**1. How do ensemble techniques work for regression and classification problems**

* Bagging, the short form for bootstrap aggregating, is mainly applied in classification and [regression](https://corporatefinanceinstitute.com/resources/knowledge/finance/regression-analysis/). It increases the accuracy of models through decision trees, which reduces variance to a large extent. The reduction of variance increases accuracy, eliminating overfitting, which is a challenge to many predictive models.
* Bagging is classified into two types, i.e., bootstrapping and aggregation. **Bootstrapping** is a sampling technique where samples are derived from the whole population (set) using the replacement procedure. The sampling with replacement method helps make the selection procedure randomized. The base learning algorithm is run on the samples to complete the procedure.
* Bagging meta-estimator is an ensembling algorithm that can be used for both classification (BaggingClassifier) and regression (BaggingRegressor) problems. It follows the typical bagging technique to make predictions. Following are the steps for the bagging meta-estimator algorithm:
* Random subsets are created from the original dataset (Bootstrapping).
* The subset of the dataset includes all features.
* A user-specified base estimator is fitted on each of these smaller sets.
* Predictions from each model are combined to get the final result.

**Sample code for Classification problem:**

from sklearn.ensemble import BaggingClassifier

from sklearn import tree

model = BaggingClassifier(tree.DecisionTreeClassifier(random\_state=1))

model.fit(x\_train, y\_train)

model.score(x\_test,y\_test)

0.75135135135135134

**Sample code for regression problem:**

from sklearn.ensemble import BaggingRegressor

model = BaggingRegressor(tree.DecisionTreeRegressor(random\_state=1))

model.fit(x\_train, y\_train)

model.score(x\_test,y\_test)

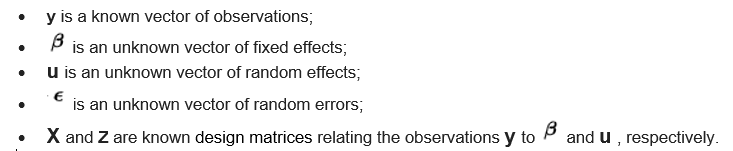
**2. What is the mixing models approach?**

* A mixing model (or more precisely mixed error-component model) is a statistical model containing both fixed effects and random effects. It is an extension of simple linear models. These models are useful in a wide variety of disciplines in the physical, biological and social sciences. It is the *regression models* which is one of the powerful tool for linear regression models when your data contains global and group-level trends.
* They are particularly useful in settings where repeated measurements are made on the same statistical units (longitudinal study), or where measurements are made on clusters of related statistical units.
* In the field of ecological and biological data are often complex and messy and sometimes bi-modal. We may have different grouping factors like populations, species, sites, gender ,etc. Sample sizes might leave something to be desired too, especially if we are trying to fit complicated models with many parameters.
* This is why mixed models were developed, to deal with such messy data and to allow us to use all our data, even when we have low sample sizes, structured data and many co-variate to fit.
* *Following is the representation of mixed model:*

Y = Fixed Effect + Random Effect + Error

https://miro.medium.com/max/296/1*raxNnXHWytoywkCx9ZzcJQ.png

Mixed Model syntax



**3. What are the XGBoost hyperparameters?**

* Generally, the XGBoost hyperparameters have been divided into 4 categories. They are as follows -
  + - * General parameters
      * Booster parameter
      * Learning task parameter
      * Command line parameters
* Before running a XGBoost model, we must set three types of parameters -**general parameters**, **booster parameters** and **task parameters**.
* The fourth type of parameters are **command line parameters**. They are only used in the console version of XGBoost. So, we will skip these parameters and limit our discussion to the first three type of parameters.
* **General Parameters**
* These parameters guide the overall functioning of the XGBoost model.
* In this section, we will discuss three hyperparameters - **booster**, **verbosity** and **nthread**.
* Please visit [XGBoost General Parameters](https://xgboost.readthedocs.io/en/latest/parameter.html#general-parameters) for detailed discussion on general parameters.
* **booster**
* **booster[default = gbtree]**
  + **booster** parameter helps us to choose which booster to use.
  + It helps us to select the type of model to run at each iteration.
  + It has 3 options - **gbtree**, **gblinear** or **dart**.
    - **gbtree** and **dart** - use tree-based models, while
    - **gblinear** uses linear models.
* **verbosity**
* **verbosity[default = 1]**
  + Verbosity of printing messages.
  + Valid values are 0 (silent), 1 (warning), 2 (info), 3 (debug).

**nthread**

* **nthread [default = maximum number of threads available if not set]**
  + This is number of parallel threads used to run XGBoost.
  + This is used for parallel processing and number of cores in the system should be entered.
  + If you wish to run on all cores, value should not be entered and algorithm will detect automatically.
* There are other general parameters like **disable\_default\_eval\_metric [default=0]**, **num\_pbuffer [set automatically by XGBoost, no need to be set by user]** and **num\_feature [set automatically by XGBoost, no need to be set by user]**.
* So, these parameters are taken care by XGBoost algorithm itself. Hence,we will not discuss these further.

**Booster Parameters**

* We have 2 types of boosters - **tree booster** and **linear booster**.
* We will limit our discussion to **tree booster** because it always outperforms the **linear booster** and thus the later is rarely used.
* Please visit, [Parameters for Tree Booster](https://xgboost.readthedocs.io/en/latest/parameter.html#parameters-for-tree-booster), for detailed discussion on booster parameters.

**eta**

* **eta [default=0.3, alias: learning\_rate]**
  + It is analogous to learning rate in GBM.
  + It is the step size shrinkage used in update to prevent overfitting.
  + After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
  + It makes the model more robust by shrinking the weights on each step.
  + range : [0,1]
  + Typical final values : 0.01-0.2.
* **gamma**
* **gamma [default=0, alias: min\_split\_loss]**
  + A node is split only when the resulting split gives a positive reduction in the loss function.
  + Gamma specifies the minimum loss reduction required to make a split.
  + It makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
  + The larger gamma is, the more conservative the algorithm will be.
  + Range: [0,∞]
* **max\_depth**
* **max\_depth [default=6]**
  + The maximum depth of a tree, same as GBM.
  + It is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
  + Increasing this value will make the model more complex and more likely to overfit.
  + The value 0 is only accepted in loss guided growing policy when tree\_method is set as hist and it indicates no limit on depth.
  + We should be careful when setting large value of max\_depth because XGBoost aggressively consumes memory when training a deep tree.
  + range: [0,∞] (0 is only accepted in lossguided growing policy when tree\_method is set as hist.
  + Should be tuned using CV.
  + Typical values: 3-10
* **min\_child\_weight**
* **min\_child\_weight [default=1]**
  + It defines the minimum sum of weights of all observations required in a child.
  + This is similar to min\_child\_leaf in GBM but not exactly. This refers to min “sum of weights” of observations while GBM has min “number of observations”.
  + It is used to control over-fitting.
  + Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
  + Too high values can lead to under-fitting.
  + Hence, it should be tuned using CV.
  + The larger min\_child\_weight is, the more conservative the algorithm will be.
  + range: [0,∞]

**max\_delta\_step**

* **max\_delta\_step [default=0]**
  + In maximum delta step we allow each tree’s weight estimation to be.
  + If the value is set to 0, it means there is no constraint.
  + If it is set to a positive value, it can help making the update step more conservative.
  + Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
  + Set it to value of 1-10 might help control the update.
  + range: [0,∞]
* **subsample**
* **subsample [default=1]**
  + It denotes the fraction of observations to be randomly samples for each tree.
  + Subsample ratio of the training instances.
  + Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. - This will prevent overfitting.
  + Subsampling will occur once in every boosting iteration.
  + Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
  + Typical values: 0.5-1
  + range: (0,1]

**colsample\_bytree, colsample\_bylevel, colsample\_bynode**

* **colsample\_bytree, colsample\_bylevel, colsample\_bynode [default=1]**
  + This is a family of parameters for subsampling of columns.
  + All **colsample\_by** parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
  + **colsample\_bytree** is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
  + **colsample\_bylevel** is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.
  + **colsample\_bynode** is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
  + **colsample\_by\*** parameters work cumulatively. For instance, the combination **{'colsample\_bytree':0.5, 'colsample\_bylevel':0.5, 'colsample\_bynode':0.5}** with 64 features will leave 8 features to choose from at each split.
* **lambda**
* **lambda [default=1, alias: reg\_lambda]**
  + L2 regularization term on weights (analogous to Ridge regression).
  + This is used to handle the regularization part of XGBoost.
  + Increasing this value will make model more conservative.
* **alpha**
* **alpha [default=0, alias: reg\_alpha]**
  + L1 regularization term on weights (analogous to Lasso regression).
  + It can be used in case of very high dimensionality so that the algorithm runs faster when implemented.
  + Increasing this value will make model more conservative.
* **tree\_method**
* **tree\_method string [default= auto]**
  + The tree construction algorithm used in XGBoost.
  + XGBoost supports approx, hist and gpu\_hist for distributed training. Experimental support for external memory is available for approx and gpu\_hist.
  + Choices: auto, exact, approx, hist, gpu\_hist
    - **auto**: Use heuristic to choose the fastest method.
      * For small to medium dataset, exact greedy (exact) will be used.
      * For very large dataset, approximate algorithm (approx) will be chosen.
      * Because old behavior is always use exact greedy in single machine, user will get a message when approximate algorithm is chosen to notify this choice.
    - **exact**: Exact greedy algorithm.
    - **approx**: Approximate greedy algorithm using quantile sketch and gradient histogram.
    - **hist**: Fast histogram optimized approximate greedy algorithm. It uses some performance improvements such as bins caching.
    - **gpu\_hist**: GPU implementation of hist algorithm.

**scale\_pos\_weight**

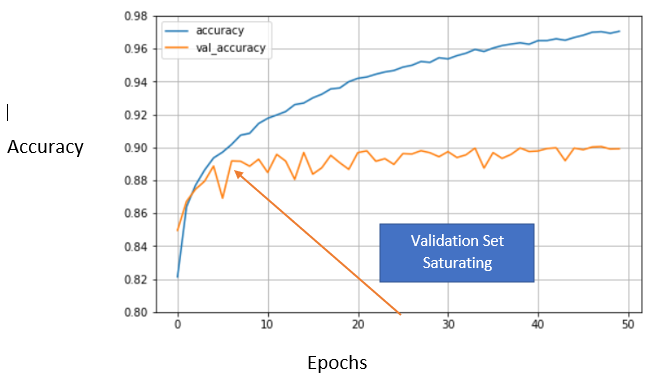
* **scale\_pos\_weight [default=1]**
  + It controls the balance of positive and negative weights,
  + It is useful for imbalanced classes.
  + A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.
  + A typical value to consider: sum(negative instances) / sum(positive instances).

**max\_leaves**

* **max\_leaves [default=0]**
  + Maximum number of nodes to be added.
  + Only relevant when grow\_policy=lossguide is set.
* There are other hyperparameters like sketch\_eps,updater, refresh\_leaf, process\_type, grow\_policy, max\_bin, predictor and num\_parallel\_tree.

**4. What is early stopping?**

* The model tries to chase the loss function crazily on the training data, by tuning the parameters. Now, we keep another set of data as the validation set and as we go on training, we keep a record of the loss function on the validation data, and when we see that there is no improvement on the validation set, we stop, rather than going all the epochs.
* This strategy of stopping early based on the validation set performance is called **Early Stopping.**



  Early Stopping Demonstration

* The training set accuracy continues to increase, through all the Epochs
* The validation set accuracy, however, saturates between 8 to 10 epochs. This is where the model can be **stopped**training.
* Early Stopping, hence does not only protect against overfitting but needs considerably less number of Epoch to train.

**5**.**What is a weak learner?**

* Week Learners have high bias and high variance.
* Weak learners are exactly what they sound like - 'weak'. They are weak in the sense that they do not perform well on our data neither train nor test. They are usually high bias models i.e they underfit but have low variance and they don’t usually overfit. As they underfit, they perform a little better than the random model i.e error rate of 50%.
* Now our prime task is to decrease this high bias without effecting the variance. We combine all these weak learners to get a strong learner which has decent bias and variance and hence performs much better than all the individual weak learners. Note that, we prefer the weak learners whose time to train is less.
* The desirable properties of a weak Learner:
* Does not overfit but high bias is obvious (that's why it performed like a random model in the first place)
* Trains fast ( we don't want out the final model to be slow)
* Predicts quickly ( we don't want out the final model to be slow)
* As you are aware we use a lot of Decision trees in Random forests as weak learners. But why only decision trees?
* The reasons come from the fact that just by varying the maximum depth of the decision tree we can get all the three desirable properties. How?
* Decrease the depth of Decision tree - We get a stump which will underfit badly, will predict faster, will run fast as the depth is low and also trees can be trained in O(log(n)\*d) time and prediction only takes O(depth of the tree)
* Here 'd' is the evaluation time for a feature with d dimensions.

**Practical Example**

Suppose you want to buy a mobile or car. You take advice from your friends, online sites, youtube videos. All these can be treated as a weak learner. When you combine all their advice(weak learner) then you finally decide which phone to buy and which is best according to your requirement.