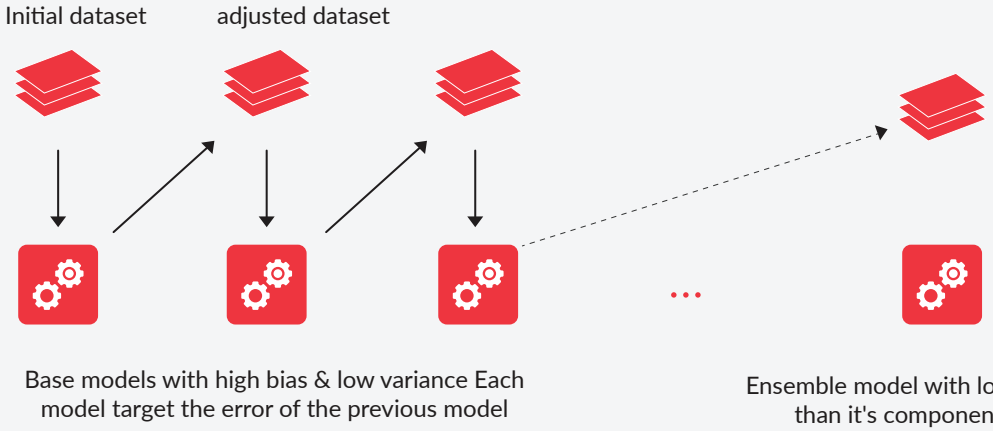
Boosting

Boosting was first introduced in 1997 by Freund and Schapire in the popular algorithm, AdaBoost. It was originally designed for classification problems. Since its inception, many new boosting algorithms have been developed those tackle regression problems also and have become famous as they are used in the top solutions of many Kaggle competitions.

Common Interview Questions

1. What is Gradient Boosting, and How Would You Define it?
2. How Does Gradient Boosting Algorithm Work?
3. What is the Reason Behind Not Using Decision Stumps as Algorithms for Gradient Boosting?
4. What are the Assumptions of the Gradient Boosting Algorithm?
5. How Can You Improve the Performance of the Gradient Boosting Algorithm?
6. What are the Advantages and Disadvantages of Using the Gradient Boosting Algorithm?
7. How is Gradient Boosting Different From XGBoost?
8. Why does Gradient Boosting perform so well?

A **weak learning algorithm** produces a model that does marginally better than a random guess. A random guess has a 50% chance of being right. Hence, any such model created by the weak learning algorithm shall have, say 60-70% chance of being correct.



Boosting

Introduction-

- An ensemble is a collection of models which ideally should predict better than individual models.
- The key idea of boosting is to create an ensemble which makes high errors only on the less frequent data points.
- Boosting leverages the fact that we can build a series of models specifically targeted at the data points which have been incorrectly predicted by the other models in the ensemble. If a series of models keep reducing the average error, we will have an ensemble having extremely high accuracy.
-Boosting is a way of generating a strong model from a weak learning algorithm.

There are essentially two steps involved in the AdaBoost algorithm:

1. Modify the current distribution to create a new distribution to generate a new model.
2. Calculation of the weights given to each of the models to get the final ensemble.

Boosting Algorithms

Pseudo Code for Adaboost Algorithm:

1. *Initialize the probabilities of the distribution as* *where n is the number of data points*
2. *For t = 0 to T, repeat the following (T is the total number of trees):*
    1. *Fit a tree h on the training data using the respective probabilities*
    2. *Compute* $\epsilon_t = \sum_i^n D_i[h_t(x_i) \neq y_i]$
    3. *Compute* $\alpha_t = \frac{1}{n} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
    4. *Update* $D_{t+1}(i) = \frac{D_t(i) * e^{-\alpha_i y_i h_t(x_i)}}{z_t}$ *where,* $z_t = \sum_{i=1}^n D_i * e^{-\alpha_i y_i h_t(x_i)}$

In other words, every subsequent model we build after changing the distribution has a misclassification rate, here the error, Єt < 0.5. With this in mind, we can see that the αt > 0 as the error Єt < 0.5, ((1- Єt)/ Єt) = positive and the ln(positive) > 0.

We continue this iteration until :

- Low training error is achieved
- A preset number of weak learners have been added We then make the final prediction by adding up the weighted prediction of every classifier.

$$H(x)=\text{sign}(\sum Tt=i\alpha tht(x))$$

Some of the ways to remove outliers are:

- Boxplots
- Cook's distance
- Z-score

Gradient Boosting Algorithm:

Gradient Boosting like AdaBoost trains many models in a gradual, additive, and sequential manner. But the major difference between the two is how they identify & handle the shortcomings of weak learners through loss functions.

To summarize here are the broader points on how does a GBM learn:

- We build the first weak learner using a sample from the training data; we will consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak i.e. still not be fully grown.
- Then the predictions are made on the training data using the decision tree just built.
- The gradient, in our case the residuals are computed and these residuals are the new response or target values for the next weak learner.
- A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
- Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.
- The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a prespecified number of weak learners or if the model starts overfitting i.e. it starts to capture the niche patterns of the training data.
- GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate)

What is PCA

Pseudo Code for Adaboost Algorithm:

At any iteration t, we repeat the following steps in the Gradient Boosting scheme of things:

1. *Initialize a crude initial function Fo as* ***argmin*** $\Sigma_{t-i}^{T} L(y_i, \hat{y})$
2. *For m = 1 to M (where M is the number of trees)*
1. *Calculate the pseudo-residuals* $r_{im} = \frac{(\partial L(y_i, F(x_i))}{\partial F(x_i)}$, *where* $F(x_i) = F_{m-1}(x_i)$,

   the pseudo residuals are the negative gradients for all data points
2. *Fit a base learner* $h_m(x)$ *to the pseudo-residuals, ie train it using the training set*

   $\Sigma_{i=1}^{n} (x_i, r_{im})$. *Here the pseudo residuals are used as the response variable.*
3. *Compute the step magnitude multiplier* γ_m *(in case of tree models, compute a* γ_m *different for every leaf/prediction)*

   $\gamma_m = argmin \Sigma_{i=1}^{n} L(y_i, (F_{m-1} + \gamma * + h_m(x_i))$
4. *Compute the next model* $F_m = F_{m-1}(x_i) + \gamma_m * h_m(x_i)$
3. *The final model is* $F_M(x)$

We see that the loss we get after fitting the model Fm is L(yi, Fm). In order to reduce this loss, we generate models Fm by adding an incremental model hm(xi) to Fm-1.

In other words, we select the hm such that L(yi,Fm-1) - L(y,Fm-1+hm) is the maximum.

The minimization is essentially a gradient descent problem. Hence, to find an hm which when added to Fm−1 reduces the loss, we take a step in the direction where the Loss L(yi,Fm−1) reduces (with respect to Fm−1).

Mathematically, we take a step of size γ in the direction −(∂L(yi,F(xi))/∂F(xi), where F(xi)=Fm−1(xi). We stop when we see that the gradients are very close to zero.

Gradient Boosting Model

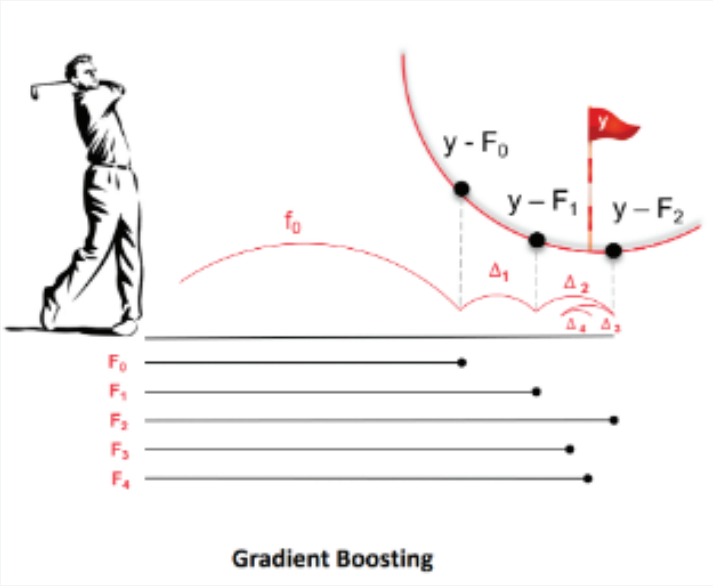
$$F_0(X) = Y_0 H_0(x, y) + e_0$$

$$F_1(X) = F_0(X) + Y_1 H_1(x, e_0) + e_1$$

$$F_2(X) = F_1(X) + Y_2 H_2(x, e_1) + e_2$$

$$\vdots$$

$$F_n(X) = F_{n-1}(X) + Y_2 H_n(x, e_{n-1}) + e_n$$



Gradient Boosting

XGBoost:

- **Extreme Gradient Boosting (XGBoost)** is similar to the gradient boosting framework but more efficient and advanced implementation of the Gradient Boosting algorithm.
- It was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. Due to its robust accuracy, it has been widely used in machine learning competitions as well. It uses more accurate approximations to tune the model and find the best fit.

Advantages of XGBoost:

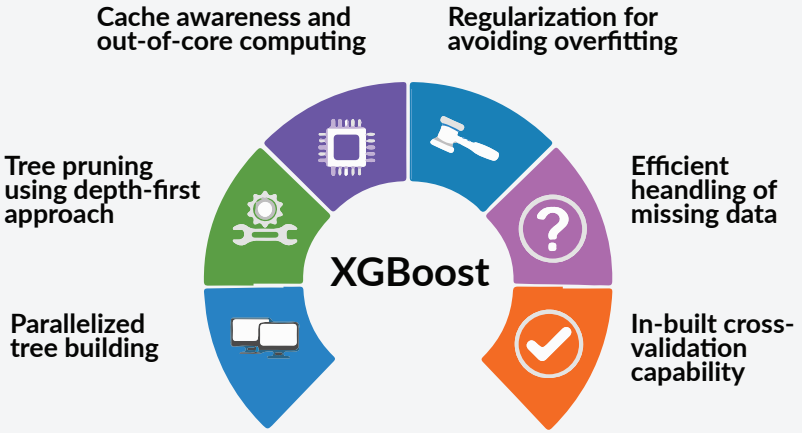
1. **Parallel Computing:** when you run xgboost, by default, it would use all the cores of your laptop/machine enabling its capacity to do parallel computation
2. **Regularization:** The biggest advantage of xgboost is that it uses regularization and controls the overfitting and simplicity of the model which gives it better performance.
3. **Enabled Cross-Validation:** XGBoost is enabled with internal Cross Validation function 4. Missing Values: XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. **Flexibility:** XGBoost is not just limited to regression, classification, and ranking problems, it supports user-defined objective functions as well. Furthermore, it supports user-defined evaluation metrics as well.

At, the **learning rate**, is also known as **shrinkage**. It can be used to regularize the gradient tree boosting algorithm. A typically varies from 0 to 1. Smaller values of At lead to a larger value of a number of trees T (called n_estimators in the Python package XGBoost). This is because, with a slower learning rate, you need a larger number of trees to reach the minima. This, in turn, leads to longer training time.

On the other hand, if A is large, we may reach the same point with a lesser number of trees (n_estimators), but there's the risk that we might actually miss the minima altogether (i.e. cross over it) because of the long stride we are taking at each iteration.

Some other ways of regularization are explicitly specifying the number of trees T and doing subsampling. Note that you shouldn't tune both Ar and number of trees T together since a high At implies a low value of T sa and vice-versa.

Subsampling is training the model in each iteration on a fraction of data (similar to how random forests build each tree). A typical value of subsampling is 0.5 while it ranges from 0 to 1. In random forests, subsampling is critical to ensure diversity among the trees, since otherwise, all the trees will start with the same training data and therefore look similar. This is not a big problem in boosting since each tree is any way built on the residual and gets a significantly different objective function than the previous one.



Python Syntaxes for Hyperparameter Tuning:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import make_classification
# Generate some example data
X, y = make_classification()
# Create an AdaBoostClassifier object
adaboost = AdaBoostClassifier()
# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [50, 100, 200],  # Number of weak learners
    'learning_rate': [0.1, 0.5, 1.0],  # Learning rate for each weak learner
    'algorithm': ['SAMME', 'SAMME.R'],  # AdaBoost algorithm to use
    'base_estimator__max_depth': [1, 2, 3],  # Maximum depth of weak
learners
    'base_estimator__min_samples_split': [2, 4, 8]  # Minimum number
of samples required to split weak learners
}
# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(adaboost, param_grid, cv=5)
grid_search.fit(X, y)
# Print the best hyperparameters found
print("Best Hyperparameters:")
print(grid_search.best_params_)
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
gbm = GradientBoostingRegressor(random_state=42)
param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.1, 0.01, 0.001],
    'max_depth': [3, 5, 7],
    'subsample': [0.8, 1.0],
    'min_samples_split': [2, 5, 10]
}
grid_search = GridSearchCV(estimator=gbm, param_grid=param_grid,
cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters:", grid_search.best_params_)
```

```
from sklearn.model_selection import GridSearchCV
from xgboost import XGBClassifier
xgb = XGBClassifier()
param_grid = {
    'learning_rate': [0.1, 0.01, 0.001],  # Learning rate
    'max_depth': [3, 5, 7],               # Maximum depth of a tree
    'n_estimators': [100, 200, 300],      # Number of trees (boosting rounds)
    'subsample': [0.8, 0.9, 1.0],         # Subsample ratio of the training instances
    'colsample_bytree': [0.8, 0.9, 1.0]   # Subsample ratio of columns when
constructing each tree
}
grid_search = GridSearchCV(estimator=xgb, param_grid=param_grid, cv=3)
grid_search.fit(X_train, y_train)
print("Best hyperparameters: ", grid_search.best_params_)
```