# 1 Performance metrics

Performance metrics in machine learning are essential for evaluating the effectiveness of models and understanding their predictive power. Key metrics include accuracy, which measures the proportion of correctly classified instances, and precision, which indicates the proportion of true positive predictions among all positive predictions. Recall, or sensitivity, assesses the model's ability to predict the fraction of positive samples correctly in total positive samples, while the F1 score provides a harmonic mean of precision and recall, offering a single metric that balances both concerns.

For regression tasks, metrics like Mean Squared Error (MSE) and R-squared value are commonly used, where MSE quantifies the average squared difference between actual and predicted values, and R-squared explains the proportion of variance in the dependent variable captured by the model. These metrics are crucial for diagnosing model performance, comparing different models, and ultimately guiding improvements in algorithm selection and tuning.

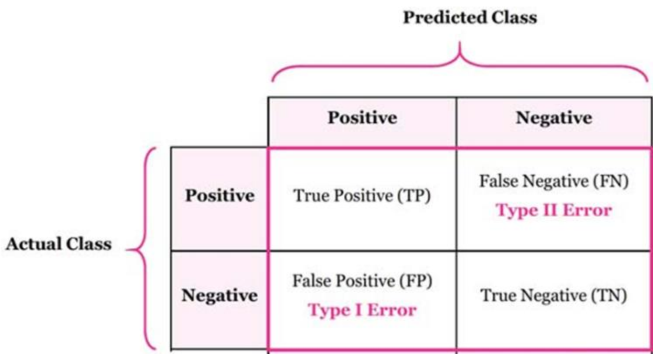
## 1.1 Classification Metrics

**Accuracy**: The simplest performance matrix for classification is Classification Accuracy.

$$Accuracy = \frac{No.of\ Correctly\ classified\ /\ predicteed\ samples}{Total\ No.of\ samples\ in\ the\ dataset}$$

However, accuracy may not exactly predict the behavior of the model. For example, in a binary classification model, the model may correctly classify all the positive samples and misclassification may occur only for negative samples.

**Confusion Matrix**: A more concise way for evaluating a model is confusion matrix. In a confusion matrix, the rows represent the actual classes, while the columns represent the predicted classes. Each cell in the matrix represents the number of instances where the actual class was predicted correctly (true positives/true negatives), incorrectly (false positives/false negatives).



| | | Predicted Class | |
|---|---|---|---|
| | | Positive | Negative |
| Actual Class | Positive | True Positive (TP) | False Negative (FN) Type II Error |
| | Negative | False Positive (FP) Type I Error | True Negative (TN) |

Accuracy can be defined as:

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

**Precision :** Precision gives the fraction of positive samples correctly predicted (True Positives) in total predictions as positive, which includes False Positives also.

$$Precision(P) = \frac{TP}{TP + FP}$$

**Recall :** Recall gives the fraction of positive samples correctly predicted in total positive samples. This is also called as sensitivity or True Positive Rate(TPR).

$$Recall(R) = \frac{TP}{TP + FN}$$

**Precision Vs Recall:** Suppose we want videos of different types to be classified as suitable for children as positive and negative. Negative videos include crime, horror and obscenity. We prefer High Precision because negative videos should not be classified as positive, which means False Positives should be ***zero***. We can afford to have some good videos being classified as negative ( False negatives). Thus, we prefer to have low recall though some positive videos are classified as negative.

A simple example where high recall is more important than precision is in medical diagnostics for a serious disease, such as COVID screening. In this context, recall (sensitivity) is crucial because it measures the proportion of actual positive cases (patients with COVID) that are correctly identified by the model. Missing a COVID diagnosis (a false negative) could have severe consequences for the patient's health and treatment outcomes. Therefore, ensuring that almost all patients with the disease are identified, even if it means some healthy individuals are incorrectly flagged as potentially having COVID (false positives), is prioritized.

**F1 Score** : F1 Score is the harmonic mean of Precision and Recall. Harmonic mean favours towards small value. F1 Score will be high only when both Precision and Recall are high. The maximum value of F1 is 1.

$$F1\ Score = \frac{2PR}{P + R}$$

**Specificity:** Specificity gives the fraction of negative samples correctly predicted in total negative samples. This is also called as True Negative Rate(TNR). This can be regarded as counterpart of recall. *Recall* is for positive samples and *specificity* is for negative samples.

$$Specificity(S) = \frac{TN}{TN + FP}$$

**False Positive Rate(FPR):** FPR gives the fraction of negative samples incorrectly predicted as positive in total negative samples.

$$FPR = \frac{FP}{TN + FP} = 1 - \frac{TN}{TN + FP} = 1 - S$$

**Receiver Operating Characteristic (ROC) :** In many models of classification, the output of the model is the probability with which the sample belongs to class 1 (Positive). The output of such models is between 0 and 1. Thus, the question arises regarding the limiting value of the output probability above which the input sample can be treated as Positive. Normally, the limiting value is 0.5, above which the input sample is regarded as Positive. The idea behind ROC is can we measure performance metrics by varying the limiting value of probability like 0.5, 0.6 and 0.7 likewise.

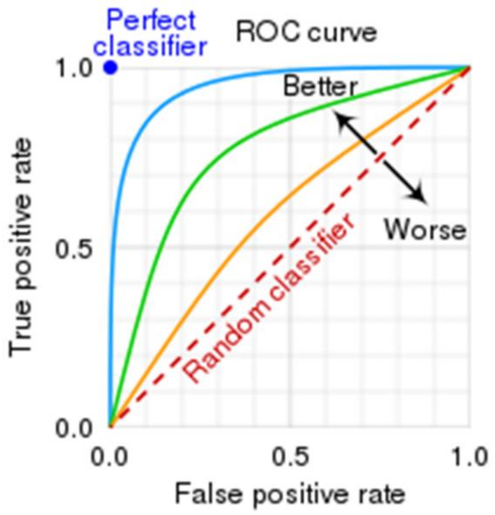
The following figure illustrates the results in case of a dataset which contains 150 positive samples and 1000 negative samples. The threshold probability $p_t$ is varied to have the values as 0.5, 0.7 and 0.9. We see that as the threshold is increased more positive samples are predicted as negative.

| $p_t$=0.5 | Positive | Negavtive |
|---|---|---|
| Positive | 100 | 50 |
| Negative | 30 | 970 |

| $p_t$=0.7 | Positive | Negavtive |
|---|---|---|
| Positive | 90 | 60 |
| Negative | 10 | 990 |

| $p_t$=0.9 | Positive | Negavtive |
|---|---|---|
| Positive | 80 | 70 |
| Negative | 5 | 995 |

For each threshold value of classification, sensitivity (recall) and specificity are calculated. The ROC curve plots sensitivity (recall) versus 1 – specificity. In other words, the ROC curve plots the True Positive Rate (another name for recall) against the 1−$TNR$. 1−$TNR$ is also equal to False Positive Rate (FPR). It can also be said that, it is plotted between TPR and FPR. The following figure shows a sample ROC.



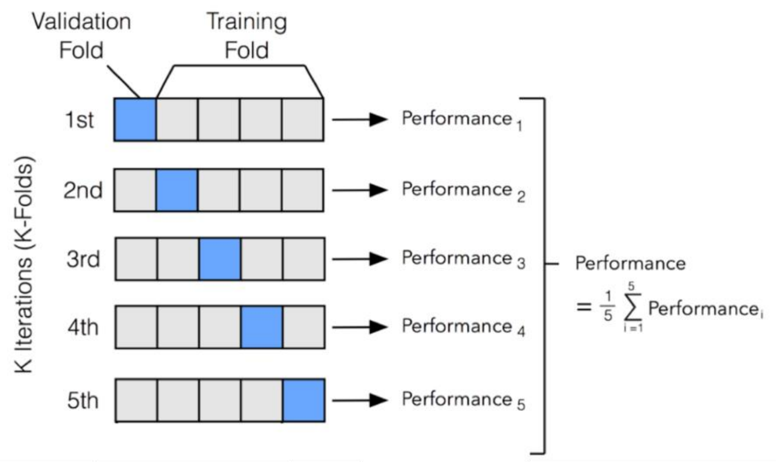
The dotted line represents the ROC curve of a purely random classifier. A good classifier stays as far away from that line as possible (toward the top-left corner). This means that all the samples are predicted correctly as belonging to their calss.

# 2 Cross-Validation

Cross-validation is a statistical technique used in machine learning to evaluate how well a predictive model will generalize to new, unseen data. Its primary goal is to assess the performance of a model and to understand how it might perform on independent datasets. Cross-validation comes in several variants, each with its own strengths and applications. The most common variants are : K-Fold Cross-Validation, Stratified K-Fold Cross-Validation, Leave-One-Out Cross-Validation (*LOOCV*), Leave-p-Out Cross-Validation (*LpOCV*), Repeated K-Fold Cross-Validation, Nested Cross-Validation and Time Series Cross-Validation.

## 2.1 K-Fold Cross-Validation

The dataset is divided into k subsets (folds) of approximately equal size. The model is trained on k-1 folds and tested on the remaining fold. The performance metrics are calculated based on the test results. This process is repeated k times, with each fold used as the testing set exactly once. The final performance metrics are averaged across all the folds. The following figure illustrates the process.



## 2.2 Stratified K-Fold Cross-Validation

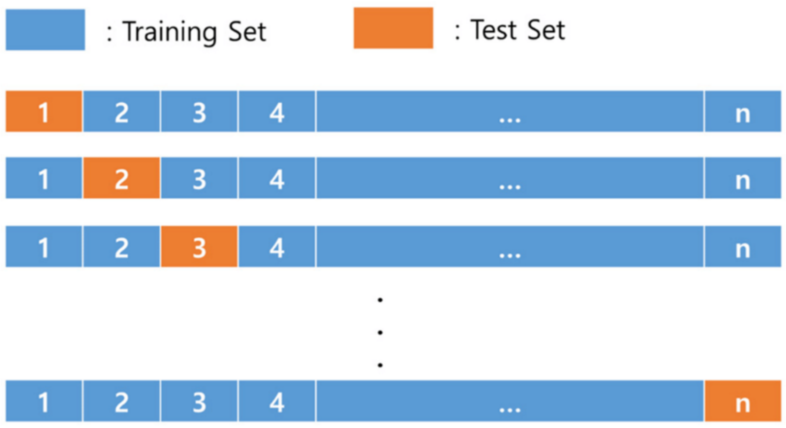
Similar to k-fold cross-validation, but the class distribution in each fold is preserved to mitigate the impact of class imbalance. Suppose we have a dataset with two classes: Class A and Class B. Let the class sizes be: Class A: 800 samples and Class B: 200 samples. Now, let's say we want to perform 5-fold cross-validation with stratification. Here's how we might split the dataset:

**Stratification**: Before splitting, the dataset is divided into two strata based on the class labels. Each stratum contains data points belonging to a particular class. Stratum 1 (Class A): 800 samples. Stratum 2 (Class B): 200 samples

**Folding**: The dataset is divided into 5 folds while maintaining the same class distribution in each fold. Each fold contains: 160 Class A and 40 Class B samples.

### 2.3 Leave-One-Out Cross-Validation (*LOOCV*)

Each data point is used as a validation set, and the model is trained on all other data points. This process is repeated for each data point. *LOOCV* is computationally expensive but provides a low bias estimate of the model's performance, especially for small datasets. The performance metrics are calculated for each fold and finally averaged. The following figure shows the details.



### 2.4 Leave-$p$-Out Cross-Validation (*LpOCV*)

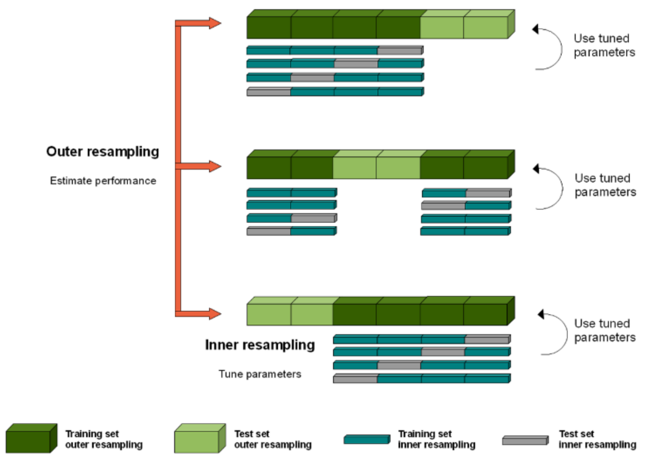
Each time $p$ data points are used as a validation set, and the model is trained on all other data points. This process is repeated for different sets of $p$ data points. *LpOOCV* is computationally expensive but provides a low bias estimate of the model's performance, especially for small datasets. The performance metrics are calculated for each fold and finally averaged.

### 2.5 Repeated K-Fold Cross-Validation

The k-fold cross-validation process is repeated multiple times with different random splits of the data. This helps in obtaining more reliable estimates of the model's performance with small datasets or when the performance is sensitive to the data split.

### 2.6 Nested Cross-Validation

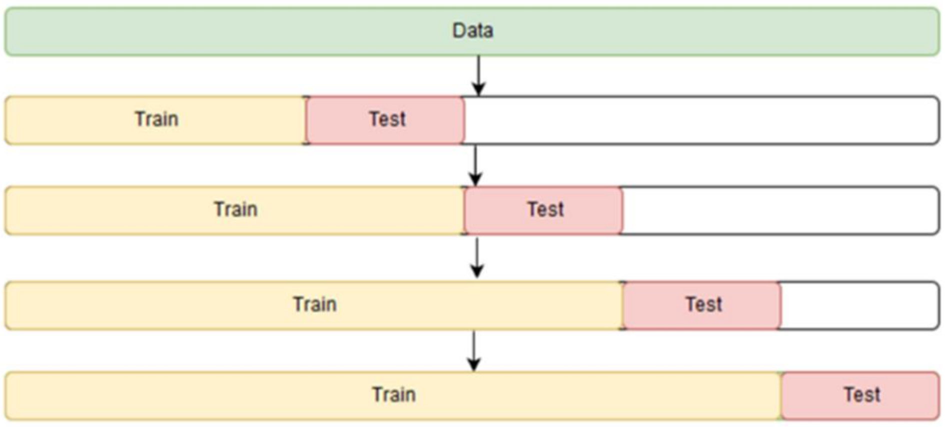
It involves using multiple cross-validation loops to evaluate the performance of nested models, such as models with hyperparameters.



The outer loop performs model evaluation using k-fold cross-validation, while the inner loop performs hyperparameter tuning using another k-fold cross-validation. The following figure ill

## 2.7 Time Series Cross-Validation:

Specifically designed for time series data, where the order of data points is important. It involves splitting the dataset into consecutive folds, ensuring that each fold contains a continuous segment of data. This helps in assessing the model's ability to generalize to future unseen data. The following figure illustrates the same.



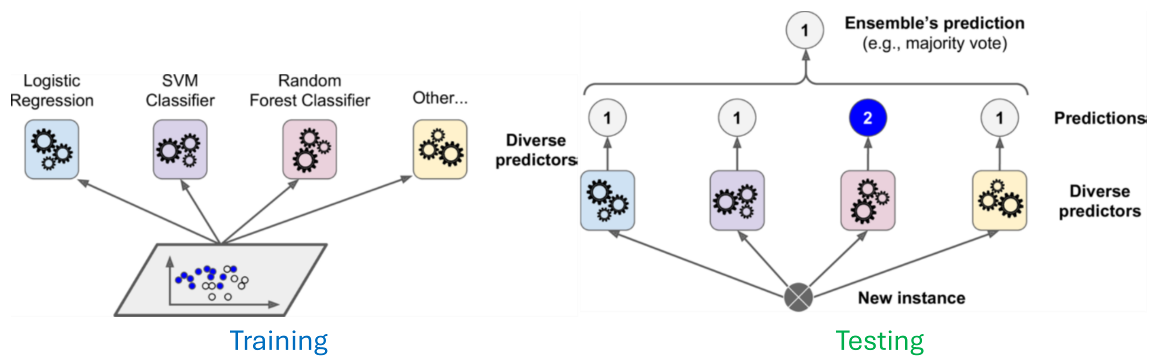
# 3 Ensemble Learning

If we aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an ensemble. This technique is called Ensemble Learning. Ensemble Learning algorithm is called an Ensemble method.

As an example of an Ensemble method, we can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, we obtain the predictions of all the individual trees, then predict the class that gets the most votes. Such an ensemble of Decision Trees is called a Random Forest. The following explains the various ensemble methods.

## 3.1 Voting Classifiers

Suppose we have trained a few classifiers on a dataset, each one achieving about 80% accuracy. They can be a Logistic Regression classifier, an SVM classifier, and a Random Forest classifier. The following diagram shows the principle of voting classifier.



A voting classifier is an ensemble machine learning technique that combines the predictions of multiple individual models (classifiers) to improve overall performance. The fundamental idea is that by aggregating the predictions from various models, the ensemble can achieve better accuracy and robustness compared to any single model. There are two main types of voting classifiers: hard voting and soft voting.

**Hard Voting**: In hard voting, each classifier in the ensemble makes a prediction (vote) for each instance. The final prediction is determined by a majority vote. In other words, the class that receives the most votes from the individual classifiers is chosen as the final prediction.

Example: Suppose you have three classifiers (A, B, and C) and you want to classify a new instance. If classifier A predicts class 0, classifier B predicts class 1, and classifier C predicts class 0, the final prediction by the voting classifier would be class 0 because it has the majority of votes (2 out of 3).

**Soft Voting**: In soft voting, each classifier outputs a probability for each class. The final prediction is made by averaging the predicted probabilities for each class across all classifiers and selecting the class with the highest average probability.

Example: Suppose you have three classifiers (A, B, and C) and they provide the following probability distributions for two classes (class 0 and class 1) for a new instance:

Classifier A: [0.7, 0.3] (70% probability for class 0, 30% for class 1)

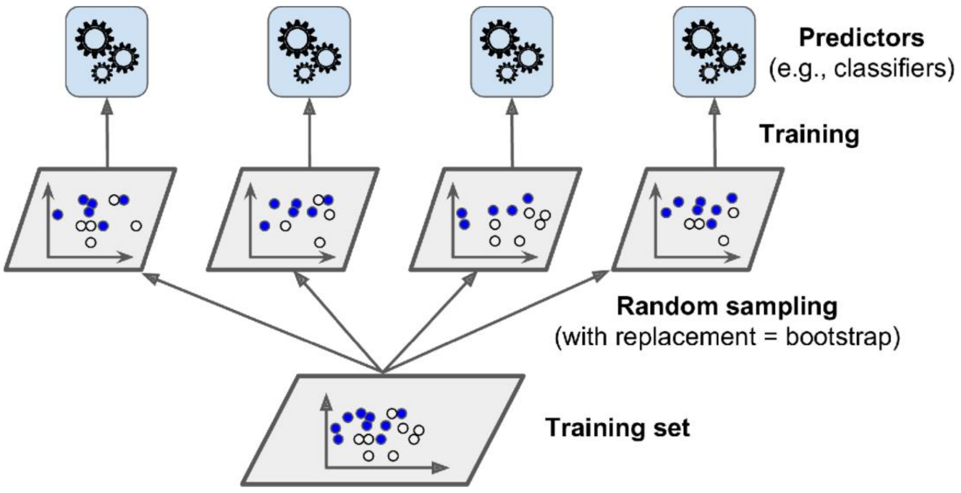
Classifier B: [0.4, 0.6]

Classifier C: [0.8, 0.2]

The average probabilities are: [(0.7+0.4+0.8)/3, (0.3+0.6+0.2)/3] = [0.63, 0.37]. The final prediction would be class 0, as it has the highest average probability (0.63).

## 3.2 Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms. Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sampling is performed with replacement, this method is called **bagging** (short for bootstrap aggregating). When sampling is performed without replacement, it is called **pasting**. The following diagram shows the bagging method.



Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the statistical mode (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. By default, a BaggingClassifier samples $m$ training instances with replacement (bootstrap=True), where $m$ is the size of the training set.

This means that only about 63% of the training instances are sampled on average for each predictor. The remaining 37% of the training instances that are not sampled are called out-of-bag (oob) instances. Note that they are not the same 37% for all predictors. Since a predictor never sees the oob instances during training, it can be evaluated on these oob instances, without the need for a separate validation set. We can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

## 3.3 Random Patches and Random Subspaces

The BaggingClassifier class supports sampling the features as well. Sampling is controlled by two hyperparameters: **max_features** and **bootstrap_features**. Thus, each predictor will be trained on a random subset of the input features. This technique is particularly useful when we are dealing with high-dimensional inputs (such as images). **Sampling both** training instances and features is called the *Random Patches* method. Keeping all training instances but **sampling features** is called the *Random Subspaces* method.

## 3.4 Random Forest

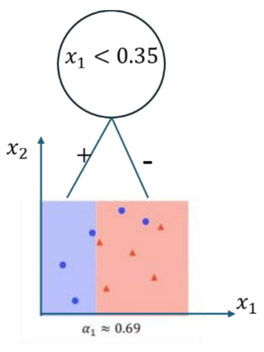
Random forests are a specific implementation of bagging that is applied to decision trees. In addition to sampling from the training data, random forests also introduce randomness in the feature selection process. Instead of considering all features at each split, random forests randomly select a subset of features to consider for splitting at each node of the tree.

This randomness further diversifies the individual trees and prevents them from being highly correlated, which can lead to improved generalization performance. When making predictions, random forests aggregate the predictions of all the trees, typically using a majority vote for classification or averaging for regression.

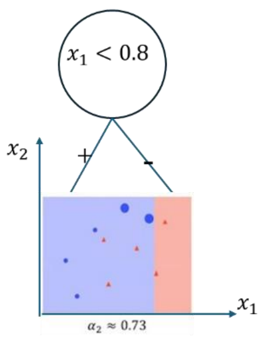
## 3.5 Boosting.

Boosting (originally called hypothesis boosting) refers to any Ensemble method that can combine several weak base learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. The two popular algorithms are **AdaBoost** and **Gradient Boosting**.

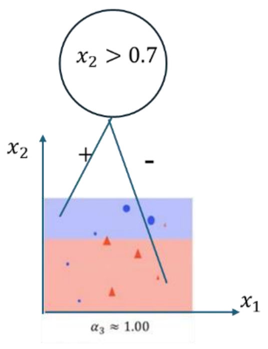
**AdaBoost**: Consider an AdaBoost classifier, which uses Decision Tree as a base classifier. Each sample is given a weight. These weighted samples are used to train a **Decision tree stump**. The resulting Decision Tree is used to make predictions on the training set. Depending on the no. of training samples correctly/incorrectly classified and their weights, the weighted error of this weak classifier is calculated. Consider the following example, where three Decision Tree stumps are trained. The dataset consists of positive (circles) and negative(triangle) samples. Each has its own classification errors. The first two trees used the feature $x_1$ at the root node. Each tree has it classification error $\alpha$.



Decision Tree - 1



Decision Tree - 2



Decision Tree - 3

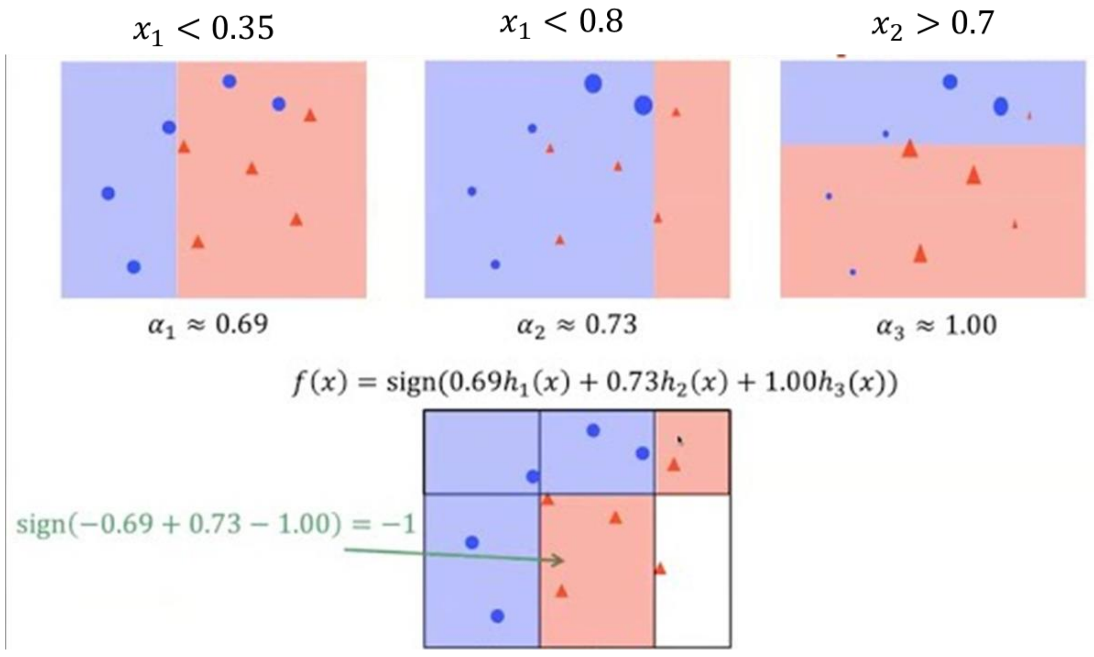
The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, calculates the weighted error of the second classifier and updates the instance weights, and so on. A new instance is classified based on the sum of predictions of each weak classifier multiplied by the weighted error of the respective weak leaner.

Let us combine these three classifiers for prediction. The resulting classifier looks as follows. The prediction is made as follows. Consider any sample and input it to each of the classfiers. The prediction is based on the sign of the following sum. $\alpha_c$ indicates the error of a particular classifier and $h_c(x)$ is the prediction of the corresponding classifier as +1(blue region) or -1(pink region).

$$S = \sum_{c=1}^{3} \alpha_c * h_c(x)$$

For the example shown, the three classifiers output -1, +1, -1 respectively. These values are multiplied with the corresponding classification errors. Thus, $S$ as shown below has negative sign and hence the sample is negative which conforms with the given label in the dataset.

$$S = -1 * 0.69 + 1 * 0.73 + -1 * 1.00$$



**Gradient Boosting:** Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor. Consider a simple regression example, using Decision Trees as the base predictors. This is called Gradient Tree Boosting, or Gradient Boosted Regression Trees (GBRT).

The following diagram shows the principle of Gradient Boosting. First, we fit a ***DecisionTreeRegressor*** to the training set (Learner-1). Then the residual error (RE) is calculated between the target value and the predicted values. The residual error is taken as the target value for the same feature set and then a new tree (Learner-2) is trained on the residual errors of the first tree. Similarly, another tree is trained on the residual errors of the second tree.

The prediction is done by giving the test sample as the input to all the learners and sum up the predictions of all the learners to get the final predicted value. An efficient implementation version of Gradient Boosting is known as **XGBoost**.

Learner-1

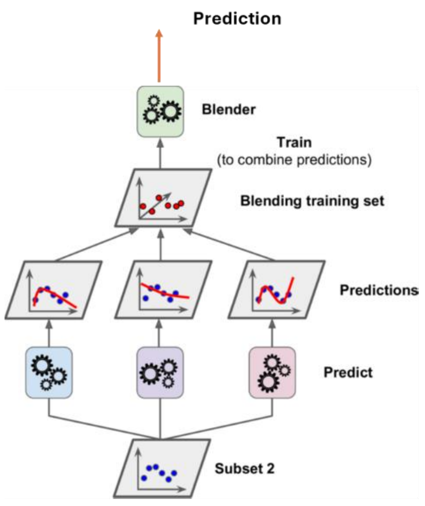
| $x_1$ | $x_2$ | $y$ | $\hat{y}$ | $RE$ |
|---|---|---|---|---|
| 1 | 3 | 8 | 6 | 2 |
| 2 | 1 | 10 | 15 | -5 |
| 4 | 2 | 12 | 9 | 3 |

Learner-2

| $x_1$ | $x_2$ | $\hat{y} = RE$ |
|---|---|---|
| 1 | 3 | 2 |
| 2 | 1 | -5 |
| 4 | 2 | 3 |

## 3.6 Stacking

In this ensemble method, instead of aggregating/voting technique used in predictions from an ensemble, another meta model is trained with Predictions of the individual predictors as the features along with the actual target value. Once the training is over, the target value for the new sample can be obtained in two steps. Initially, the predicted values are obtained for the test sample from the individual predictors and these are input to the blender to get the target value.


Prediction
Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2