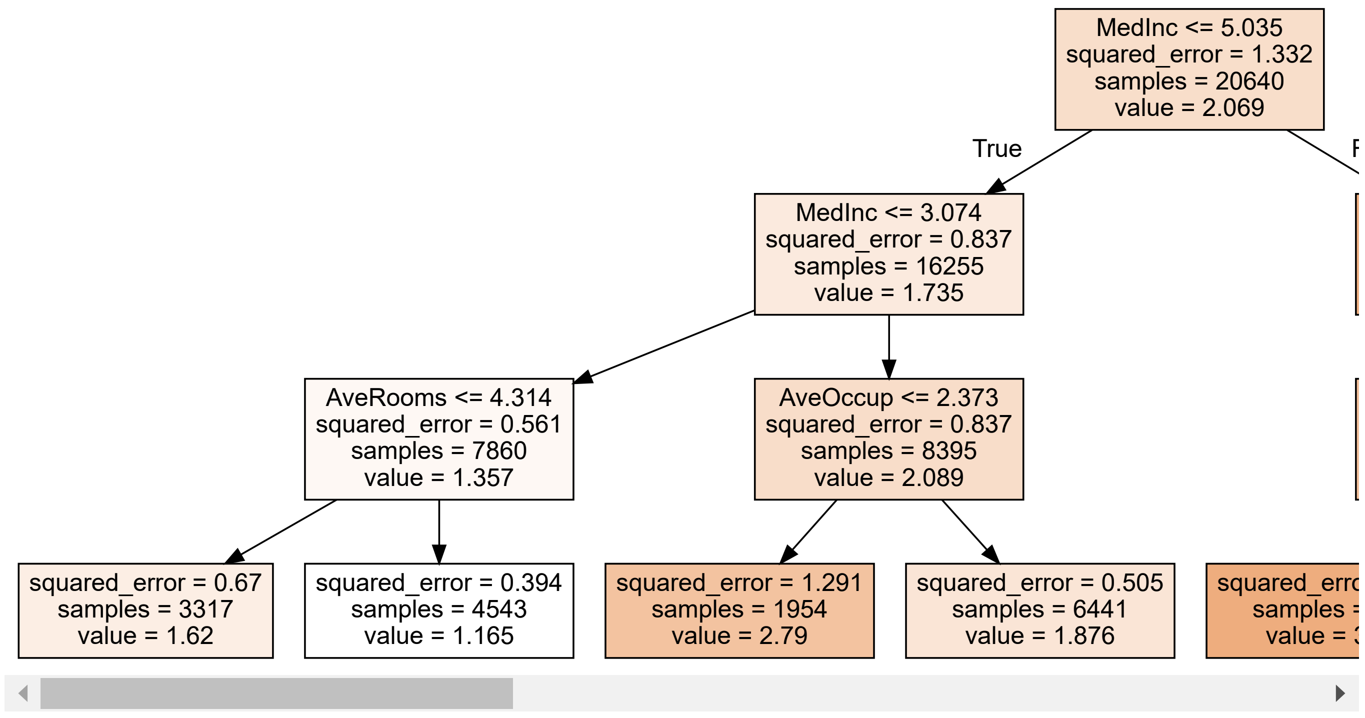
```
text_representation = tree.export_text(dt)
print(text_representation)
```

```
|--- feature_0 <= 5.04
|   |--- feature_0 <= 3.07
|   |   |--- feature_2 <= 4.31
|   |   |   |--- value: [1.62]
|   |   |--- feature_2 >  4.31
|   |   |   |--- value: [1.16]
|   |--- feature_0 >  3.07
|   |   |--- feature_5 <= 2.37
|   |   |   |--- value: [2.79]
|   |   |--- feature_5 >  2.37
|   |   |   |--- value: [1.88]
|--- feature_0 >  5.04
|   |--- feature_0 <= 6.82
|   |   |--- feature_5 <= 2.74
|   |   |   |--- value: [3.39]
|   |   |--- feature_5 >  2.74
|   |   |   |--- value: [2.56]
|   |--- feature_0 >  6.82
|   |   |--- feature_0 <= 7.82
|   |   |   |--- value: [3.73]
|   |   |--- feature_0 >  7.82
|   |   |   |--- value: [4.57]
```

```
# Visualize tree using plot_tree
fig = plt.figure(figsize=(15,10))
_ = tree.plot_tree(dt, feature_names=housing.feature_names, filled=True)
```



```
# Visualize tree using graphviz
dot_data = tree.export_graphviz(dt, out_file=None,
                                feature_names=housing.feature_names,
                                filled=True)
graphviz.Source(dot_data, format="png")
```



```
# Visualize tree using dtreeviz
import matplotlib
matplotlib.rcParams['font.family'] = 'DejaVu Sans'
from dtreeviz import dtreeviz
from dtreeviz import model
vizualize = model(dt, X, y,
                target_name="target",
                feature_names=housing.feature_names)
```

From the above `plot_tree` and `graphviz` diagram, we can see that the decision tree for regression is similar to that of classification. But with only one difference, here it is predicting a value instead of a class. So, suppose we want to make a prediction of a new instance which is RM = 7 and NOX = 0.6. Starting from the root node, we end up at the leaf node that predicts value = 33.35. This value is just the average of 43 training instances that fall into this leaf node. This prediction results in mean squared error (MSE) = 20.11 over 43 instances.

Thus, the target value predicted in each region is the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE.

Cart Cost Function For Regression

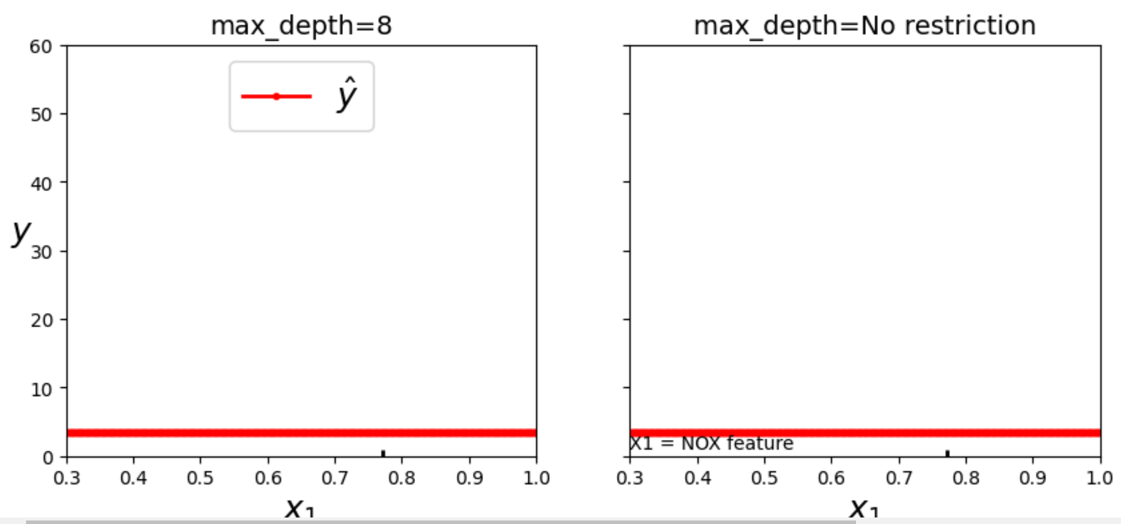
$J_{k,t_k} = \frac{m_{left}}{m} MSE_{left} + \frac{m_{right}}{m} MSE_{right}$

$$Where \begin{cases} MSE_{node} & \sum_{i \in node} (\hat{y} - y^{(i)})^2, \\ \hat{y} = \frac{1}{m_{node}} \sum_{i \in node} y^{(i)} \end{cases}$$

As we saw in classification problem, decision trees in regression with no regularization on hyperparameters are also prone to overfitting. From the graph obtained by running the below code, we see predictions obtained with `max_depth = 8` on left and on the right without any regularization.

```
# Define plot regression function

def plot_regression_predictions(tree_reg, X, y, axes=[0.3, 1, 0, 60], ylabel="$y$"):
    # creating the x-axes grid in array
    x1 = np.linspace(axes[0], axes[1], 500).reshape(-1, 1)
    # define y
    y_pred = tree_reg.predict(x1)
    plt.axis(axes)
    plt.xlabel("$x_1$", fontsize=18)

    if ylabel:
        plt.ylabel(ylabel, fontsize=18, rotation=0)

    plt.plot(X, y, "b.")
    # plot y hat (predicted values) in red line in both graphs
    plt.plot(x1, y_pred, "r.-", linewidth=2, label=r"$\hat{y}$")
```

```
# Take index value 4 i.e, NOX
X = housing.data[:,4:5]
y = housing.target


# Define model with no hyperparameter
dt = DecisionTreeRegressor(random_state=1234)
dt.fit(X, y)

# Define model with maximum depth = 8
dt2 = DecisionTreeRegressor(max_depth=8, random_state=1234)
dt2.fit(X, y)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(axes[0])

# Plot decision boundary
plot_regression_predictions(dt2, X, y)

for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    # Plot the fit regression line
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
plt.legend(loc="upper center", fontsize=18)
plt.title("max_depth=8", fontsize=14)

plt.sca(axes[1])
# Plot dcision boundary
plot_regression_predictions(dt, X, y, ylabel=None)

for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    # Plot the fit regression line
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)

plt.title("max_depth=No restriction", fontsize=14)
plt.text(0.3,1, "X1 = NOX feature", fontsize=10)
plt.show()
```



The one without any hyperparameters setting is obviously overfitting the training instances and just by restricting max_depth = 8 results in a much more reasonable model.

Limitations of Decision Tree Algorithm

The major limitations of decision tree approaches for data analysis are:

- Provide less information on the relationship between the predictors and the target variable (unlike, in linear models, we get a mathematical relation between x and y as y = θ x + c).
- Biased toward predictors with more variance.
- Can have issues with highly collinear predictors.
- Can have poor prediction accuracy for responses with low sample sizes.
- Trees can be unstable because small variations in the data might result in a completely different tree being generated.
- For classification, if some classes dominate, it can create biased trees. It is therefore recommended to balance the dataset prior to fitting.

Theory Questions