Decision Trees

1. Problem Statement

- Dataset: https://github.com/gastonstat/CreditScoring"

The raw dataset is in the file **"CreditScoring.csv"** which contains 4455 rows and 14 columns:

1 Status	credit status
2 Seniority	job seniority (years)
3 Home	type of home ownership
4 Time	time of requested loan
5 Age	client's age
6 Marital	marital status
7 Records	existance of records
8 Job	type of job
9 Expenses	amount of expenses
10 Income	amount of income
11 Assets	amount of assets
12 Debt	amount of debt
13 Amount	amount requested of loan
14 Price	price of good

In this session we'll learn about decision trees and ensemble learning algorithms. The questions that we try to address this week are, "What are decision trees? How are they different from ensemble algorithms? How can we implement and fine-tune these models to make binary classification predictions?"

To be specific, we'll use credit scoring data to build a model that predicts whether a bank should lend loan to a client or not. The bank takes these decisions based on the historical record.

In the credit scoring classification problem,

- if the model returns 0, this means, the client is very likely to payback the loan and the bank will approve the loan.
- if the model returns 1, then the client is considered as a defaulter and the bank may not approval the loan.

Loading Data

```
#Libraries
import pandas as pd
```

```
df = pd.read_csv('credit_score.csv')
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	0	9	rent	60	30	married	no	freelance	73	129.0	0.0	0.0	800	846
1	0	17	rent	60	58	widow	no	fixed	48	131.0	0.0	0.0	1000	1658
2	1	10	owner	36	46	married	yes	freelance	90	200.0	3000.0	0.0	2000	2985
3	0	0	rent	60	24	single	no	fixed	63	182.0	2500.0	0.0	900	1325
4	0	0	rent	36	26	single	no	fixed	46	107.0	0.0	0.0	310	910

Data Understanding

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4454 entries, 0 to 4453
Data columns (total 14 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   status     4454 non-null   int64
```

```
 1   seniority  4454 non-null   int64
 2   home       4454 non-null   object
 3   time       4454 non-null   int64
 4   age        4454 non-null   int64
 5   marital    4454 non-null   object
 6   records    4454 non-null   object
 7   job        4454 non-null   object
 8   expenses   4454 non-null   int64
 9   income     4420 non-null   float64
 10  assets     4407 non-null   float64
 11  debt       4436 non-null   float64
 12  amount     4454 non-null   int64
 13  price      4454 non-null   int64
dtypes: float64(3), int64(7), object(4)
memory usage: 487.3+ KB
```

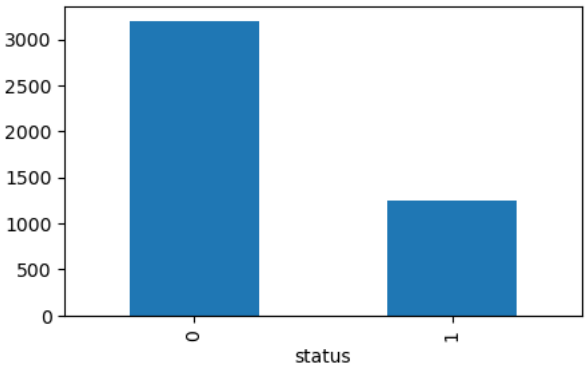
```
#missing values
df.isna().sum().sort_values(ascending=False)
```

```
assets       47
income       34
debt         18
status        0
seniority     0
home          0
time          0
age           0
marital       0
records       0
job           0
expenses      0
amount        0
price         0
dtype: int64
```

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(5,3))
df.status.value_counts().plot(kind='bar')
```

```
<Axes: xlabel='status'>
```



Data Preparation

```
X = df.drop('status', axis=1)
y = df['status']
```

```
#splitting
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

```
import numpy as np
```

```
# categorical and numerical columns
num_cols = X_train.select_dtypes(include=np.number).columns.tolist()
cat_cols = X_train.select_dtypes('object').columns.tolist()
```

```
# fill missing values with mean
from sklearn.impute import SimpleImputer
```

```
imputer = SimpleImputer(strategy='mean')
```

```
num_cols
```

```
['seniority',
 'time',
 'age',
 'expenses',
 'income',
 'assets',
 'debt',
 'amount',
 'price']
```

```
X_train[num_cols] = imputer.fit_transform(X_train[num_cols])
X_test[num_cols] = imputer.transform(X_test[num_cols])
```

Encoding

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False).fit(df[cat_cols])
```

```
encoded_cols = list(encoder.get_feature_names_out(cat_cols))
encoded_cols
```

```
['home_ignore',
 'home_other',
 'home_owner',
 'home_parents',
 'home_private',
 'home_rent',
 'home_unk',
 'marital_divorced',
 'marital_married',
 'marital_separated',
 'marital_single',
 'marital_unk',
 'marital_widow',
 'records_no',
 'records_yes',
 'job_fixed',
 'job_freelance',
 'job_others',
 'job_partime',
 'job_unk']
```

```
X_train[encoded_cols] = encoder.transform(X_train[cat_cols])
X_test[encoded_cols] = encoder.transform(X_test[cat_cols])
```

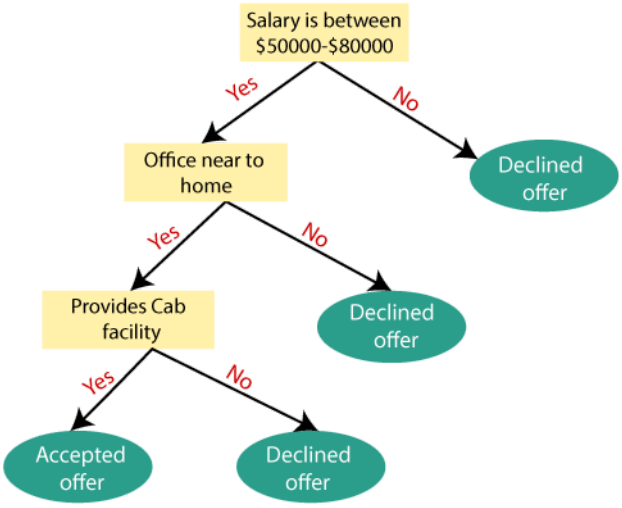
Scaling Numeric Columns

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
X_train[num_cols] = scaler.fit_transform(X_train[num_cols])
X_test[num_cols] = scaler.transform(X_test[num_cols])
```

```
# combining
X_train = X_train[num_cols + encoded_cols]
X_test = X_test[num_cols + encoded_cols]
```

Training and Visualizing Decision Trees



A Decision Tree is a flowchart-like structure where:

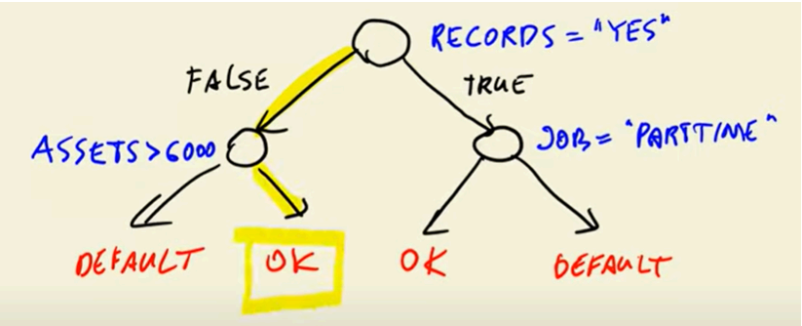
- Internal nodes represent decisions based on features.
- Branches represent outcomes of those decisions.
- Leaf nodes represent the final prediction (class label or continuous value).

It splits the data into subsets based on feature values, aiming to create homogeneous subsets.

```
['records', 'job', 'assets']
```

```
['records', 'job', 'assets']
```

A decision tree in general parlance represents a hierarchical series of binary decisions:



A decision tree in machine learning works in exactly the same way, and except that we let the computer figure out the optimal structure & hierarchy of decisions, instead of coming up with criteria manually.

```
def risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'
```

```
df.columns
```

```
Index(['status', 'seniority', 'home', 'time', 'age', 'marital', 'records',
       'job', 'expenses', 'income', 'assets', 'debt', 'amount', 'price'],
      dtype='object')
```

```
x = df.iloc[140].to_dict()
```

```
x
```

```
{'status': 1,
 'seniority': 13,
 'home': 'rent',
 'time': 60,
 'age': 42,
 'marital': 'separated',
 'records': 'yes',
 'job': 'fixed',
 'expenses': 47,
 'income': 120.0,
 'assets': 0.0,
 'debt': 0.0,
 'amount': 800,
 'price': 1395}
```

```
risk(x)
```

```
'ok'
```

```
from sklearn.tree import DecisionTreeClassifier
```

```
model = DecisionTreeClassifier(random_state=42)
```

```
%%time
model.fit(X_train, y_train)
```

```
CPU times: total: 15.6 ms
Wall time: 40.1 ms
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

```
from sklearn.metrics import accuracy_score
```

```
train_pred = model.predict(X_train)
train_score = accuracy_score(train_pred, y_train)

test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

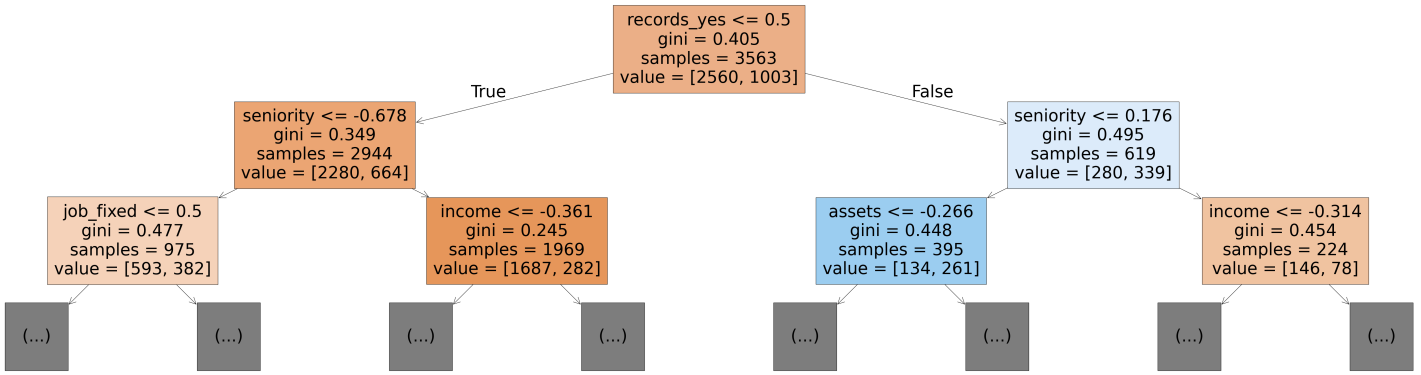
```
Train Score:  0.9997193376368229
Test Score:  0.7396184062850729
```

It appears that the model has learned the training examples perfect, and doesn't generalize well to previously unseen examples. This phenomenon is called "overfitting", and reducing overfitting is one of the most important parts of any machine learning project.

Overfitting is a common problem in machine learning and statistical modeling where a model learns the training data to an extent that it captures noise or random fluctuations in the data rather than the underlying patterns. In other words, an overfit model fits the training data too closely and fails to generalize well to unseen or new data. This can result in poor model performance and accuracy when applied in real-world scenarios. Overfitting can occur in various types of machine learning models, including decision trees, neural networks, support vector machines, and more.

```
from sklearn.tree import plot_tree, export_text
```

```
plt.figure(figsize=(80, 20))

plot_tree(model, feature_names=X_train.columns.tolist(), filled=True, max_depth=2);
```




```
model.tree_.max_depth
```

```
25
```

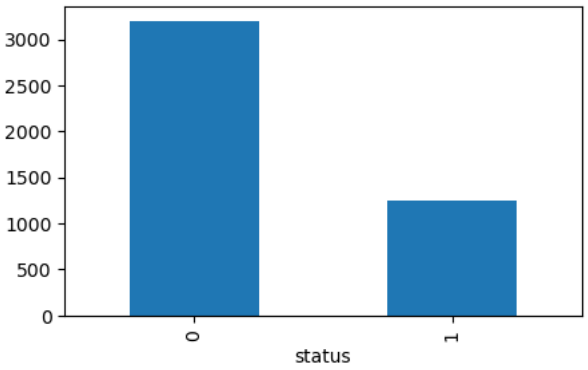
```
len(X_train.columns.tolist())
```

```
29
```

Dealing with Imbalance Class

```
plt.figure(figsize=(5,3))
df.status.value_counts().plot(kind='bar')
```

```
<Axes: xlabel='status'>
```



Which Method to Choose?

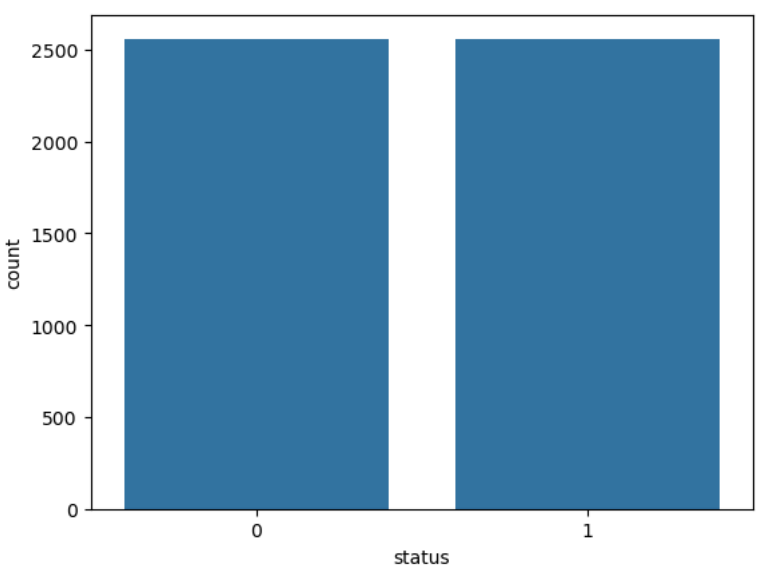
- **Oversampling (e.g., SMOTE)**:
  Use when you have a small dataset and want to increase the minority class.
- **Undersampling**:
  Use when you have a large dataset and can afford to lose some majority class samples.
- **Class Weights**:
  Use when you want to avoid modifying the dataset and prefer to adjust the model's behavior.
- **Ensemble Methods**:
  Use when you want a robust solution designed for imbalanced datasets.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
```

```
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

```
import seaborn as sns
```

```
sns.countplot(x=y_train_resampled)
plt.show()
```




```
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train_resampled, y_train_resampled)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

```
train_pred = model.predict(X_train_resampled)
train_score = accuracy_score(train_pred, y_train_resampled)

test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

```
Train Score:  0.9998046875
Test Score:  0.7261503928170595
```

Can you see how the model classifies a given input as a series of decisions? The tree is truncated here, but following any path from the root node down to a leaf will result in "Yes" or "No". Do you see how a decision tree differs from a logistic regression model?

How a Decision Tree is Created

Note the `gini` value in each box. This is the loss function used by the decision tree to decide which column should be used for splitting the data, and at what point the column should be split. A lower Gini index indicates a better split. A perfect split (only one class on each side) has a Gini index of 0.

For a mathematical discussion of the Gini Index, watch this video: https://www.youtube.com/watch?v=-W0DnxQK1Eo . It has the following formula:

Gini Index

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

p_j: proportion of the samples that belongs to class c for a particular node

Conceptually speaking, while training the models evaluates all possible splits across all possible columns and picks the best one. Then, it recursively performs an optimal split for the two portions. In practice, however, it's very inefficient to check all possible splits, so the model uses a heuristic (predefined strategy) combined with some randomization.

Let's check the depth of the tree that was created.

The situation where training accuracy is 100% while validation accuracy is lower typically indicates overfitting. Overfitting occurs when a model learns the training data too well, capturing noise and details that don't generalize to new data. Possible reasons include memorizing the training set, having a too complex model, lack of regularization, and limited data. To address overfitting, consider using regularization techniques, cross-validation, feature engineering, data augmentation, and finding a balance between model complexity and generalization. Monitoring both training and validation metrics helps identify and address overfitting during model development.

Feature Importance

Based on the gini index computations, a decision tree assigns an "importance" value to each feature. These values can be used to interpret the results given by a decision tree.

```
model.feature_importances_
```

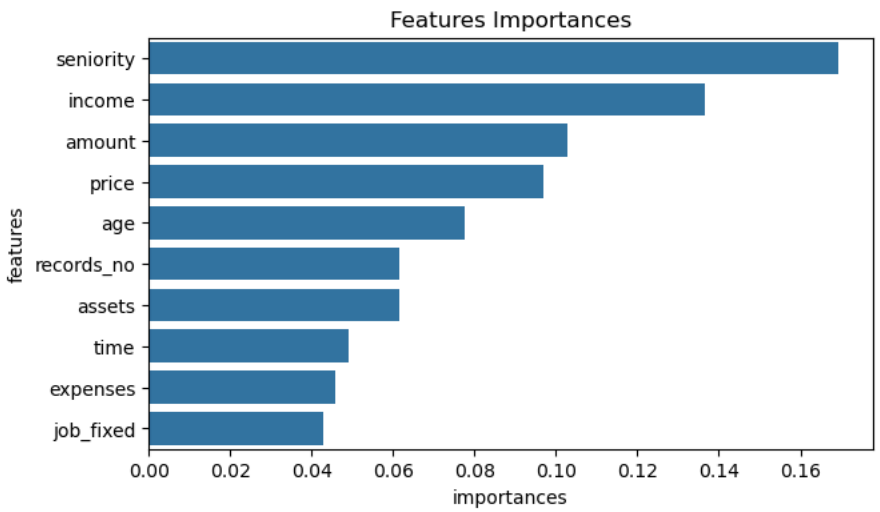
```
array([0.16926723, 0.04895828, 0.07765284, 0.04579937, 0.13672332,
       0.06156006, 0.02385743, 0.1027007 , 0.09677102, 0.0003204 ,
       0.00575844, 0.02396634, 0.00708383, 0.00412534, 0.01297677,
       0.00110923, 0.00066638, 0.00480843, 0.00392577, 0.00852212,
       0.        , 0.00275134, 0.0616683 , 0.03656394, 0.04278272,
       0.00576654, 0.00749667, 0.00641717, 0.        ])
```

```
importance_df = pd.DataFrame({
    'features' : X_train.columns,
    'importances' : model.feature_importances_
}).sort_values('importances', ascending=False)
```

```
importance_df
```

	features	importances
0	seniority	0.169267
4	income	0.136723
7	amount	0.102701
8	price	0.096771
2	age	0.077653
22	records_no	0.061668
5	assets	0.061560
1	time	0.048958
3	expenses	0.045799
24	job_fixed	0.042783
23	records_yes	0.036564
11	home_owner	0.023966
6	debt	0.023857
14	home_rent	0.012977
19	marital_single	0.008522
26	job_others	0.007497
12	home_parents	0.007084
27	job_partime	0.006417
25	job_freelance	0.005767
10	home_other	0.005758
17	marital_married	0.004808
13	home_private	0.004125
18	marital_separated	0.003926
21	marital_widow	0.002751
15	home_unk	0.001109
16	marital_divorced	0.000666
9	home_ignore	0.000320
20	marital_unk	0.000000
28	job_unk	0.000000

```
plt.figure(figsize=(7, 4))
plt.title('Features Importances')
sns.barplot(data=importance_df.head(10), x='importances', y='features')
plt.show()
```



Hyperparameter Tuning and Overfitting

As we saw in the previous section, our decision tree classifier memorized all training examples, leading to a 100% training accuracy, while the validation accuracy was only marginally better than a dumb baseline model. This phenomenon is called overfitting, and in this section, we'll look at some strategies for reducing overfitting. The process of reducing overfitting is known as **`Regularization`**

The `DecisionTreeClassifier` accepts several arguments, some of which can be modified to reduce overfitting.

```
?DecisionTreeClassifier
```

```
Init signature:
DecisionTreeClassifier(
    *,
    criterion='gini',
    splitter='best',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_features=None,
    random_state=None,
    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    class_weight=None,
    ccp_alpha=0.0,
    monotonic_cst=None,
)
Docstring:
A decision tree classifier.

Read more in the :ref:`User Guide <tree>`.

Parameters
----------
criterion : {"gini", "entropy", "log_loss"}, default="gini"
    The function to measure the quality of a split. Supported criteria are
    "gini" for the Gini impurity and "log_loss" and "entropy" both for the
    Shannon information gain, see :ref:`tree_mathematical_formulation`.

splitter : {"best", "random"}, default="best"
    The strategy used to choose the split at each node. Supported
    strategies are "best" to choose the best split and "random" to choose
    the best random split.

max_depth : int, default=None
    The maximum depth of the tree. If None, then nodes are expanded until
    all leaves are pure or until all leaves contain less than
    min_samples_split samples.

min_samples_split : int or float, default=2
    The minimum number of samples required to split an internal node:

    - If int, then consider `min_samples_split` as the minimum number.
    - If float, then `min_samples_split` is a fraction and
      `ceil(min_samples_split * n_samples)` are the minimum
      number of samples for each split.

    .. versionchanged:: 0.18
       Added float values for fractions.

min_samples_leaf : int or float, default=1
    The minimum number of samples required to be at a leaf node.
    A split point at any depth will only be considered if it leaves at
    least ``min_samples_leaf`` training samples in each of the left and
    right branches.  This may have the effect of smoothing the model,
    especially in regression.

    - If int, then consider `min_samples_leaf` as the minimum number.
    - If float, then `min_samples_leaf` is a fraction and
      `ceil(min_samples_leaf * n_samples)` are the minimum
      number of samples for each node.

    .. versionchanged:: 0.18
       Added float values for fractions.

min_weight_fraction_leaf : float, default=0.0
    The minimum weighted fraction of the sum total of weights (of all
    the input samples) required to be at a leaf node. Samples have
    equal weight when sample_weight is not provided.

max_features : int, float or {"sqrt", "log2"}, default=None
    The number of features to consider when looking for the best split:

        - If int, then consider `max_features` features at each split.
        - If float, then `max_features` is a fraction and
          `max(1, int(max_features * n_features_in_))` features are considered at
          each split.
        - If "sqrt", then `max_features=sqrt(n_features)`.
        - If "log2", then `max_features=log2(n_features)`.
        - If None, then `max_features=n_features`.

    Note: the search for a split does not stop until at least one
    valid partition of the node samples is found, even if it requires to
    effectively inspect more than ``max_features`` features.

random_state : int, RandomState instance or None, default=None
    Controls the randomness of the estimator. The features are always
    randomly permuted at each split, even if ``splitter`` is set to
```

```
    randomly permuted at each split, even if ``splitter`` is set to
    ``"best"``. When ``max_features < n_features``, the algorithm will
    select ``max_features`` at random at each split before finding the best
    split among them. But the best found split may vary across different
    runs, even if ``max_features=n_features``. That is the case, if the
    improvement of the criterion is identical for several splits and one
    split has to be selected at random. To obtain a deterministic behaviour
    during fitting, ``random_state`` has to be fixed to an integer.
    See :term:`Glossary <random_state>` for details.

max_leaf_nodes : int, default=None
    Grow a tree with ``max_leaf_nodes`` in best-first fashion.
    Best nodes are defined as relative reduction in impurity.
    If None then unlimited number of leaf nodes.

min_impurity_decrease : float, default=0.0
    A node will be split if this split induces a decrease of the impurity
    greater than or equal to this value.

    The weighted impurity decrease equation is the following::

        N_t / N * (impurity - N_t_R / N_t * right_impurity
                            - N_t_L / N_t * left_impurity)

    where ``N`` is the total number of samples, ``N_t`` is the number of
    samples at the current node, ``N_t_L`` is the number of samples in the
    left child, and ``N_t_R`` is the number of samples in the right child.

    ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
    if ``sample_weight`` is passed.

    .. versionadded:: 0.19

class_weight : dict, list of dict or "balanced", default=None
    Weights associated with classes in the form ``{class_label: weight}``.
    If None, all classes are supposed to have weight one. For
    multi-output problems, a list of dicts can be provided in the same
    order as the columns of y.

    Note that for multioutput (including multilabel) weights should be
    defined for each class of every column in its own dict. For example,
    for four-class multilabel classification weights should be
    [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
    [{1:1}, {2:5}, {3:1}, {4:1}].

    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as ``n_samples / (n_classes * np.bincount(y))``

    For multi-output, the weights of each column of y will be multiplied.

    Note that these weights will be multiplied with sample_weight (passed
    through the fit method) if sample_weight is specified.

ccp_alpha : non-negative float, default=0.0
    Complexity parameter used for Minimal Cost-Complexity Pruning. The
    subtree with the largest cost complexity that is smaller than
    ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
    :ref:`minimal_cost_complexity_pruning` for details.

    .. versionadded:: 0.22

monotonic_cst : array-like of int of shape (n_features), default=None
    Indicates the monotonicity constraint to enforce on each feature.
      - 1: monotonic increase
      - 0: no constraint
      - -1: monotonic decrease

    If monotonic_cst is None, no constraints are applied.

    Monotonicity constraints are not supported for:
      - multiclass classifications (i.e. when `n_classes > 2`),
      - multioutput classifications (i.e. when `n_outputs_ > 1`),
      - classifications trained on data with missing values.

    The constraints hold over the probability of the positive class.

    Read more in the :ref:`User Guide <monotonic_cst_gbdt>`.

    .. versionadded:: 1.4

Attributes
----------
classes_ : ndarray of shape (n_classes,) or list of ndarray
    The classes labels (single output problem),
    or a list of arrays of class labels (multi-output problem).

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
```

```
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature.  It is also
    known as the Gini importance [4]_.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

max_features_ : int
    The inferred value of max_features.

n_classes_ : int or list of int
    The number of classes (for single output problems),
    or a list containing the number of classes for each
    output (for multi-output problems).

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

tree_ : Tree instance
    The underlying Tree object. Please refer to
    ``help(sklearn.tree._tree.Tree)`` for attributes of Tree object and
    :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py`
    for basic usage of these attributes.

See Also
--------
DecisionTreeRegressor : A decision tree regressor.

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The :meth:`predict` method operates using the :func:`numpy.argmax`
function on the outputs of :meth:`predict_proba`. This means that in
case the highest predicted probabilities are tied, the classifier will
predict the tied class with the lowest index in :term:`classes_`.

References
----------

.. [1] https://en.wikipedia.org/wiki/Decision_tree_learning

.. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification
       and Regression Trees", Wadsworth, Belmont, CA, 1984.

.. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical
       Learning", Springer, 2009.

.. [4] L. Breiman, and A. Cutler, "Random Forests",
       https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Examples
--------
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...                             # doctest: +SKIP
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
File:           c:\users\user\anaconda3\lib\site-packages\sklearn\tree\_classes.py
Type:           ABCMeta
Subclasses:     ExtraTreeClassifier
```

These arguments are called hyperparameters because they must be configured manually (as opposed to the parameters within the model which are *learned* from the data. We'll explore a couple of hyperparameters:

- `max_depth`
- `max_leaf_nodes`

`max_depth`

By reducing the maximum depth of the decision tree, we can prevent the tree from memorizing all training examples, which may lead to better generalization

```
model.tree_.max_depth
```

```
24
```

```
#
model = DecisionTreeClassifier(max_depth=7, random_state=42)
model.fit(X_train, y_train)
```

```
DecisionTreeClassifier(max_depth=7, random_state=42)
```

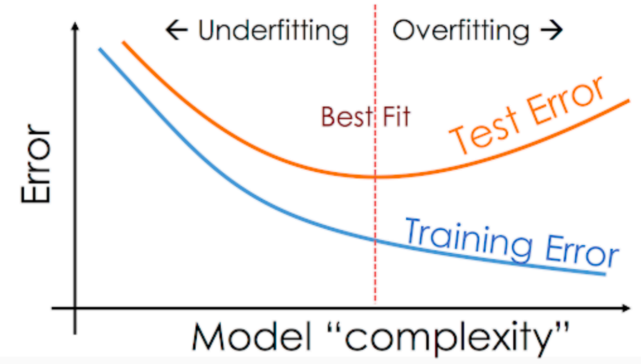
```
train_pred = model.predict(X_train_resampled)
train_score = accuracy_score(train_pred, y_train_resampled)

test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

```
Train Score:  0.76171875
Test Score:  0.7586980920314254
```

```
result = []

for depth in range(1, 11):
    model = DecisionTreeClassifier(max_depth=depth, random_state=42)
    model.fit(X_train, y_train)

    #  train Error
    train_pred = model.predict(X_train)
    train_error = 1 - accuracy_score(train_pred, y_train)

    test_pred = model.predict(X_test)
    test_error = 1 - accuracy_score(test_pred, y_test)

    result.append({
        'Max Depth' : depth,
        'Training Error' : train_error,
        'Test Error' : test_error
    })

result_df = pd.DataFrame(result)
result_df
```

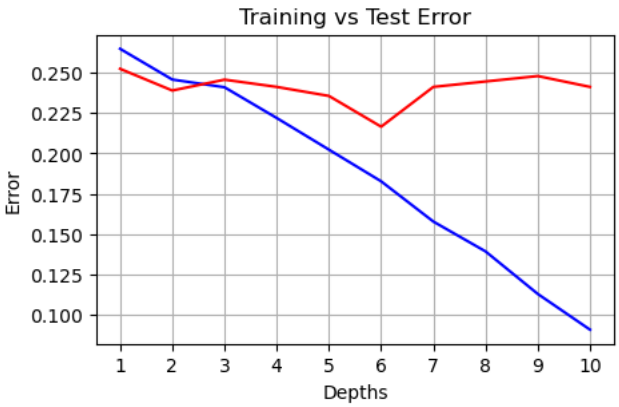
	Max Depth	Training Error	Test Error
0	1	0.264945	0.252525
1	2	0.245860	0.239057
2	3	0.241089	0.245791
3	4	0.222004	0.241302
4	5	0.202358	0.235690
5	6	0.182711	0.216611
6	7	0.157732	0.241302
7	8	0.139209	0.244669
8	9	0.112826	0.248036
9	10	0.090654	0.241302

This is a common pattern you'll see with all machine learning algorithms:



You'll often need to tune hyperparameters carefully to find the optimal fit. In the above case, it appears that a maximum depth of 7 results in the lowest validation error.

```
plt.figure(figsize=(5,3))
plt.plot(result_df['Max Depth'], result_df['Training Error'], color='blue')
plt.plot(result_df['Max Depth'], result_df['Test Error'], color='red')
plt.title('Training vs Test Error')
plt.xlabel('Depths')
plt.ylabel('Error')
plt.xticks(range(1, 11))
plt.grid()
plt.show()
```




```
model = DecisionTreeClassifier(max_depth=5, random_state=42)
model.fit(X_train, y_train)

train_pred = model.predict(X_train_resampled)
train_score = accuracy_score(train_pred, y_train_resampled)
```

```
test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

```
Train Score:  0.7068359375
Test Score:  0.7643097643097643
```

```
scores = []

for depth in [3, 4, 5, 6]:
    for leaf in [1, 3, 5, 7, 9, 11, 13, 15]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=leaf, random_state=42)
        dt.fit(X_train, y_train)
        y_pred = dt.predict(X_test)
        score = accuracy_score(y_test, y_pred)
        scores.append((depth, leaf, score))
```

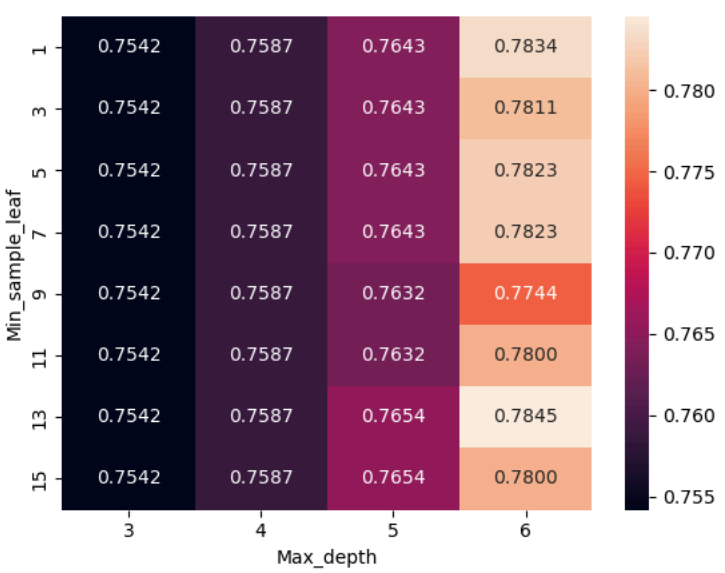
```
df_scores = pd.DataFrame(scores,
                         columns=['Max_depth', 'Min_sample_leaf', 'scores'])
df_scores
```

	Max_depth	Min_sample_leaf	scores
0	3	1	0.754209
1	3	3	0.754209
2	3	5	0.754209
3	3	7	0.754209
4	3	9	0.754209
5	3	11	0.754209
6	3	13	0.754209
7	3	15	0.754209
8	4	1	0.758698
9	4	3	0.758698
10	4	5	0.758698
11	4	7	0.758698
12	4	9	0.758698
13	4	11	0.758698
14	4	13	0.758698
15	4	15	0.758698
16	5	1	0.764310
17	5	3	0.764310
18	5	5	0.764310
19	5	7	0.764310
20	5	9	0.763187
21	5	11	0.763187
22	5	13	0.765432
23	5	15	0.765432
24	6	1	0.783389
25	6	3	0.781145
26	6	5	0.782267
27	6	7	0.782267
28	6	9	0.774411
29	6	11	0.780022
30	6	13	0.784512
31	6	15	0.780022

```
df_scores = df_scores.pivot(index='Min_sample_leaf',
                            columns='Max_depth',
                            values='scores')
df_scores
```

Min_sample_leaf	3	4	5	6
1	0.754209	0.758698	0.764310	0.783389
3	0.754209	0.758698	0.764310	0.781145
5	0.754209	0.758698	0.764310	0.782267
7	0.754209	0.758698	0.764310	0.782267
9	0.754209	0.758698	0.763187	0.774411
11	0.754209	0.758698	0.763187	0.780022
13	0.754209	0.758698	0.765432	0.784512
15	0.754209	0.758698	0.765432	0.780022

```
sns.heatmap(df_scores, annot=True, fmt='.4f')
plt.show()
```




```
model = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15, random_state=42)
model.fit(X_train, y_train)

train_pred = model.predict(X_train_resampled)
train_score = accuracy_score(train_pred, y_train_resampled)

test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

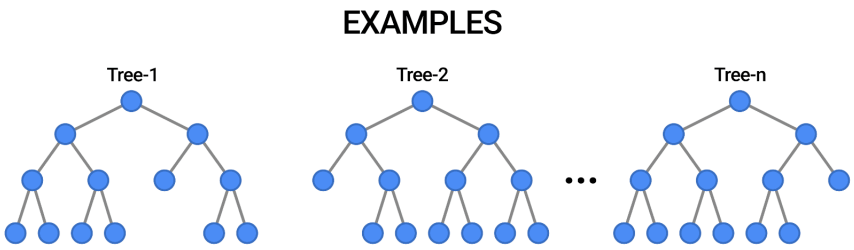
```
Train Score:  0.724609375
Test Score:  0.7800224466891134
```

Training a Random Forest

While tuning the hyperparameters of a single decision tree may lead to some improvements, a much more effective strategy is to combine the results of several decision trees trained with slightly different parameters. This is called a random forest model.

The key idea here is that each decision tree in the forest will make different kinds of errors, and upon averaging, many of their errors will cancel out. This idea is also commonly known as the "wisdom of the crowd":

A random forest works by averaging/combining the results of several decision trees:



We'll use the `RandomForestClassifier` class from `sklearn.ensemble`.

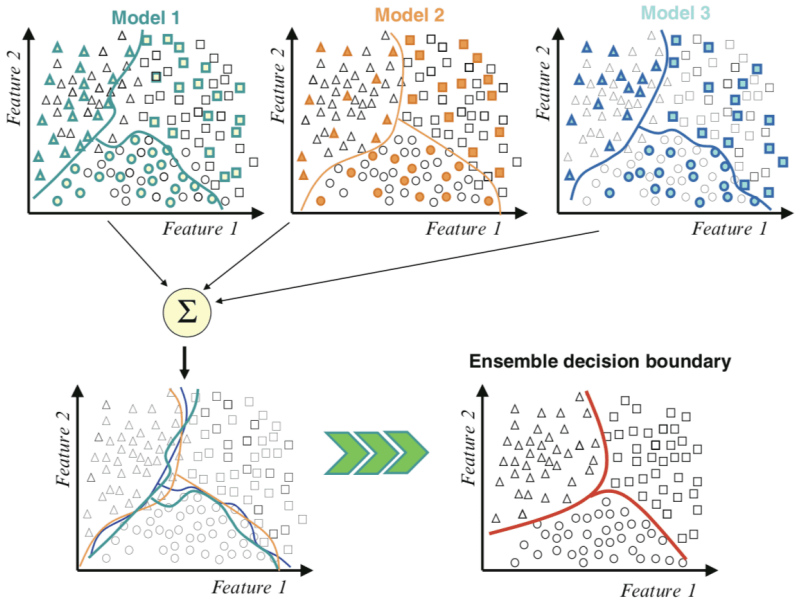
```
from sklearn.ensemble import RandomForestClassifier
```

```
model = RandomForestClassifier(random_state=42, n_jobs=-1)
model.fit(X_train, y_train)

train_pred = model.predict(X_train_resampled)
train_score = accuracy_score(train_pred, y_train_resampled)

test_pred = model.predict(X_test)
test_score = accuracy_score(test_pred, y_test)

print('Train Score: ', train_score)
print('Test Score: ', test_score)
```

```
Train Score:  0.9400390625
Test Score:  0.8103254769921436
```

Once again, the training accuracy is almost 100%, but this time the validation accuracy is much better. In fact, it is better than the best single decision tree we had trained so far. Do you see the power of random forests?

This general technique of combining the results of many models is called "ensembling", it works because most errors of individual models cancel out on averaging. Here's what it looks like visually:



We can also look at the probabilities for the predictions. The probability of a class is simply the fraction of trees which that predicted the given class.

Notice that the distribution is a lot less skewed than that for a single decision tree.

Hyperparameter Tuning with Random Forests

Just like decision trees, random forests also have several hyperparameters. In fact many of these hyperparameters are applied to the underlying decision trees.

Let's study some the hyperparameters for random forests. You can learn more about them here: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

```
?RandomForestClassifier
```