Day 2: Classification and Model Evaluation

Machine Learning Course

Sammie Omranian

6/11/2024

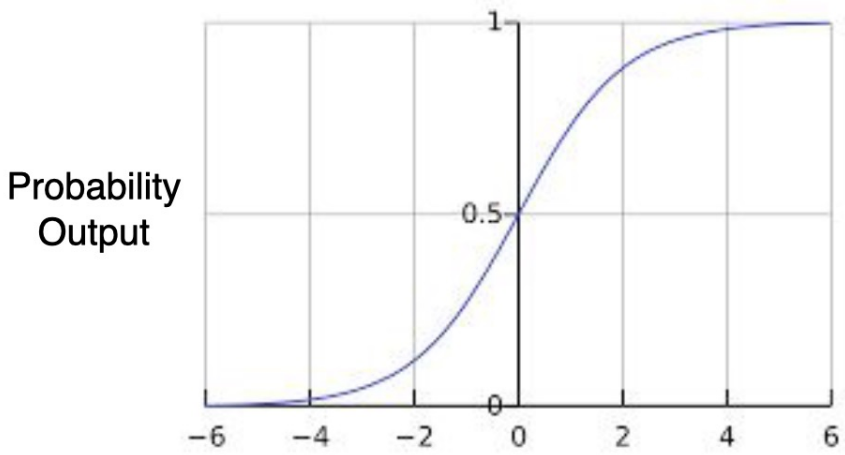
Classification Algorithms Overview

- Logistic Regression
- K-Nearest Neighbors (KNN)
- Support Vector Machines (SVM)

Logistic Regression

- Logistic regression is a linear model used for binary classification problems.
- It estimates the probability that a given input belongs to a particular class.
- **Use Case:**
  - Spam detection, disease diagnosis.

  - The model uses the logistic (sigmoid) function to map predicted values to probabilities.
  - Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$



Logistic Regression

Mathematical Foundation:

1. Sigmoid Function (Logistic Function):
   - The sigmoid function is used to map any real-valued number into the [0, 1] interval.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

   - Here, z is the input to the function, which is typically the linear combination of the input features and their corresponding coefficients.

2. Logistic Regression Equation:
   - The logistic regression model predicts the probability that the input X belongs to the positive class (class 1) as follows:

$$P(y = 1|X) = \sigma(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n)$$

   1. Here:
      - $P(y = 1|X)$ is the probability that the output y is 1 given the input features X.
      - β_0 is the intercept (bias term).
      - $\beta_1, , \beta_2 , \dots , \beta_n$ are the coefficients (weights) for the input features $X_1, X_2 , \dots , X_n$.

Logistic Regression

Example:

Let's consider a simple example where we have two features (X_1 and X_2):

1. Logistic regression equation:

$$P(y = 1|X) = \sigma(\beta_0 + \beta_1 X_1 + \beta_2 X_2)$$

2. Suppose we have the following coefficients:

   - $\beta_0 = -3$
   - $\beta_1 = 2$
   - $\beta_2 = 1$

3. For an input X where $X_1 = 1$ and $X_2 = 2$:

$$z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 = -3 + 2(1) + 1(2) = 1$$

4. Apply the sigmoid function:

$$\sigma(1) = \frac{1}{1 + e^{-1}} \approx 0.731$$

5. The probability that the input X belongs to class 1 is approximately 0.731 (or 73.1%).

Logistic Regression

- **from** sklearn.linear_model **import** LogisticRegression

- ***class*** **sklearn.linear_model.LogisticRegression(*penalty*='l2', *, *dual*=False, *tol*=0.0001, *C*=1.0, *fit_intercept*=True, *intercept_scaling*=1, *class_weight*=None, *random_state*=None, *solver*='lbfgs', *max_iter*=100, *multi_class*='deprecated', *verbose*=0, *warm_start*=False, *n_jobs*=None, *l1_ratio*=None)**

Source

When training your model do hyperparameter tuning using GridSearchCV.

By hyperparameter tuning you are adjusting hyperparameters to find the optimal settings that result in the best model performance.

K-Nearest Neighbors (KNN)

KNN is a simple non-parametric algorithm that classifies data based on the majority class of its k-nearest neighbors.

- It makes predictions by finding the most similar training examples to the test data points.
- Use Case:
  - Image recognition, recommendation systems.

- Distance Metrics:
  - Commonly uses Euclidean distance, but other metrics can also be used.

$\mathbf{p} = (p_1, p_2, \ldots, p_n)$ and $\mathbf{q} = (q_1, q_2, \ldots, q_n)$ n-dimensional space

Euclidean Distance $d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$

K-Nearest Neighbors (KNN)

- Training Phase:
  - During the training phase, KNN simply stores the labeled training data. There is no explicit model training or parameter estimation in the traditional sense because KNN is a lazy learning algorithm.

- Prediction Phase:
  - When making a prediction for a new, unseen data point, KNN searches through the stored training data to find the k-nearest neighbors.

  - The algorithm then uses the labels of these neighbors to determine the label of the new data point, usually through majority voting (for classification) or averaging (for regression).

K-Nearest Neighbors (KNN)

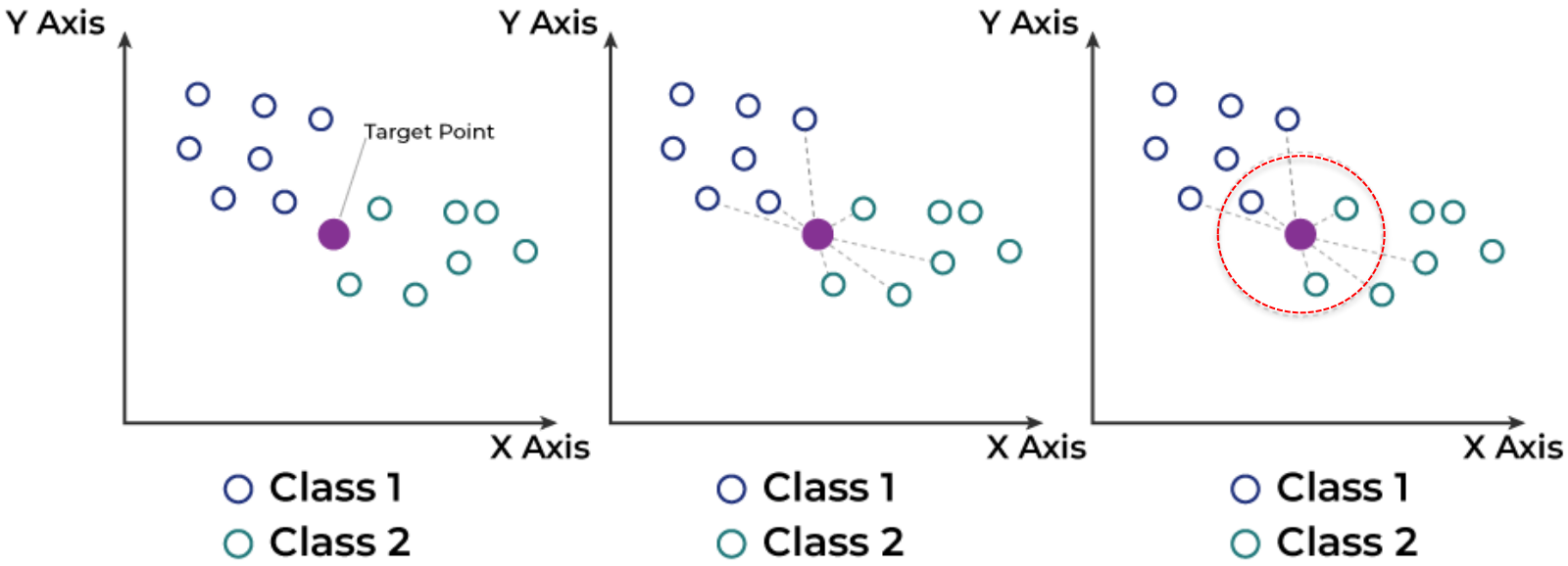


Image source

K-Nearest Neighbors (KNN)

- **from** sklearn.neighbors **import** NearestNeighbors

- ***class*** **sklearn.neighbors.KNeighborsClassifier(*n_neighbors*=5, *, *weights*='***uniform***', *algorithm*='auto', *leaf_size*=30, *p*=2, *metric*='minkowski', *metric_params*=None, *n_jobs*=None)**

- source

Support Vector Machines (SVM)

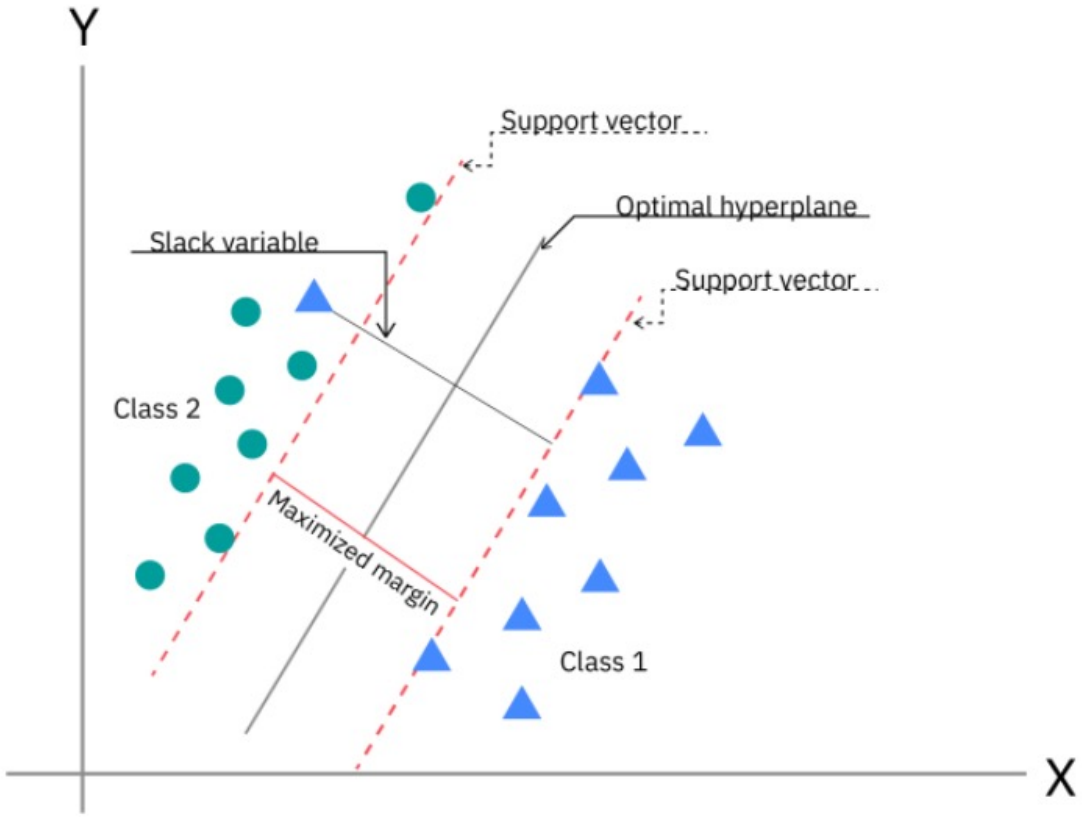
A Support Vector Machine is a supervised learning algorithm that classifies data by finding an optimal line or hyperplane that maximizes the distance between each class in an N-dimensional space.

- SVMs are commonly used within classification problems.
- They distinguish between two classes by finding the optimal hyperplane that maximizes the margin between the closest data points of opposite classes.
- The number of features in the input data determine if the hyperplane is a line in a 2-D space or a plane in a n-dimensional space.

Support Vector Machines (SVM)

- Since multiple hyperplanes can be found to differentiate classes, maximizing the margin between points enables the algorithm to find the best decision boundary between classes.

The lines that are adjacent to the optimal hyperplane are known as **support vectors** as these vectors run through the data points that determine the maximal margin.



Support Vector Machines (SVM)

- **from** sklearn.svm **import** SVC

***class* sklearn.svm.SVC(*, *C*=*1.0*, *kernel*='*rbf*', *degree*=*3*, *gamma*='*scale*', *coef0*=*0.0*, *shrinking*=*True*, *probability*=*False*, *tol*=*0.001*, *cache_size*=*200*, *class_weight*=*None*, *verbose*=*False*, *max_iter*=*-1*, *decision_function_shape*='*ovr*', *break_ties*=*False*, *random_state*=*None*)**

source

Model Evaluation Metrics

- Accuracy:
- Accuracy is one metric for evaluating classification models.
- Informally, **accuracy** is the fraction of predictions our model got right.
- Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Where *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

Model Evaluation Metrics

Precision

- **Precision** attempts to answer the following question:
- What proportion of positive identifications was actually correct?

- Precision is defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Model Evaluation Metrics

Recall

- **Recall** attempts to answer the following question:
- What proportion of actual positives was identified correctly?
- Mathematically, recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Model Evaluation Metrics

F1 Score

- The harmonic mean of precision and recall.
- The F1 score provides a single metric that considers both false positives and false negatives. It helps to understand how well the model is performing in terms of both detecting positive instances and avoiding incorrect positive predictions.
- Mathematically, f1 score is defined as follows:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Model Evaluation Metrics

ROC curve

- An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

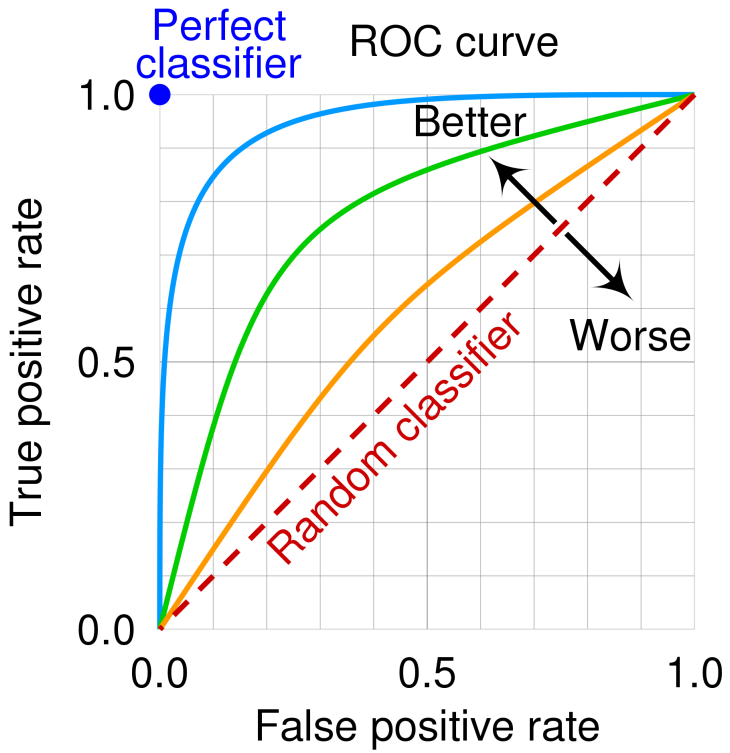
- True Positive Rate (TPR is a synonym for recall)

- False Positive Rate, (FPR) is defined as follows:

$$FPR = \frac{FP}{FP + TN}$$

Model Evaluation Metrics

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.



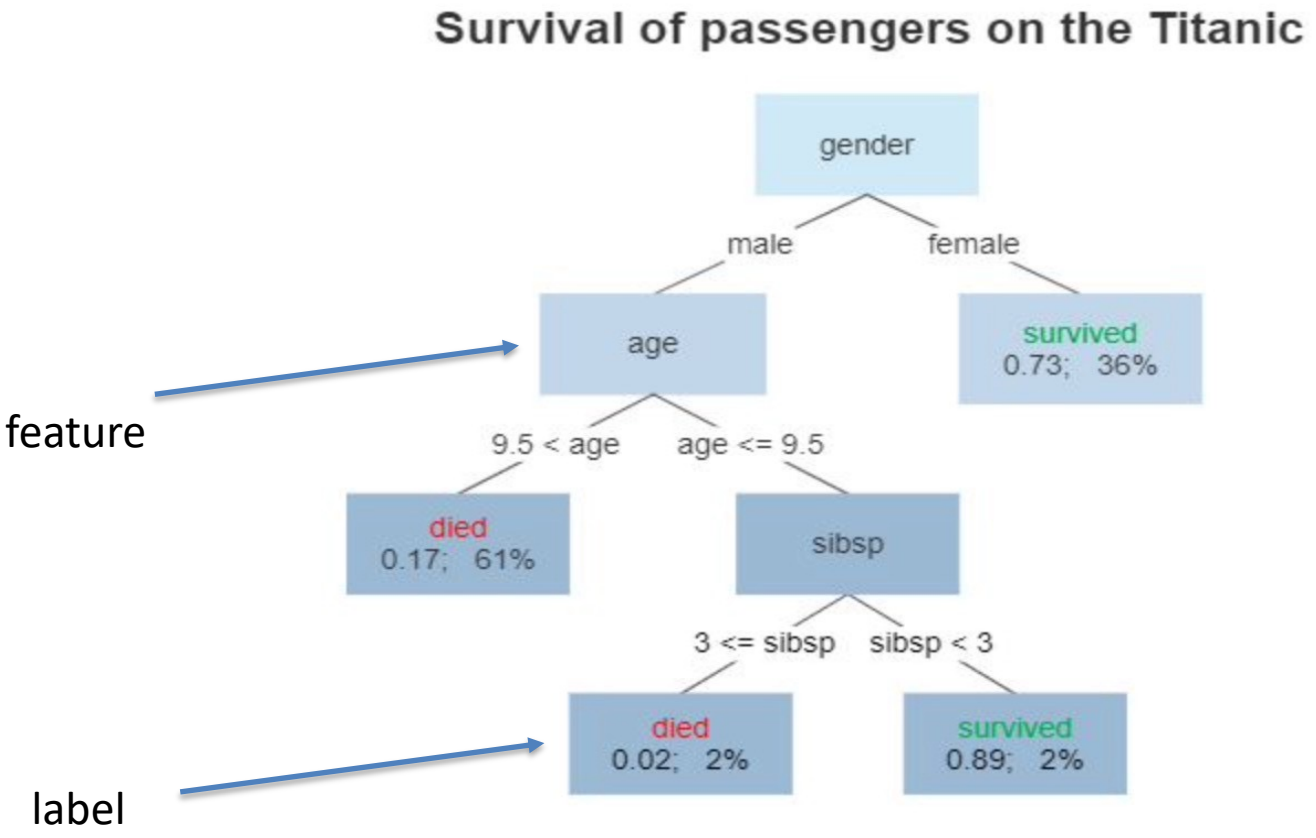
Decision Trees and Ensemble Methods Overview

- Decision Trees
- Random Forests
- GBM
- AdaBoost
- XGBoost

Decision Trees

- Definition: A tree-like model used to make decisions based on the values of input features.

if-then-else true/false



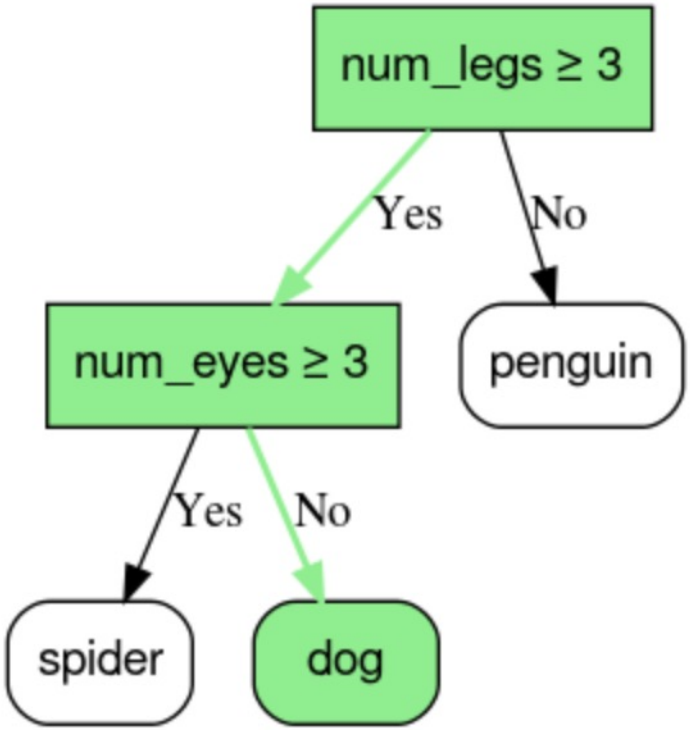
Decision Trees

Example: Consider the following feature values:

num_legs : 4, num_eyes : 2}

The prediction would be *dog*.

1. num_legs ≥ 3 → Yes
2. num_eyes ≥ 3 → No



Decision Trees

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

class **sklearn.tree.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)**

source

Ensemble Methods

In machine learning, an **ensemble** is a collection of models whose predictions are averaged (or aggregated in some way).

Combine multiple base models to create a more robust and accurate overall model.

Types:

- Bagging: Trains multiple models independently on different subsets of the data. ex, Random forest

- Boosting: Trains models sequentially, with each new model focusing on correcting errors of the previous ones. ex, XGBoost

Random Forest

- A **random forest** (**RF**) is an ensemble of decision trees that combines the predictions of multiple decision trees to improve accuracy, robustness, and generalization.

- How it works:
- **Bootstrap Sampling**: Each tree in the ensemble is trained on a different subset of the data, created by randomly sampling with replacement from the original dataset, called the bootstrap sample.

- **Random Feature Selection**: During the training of each tree, a random subset of features is selected at each split, ensuring that the trees are diverse.

- **Voting/Averaging**: For classification, the final prediction is made by majority voting among the trees. For regression, the final prediction is the average of the predictions of all trees.

Random Forest

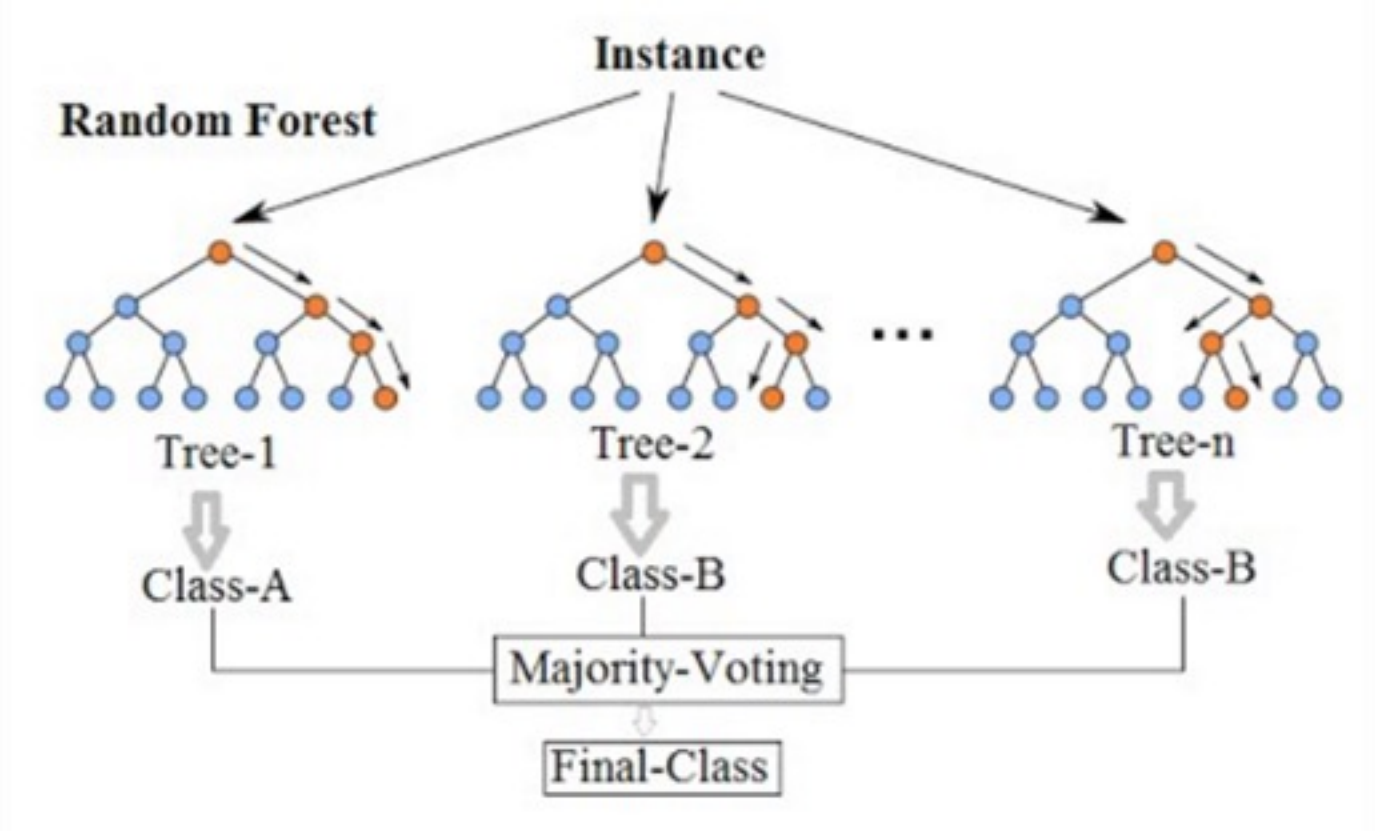


Image credit

Random Forest

- **from** sklearn.ensemble **import** RandomForestClassifier

- ***class*** **sklearn.ensemble.RandomForestClassifier(*n_estimators*=*100*, *, *criterion*=*'gini'*, *max_depth*=*None*, *min_samples_split*=*2*, *min_samples_leaf*=*1*, *min_weight_fraction_leaf*=*0.0*, *max_features*=*'sqrt'*, *max_leaf_nodes*=*None*, *min_impurity_decrease*=*0.0*, *bootstrap*=*True*, *oob_score*=*False*, *n_jobs*=*None*, *random_state*=*None*, *verbose*=*0*, *warm_start*=*False*, *class_weight*=*None*, *ccp_alpha*=*0.0*, *max_samples*=*None*, *monotonic_cst*=*None*)**

- **Sample**

- Number of Trees (n_estimators)

XGBoost

- XGBoost (Extreme Gradient Boosting) builds an ensemble of decision trees sequentially, where each new tree is trained to correct the errors made by the previous trees.

- XGBoost is a highly optimized and efficient implementation of the gradient boosting algorithm.

- The term “gradient boosting” comes from the idea of “boosting” or improving a single weak model by combining it with a number of other weak models in order to generate a collectively strong model. It generalizes boosting by using gradient descent to minimize the loss function.

XGBoost

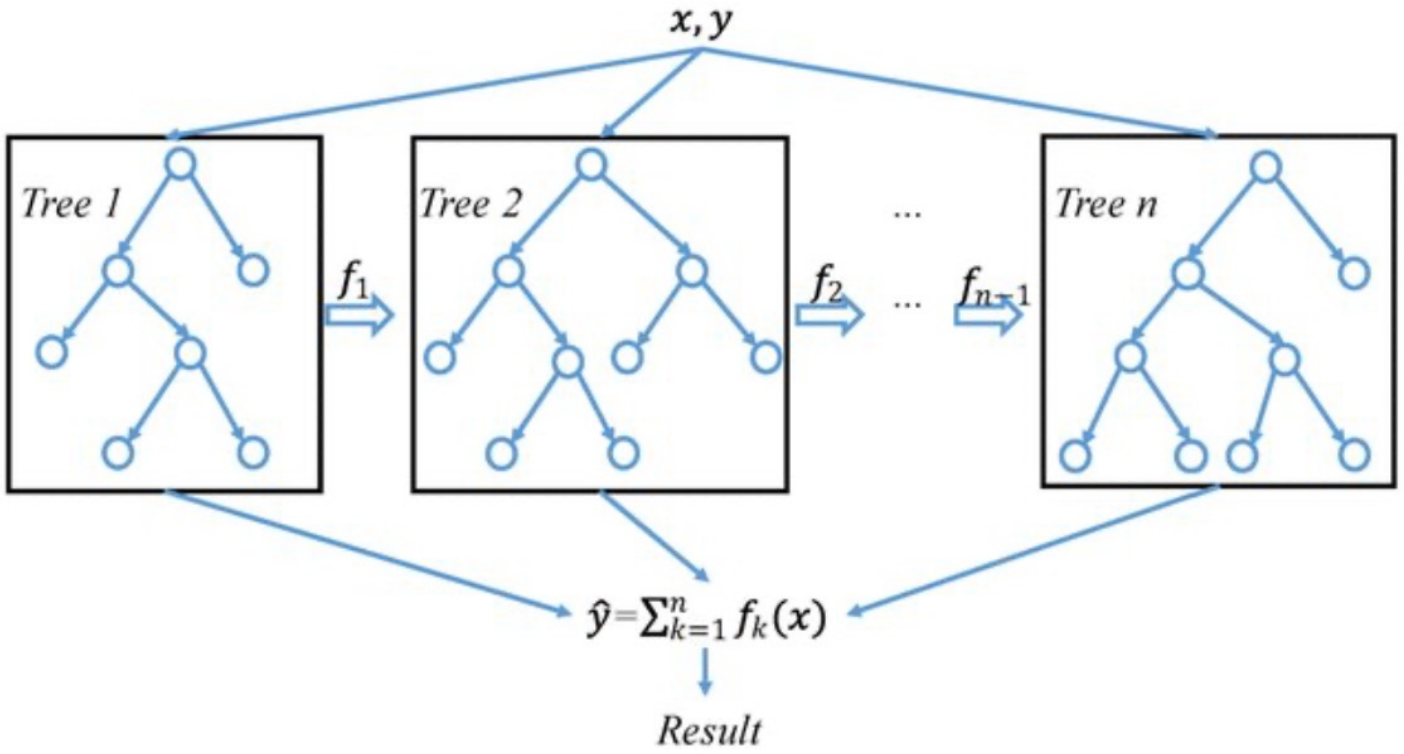


Image source

XGBoost

- **Regularization**: XGBoost includes L1 (Lasso) and L2 (Ridge) regularization to reduce overfitting.
- **Parallel Processing**: It supports parallel processing during tree construction, training, and can distribute data across multiple cores, making it faster than many other gradient boosting implementations.
- **Handling Missing Data**: Automatically learns the best direction to handle missing data.
- **Built-in Cross-Validation**: Supports k-fold cross-validation directly in the training process.
- **Early Stopping**: Allows the training process to stop early if the model performance does not improve after a certain number of iterations.

XGBoost

- import xgboost as xgb
- *class* xgboost.XGBClassifier(*, *objective='binary:logistic'*, ***kwargs*)
- source