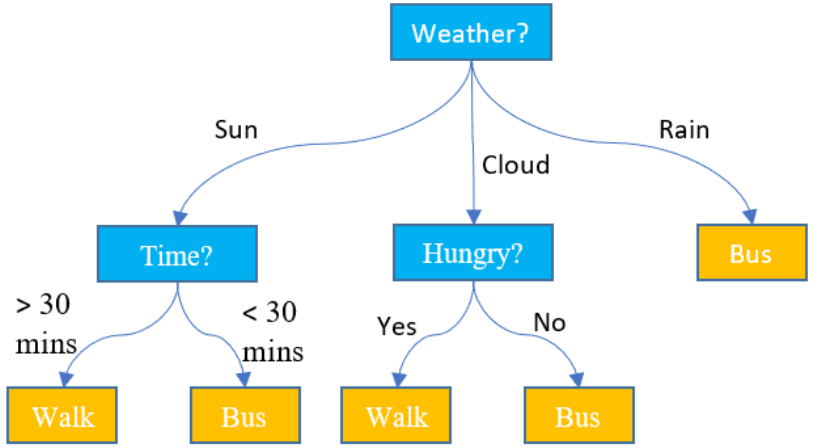
Decision Trees

A **decision tree** is a Supervised Learning Algorithm used for both Classification and Regression problems. It splits the dataset using conditions and forms a tree like structure which acts as a flow chart to classify the records. It consists of branches and nodes.



Source: https://www.displayr.com/what-is-a-decision-tree/

Root Node: The first node with all the data points and maximum impurity. It identifies the best split and divides the data accordingly

Decision Node: It is an intermediate node that further splits the data with the best splitting condition

Leaf nodes: They are the final nodes in the decision tree which makes the final prediction or classification

Impurity measures in Decision Trees are used to select the best split for that node. The lower the impurity the better the split, yielding the maximum purity in each node.

The tree keeps growing until all the nodes are completely pure.

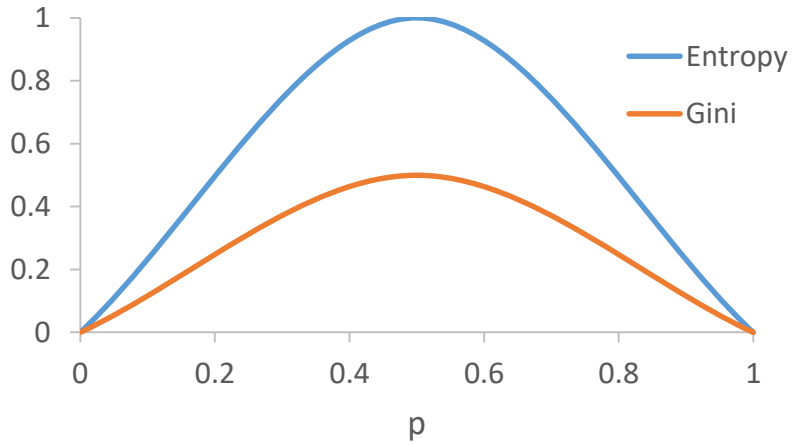
Commonly used impurity measures are: Gini, Entropy and Information gain

Gini Impurity: $1 - \sum_i p_i^2$ (ranges from 0 to 0.5)

Entropy (E): $1 - \sum_{i=1}^{C} -p_i \log_2 p_i$ (ranges from 0 to 1)

Information Gain:

$$Entropy_{(previous\ node)} - \frac{m}{m+n} Entropy(First\ child) - \frac{m}{m+n} Entropy(Second\ child)$$



- Higher Gini and Entropy values indicate higher impurity
- Reduction in impurity after the split is measured by information gain

Pruning and Parameters in Decision Trees

One of the dis-advantage of Decision Tree is, it is prone to overfitting. The tree grows to the maximum depth until all the samples in the training data are perfectly fitted.

Pruning: It helps us to overcome the problem of overfitting by reducing or cutting off the branches or nodes of the decision tree to improve the model.

Cost Complexity Pruning: It is one of the pruning techniques which uses cost complexity parameter ccp_alpha to prune the nodes.
Reference Link: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html

max_depth: It is the length between the root node and last nodes. By assigning max depth value we can limit the tree growth and simplify the model.
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

min_samples_leaf: The minimum number of samples required in the leaf node. Further splitting will be stopped if the child nodes cannot meet this minimum samples requirement.
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

min_impurity_decrease: The tree grows by decreasing the impurity in each node it builds and sometimes it even grows if there no decrease in the impurity leading to overfit model. By setting some value to this parameter will help us limit the tree growth
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

min_samples_split: The minimum of samples required to split an internal mode. If the number of samples is less than the specified value, then split will not happen and the tree stops growing.
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

max_leaf_nodes: This parameter helps us to limit the tree based on the number of leaf nodes. The tree stops growing once it attains the maximum leaf nodes as specified in the parameter.
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

max_features: It is used to specify the maximum number of features to be considered for the best split.
Reference Link: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

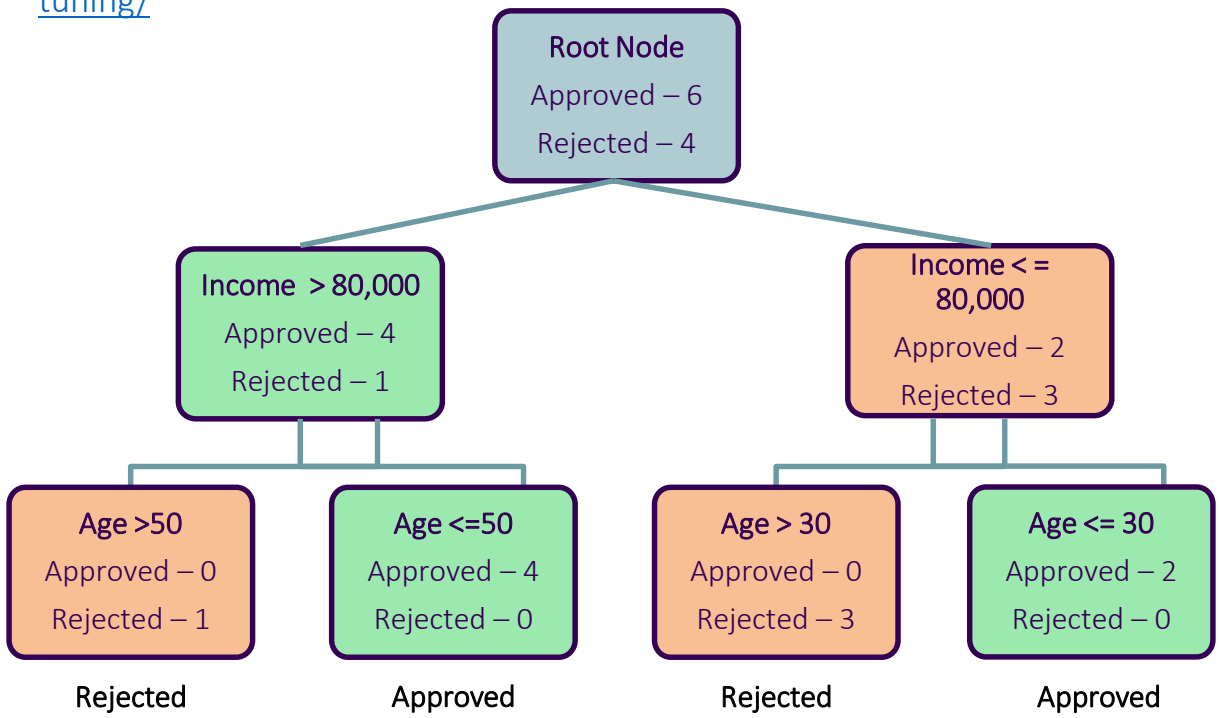
Pruning and Visualizing Trees

```
#Fitting the model
d_tree = DecisionTreeClassifier()
d_tree.fit(X_train,y_train)

#Pruning with cost complexity parameter
path = d_tree.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

#Next, we train a decision tree using the effective alphas. The last value in
ccp_alphas is the alpha value that prunes the whole tree, leaving the tree,
clfs[-1], with one node.
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)
print(
    "Number of nodes in the last tree is: {} with ccp_alpha: {}".format(
        clfs[-1].tree_.node_count, ccp_alphas[-1]
    )
)
```

Reference: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html

```
#Visualizing a Decision Tree
from sklearn.tree import plot_tree, export_text
plt.figure(figsize =(80,20))

plot_tree(d_tree, feature_names=X_train.columns, max_depth=2,
filled=True)
```

Reference: https://www.section.io/engineering-education/hyperparmeter-tuning/



Ensemble Methods

Ensemble Methods combines the multiple individual models to derive an output or prediction. It is used for both classification and regression problems.
We get an highly accurate and generalizable model.

Averaging methods, the driving principle is to build several estimators independently and then to average / vote their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.
E.g. Bagging methods, Forests of randomized trees, ...

Boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. E.g.
AdaBoost, Gradient Tree Boosting, ...

Bagging: It is also known as Bootstrap Aggregation, uses sampling with replacement to generate multiple samples of a given size to build an ensemble of models that improve performance and accuracy. It learns from a homogenous weak learners independently and aggregates them.

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

Random Forest: One of the bagging methods where each tree is built from a sample drawn with replacement from the training set. It uses a random subset of features instead of all features and finds the best split among them.

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

Ada Boosting: In AdaBoost, the successive models are created with a focus on the ill fitted data of the previous learner. Each successive model focuses more and more on the harder to fit data i.e. their residuals in the previous model. Model instances are created sequentially; except for the first, each subsequent model is grown from previously grown learners

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> X, y = load_iris(return_X_y=True)
>>> clf = AdaBoostClassifier(n_estimators=100)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
```

Ensemble Methods

Gradient Boosting: Similarly to AdaBoost, gradient tree boosting is built from a set of small trees, though usually slightly deeper than decision stumps. The trees are trained sequentially, just like in AdaBoost, but the training of individual trees is not the same

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

XGBoost: It is called as Extreme Gradient Boosting which uses gradient boosting framework with better performance and speed.

```
>>> from xgboost import XGBClassifier
>>> xgb_classifier = XGBClassifier(random_state=1, eval_metric= "error")
>>> xgb_classifier.fit(X_train.astype('int'),y_train)
```

Stacking: It uses a meta-learning algorithm to learn how to best combine the predictions from two or more base machine learning algorithms.

```
>>> from sklearn.linear_model import RidgeCV, LassoCV
>>> from sklearn.neighbors import KNeighborsRegressor

>>> estimators = [('ridge', RidgeCV()),
...               ('lasso', LassoCV(random_state=42)),
...               ('knr', KNeighborsRegressor(n_neighbors=20,
...                                           metric='euclidean'))]

>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import StackingRegressor
>>> final_estimator = GradientBoostingRegressor(
...     n_estimators=25, subsample=0.5, min_samples_leaf=25,
max_features=1,
...     random_state=42)
>>> reg = StackingRegressor(
...     estimators=estimators,
...     final_estimator=final_estimator)
```

Reference: https://scikit-learn.org/stable/modules/ensemble.html