Extreme Gradient Boosting

XGBoost Advantage

1. **Regularization:**
    - Standard GBM implementation has no regularization like XGBoost, therefore it also helps to reduce overfitting.
    - In fact, XGBoost is also known as a 'regularized boosting' technique.
2. **Parallel Processing:**
    - XGBoost implements parallel processing and is blazingly faster as compared to GBM.
    - But hang on, we know that boosting is a sequential process so how can it be parallelized? We know that each tree can be built only after the previous one, so what stops us from making a tree using all cores? I hope you get where I'm coming from. Check this link out to explore further.
    - XGBoost also supports implementation on Hadoop.
3. **High Flexibility:**
    - XGBoost allows users to define custom optimization objectives and evaluation criteria.
    - This adds a whole new dimension to the model and there is no limit to what we can do.
4. **Handling Missing Values:**
    - XGBoost has an in-built routine to handle missing values.
    - The user is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounters a missing value on each node and learns which path to take for missing values in future.
5. **Tree Pruning:**
    - A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a greedy algorithm.
    - XGBoost on the other hand make splits upto the max_depth specified and then start pruning the tree backwards and remove splits beyond which there is no positive gain.
    - Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. GBM would stop as it encounters -2. But XGBoost will go deeper and it will see a combined effect of +8 of the split and keep both.
6. **Built-in Cross-Validation:**
    - XGBoost allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.
    - This is unlike GBM where we have to run a grid-search and only a limited values can be tested.
7. **Continue on Existing Model:**
    - User can start training an XGBoost model from its last iteration of previous run. This can be of significant advantage in certain specific applications.
    - GBM implementation of sklearn also has this feature so they are even on this point.

XGBoost Training Parameters

There are 3 categories of XGBoost:

- **General Parameters**: Guide the overall functioning
- **Booster Parameters**: Guide the individual booster (tree/regression) at each step
- **Learning Task Parameters**: Guide the optimization performed

```
xgb.train(params = list(), data, nrounds,
      watchlist = list(), obj = NULL,
      feval = NULL, verbose = 1,
      print_every_n = 1L,
      early_stopping_rounds = NULL,
      maximize = NULL, save_period = NULL,
      save_name = "xgboost.model",
      xgb_model = NULL, callbacks = list(), ...)


xgboost(data = NULL, label = NULL,
      missing = NA, weight = NULL,
      params = list(), nrounds, verbose = 1,
      print_every_n = 1L,
      early_stopping_rounds = NULL,
      maximize = NULL, save_period = NULL,
      save_name = "xgboost.model",
      xgb_model = NULL, callbacks = list(), ...)
```

General Parameters

1. **booster [default=gbtree]**
   Select the type of model to run at each iteration. It has 2 options:
    - gbtree: tree-based models
    - gblinear: linear models
2. **silent [default=0]:**
    - Silent mode is activated is set to 1, i.e. no running messages will be printed.
    - It's generally good to keep it 0 as the messages might help in understanding the model.
3. **nthread [default to maximum number of threads available if not set]**
    - 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

Booster Parameters--tree

Here I only include some common parameters of XGBoost Model.

1. **eta [default = 0.3]**
   Learning rate, smaller eta requires more nrounds
    - Makes the model more robust by shrinking the weights on each step
    - Typical final values to be used: 0.01-0.2
2. **min_child_weight [default=1]**
   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".
    - 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.

3. **max_depth [default=6]**
    - Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
    - Should be tuned using CV.
    - Typical values: 3-10
4. **max_leaf_nodes**
   The maximum number of terminal nodes or leaves in a tree.
    - Can be defined in place of max_depth. Since binary trees are created, a depth of 'n' would produce a maximum of 2^n leaves.
    - If this is defined, GBM will ignore max_depth.
5. **gamma [default=0]**
   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.
   Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
6. **subsample [default=1]**
   Denotes the fraction of observations to be randomly samples for each tree.
    - Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
    - Typical values: 0.5-1
7. **colsample_bytree [default=1]**
   Denotes the fraction of columns to be randomly samples for each tree.
    -
    - Typical values: 0.5-1
8. **scale_pos_weight [default=1]**
   A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

Learning Task Parameters--tree

These parameters are used to define the optimization objective the metric to be calculated at each step.. Here I only include some common parameters of XGBoost Model.

1. **objective [default=reg:linear]**
   This defines the loss function to be minimized. Mostly used values are:
    - **binary:logistic** –logistic regression for binary classification, returns predicted probability (not class)
    - **multi:softmax** –multiclass classification using the softmax objective, returns predicted class (not probabilities)
      For this objective, you need to set an additional **num_class** parameter define the number of unique classes.
    - **multi:softprob** –same as softmax, but returns predicted probability of each data point belonging to each class.
2. **eval_metric [default according to objective]**
   The metric to be used for validation data.The default values are **rmse** for regression and **error** for classification. Typical values are:
    - **rmse** – root mean square error
    - **mae** – mean absolute error
    - **logloss** – negative log-likelihood
    - **error** – Binary classification error rate (0.5 threshold)
    - **merror** – Multiclass classification error rate
    - **mlogloss** – Multiclass logloss
    - **auc** - Area under the curve
3. **seed [default=0]**
   The random number seed. Can be used for generating reproducible results and also for parameter tuning.

Model Implementation

1. **xgb**
   – this is the direct xgboost library. I will use a specific function "cv" from this library
2. **XGBClassifier**
   – this is an sklearn wrapper for XGBoost. This allows us to use sklearn's Grid Search with parallel processing in the same way we did for GBM.

Parameter Tuning

There are many ways of tuning parameters, for XGBClassifier specifically, tuning **learning rate** and **n_estimator** together and other parameters together. Common Tuning methods:

- **RandomizedSearchCV**
  which randomly choose parameters from the param_distributions.The number of parameter settings that are tried is given by n_iter.

```
class sklearn.model_selection.
RandomizedSearchCV(
  estimator, param_distributions, n_iter=10,
  scoring=None, n_jobs=None, iid='warn',
  refit=True, cv='warn', verbose=0,
  pre_dispatch='2*n_jobs', random_state=None,
  error_score='raise-deprecating',
  return_train_score=False)
```

- **GridSearchCV**
  which do bruce force of each parameter inside the param_grid.

```
class sklearn.model_selection.
GridSearchCV(
 estimator, param_grid, scoring=None,
 n_jobs=None, iid='warn', refit=True,
 cv='warn', verbose=0,
 pre_dispatch='2*n_jobs',
 error_score='raise-deprecating',
 return_train_score=False)
```

Common parameters descriptions in RandomizedSearchCV and GridSearchCV are:

- **estimator: estimator object.** the model you specify
- **param_distributions/param_grid: dict** Dictionary with parameters names (string) as keys and distributions or lists of parameters to try.
- **n_iter : int, default=10** Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.
- **scoring : string, callable, list/tuple, dict or None, default: None** A single string custom evalution metric or a callable to evaluate the predictions on the test set. Check Example
  If you have multiple metrics, also use **refit**.
- **n_jobs: int or None, optional (default=None)** Number of jobs to run in parallel. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors.
- **iid : boolean, default='warn'** If True, return the average score across folds, weighted by the number of samples in

each test set. In this case, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds. If False, return the average score across folds.

- **cv : int, cross-validation generator or an iterable, optional** Determines the cross-validation splitting strategy. Possible inputs for cv are: None(default cv = 3); integer(using StratifiedKFold)
- **refit : boolean, string, or callable, default=True** Refit an estimator using the best found parameters on the whole dataset.
- **random_state : int, RandomState instance or None, optional, default=None**

Common attributes are:

- **cv_results_ : dict of numpy (masked) ndarrays**
- **best_estimator_ : estimator or dict** Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if *refit*=False.
- **best_score_ : float** Mean cross-validated score of the best_estimator.
- **best_params_ : dict** Parameter setting that gave the best results on the hold out data.
- **best_index_ : int** The index (of the cv_results_ arrays) which corresponds to the best candidate parameter setting.
- **scorer_ : function or a dict** Scorer function used on the held out data to choose the best parameters for the model.
- **n_splits_ : int** The number of cross-validation splits (folds/iterations).

Detailed documents, see sklearn.model_selection.RandomizedSearchCV and sklearn.model_selection.GridSearchCV