Ensemble Learning

By Waad ALMASRI

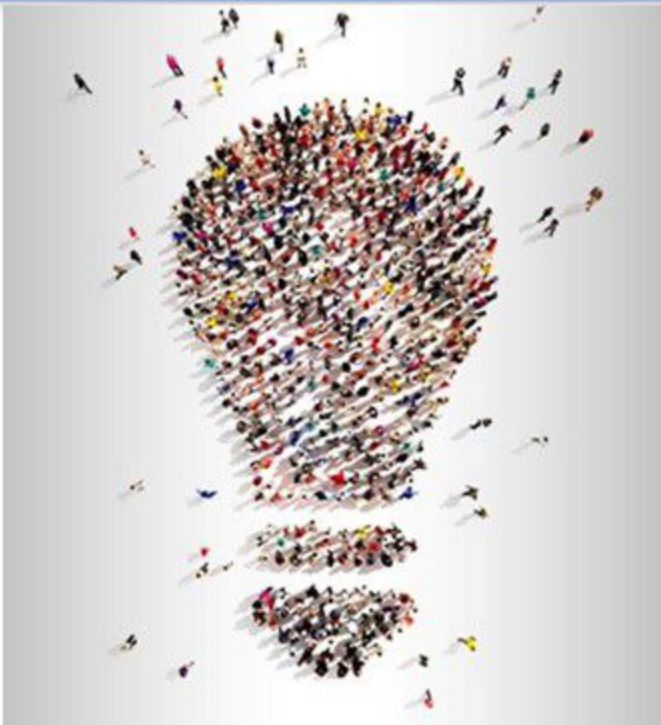
Introduction

The wisdom of the crowd

Aggregating the predictions of a group of predictors (classifiers or regressors) will often yield to better predictions than with the best individual predictor.

A group of predictors is called an ensemble.

Ensemble methods are usually used near the end of a project, after building several good predictors. The predictors are combined to an even better predictor.



Topmost common ensemble method

- Voting classifier
- Bagging & Pasting
- Random Forest
- Boosting
- Stacking

Voting Classifiers

Voting Classifier

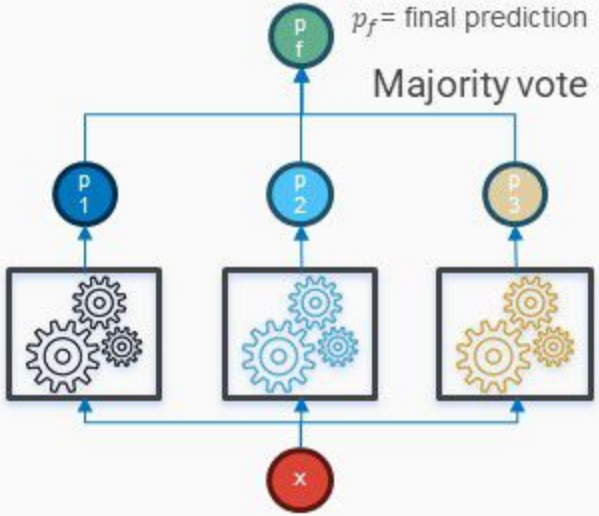
It consists of training several classifiers (e.g., a logistic regression, an SVM, a random forest, etc.,) and aggregating the predictions of each of these classifiers; the majority class is the ensemble's prediction.

This majority-vote classifier is called a *hard voting classifier.*

As a matter of fact, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble.

Moreover, even if each classifier is a *weak learner* (i.e., it is slightly better than random guessing), the ensemble can still be a *string learner* (i.e., achieving high accuracy) if there are a sufficient number of weak learners that are diverse. This is due to the law of large numbers.

If we have an ensemble of 1000 classifiers, each with a 51% of accuracy, we can obtain 75% of accuracy if and only if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data; they are likely to make the same types of errors, so there will be many votes for the wrong class, reducing the ensemble's accuracy. One solution would be to use diverse classifiers, which make different types of errors, to improve the ensemble's accuracy.



Voting classifier

Scikit-Learn: VotingClassifier, it has 4 important attributes: *estimators* : original classifiers, *estimators_* : the fitted/trained predictors, *named_estimators*: the original predictors with their names, and *named_estimators_*: the trained predictors with their names.

NB: if all classifiers are able to estimate class probabilities (i.e., they all have the *predict_proba()* method), then the predicted class will be the one with the highest probability. This is *soft voting*. To enable soft voting, set the *voting* hyperparameter of VotingClassifier to *"soft"* .

Soft Voting achieves higher performance than hard voting because it gives more weight to highly confident votes (i.e., class probabilities close to 1.0).

Law of large numbers

```
from scipy.stats import binom

n = 1000 # tosses
p = 0.51 # probability of head (coin is biased)
x = 501 # to get majority of heads out of 1000 tosses I need just 501
print(binom.pmf(x, n, p)) # scipy library
print(1 - binom.cdf(x-1, n, p))
n = 10000 # tosses
p = 0.51 # probability of head (coin is biased)
x = 5001 # to get majority of heads out of 10000 tosses I need just 5001
print(binom.pmf(x, n, p)) # scipy library
# You got the probability of exactly 501/1000 heads and then exactly 5001/10000 heads rather than at least that many.
# ut to this you need to add probability of 502/100, 503/100, ... 1000/1000 (which is highly improbable), which is
equivalent to using the cumulative distribution function or its complement: 1 - binom.cdf(x-1, n, p)
print(1 - binom.cdf(x-1, n, p))
```

Cumulative distribution function [edit]

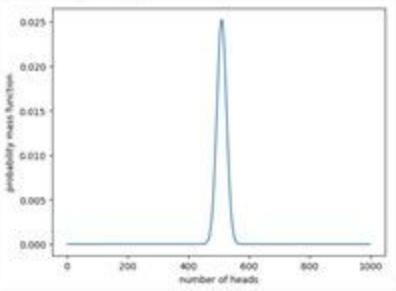
The cumulative distribution function can be expressed as:

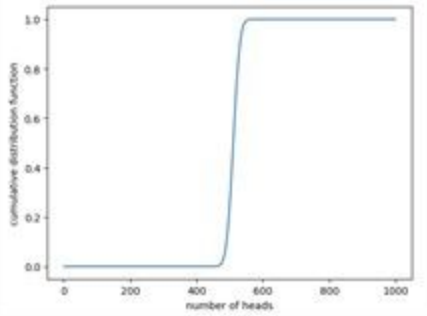
$$F(k; n, p) = \Pr(X \leq k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i},$$

where $\lfloor k \rfloor$ is the "floor" under k, i.e. the greatest integer less than or equal to k.

Law of large numbers

```
import numpy as np, matplotlib.pyplot as plt
n = 1000 # tosses
p = 0.51 # probability of head (coin is biased)
X = np.arange(1, n)
y = [binom.cdf(x-1, n, p) for x in X]
y2 = [binom.pmf(x,n,p) for x in X]
plt.plot(X,y)
plt.plot(X,y2)
sum(y2[500:])# this includes the pmf of the times where the number of heads
were 501, 502, …, 1000, i.e., everytime the heads were the majority.
>>> 0.7467502275563255
```





Bagging & Pasting

Bagging & Pasting

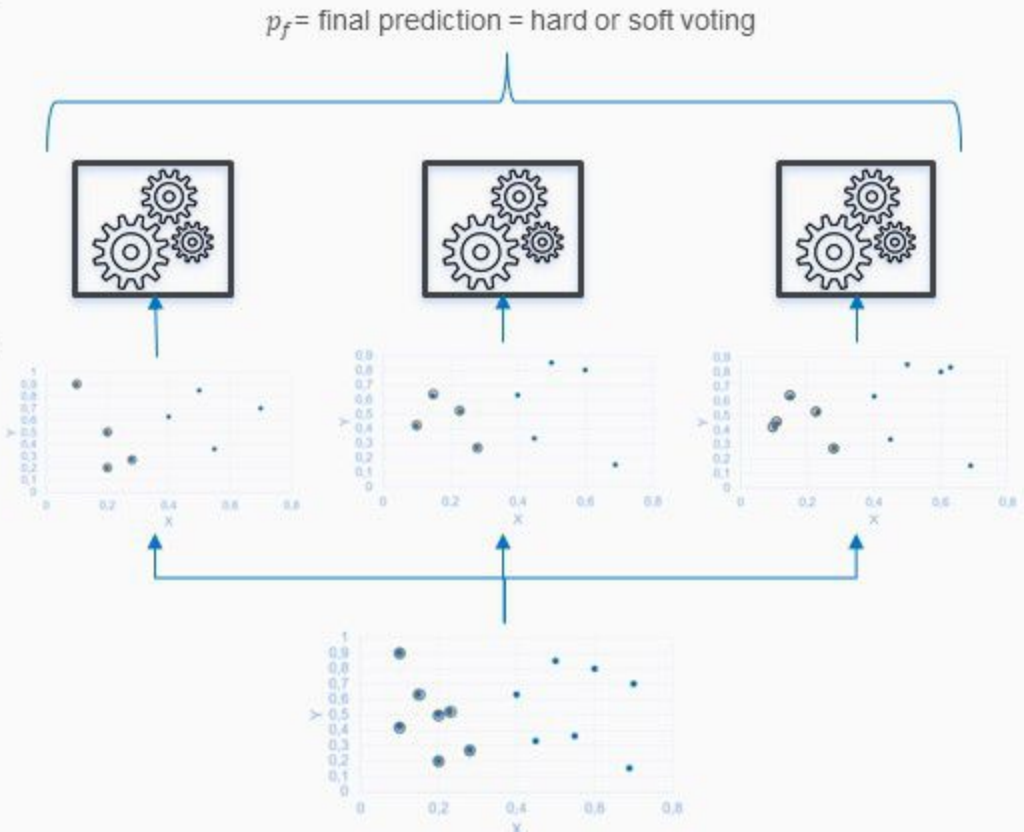
Instead of using different set of algorithms like in Voting Classifiers, we can use the same training algorithm but train them on different random subsets of the training set.

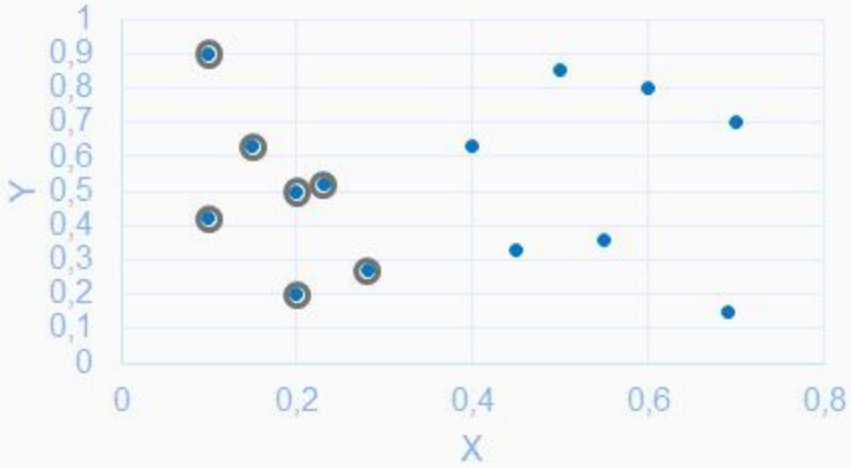
- Bootstrap aggregating or Bagging: is when sampling with replacement; training instances can be samples several times across multiple predictors.
- Pasting: is when sampling without replacement.

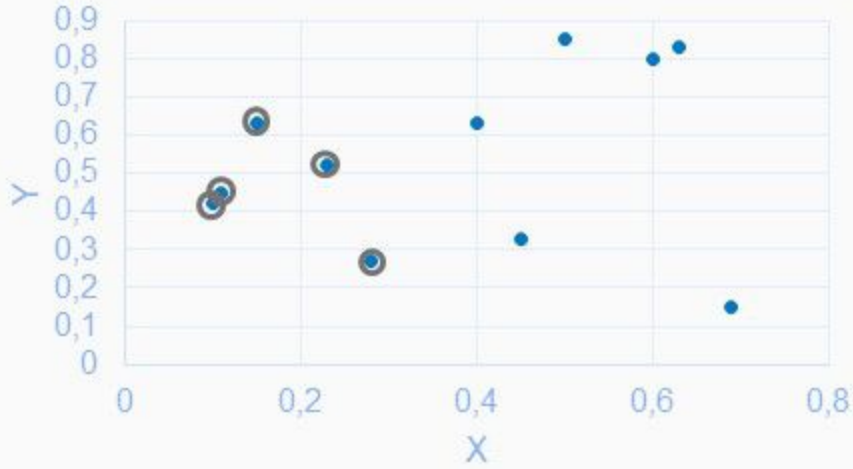
Once all predictors are trained, the ensemble can make a prediction for a new instance by aggregating the predictions of all predictors.

For classification, the aggregation function is the statistical *mode* (i.e., the majority class).

For regression, the aggregation function is the average.




1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Y
0
0,2
0,4
0,6
0,8
X


0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Y
0
0,2
0,4
0,6
0,8
X

Pros & cons

Each individual predictor has a higher bias than if it were trained on the original training set. However, the aggregation of all the predictors reduces both bias and variance.

Generally, the result of the ensemble model has a similar bias but a lower variance than a single predictor trained on the original training set; it makes roughly the same errors on the training set, but the decision boundary is less irregular.

Advantage of Bagging/Pasting: they scale very well in training & inference times; all predictors can be trained in parallel, via different CPU cores or even different servers. Moreover, predictions are output in parallel.

Bagging vs Pasting:
Bagging introduces more diversity in the subsets that each predictor is trained on → Bagging ends up with a slightly higher bias than pasting. However, this additional diversity results in less correlated predictors → Bagging achieves a lower variance than pasting i.e., less overfitting. Bagging usually gives better results, which explains its popularity, unlike Pasting.

Out-of-Bag Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all.

By default a BaggingClassifier samples n training instances with replacement where n is the size of training set.

On average, about 63% of the training instances are sampled for each predictor.

The remaining 37% that are not sampled are called out-of-bag (OOB) instances. NB: they are not the same 37% for all predictors.

TThe core idea of OOB-evaluation is as follows:

- To evaluate the random forest on the training set.
- For each example, only use the decision trees that did not see the example during training.

Thus, we can evaluate a bagging ensemble model using these OOB instances without needing to create a separate validation set.

If there are enough estimators, then each instance in the training set will likely be an OOB instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance.

Once you have a prediction for each instance, you can compute the ensemble's prediction accuracy.

Random patches & random subspaces

Random patches is when we sample not only training instances but also the features (i.e., columns, i.e., X). This is beneficial when we are dealing with high-dimensional inputs; it accelerates the training.

Random subspaces is when we keep all training instances but we sample the features.

Sampling features results in more diverse predictors → a slightly higher bias but a lower variance.

Bagging & Pasting

Scikit-Learn: BaggingClassifier & BaggingRegressor

To choose Bagging, set the hyperparameter *bootstrap* to True and vice versa for Pasting.

The *max_samples* hyperparameter is the maximum number of training instances randomly sampled from the training set.

To use OOB evaluation, you should set the *oob_score* hyperparameter to True. NB: the OOB decision function for each training instance is available through the *oob_decision_function_* attribute.

For random patches, we set the hyperparameters: *bootstrap* to True, *bootstrap_featutres* to True, and/or *max_features* to a value less than 1.0.

For random subspaces, we set the hyperparameters: *bootstrap* to False, *bootstrap_featutres* to True, and/or *max_features* to a value less than 1.0.

To parallelize training & inference on several CPUs, the *n_job* hyperparameter needs to be changed; *n_job=-1 is equivalent to use all available cores.*

Random Forest

Random forest

Random forest = ensemble of decision trees trained via the bagging (or pasting method if the *max_samples* = the size of the training set).

It is equivalent to using a BaggingClassifier with a DecisionTreeClassifier as base estimator. However, RandomForestClassifier is more optimized for decision trees.

Random forest model encapsulates all decision trees and bagging hyperparameters.

An additional characteristic of random forest is when growing trees, instead of searching for the best feature when splitting a node, it searches for the best feature among a random subset of features. By default, it samples $\sqrt{m}$ features with m being the total number of features.

This results in more tree diversity → higher bias and a lower variance.

Extra-Trees

An ensemble method that instead of searching for the best possible thresholds for each feature at every node, it uses random thresholds. This could be done by setting the DecisionTreeClassifier's hyperparameter *splitter* to *"random"* .

A forest of such random trees is called *extremely randomized trees or extra-trees.*

Similarly to all ensemble methods, the method trades more bias for a lower variance.

Their training is much faster than random forests because finding the best threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

Scikit-Learn: ExtraTreesClassifier & ExtraTreesRegressor.

Which is better Extra-Trees or RandomForest? And Why?

Feature importance

With random forest, we could measure the relative importance of each feature.

Scikit-Learn measures a feature importance by looking at how much the tree nodes using this feature reduce impurity on average, across all trees in the forest. It is a weighted average; the node's weight is equal to the number of training samples that are associated with it.

Scikit-Learn computes features importances after the training for each feature; they are then scaled so the sum of the features importances of all features is equal to 1.0. It is the randomforest model's attribute *feature_importances_*.

→ this characteristic makes random forests particularly interesting as a feature selection method.

Boosting

Introduction

- Boosting or hypothesis boosting
- Ensemble methods combining several weak learners into a strong learner.
- It trains predictors sequentially; each predictor tries to correct its predecessor.
- AdaBoost, Gradient Boosting, Histogram-Based Gradient Boosting, XGBoost, CatBoost, LightGBM

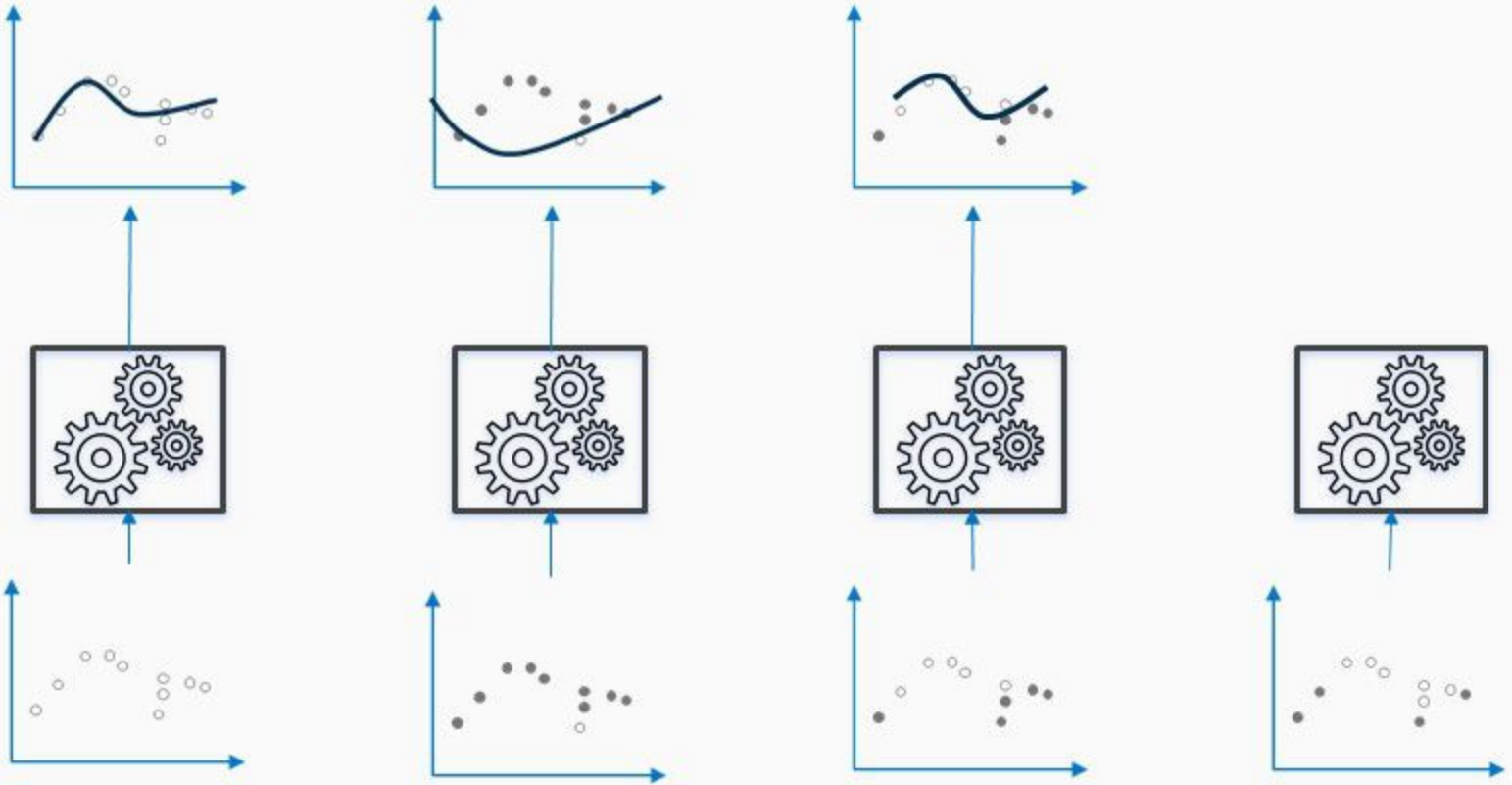
AdaBoost - Adaptive Boosting

AdaBoost is built on the idea that for the ensemble model to be accurate, every new predictor needs to pay more attention to the training instances that the predecessor predictor under-fitted.

AdaBoost Training is as follows:

- The first predictor is trained and used to make predictions on the training set.
- AdaBoost increases the relative weight of misclassified training instances.
- A second predictor is trained, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on.

AdaBoost Sequential Training with instance weight updates

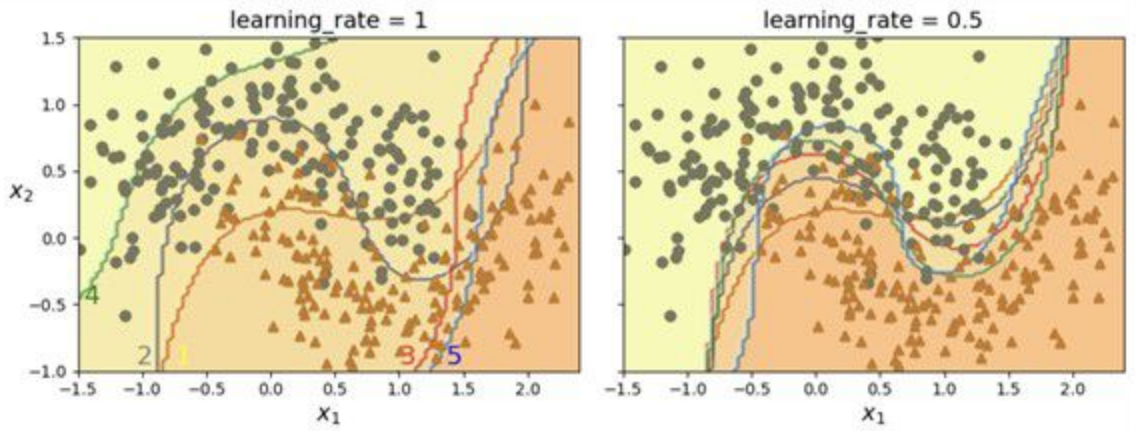


Training Algorithm of AdaBoost

1. The first predictor is trained with each instance weight w_i set to 1/n; n being the total number of training instances.
2. Then, this predictor is used to predict on the training set.
3. Afterwards, the weighted error r_1 is computed on these training predictions:
   a. $r_j = \sum_{i=1, \widehat{y_{i,j}} \neq y_i}^{n} w_i$
   b. where $j = 1$ (the first predictor)
   c. $\widehat{y_{i,j}}$ is the prediction for the i th instance predicted by the j th predictor
   d. y_i is the true target value
4. Then, we compute the predictor's weight α_j, here α_1:
   a. $\alpha_j = \eta . \log(\frac{1-r_j}{r_j})$
   b. With η = the learning rate hyperparameter
   c. The more accurate the predictor is, the higher its weight (α_j) will be.
   d. If the predictor is worse than random guessing, its weight will be negative.
5. Next, the weights of the instances (that were initially set to $1/n$) are updated in a way to give more weight for the misclassified instances:
   a. For $i = 1,2,..n$:
      i. $w_i = w_i$ if $\widehat{y_{i,j}} = y_i$
      ii. $w_i = w_i . \exp(\alpha_j)$ if $\widehat{y_{i,j}} \neq y_i$
      iii. Then all the instances weights are normalized (i.e., divided by $\sum_{i=1}^{n} w_i$; w_i are the new updated weights.
6. Finally, a new predictor is the trained using the updated weights, and the steps 2 to 5 are repeated.
7. The algorithm stops when the desired number of predictors is reached or when a perfect predictor is reached.

Example

This figure shows the decision boundaries of 5 consecutive predictors on the moons dataset.



AdaBoost - Inference

AdaBoost's prediction is the majority of the weighted votes coming from the predictors:

$\hat{y}_i = argmax_k \sum_{j=1, \hat{y}_j(x)=k}^{N} \alpha_j$ where N = the number of predictors.

Scikit-Learn:

- AdaBoostClassifier (for binary classification)
- SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function) for multiclass classification
- If the predictors have the predict_proba method, Scikit-Learn uses SAMME.R (SAMME Real), which uses class probabilities rather than predictions and generally performs better.

Gradient Boosting

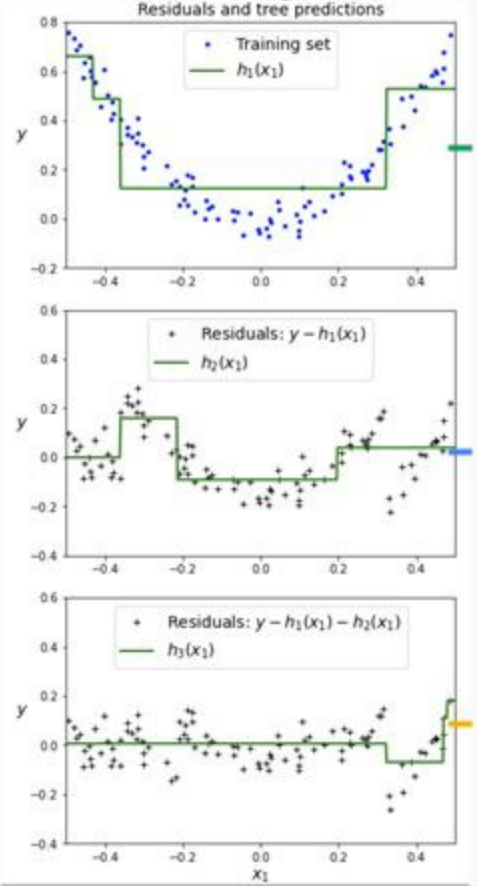
Gradient Boosting is similar to AdaBoost in the sequential training of its predictors.

However, instead of updating the weights of the instances at every iteration like AdaBoost does, it tries to fit the new predictor on the residual errors made by the previous predictor.

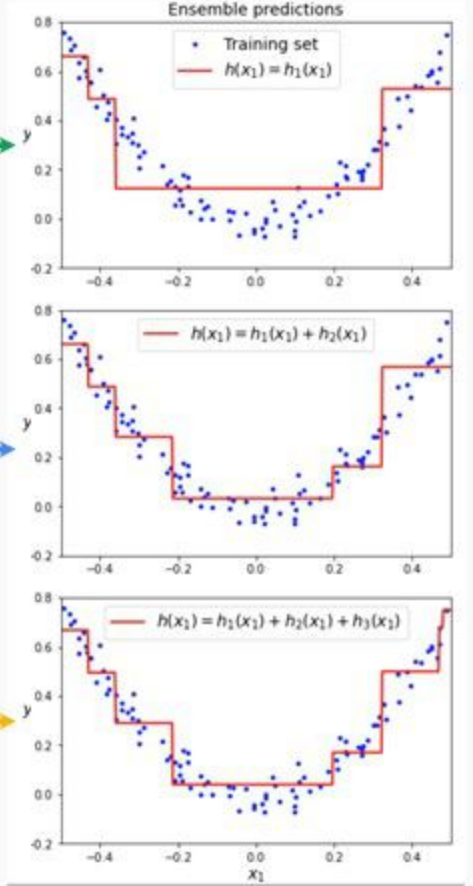
When the decision tree is the base estimator for the gradient boosting, the model is called Gradient Tree Boosting or Gradient Boosted Regression Tree GBRT

Gradient Boosting - Training vs Inference

Training



Inference



The ensemble predictions (on the right) are exactly the same because this the first tree in the ensemble (plot on the left).

The ensemble predictions are equal to the sum of the predictions from the first predictor and the second one trained on the residuals of the first predictor.

The ensemble predictions are equal to the sum of the predictions from the first predictor, the second, and the third one. ***The ensemble's performance gradually improves as more predictors are added***

Hyperparameters

Scikit-Learn: GradientBoostingClassifier & GradientBoostingRegressor (decision tree is the base estimator)

Hyperparameters are: tree-like hyperparameters: max_depth, min_samples_leaf, etc., and ensemble-like hyperparameters such as n_estimators.

Other hyperparameters worth mentioning:

- Learning_rate: it scales the contribution if each tree. If it is set too low, more trees are needed to fit the training set, but a better generalization is guaranteed. This regularization is called shrinkage.
- n_iter_no_change: the model automatically stops adding more trees during training if the last *n_iter_no_change* trees did not help. This is an early stopping technique that tolerates no progress for a few iterations (n_iter_no_change iterations) before it stops. Setting the value of n_iter_no_change too low can result in underfitting. Reversely, setting it too high provokes overfitting.
- Subsample: the percentage of training instances sampled randomly and used for training each tree. Subsampling trades a higher bias for a lower variance and accelerates the training. When subsample is used, the model is called *stochastic gradient boosting*.

Histogram-based Gradient Boosting

- It is an optimized implementation of gradient boosting for large datasets.
- The input features (X) are binned and replaced with integers, with the max number of bins per feature equals to 255=max_bin hyperparameter.The advantages of this are:
  - Binning a feature reduces the number of possible thresholds that the training algorithm needs to evaluate.
  - Working with integers allows to benefit from faster and more memory-efficient data structures.
  - Sorting the features when training each tree is no longer required because of the way bin-integers are build.
  - The computational complexity is reduced from $\Theta(n.m.\log_2(n))\ to\ \Theta(n.b)$ where n is the number of training instances, m is the number of features, and b is the number of binned features (bins). Thus, HGB trains hundreds of times faster than regular gradient boosting on large datasets. However, it trades this with a precision loss.
  - Binning acts like a regularizer; depending on the dataset, this can reduce overfitting or result in underfitting.
- Scikit-Learn: HistGradientBoostingClassifier & HistGradientBoostingRegressor

Pros & cons

Training AdaBoost/Gradient Boosting cannot be parallelized because of the sequential training; each predictor can only be trained after the previous predictor got trained & evaluated on the training set. Both do not scale well like bagging/pasting. → Histogram-based Gradient Boosting scales better but has a lower precision.

Prone to overfitting the training set→ reducing the number of predictors or regularizing the base predictor could reduce this effect.

To find the best number of predictors, cross-validation could be performed.

Histogram-based Gradient Boosting has additional characteristics: it supports v=categorical features (they must be represented as integers and cannot be higher than max_bins (255 is the max) and missing values → preprocessing of data is simplified.

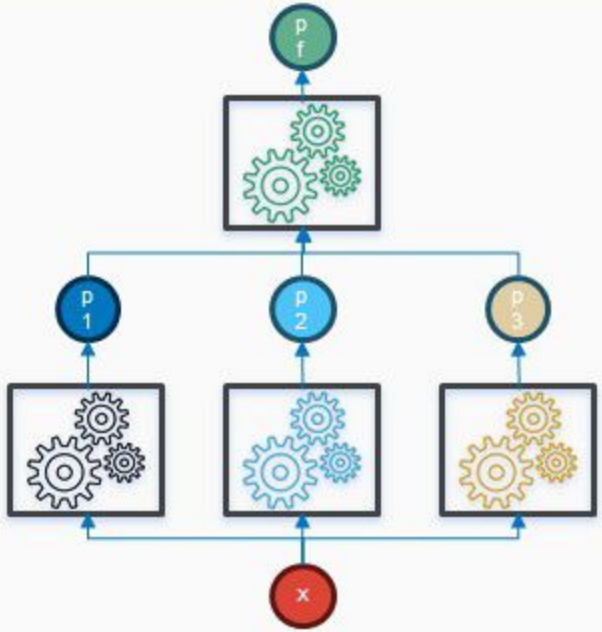
Stacking

Stacked Generalization or Stacking

Intuition behind: instead of using functions like hard voting to aggregate the predictions of all the predictors in the ensemble, let us train a model to perform this aggregation; this aggregator model is called blender or meta learner.

- Input: data point x
- First layer: x is input to the ensemble predictors, here 3 models.
- Output: 3 predictions (p1, p2, p3)
- These predictions are then input to the blender to output the final prediction pf.

Scikit-Learn: StackingRegressor, StackingClassifier



Stacking - How do we train the blender?

1. Build the blending training dataset:
    1. Using the cross_val_predict() on every predictor in the ensemble, we can get out-of-sample predictions for each instance in the original training set
    2. Targets are the one in the original dataset
    3. The blending training dataset consists of new features = the number of the predictors (in the previous slide, 3)
2. Train the blender with this dataset
3. Once the blender is trained, retrain the base predictors on the full original training set.

NB: one could train several blenders (linear blender, random forest blender, etc.,) to get an additional layer of blenders and finally train a final blender on top. This will will boost performance but will cost in both training time & system complexity.

Last slide put the schemas of figures 7.17.4, etc. to show the different ensemble types.

References

Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.

https://developers.google.com/machine-learning/decision-forests/out-of-bag