XGBoost: algorithm introduction, parameter tuning, and applications

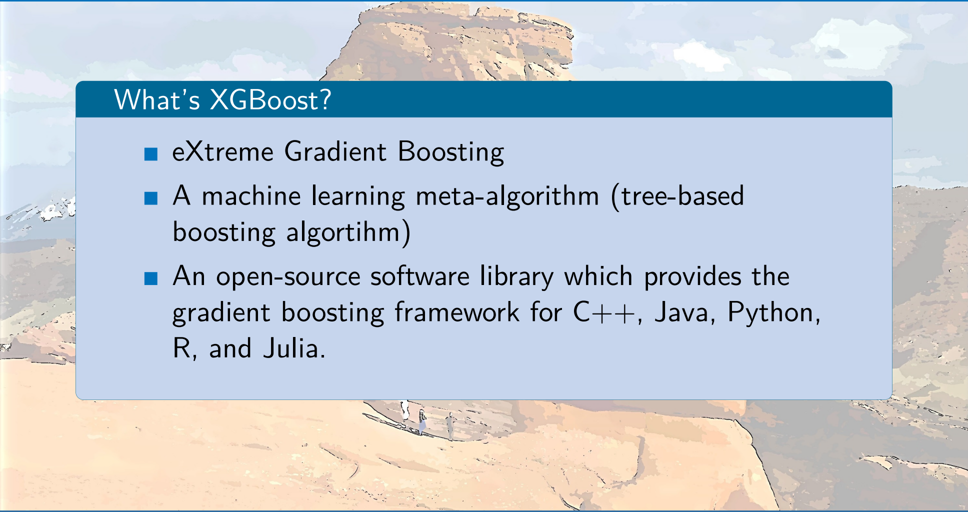
Shiu-Tang Li

University of Utah

May 7th, 2017

Outline

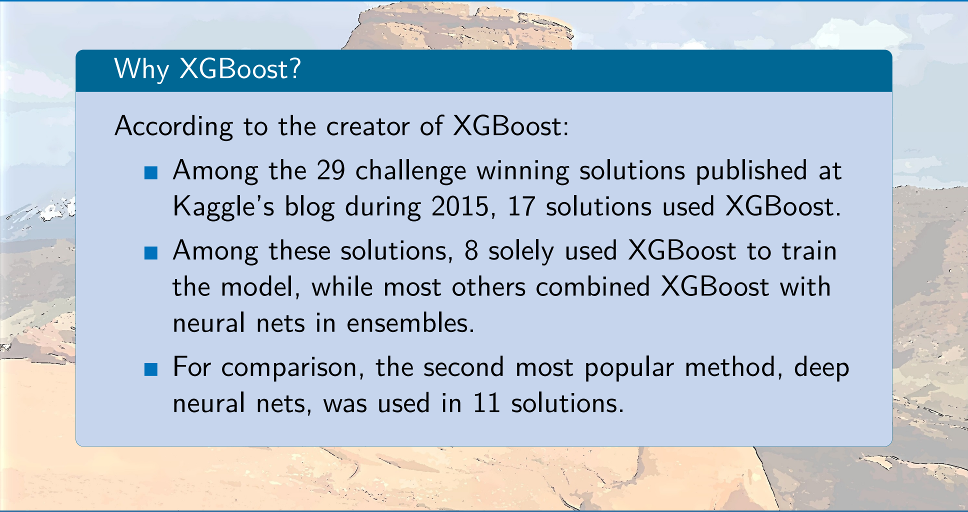
Introduction



What's XGBoost?

- eXtreme Gradient Boosting
- A machine learning meta-algorithm (tree-based boosting algortihm)
- An open-source software library which provides the gradient boosting framework for C++, Java, Python, R, and Julia.

Introduction



Why XGBoost?

According to the creator of XGBoost:

- Among the 29 challenge winning solutions published at Kaggle's blog during 2015, 17 solutions used XGBoost.
- Among these solutions, 8 solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles.
- For comparison, the second most popular method, deep neural nets, was used in 11 solutions.

Introduction

Recent kaggle winner's report which XGBoost is used

- Outbrain Click Prediction (2nd) 3.17.2017
- Allstate Claims Severity Competition (2nd) 2.27.2017
- Santander Product Recommendation Competition (2nd, 3rd) 01.12.2017, 2.22.2017
- Bosch Production Line Performance Competition (1st, 3rd) 12.15.2016, 12.30.2016
- TalkingData Mobile User Demographics Competition (3rd) 10.19.2016
- Red Hat Business Value Competition (1st) 11.03.2016

Notations setup

- Suppose we have n training data points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$, each data point $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_d^{(i)})$ is a d-dimensional real-valued vector, d is the number of features in the training set.
- $\mathbf{y} = (y_1, \ldots, y_n)$ is the vector containing training labels. y_i is the training label of $\mathbf{x}^{(i)}$.
- $\widehat{\mathbf{y}} = (\widehat{y}_1, \ldots, \widehat{y}_n)$ is the vector containing predicted values. $\widehat{y}_i$ is the predicted value of $\mathbf{x}^{(i)}$.
- For a regression problem, for $1 \le i \le n$, $y_i, \widehat{y}_i \in \mathbb{R}$. For a two-class classification problem, $y_i, \widehat{y}_i \in [0, 1]$.
- If we put a superscript on $\widehat{\mathbf{y}}$, say $\widehat{\mathbf{y}}^{(t)}$, it means it's the predicted value for t-th interation.

Loss function

Let $L(\mathbf{y}, \widehat{\mathbf{y}})$ denote the loss function, used to measure the difference of training labels and predicted values. If $\mathbf{y} = \widehat{\mathbf{y}}$, $L(\mathbf{y}, \widehat{\mathbf{y}}) = 0$. Examples:

- RMSE (root mean square error) $(y_i, \widehat{y}_i \in \mathbb{R})$:

$$L(\mathbf{y}, \widehat{\mathbf{y}}) := \sqrt{\frac{\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2}{n}}$$

- log loss for binary classification $(y_i \in \{0, 1\}, \widehat{y}_i \in (0, 1))$:

$$L(\mathbf{y}, \widehat{\mathbf{y}}) := \sum_{i=1}^{n} -y_i \ln(\widehat{y}_i) - (1 - y_i) \ln(1 - \widehat{y}_i)$$

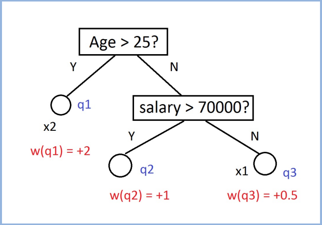
Think of a tree as a real-valued function on $\mathbb{R}^d$

We can think of a tree as a function F of two composite functions $w \circ q$, where $q : \mathbb{R}^d \to \{q_1, \ldots, q_T\}$ maps data points that contains d features to tree leaves $\{q_1, \ldots, q_T\}$, and $w : \{q_1, \ldots, q_T\} \to \mathbb{R}$ is a weight function that assigns a real value to each leaf. So $F := w \circ q$ is a function that maps $\mathbb{R}^d$ to $\mathbb{R}$.

Features=(Age,Salary(USD))

$x^{(1)} = (25, 45000)$

$x^{(2)} = (32, 65000)$



$F(x^{(1)}) = w \circ q(x^{(1)}) = 0.5$

$F(x^{(2)}) = w \circ q(x^{(2)}) = 2$

Tree boosting algorithm (1)

1. Begin with an initial estimator $Model \leftarrow F^{(0)}$ and an initial estimate $\widehat{\mathbf{y}}^{(0)}$ $(\widehat{y}_i^{(0)} = F^{(0)}(\mathbf{x}^{(i)}))$.
2. Calculate the loss $L(\mathbf{y}, \widehat{\mathbf{y}}^{(0)}) = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(0)})$.
3. In the first iteration, pick a tree $F(\mathbf{x}, \mathbf{a})$. $\mathbf{a}$ are the parameters (e.g. weights) to be tuned for the tree.
4. Pick $\mathbf{a}$ such that

$$\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(0)} + F(\mathbf{x}^{(i)}, \mathbf{a})) < \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(0)}),$$

and $\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(0)} + F(\mathbf{x}^{(i)}, \mathbf{a}))$ is minimized among all choices of $\mathbf{a}$'s.

Tree boosting algorithm (2)

5. Perform node splitting to the tree in the previous step. We would have many different tree structures $F_1(\mathbf{x}, \mathbf{a}_k), \ldots, F_k(\mathbf{x}, \mathbf{a}_k)$, along with the weights $\mathbf{a}_1, \ldots, \mathbf{a}_k$ to be tuned.
6. The goal is to find the tree structure along with the best weights $\widetilde{F}(\mathbf{x}, \widetilde{\mathbf{a}})$ such that the new loss $\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(0)} + \widetilde{F}(\mathbf{x}^{(i)}, \widetilde{\mathbf{a}}))$ is minimized and is smaller than the previous loss.
7. If we can find smaller loss in Step 6, then we continue Step 5 and 6 (keep splitting the tree) until we can't find smaller loss anymore.

Tree boosting algorithm (3)

8. Call the tree we get after several splits $F^{(1)}$. This is the tree we get when the first iteration is done.
9. Update our prediction $\widehat{y}_i^{(1)} = \widehat{y}_i^{(0)} + F^{(1)}(\mathbf{x}^{(i)})$ for all $1 \leq i \leq n$ and update the model by $Model \leftarrow Model + F^{(1)}$. The new loss becomes

$$L(\mathbf{y}, \widehat{\mathbf{y}}^{(1)}) = \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(1)}).$$

10. Repeat all the previous steps until we have N trees, or the loss can't be improved anymore. The reader can think of this process like growing a tree in each iteration step, and finally we have a forest of N trees.

The XGBoost algorithm (1)

Let's look at the main features of XGBoost that may differ from a typical tree boosting algorithm.

1. **Regularization**. When we're looking for a new tree F in t-th iteration, additional regularization terms are added to the loss function:

$$\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)} + F(\mathbf{x}^{(i)})) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 .$$

   Here T is the number of nodes of a tree, and w_j's are weights of the nodes (For notational simplicity, $w_j := w(q_j)$). The regularization terms limit the size of the tree and also limit the size of weights of nodes.

The XGBoost algorithm (2)

2. **Shrinkage (learning rate)**. After each iteration step, when updating the prediction and the model, a shrinkage constant $0 < \eta \leq 1$ is applied to reduce the learning rate.

$$\mathbf{y}^{(t)} = \mathbf{y}^{(t-1)} + \eta \cdot (F^{(t)}(\mathbf{x}^{(1)}), \cdots, F^{(t)}(\mathbf{x}^{(n)}))$$
$$Model \leftarrow Model + \eta \cdot F^{(t)}$$

3. **Column subsampling**. For each iteration step, when growing a new tree, a portion of features are randomly selected from $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ instead of using all features in the training data. This is to avoid overfitting.

The XGBoost algorithm (3)

4 **Apply 2nd order approximation to loss function**. By 2nd order Taylor approximation formula $f(x+\Delta x) \approx f(x) + f'(x) \cdot (\Delta x) + \frac{1}{2} f''(x) \cdot (\Delta x)^2$, the loss function ($t$-th iteration step, with tree F) can approximated by

$$\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)} + F(\mathbf{x}^{(i)})) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

$$\approx \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)}) + \sum_{i=1}^{n} g(y_i, \widehat{y}_i^{(t-1)}) \cdot F(\mathbf{x}^{(i)})$$

$$+\frac{1}{2} \sum_{i=1}^{n} h(y_i, \widehat{y}_i^{(t-1)}) \cdot (F(\mathbf{x}^{(i)}))^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 . \qquad (1)$$

(Here $g(z_1, z_2) := \frac{\partial}{\partial z_2} l(z_1, z_2)$, $h(z_1, z_2) := \frac{\partial^2}{\partial z_2^2} l(z_1, z_2)$.)
(Remark: GBM uses linear approximation instead)

The XGBoost algorithm (4)

In eq (1), the structure of the tree (i.e., $q(\mathbf{x})$) is fixed. We'd like to find the weights $w_1, \ldots, w_T$ that minimize eq (1).

Let $I_j := \{i : q(\mathbf{x}_i) = q_j\}$, the set of indices of $\mathbf{x}_i$ that are mapped to tree leaf q_j. We can rewrite (1) as

$$\sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)}) + \sum_{j=1}^{T} \sum_{i \in I_j} g(y_i, \widehat{y}_i^{(t-1)}) \cdot w_j$$
$$+\frac{1}{2} \sum_{j=1}^{T} \sum_{i \in I_j} h(y_i, \widehat{y}_i^{(t-1)}) \cdot w_j^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2. \quad (2)$$

which is the sum of T quadratics.

The XGBoost algorithm (5)

To minimize eq (2), the optimal weights w_j^* are given by

$$w_j^* = -\frac{\sum_{i\in I_j} g(y_i, \widehat{y}_i^{(t-1)})}{\sum_{i\in I_j} h(y_i, \widehat{y}_i^{(t-1)}) + \lambda} \tag{3}$$

and the optimal loss (with weight w_j^*'s) is given by

$$-\frac{1}{2}\sum_{j=1}^{T} \frac{\left(\sum_{i\in I_j} g(y_i, \widehat{y}_i^{(t-1)})\right)^2}{\sum_{i\in I_j} h(y_i, \widehat{y}_i^{(t-1)}) + \lambda} + \gamma T + \sum_{i=1}^{n} l(y_i, \widehat{y}_i^{(t-1)}). \tag{4}$$

The XGBoost algorithm (6)

Without loss of generality, let's split last tree node q_T into two nodes $q(L)$ and $q(R)$, and thus $I_T = I(L) \cup I(R)$. The optimal loss for the tree after we split the node becomes

$$-\frac{1}{2}\left(\sum_{j=1}^{T-1}\frac{\left(\sum_{i\in I_j} g(y_i,\widehat{y}_i^{(t-1)})\right)^2}{\sum_{i\in I_j} h(y_i,\widehat{y}_i^{(t-1)})+\lambda}+\frac{\left(\sum_{i\in I(L)} g(y_i,\widehat{y}_i^{(t-1)})\right)^2}{\sum_{i\in I(L)} h(y_i,\widehat{y}_i^{(t-1)})+\lambda}\right.$$
$$\left.+\frac{\left(\sum_{i\in I(R)} g(y_i,\widehat{y}_i^{(t-1)})\right)^2}{\sum_{i\in I(R)} h(y_i,\widehat{y}_i^{(t-1)})+\lambda}\right)+\gamma(T+1)+\sum_{i=1}^{n} l(y_i,\widehat{y}_i^{(t-1)}). \quad (5)$$

The XGBoost algorithm (7)

Therefore, for a tree with leaves $\{q_1, \ldots, q_T\}$, the optimal loss difference after we split tree leaf q_T would be

$$eq(5) - eq(4) = \gamma - \frac{1}{2}\left(\frac{\left(\sum_{i \in I(L)} g(y_i, \widehat{y}_i^{(t-1)})\right)^2}{\sum_{i \in I(L)} h(y_i, \widehat{y}_i^{(t-1)}) + \lambda} + \frac{\left(\sum_{i \in I(R)} g(y_i, \widehat{y}_i^{(t-1)})\right)^2}{\sum_{i \in I(R)} h(y_i, \widehat{y}_i^{(t-1)}) + \lambda} - \frac{\left(\sum_{i \in I_T} g(y_i, \widehat{y}_i^{(t-1)})\right)^2}{\sum_{i \in I_T} h(y_i, \widehat{y}_i^{(t-1)}) + \lambda}\right). \quad (6)$$

Exact greedy algorithm enumerates over all possible nodes and features in order to find the best loss difference. (i.e., the quantity in eq (6))

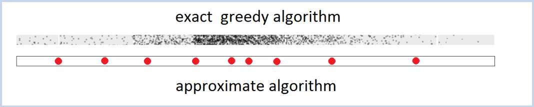
The XGBoost algorithm (8)

Exact greedy algorithm can take a ton of time searching values to split a feature. Therefore, XGBoost uses an approximation algorithm instead when the training data size is large.

5 **Approximate algorithm**.

I'm not going to talk about the details here (those who are interested can read the appendix A of [CG16]). The idea is instead of looking for all possible values of a feature to split, some calculations are done first to find a few good splitting values, and only split on these values.

Example.



The XGBoost algorithm (9)

6. **Handling missing values**. When the training data set contains lots missing values, a default direction is assigned. For each data point, when the features needed for the split are missing, it will follow the default direction.
7. Other features about system computation details (parallel computation, cache, . . .) are not discussed here.

Some suggestions when using XGBoost

- XGBoost is an algorithm for numeric features, not categorical features.
- Categorical features can be encoded in numeric features with many different ways (Say tf-idf, one-hot, lexical). If the categorical feature has orders, we better encode this feature with number indicating its levels. (Say Perfect $= 1$, Good $= 2$, Soso $= 3$, Bad $= 4$).
- In most cases, when you add more features to the model (XGBoost will figure out itself what features are better features), the cross validation score gets better. But you have to do feature engineering carefully (say throwing out some features) if you want your model to fit other test sets well.

Parameter tuning

Parameter tuning

In this section I'll talk about the major parameters of XGBoost, and how we tune it in python. More resources can be found in:
XGBoost parameter documentation page
http://xgboost.readthedocs.io/en/latest/parameter.html
Python XGBoost package introduction
https://xgboost.readthedocs.io/en/latest/python/

Getting started

There're two different ways to tune parameters in Python. **The first approach (recommended!** DMatrix is a internal data structure that used by XGBoost which is optimized for both memory efficiency and training speed):

import xgboost as xgb
xgb.DMatrix(parameters) # datatype conversion
model = xgb.train(parameters) # for both classification and regression

The second approach (Scikit-Learn wrapper interface):

import xgboost as xgb
clf = xgb.XGBClassifier(parameters) # for classification
reg = xgb.XGBRegressor(parameters) # for regression

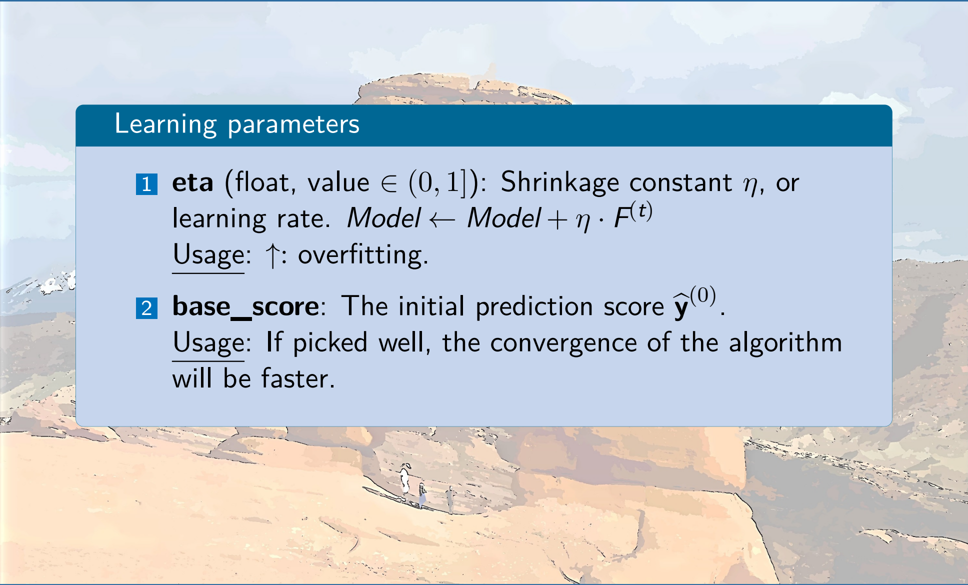
Category of parameters

Let's classify the major parameters into a few groups:

1. **Tree parameters**. num_boost_round, max_depth, min_child_weight
2. **Learning parameters**. eta, base_score
3. **Regularization parameters**. gamma, alpha, lambda
4. **Randomness parameters**. subsample, colsample_bytree, colsample_bylevel, seed
5. **Imbalanced dataset parameters**. scale_pos_weight, max_delta_step

Tree parameters

1. **num_boost_round** (int, value $\in \{1, 2, 3, \ldots\}$): number of boosted trees to be used (number of iterations).
   Usage: increase as training set size increases. $\uparrow$: overfitting.
2. **max_depth** (int, value $\in \{1, 2, 3, \ldots\}$): the maximum depth for each tree.
   Usage: increase as training set size increases. $\uparrow$: overfitting.
3. **min_child_weight** (int, value $\in \{1, 2, 3, \ldots\}$): minimum weight for every leaf.
   Usage: $\downarrow$: overfitting.



Learning parameters

1. **eta** (float, value $\in (0,1]$): Shrinkage constant η, or learning rate. $Model \leftarrow Model + \eta \cdot F^{(t)}$
   Usage: $\uparrow$: overfitting.
2. **base_score**: The initial prediction score $\widehat{\mathbf{y}}^{(0)}$.
   Usage: If picked well, the convergence of the algorithm will be faster.

Regularization parameters

The regularization term added to loss function is given by (in [CG16], $\alpha := 0$)

$$\gamma T + \alpha \sum_{j=1}^{T} |w_j| + \frac{1}{2}\lambda \sum_{k=1}^{T} w_k^2 \tag{7}$$

1. **gamma** (float, value $\in [0, \infty)$): γ in eq (7)
   Usage: $\downarrow$: overfitting.
2. **alpha**: (float, value $\in [0, \infty)$): α in eq (7).
   Usage: $\downarrow$: overfitting.
3. **lambda** (float, value $\in [0, \infty)$): λ in eq (7).
   Usage: $\downarrow$: overfitting.

Randomness parameters

1. **seed** (int, value $\in \{1, 2, 3, \ldots\}$): Random seed.
2. **subsample** (float, value $\in (0, 1]$): Proportion of training data used to build prediction models.
   Usage: Suggested values: 0.5-1.
3. **colsample_bytree** (float, value $\in (0, 1]$): subsample ratio of columns when constructing each tree.
   Usage: Suggested values: 0.5-1.
4. **colsample_bylevel** (float, value $\in (0, 1]$): subsample ratio of columns for each split, in each level.
   Usage: Suggested values: 0.5-1.

Imbalanced dataset parameters

These two parameters are used in imbalanced datasets, say in a binary classification problem, 95% of the training data are labeled '1' and the rest 5% are labeled '0'.

1. **scale_pos_weight** (float, value $\in [0, \infty)$): Control the balance of positive and negative weights.
Usage: When the evaluation metric is 'AUC' (area under ROC curve), scale_pos_weight can be tuned to

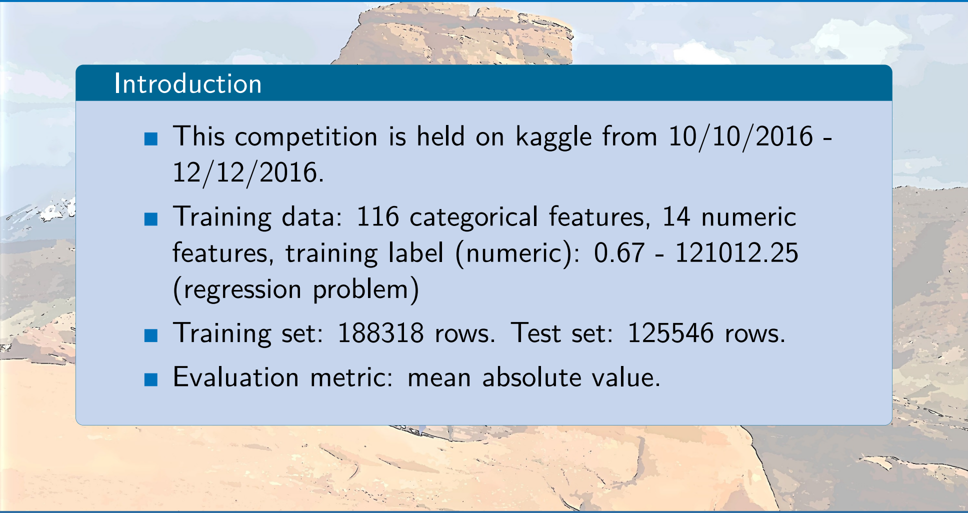
$$\frac{\text{number of 0s}}{\text{number of 1s}}$$

2. **max_delta_step** (float, value $\in (0, \infty)$): Maximum delta step we allow each tree weight estimation to be.
Usage: If the goal is to predict the right probability, we can set max_delta_step to a finite number, say 1-10, to limit the growth for each iteration step.

Some advice for parameter tuning

1. Set ‘eta’ to 0.1-0.2. Pick an appropriate value for ‘base_score’. (0.5 for binary classification, average value for regression)
2. Adjust ‘eta’ such that the cross validation score hits minimum at 200 rounds at least.
3. Tune tree parameters and regularization parameters. When good parameters are found, lower ‘eta’.
4. Tune randomness parameters.
5. There’re other useful parameters not discussed here, say to allow users to define their own loss function, set up criterion for early stopping, etc.

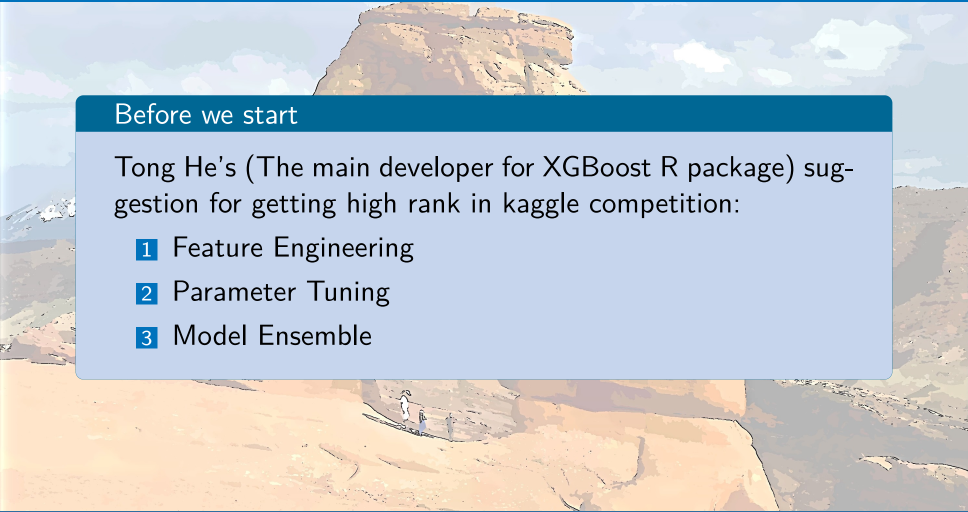
Kaggle competition: Allstate Claims Severity



Introduction

- This competition is held on kaggle from 10/10/2016 - 12/12/2016.
- Training data: 116 categorical features, 14 numeric features, training label (numeric): 0.67 - 121012.25 (regression problem)
- Training set: 188318 rows. Test set: 125546 rows.
- Evaluation metric: mean absolute value.

Kaggle competition: Allstate Claims Severity



Before we start

Tong He's (The main developer for XGBoost R package) suggestion for getting high rank in kaggle competition:

1. Feature Engineering
2. Parameter Tuning
3. Model Ensemble

Kaggle competition: Allstate Claims Severity

Issues

- How to deal with categorical variables?
- The target loss function 'mean absolute value'

$$L(\mathbf{y}, \widehat{\mathbf{y}}) := \frac{\sum_{i=1}^{n} |y_i - \widehat{y}_i|}{n}$$

is not differentiable when any $y_i = \widehat{y}_i$. And the second derivatives are zero everywhere except where L can't be differentiated. (Non-smoothness means horrible approximation with gradient decent)

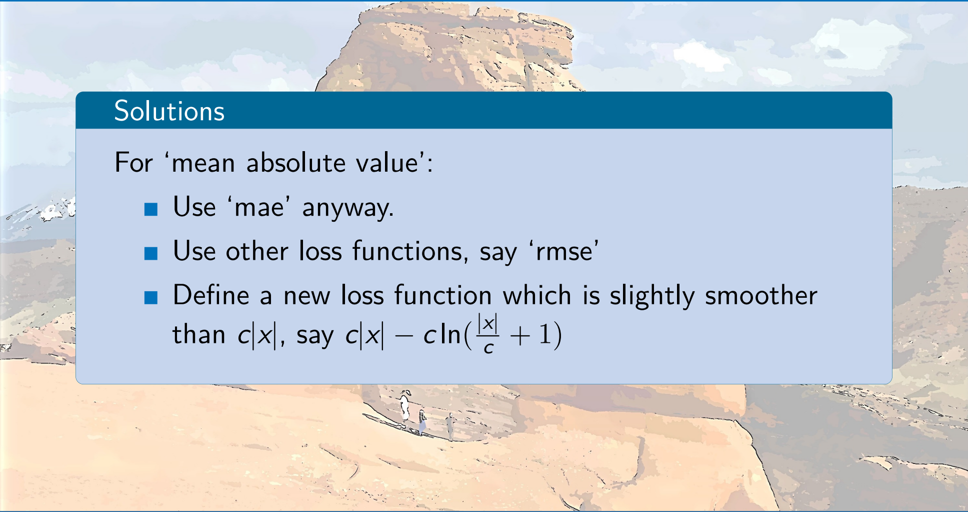
Kaggle competition: Allstate Claims Severity

Solutions

Tianqi Chen's comments: Xgboost treat every input feature as numerical, with support for missing values and sparsity. So if you want ordered variables, you can transform the variables into **numerical levels** (say age). Or if you prefer treating it as categorical variable, do **one hot encoding**.

numerical levels (lexical encoding): 'A', 'B', 'C' $\rightarrow$ 1, 2, 3 (pandas.facorize)
one hot encoding: 'A', 'B', 'C' $\rightarrow$ [1,0,0], [0,1,0], [0,0,1] (pandas.get_dummies)

Kaggle competition: Allstate Claims Severity



Solutions

For ‘mean absolute value’:

- Use ‘mae’ anyway.
- Use other loss functions, say ‘rmse’
- Define a new loss function which is slightly smoother than $c|x|$, say $c|x| - c\ln(\frac{|x|}{c} + 1)$

Kaggle competition: Allstate Claims Severity

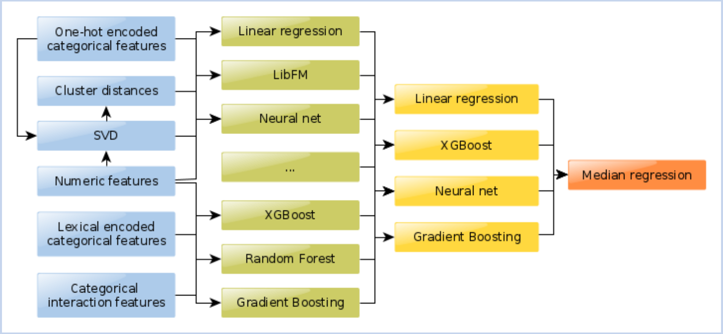
2nd place olutions

- Used Xgbfi to do feature selection
- Define a new loss function instead of ‘mae’
- Applied log(y + 200) transformation to ’loss’, to make the distribution like normal distribution
- Combined various predictors (XGBoost, Neural Networks, and others) with 3-level stacking (figure in the next page)

Kaggle competition: Allstate Claims Severity

2nd place olutions

[The figure below is taken from kaggle official blog.]



Kaggle competition: Allstate Claims Severity

Demo

In the very end I would like to do a quick demo on Allstate Claims Severity dataset with only XGBoost. This is an easy predictor which

- lexical encoding done to all categorical features.
- applied $\log(y + 200)$ transformation to 'loss'
- No feature engineering. No stacking. Only parameter tuning.
- This naive version could get you into top 30% on Kaggle private leader board

References

- Tianqi Chen and Carlos Guestrin (2016), Xgboost: A scalable tree boosting system, (2016), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 13-17, 2016, pages 785-794.*
- XGBoost documentation: http://xgboost.readthedocs.io/
- Kaggle contest page: https://www.kaggle.com/c/allstate-claims-severity/
- Kaggle official blog: http://blog.kaggle.com/