Decision Tree

Decision trees are versatile machine learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. It works by recursively partitioning the data into subsets based on **statements** of features, making decisions whether or not the statement is **True** or **False**.

Important notes to remember

- Handle both numerical and categorical data.
- Do not require feature scaling.
- The algorithm selects the best feature at each decision point, aiming to maximize information gain (for classification) or variance reduction (for regression). Decision trees are interpretable, easy to visualize, and can handle both numerical and categorical data. However, they may be prone to overfitting, which can be addressed using techniques like pruning.
- Ensemble methods, such as Random Forests and Gradient Boosting, often use multiple decision trees to enhance predictive performance and address the limitations of individual trees.
- Decision Tree *(white box model, interpretable ML)* vs Random Forest, NNs *(black box model)*
- Some DT algorithm:
  - ID3, C4.5 (Information Gain/ Information Gain Ratio)
  - CART (Gini Index)
  - SLIQ, SPRINT (Gini Index)

Training and Visualizing

```
## Train a decision tree

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
```

```
iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target
some_flower = X_iris[100,:]

dt_clf = DecisionTreeClassifier(max_depth=2, random_state=29)
dt_clf.fit(X_iris, y_iris)
print(dt_clf.predict([some_flower]))
print(dt_clf.predict_proba([some_flower]))
```

```
[2]
[[0.         0.02173913 0.97826087]]
```

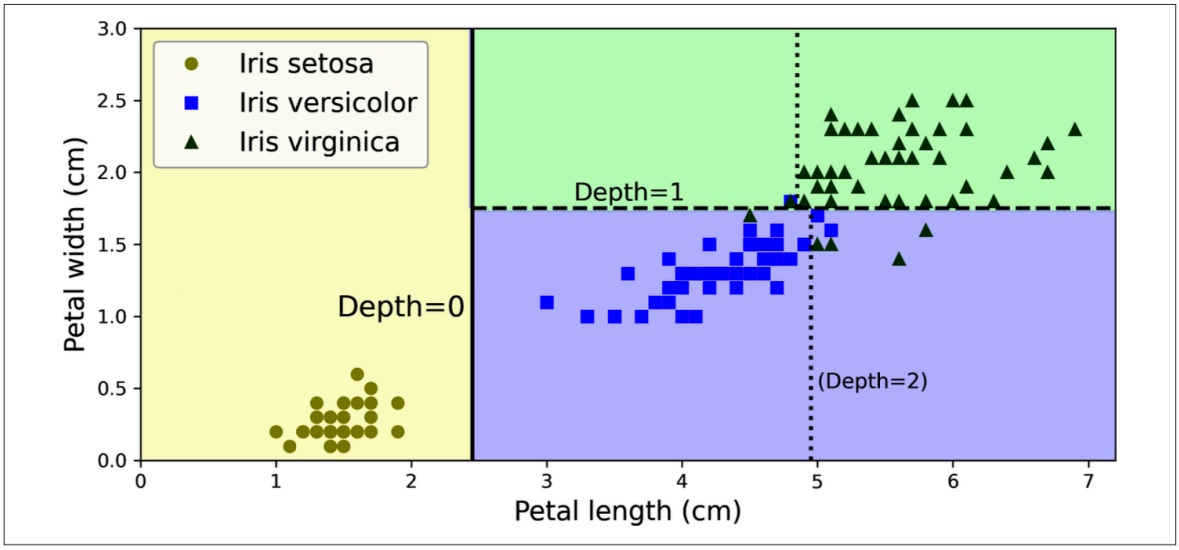


Figure 0.1: Decsion tree decision boudaries

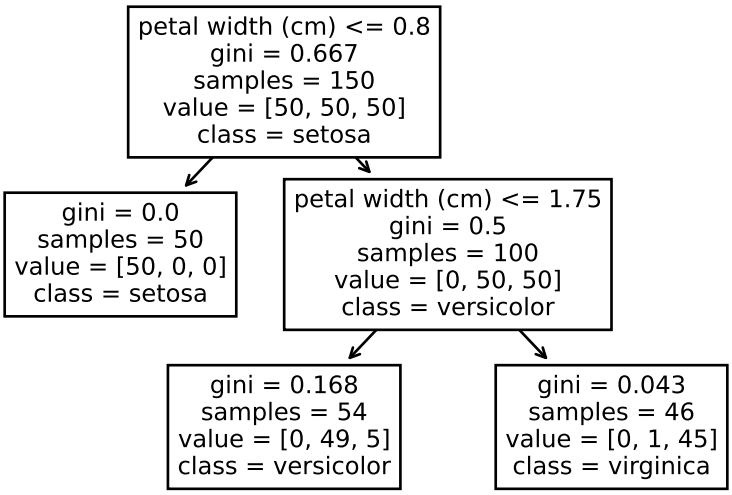
```
## Plot a decision tree

from sklearn.tree import plot_tree

plot_tree(dt_clf, feature_names=["petal length (cm)", "petal width (cm)"], class_names=iri
```

```
[Text(0.4, 0.8333333333333334, 'petal width (cm) <= 0.8\ngini = 0.667\nsamples = 150\nvalue =
 Text(0.2, 0.5, 'gini = 0.0\nsamples = 50\nvalue = [50, 0, 0]\nclass = setosa'),
```

```
Text(0.6, 0.5, 'petal width (cm) <= 1.75\ngini = 0.5\nsamples = 100\nvalue = [0, 50, 50]\ncl
Text(0.4, 0.16666666666666666, 'gini = 0.168\nsamples = 54\nvalue = [0, 49, 5]\nclass = vers
Text(0.8, 0.16666666666666666, 'gini = 0.043\nsamples = 46\nvalue = [0, 1, 45]\nclass = virg
```



The structure of a decision tree:

1. Root node: depth 0, at top
2. Branch nodes: split data based on statements
3. Leaf nodes: output, no child nodes

Figure 0.2, in a node:
- Samples: count instances in that node
- Value: count instances of each class
- Gini (Gini impurity): a measurement, measure a node is pure (0) or not (-> 1)

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

```
Gi: Gini impurity of node i\
pi,k: probability of `class k instances` in total instances in node `i`
```

Other measurements:

- Entropy (Entropy impurity) : measure randomness or uncertainty

$$H_i = -\sum p_{i,k} log_2(p_{i,k})$$

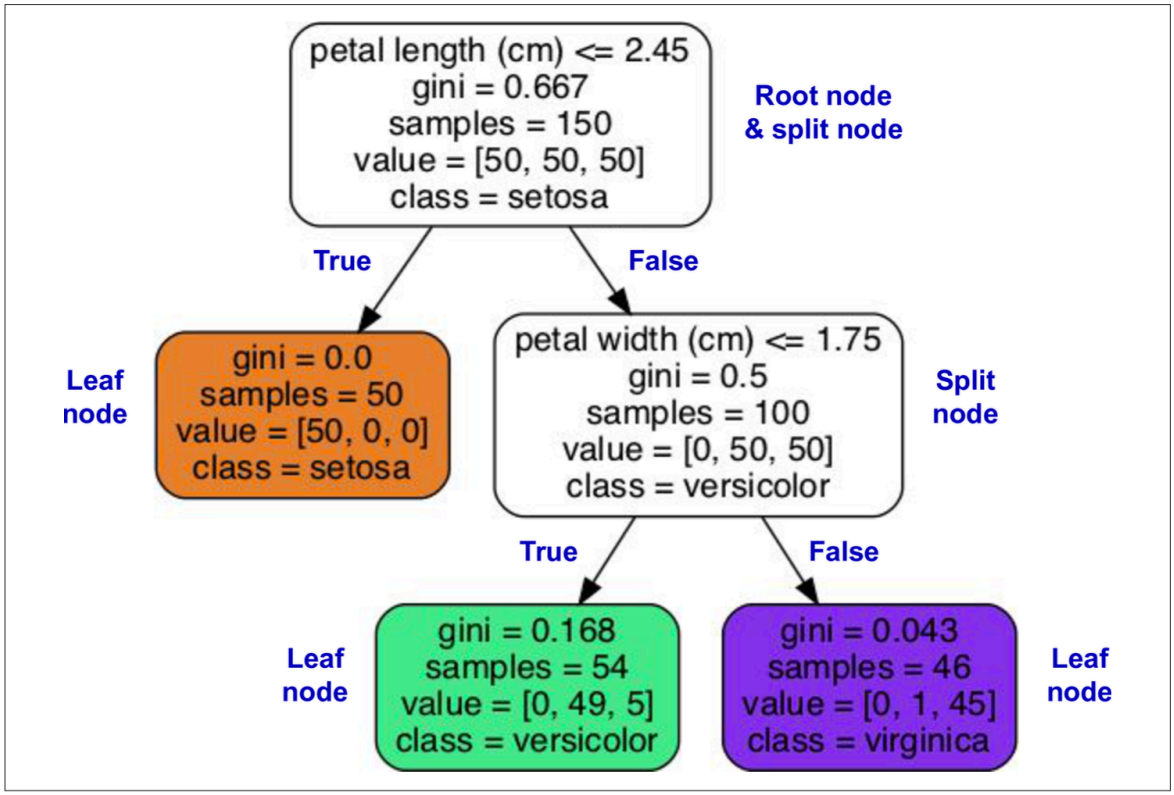


Figure 0.2: Decision tree

- Information Gain = (Entropy before split) - (weighted entropy after split)

- Information Gain Ratio = Information Gain / SplitInfo

- SplitInfo (Split Information): potential worth of splitting a branch from a node

$$SplitInfo(A) = -\sum p_k log_2(p_k)$$

> **Note**
>
> Example:
> Gi = 1 – (0/54)2 – (49/54)2 – (5/54)2 0.168
> Hi = –(49/54) log2 (49/54) – (5/54) log2 (5/54) 0.445

- Use cases:
- `Gini` and `Entropy`: No big difference
- `Gini` is faster => good default
- `Gini`: isolate the most frequent class in its own branch
- `Entropy`: produce slightly more balanced trees
- `Information Gain` tends to prefer attributes that create many branches (e.g. 12 instances with 12 classes => Entropy = 0)
- `Information Gain Ratio`: regularize Information Gain
- Sometimes, attribute A is chosen because its Information Gain Ratio is really low => Set Information Gain threshold

CART algorithm

- sklearn use *CART* algorithm produce *binary tree* (2 children only)
- Other algorithms such as *ID3* have 2 or more children
- Algorithm choose *feature k* and *threshold tk* producing purest subsets, weighted by their sizes. CART cost function for `classification`:

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

- G: impurity

- m: number of instances

- CART split training set recursively until: *purest*, *max_depth*, *min_samples_split*, *min_samples_leaf*, *min_weight_fraction_leaf*, and *max_leaf_nodes*.

- Find optimal tree is `NP-complete problem`, **O(exp(m))** => Have to find 'resonably good' solution when training a decision tree
- But making predictions is just **O(log2(m))**

> NP-complete problem
>
> P is the set of problems that can be solved in `polynomial time` (i.e., a polynomial of the dataset > size). NP is the set of problems whose solutions can be verified in polynomial time. An `NP-hard` problem is a problem that can be reduced to a known NP-hard problem in polynomial time. An `NP-complete` problem is both NP and NP-hard. A major open mathematical question is whether or not P = NP. If P = NP (which seems likely), then no polynomial algorithm will ever be found for any NP-complete problem (except perhaps one day on a quantum computer).

Regularization

- Parametric model (Linear Regression): predetermined parameters => degree of freedom is limited => reducing the risk of overfitting
- Non-parametric model: parameters not determined prior to training => go freely => overfitting => **regularization**
- Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will **regularize** the model:
  - max_depth, max_features, max_leaf_nodes
  - min_samples_split, min_samples_leaf, min_weight_fraction_leaf

```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

dt_clf1 = DecisionTreeClassifier(random_state=29)
dt_clf2 = DecisionTreeClassifier(max_depth=5, min_samples_leaf=5, random_state=29)
dt_clf1.fit(X_moons, y_moons)
dt_clf2.fit(X_moons, y_moons)

X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2, random_state=43)

print(f'Non-regularized decision tree: {dt_clf1.score(X_moons_test, y_moons_test):.4f}')
print(f'Regularized decision tree: {dt_clf2.score(X_moons_test, y_moons_test):.4f}')
```

```
Non-regularized decision tree: 0.8940
Regularized decision tree: 0.9200
```

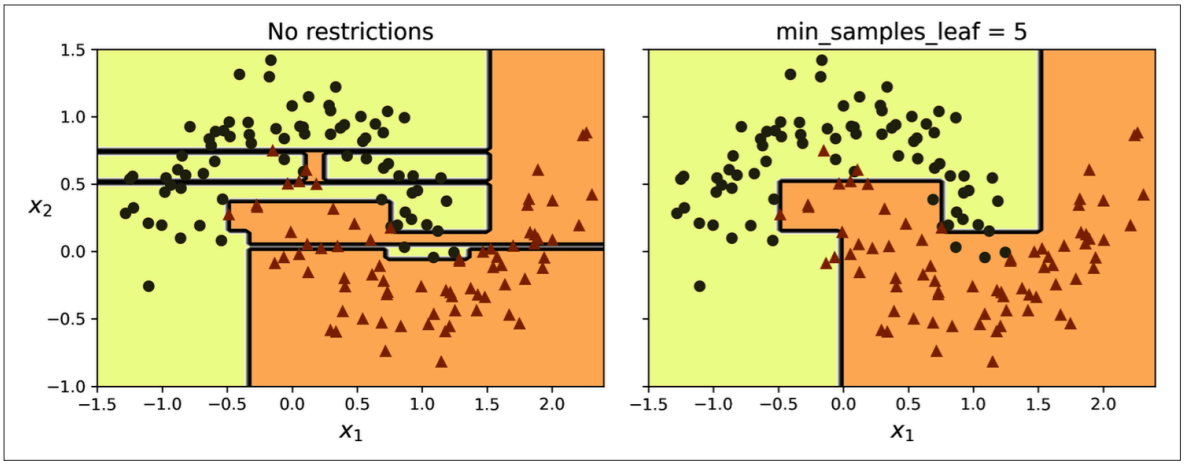


Figure 0.3: Decision boudaries of unregularized tree (left) and regularized tree (right)

! Pruning Trees

- Pruning: deleting unnecessary nodes
- Algorithms work by first training the decision tree without restrictions, then pruning (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the 2 test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the null hypothesis). If this probability, called the p-value, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.
- There are 3 types of Pruning Trees
  - Pre-Tuning
  - Post-Tuning
  - Combines

Regression

- *DecisionTreeRegressor* splits each region in a way that makes most training instances `as close as possible` to that predicted value (average of instances in the region)
- CART cost function for `regression`:

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

- MSE: mean squared error
- m: number of instances

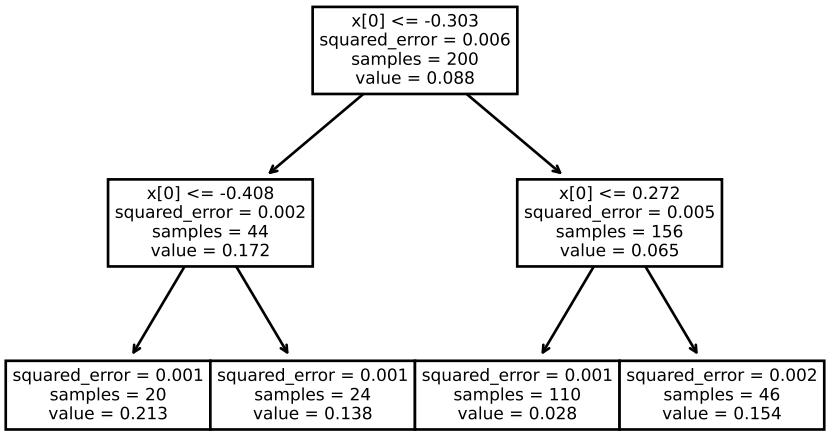
```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X_quad = np.random.rand(200, 1) - 0.5
y_quad = X_quad ** 2 + 0.025 * np.random.randn(200, 1)

dt_reg = DecisionTreeRegressor(max_depth=2, min_samples_leaf=5, random_state=29)
dt_reg.fit(X_quad, y_quad)

plot_tree(dt_reg)
```

```
[Text(0.5, 0.8333333333333334, 'x[0] <= -0.303\nsquared_error = 0.006\nsamples = 200\nvalue =
 Text(0.25, 0.5, 'x[0] <= -0.408\nsquared_error = 0.002\nsamples = 44\nvalue = 0.172'),
 Text(0.125, 0.16666666666666666, 'squared_error = 0.001\nsamples = 20\nvalue = 0.213'),
 Text(0.375, 0.16666666666666666, 'squared_error = 0.001\nsamples = 24\nvalue = 0.138'),
 Text(0.75, 0.5, 'x[0] <= 0.272\nsquared_error = 0.005\nsamples = 156\nvalue = 0.065'),
 Text(0.625, 0.16666666666666666, 'squared_error = 0.001\nsamples = 110\nvalue = 0.028'),
 Text(0.875, 0.16666666666666666, 'squared_error = 0.002\nsamples = 46\nvalue = 0.154')]
```



If we set *max_depth* larger, the model will predict more strictly. As same as Figure 0.4, if we keep the default hypeparameters, the model will grow as much as it can Figure 0.5.

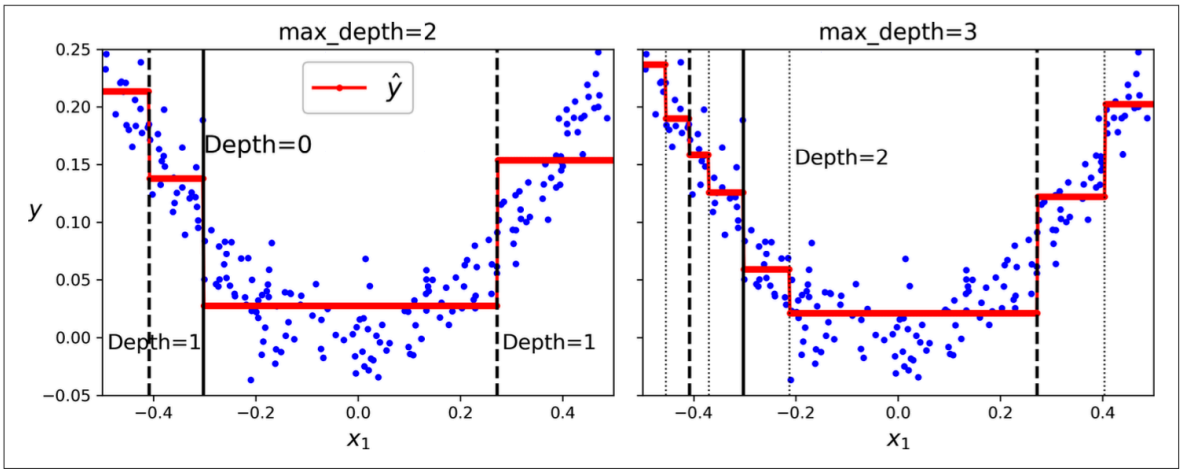


Figure 0.4: Different max_depth

Limitations of Decision Tree

- Decision tree tends to make `orthogonal` decision boudaries. => Sensitive to the data's orientation.

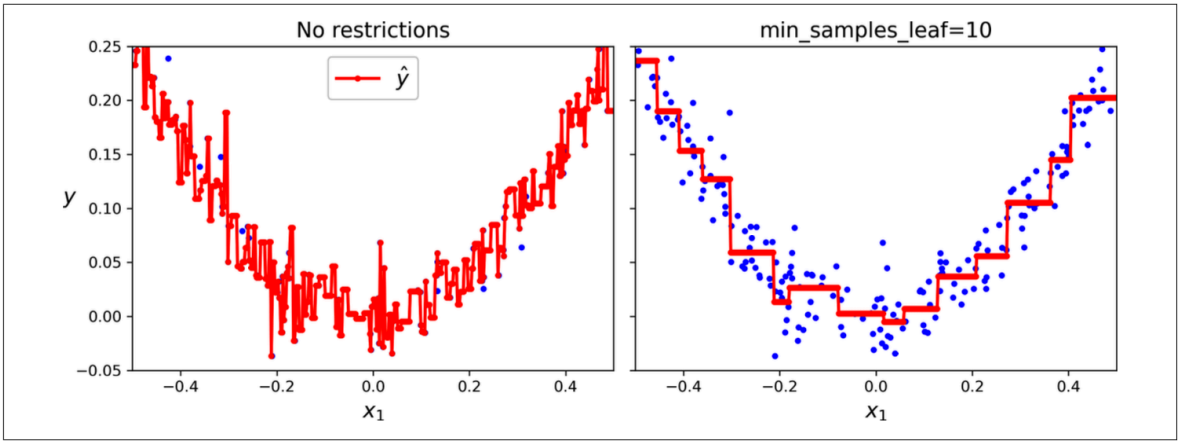


Figure 0.5: Unregularized tree (left) and regularized tree (right

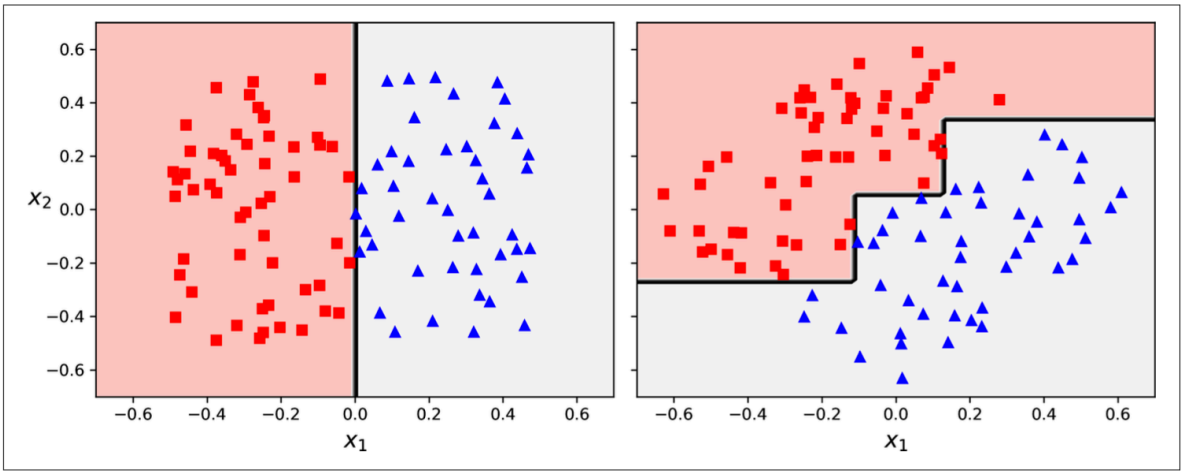


Figure 0.6: Decision tree is sensitive to data's orientation

=> **Solution**: Scale data -> PCA: reduce dimensions but do not loss too much information, rotate data to reduce correlation between features, which often (not always) makes things easier for trees.

Compare to Figure 0.2, the scaled and PCA-rotated iris dataset is separated easier.

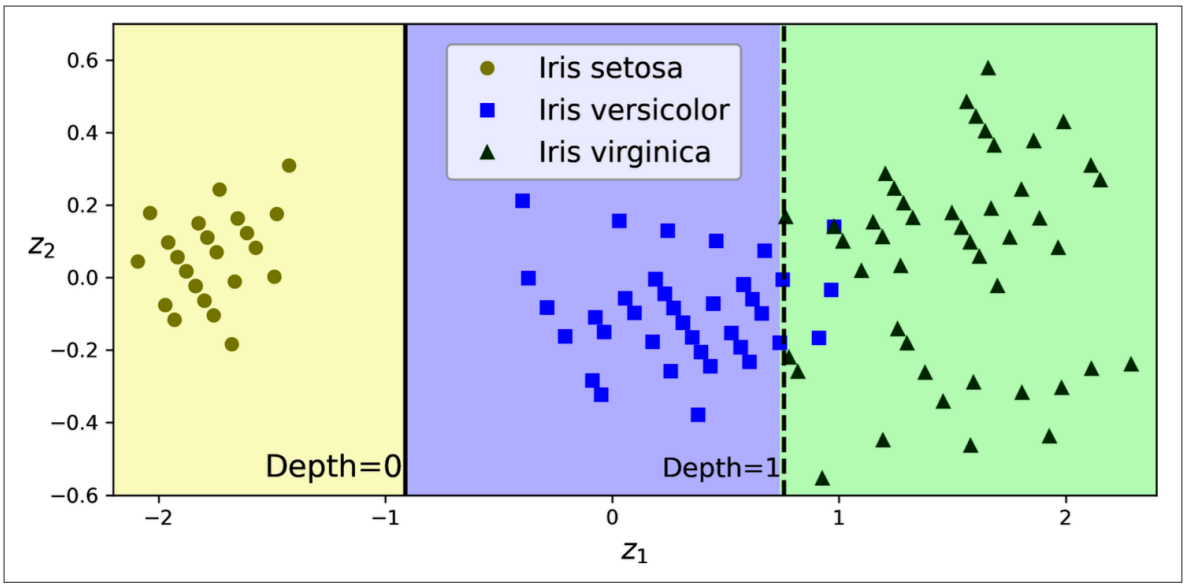


Figure 0.7: Scaled and PCA-rotated data

- High variance: **randomly** select set of features to evaluate at each node, so that if we retrain model on the same dataset, it can behave really different => high variance, unstable
  => **Solution**: Ensemble methods (Random Forest, Boosting methods) averaging predictions over many trees.