Boosting Machines

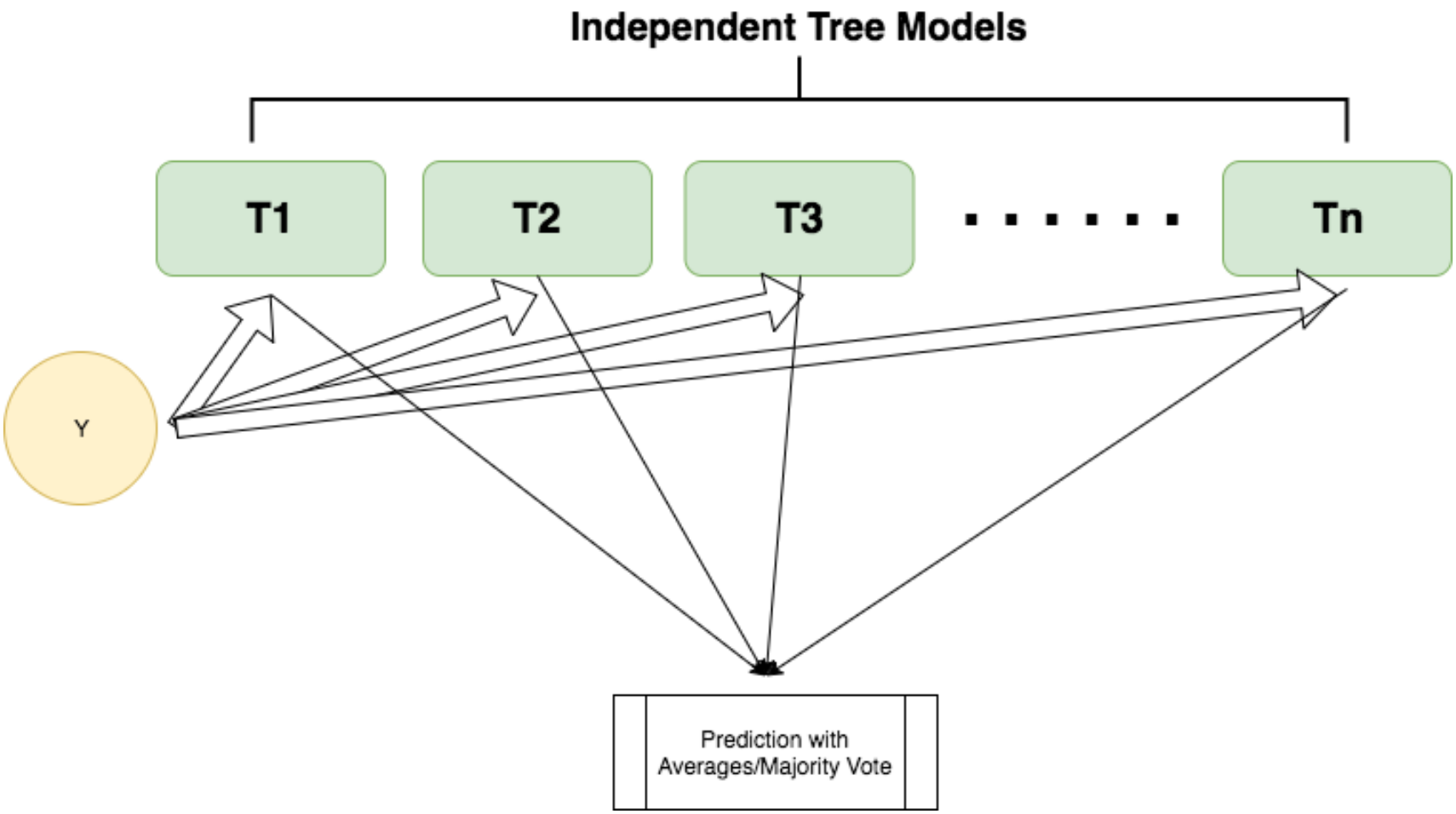
Agenda

Discussion Flow

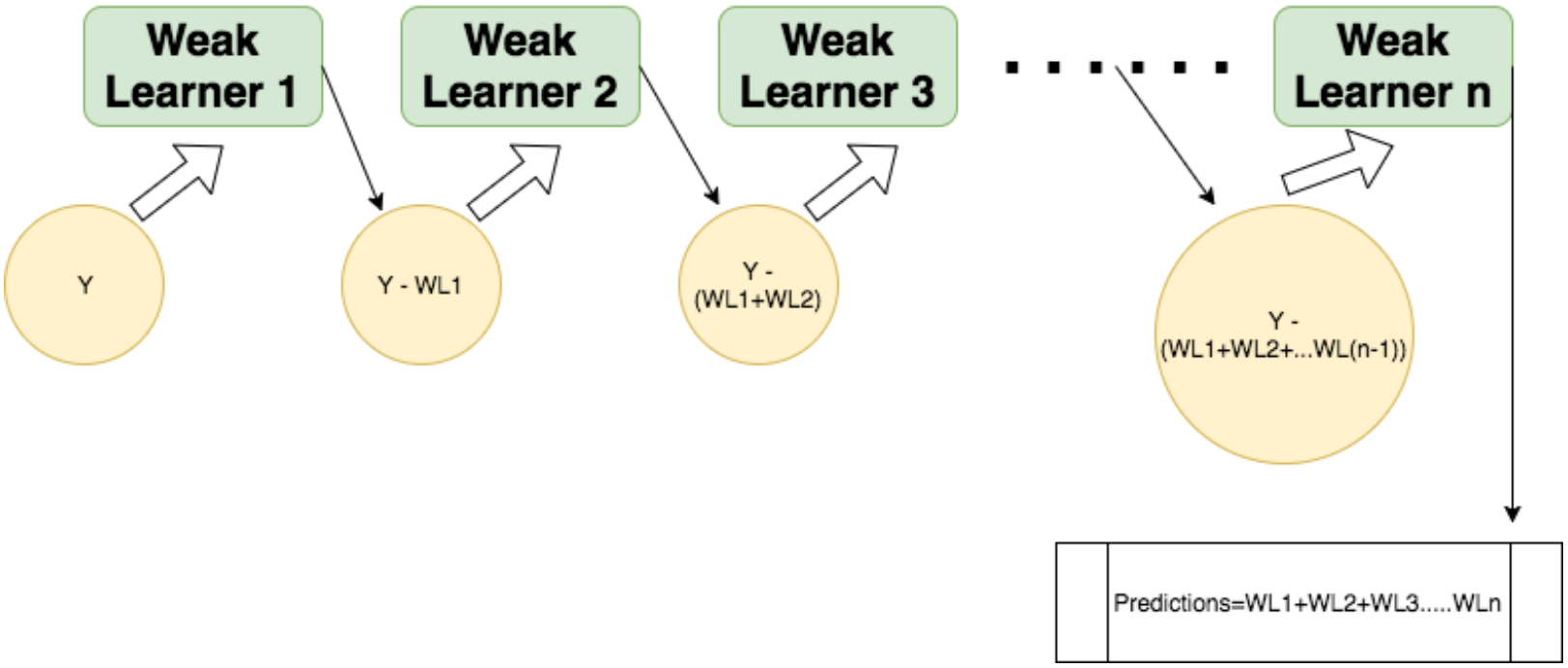
- Bagging Vs Boosting
- Decision tree stumps as weak learners
- Gradient boosting machines
- Boosting machines for Regression
- Boosting machines for classification
- Sklearn Implementation
- Xgboost
- Sklearn Implementation

Bagging Vs Boosting

Bagging



Boosting



Weak Learner

What ?

- Simple models which can capture generic patterns
- Examples :
  - Linear models with subset of variables
  - Linear models with heavy penalty
  - Decision Tree Stumps (Shallow tree with low number of splits)

Why ?

- Inability to learn niche patterns , difficult to overfit
- Strong learners for the same reason will lead to overfit
- Weak learners emphasise capture of patterns which are generic and yet in combination can make very strong model overall

Decision Tree Stumps as Weak Learner

- Decision Tree Stumps are a popular choice of weak learner
- Easy to implement and shallow trees can learn simple non-linear patterns too

Boosting Machines

Incremental Nature of Boosting Machine Models

$$Prediction\ at\ iteration\ t = F_t(X) = \sum_{i=0}^{t} f_t(X)$$

$$where\ f_t(X)\ is\ t^{th}\ weak\ learner$$

Gradient Descent in Functional Space

$$J = \sum L(y_i, F_t(X_i))$$

$$\frac{\delta J}{\delta F_t(X)} = \sum \frac{\delta L(y_i, F_t(X_i))}{\delta F_t(X)}$$

$$f_{t+1}(X) \rightarrow -\eta \frac{\delta J}{\delta F_t(X)}$$

$$F_{t+1}(X) = F_t(X) + f_{t+1}(X)$$

GBM for regression

$$L(y_i, F_t(x_i)) = (y_i - F_t(x_i))^2$$

$$\frac{\delta L}{\delta F_t(x)} \sim -(y_i - F_t(x_i))$$

$$f_{t+1}(x) \rightarrow -\eta \frac{\delta L}{\delta F_t(x)} \rightarrow \eta(y_i - F_t(x_i))$$

GBM for classification

Model

$$p_i^{(t)} = \frac{1}{1 + e^{-F_t(x_i)}}$$

Loss Function

$$L(y_i, F_t(x_i)) = -(y_i * log(p_i^{(t)}) + (1 - y_i) * log(1 - p_i^{(t)}))$$
$$= log(1 + e^{F_t(x_i)}) - y_i * F_t(x_i)$$

Gradient

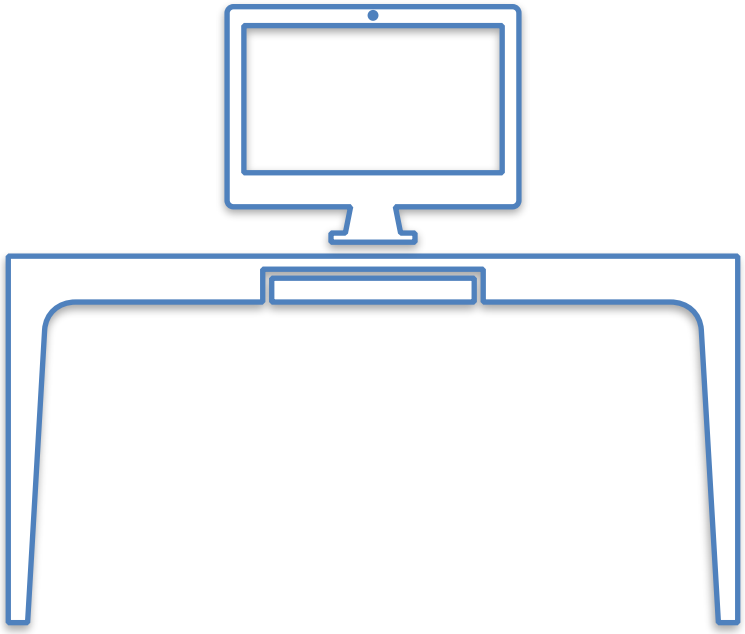
$$\frac{\delta J}{\delta F_t(x_i)} = -(y_i - \frac{1}{1 + e^{-F_t(x_i)}})$$
$$= -(y_i - p_i^{(t)})$$

A Regression Tree

Next Weak Learner

$$f_{t+1}(x) \rightarrow \eta(y_i - p_i^{(t)})$$

Lets see it in action in Python



Xgboost : Theoretical Details

Issues with usual GBM

- Loss function doesn't consider complexity of the model
- Leads to overfit and not so generalised error performance
- xgboost uses new objective function, which adds model complexity to the traditional loss function

Objective Function for Xgboost

Tradition loss function

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_t(x_i)) + \sum_{i=1}^{t} \Omega(f_i)$$

Model Complexity

Taylor's Expansion of loss function

$$= \sum_{i=1}^{n} L(y_i, F_t(x_i)) + \sum_{i=1}^{t} \Omega(f_i)$$

$$= \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + f_t(x_i)) + \sum_{i=1}^{t} \Omega(f_i)$$

$$= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \sum_{i=1}^{t} \Omega(f_i) \; + \; constant$$

$$g_i = \frac{\delta L(y_i, F_{t-1}(x_i))}{\delta F_{t-1}}$$

$$h_i = \frac{\delta^2 L(y_i, F_{t-1}(x_i))}{\delta^2 F_{t-1}}$$

Model Complexity : Defining a tree

- Predictions of a tree are nothing but score/weights at the leaf nodes

$$f_t(x) = w_{q(x)}, w \in R^T, q : R^d \rightarrow \{1,2,\ldots,T\}$$

- here q(x) maps each data point to its leaf , output of q is the leaf index
- w is score/weight for each leaf
- T is the number of leaves in the tree

Model Complexity contd..

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

- This has penalty on size of the tree, as larger the number of leaves more complex is the tree
- Each individual tree (weak learner) shouldn't result in huge update in the over all model, to control this , leaf scores are penalised too

Revisiting the objective function

$$obj^{(t)} \sim \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2$$

$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

- I_j here is the set of indices of data points in j^{th} leaf
- All the data points in the same leaf get same score

Revisiting the objective function contd..

$$G_j = \sum_{i \in I_j} g_i$$

$$H_j = \sum_{i \in I_j} h_i$$

$$obj^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T$$

Optimisation

- For a given tree structure , score on a leaf doesn't have to be simple proportion or average or majority vote
- We can in fact find an efficient score which minimises the objective function given a tree structure
- We can then use the optimal value weight to see how good the tree structure is (free of w_j)

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

$$obj^* = -\frac{1}{2}\sum_{i=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T$$

Building the Trees

$$Gain = \frac{1}{2}\left[\frac{G_L{}^2}{H_L + \lambda} + \frac{G_R{}^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma$$

- This formula can be decomposed as
  - the score on the new left leaf
  - the score on the new right leaf
  - The score on the original leaf
  - regularization on the additional leaf.
- If the gain is smaller than gamma for all rules , no branch is added

Xgboost : Paramters

Objective Functions to use for different problems

- “reg:linear” –linear regression
- “binary:logistic” –logistic regression for binary classification, output probability
- “multi:softmax” –set XGBoost to do multiclass classification using the softmax objective, you also need to set num_class(number of classes)

Reference : http://xgboost.readthedocs.io/en/latest/parameter.html

Hyperparameters

Control Overfitting

When you observe high training accuracy, but low tests accuracy, it is likely that you encounter overfitting problem.

There are in general two ways that you can control overfitting in xgboost

- The first way is to directly control model complexity
  - This include `max_depth`, `min_child_weight` and `gamma`
- The second way is to add randomness to make training robust to noise
  - This include `subsample`, `colsample_bytree`
  - You can also reduce stepsize `eta`, but needs to remember to increase `num_round` when you do so.

Reference : http://xgboost.readthedocs.io/en/latest/how_to/param_tuning.html

Hyperparameters

Handle Imbalanced Dataset

For common cases such as ads clickthrough log, the dataset is extremely imbalanced. This can affect the training of xgboost model, and there are two ways to improve it.

- If you care only about the ranking order (AUC) of your prediction
  - Balance the positive and negative weights, via `scale_pos_weight`
  - Use AUC for evaluation
- If you care about predicting the right probability
  - In such a case, you cannot re-balance the dataset
  - In such a case, set parameter `max_delta_step` to a finite number (say 1) will help convergence

Reference : http://xgboost.readthedocs.io/en/latest/how_to/param_tuning.html

Understanding individual hyperparamters

- eta [default=0.3, alias: learning_rate]
  - step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features. and eta actually shrinks the feature weights to make the boosting process more conservative.
  - range: [0,1]
- gamma [default=0, alias: min_split_loss]
  - minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm will be.
  - range: [0,∞]
- max_depth [default=6]
  - maximum depth of a tree, increase this value will make the model more complex / likely to be overfitting. 0 indicates no limit, limit is required for depth-wise grow policy.
  - range: [0,∞]

Reference : http://xgboost.readthedocs.io/en/latest/parameter.html

Understanding individual hyperparamters

- min_child_weight [default=1]
  - minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression mode, this simply corresponds to minimum number of instances needed to be in each node. The larger, the more conservative the algorithm will be.
  - range: [0,∞]
- max_delta_step [default=0]
  - 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 [default=1]
  - subsample ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collected half of the data instances to grow trees and this will prevent overfitting.
  - range: (0,1]

Reference : http://xgboost.readthedocs.io/en/latest/parameter.html

Understanding individual hyperparamters

- colsample_bytree [default=1]
  - subsample ratio of columns when constructing each tree.
  - range: (0,1]
- colsample_bylevel [default=1]
  - subsample ratio of columns for each split, in each level.
  - range: (0,1]
- lambda [default=1, alias: reg_lambda]
  - L2 regularization term on weights, increase this value will make model more conservative.
- alpha [default=0, alias: reg_alpha]
  - L1 regularization term on weights, increase this value will make model more conservative.
- scale_pos_weight, [default=1]
  - Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider:

sum(negative cases)/sum(positive cases)

Reference : http://xgboost.readthedocs.io/en/latest/parameter.html

Lets see it in action in Python