


# XGBoost: algorithm introduction, parameter tuning, and applications

Shiu-Tang Li

Big Data Utah Meetup

Jun 14, 2017

# Outline

- 
- 1 Introduction**
    - Introduction
  - 2 Algorithm**
    - Fundamentals
    - The XGBoost algorithm
  - 3 Parameter tuning**
    - Parameter tuning
  - 4 Applications**
    - Applications

# Introduction

## What's XGBoost?

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

# 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 for training sets and labels

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

# Loss function: measures how good your prediction is

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

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

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

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

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

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

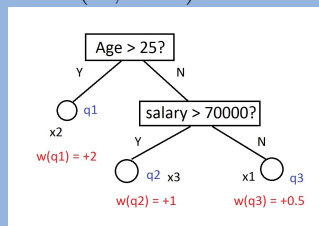
We can think of a tree as a function  $F$  of two composite functions  $w \circ q$ , where  $q : \mathbb{R}^d \rightarrow \{q_1, \dots, q_T\}$  maps data points that contains  $d$  features to tree leaves  $\{q_1, \dots, q_T\}$ , and  $w : \{q_1, \dots, q_T\} \rightarrow \mathbb{R}$  is a weight function 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))

$$x^{(1)} = (23, 45000)$$

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

$$x^{(3)} = (22, 80000)$$



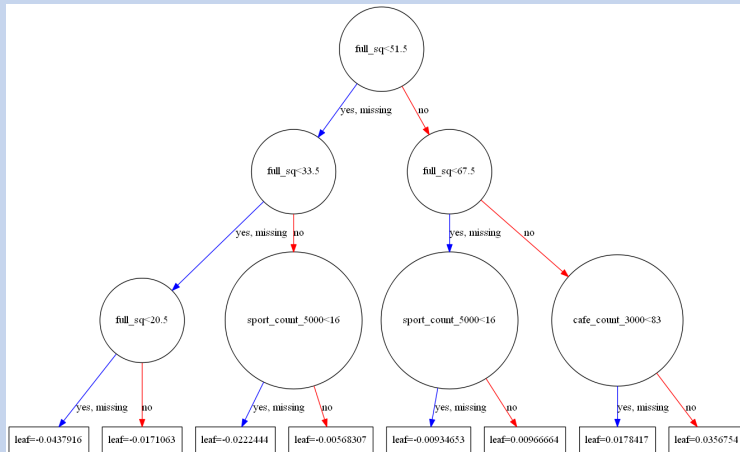
$$F(x^{(1)}) = w \circ q(x^{(1)}) = 0.5$$

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

$$F(x^{(3)}) = w \circ q(x^{(3)}) = 1$$



# Take a glance at XGBoost classification tree



# Decision tree / Random forests / Tree boosting algorithms: what's the difference?

## Decision Tree

Build a single tree. If ID3 algorithm is used, each node is split in a way that information gain is maximized.



# Decision tree / Random forests / Tree boosting algorithms: what's the difference?

## Random forest

Build a bunch of decision trees, for each decision tree, only a random proportion of features are used to split nodes. Then we take average of these trees.



# Decision tree / Random forests / Tree boosting algorithms: what's the difference?

## Tree boosting algorithms

build one tree in each round. The trees built in early rounds usually have larger weights and larger sizes.



## Tree boosting algorithm (1)

- 1 Begin with an initial tree  $F(\mathbf{x}, \theta = \theta_0)$  and an initial estimate  $\hat{\mathbf{y}}^{(0)} \equiv c$  (a constant). Initialize the model by  $Model \leftarrow \hat{\mathbf{y}}^{(0)}$ .  $\theta$  are the parameters (e.g. weights, height) to be tuned for the tree.
- 2 Calculate the loss function  $L(\mathbf{y}, \hat{\mathbf{y}}^{(0)}) = \sum_{i=1}^n l(y_i, \hat{y}_i^{(0)})$ .
- 3 By performing several node splits and assigning different weights to  $F$ , we have different choices of  $\theta$ . We pick  $\theta = \hat{\theta}$  such that

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

## Tree boosting algorithm (2)

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

- 5 When the first iteration is done, the tree we get is denoted by  $F^{(1)}(\mathbf{x}) = F(\mathbf{x}, \hat{\boldsymbol{\theta}})$ .
- 6 Update our prediction  $\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + F^{(1)}(\mathbf{x}^{(i)})$  for all  $1 \leq i \leq n$  and add this tree to the model by  $Model \leftarrow Model + F^{(1)}$ . The new loss becomes

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

- 7 Repeat all the previous steps until the user stops the algorithm.

## The XGBoost algorithm (1)

Let's look at the main features of XGBoost that 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:

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

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.

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

$$Model \leftarrow Model + \eta \cdot F^{(t)}$$

- 3 Column subsampling.** For each iteration step, when growing a new tree, a portion of features are randomly selected from  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$  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 be approximated by

$$\begin{aligned} & \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + F(\mathbf{x}^{(i)})) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ & \approx \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)}) + \sum_{i=1}^n g(y_i, \hat{y}_i^{(t-1)}) \cdot F(\mathbf{x}^{(i)}) \\ & \quad + \frac{1}{2} \sum_{i=1}^n h(y_i, \hat{y}_i^{(t-1)}) \cdot (F(\mathbf{x}^{(i)}))^2 + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2. \end{aligned} \quad (1)$$

(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)

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

Let  $I_j := \{i : q(\mathbf{x}_i) = q_j\}$ , the set of subscripts of  $\mathbf{x}_i$  that are mapped to tree leaf  $q_j$ . Therefore,  $F(\mathbf{x}^{(i)}) = w_j$  if  $i \in I_j$ , and we can rewrite (1) as

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

which is the sum of  $T$  quadratic equations.

## To minimize the sum of quadratic equations... (1)

Let  $f(w) = aw^2 + bw + c$ . To make sure the minimum exists,  $a$  must be  $> 0$ . In this case,  $f(w)$  gets minimized when  $w = \frac{-b}{2a}$ , and the minimum is given by  $\frac{4ac - b^2}{4a}$ .

To minimize the sum of quadratic equations  $f(w_1, \dots, w_n) = \sum_{i=1}^n (a_i w_i^2 + b_i w_i + c_i)$ , you have to make sure  $a_i > 0$  for all  $i = 1, \dots, n$ . The minimum for  $f(w_1, \dots, w_n)$  is then  $\sum_{i=1}^n \frac{4a_i c_i - b_i^2}{4a_i}$ .

## To minimize the sum of quadratic equations... (2)

Recall that in equation (2), the coefficient for  $w_j^2$  ( $1 \leq j \leq T$ ) is  $\frac{1}{2} \sum_{i \in I_j} h(y_i, \hat{y}_i^{(t-1)}) + \frac{1}{2} \lambda$  (here  $h(z_1, z_2) = \frac{\partial^2}{\partial z_2^2} l(z_1, z_2)$ ). To guarantee the existence of minimum, these coefficients have to be greater than zero.

So there is one important assumption that we have to make on the loss function  $l$ :  $l$  has to be a convex function! (So the 2nd partial derivatives of a convex function will be positive). You can't use an arbitrary function as the loss function!

## The XGBoost algorithm (5)

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

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

and the minimum for the loss function (with weight  $w_j^*$ 's) is given by

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

## The XGBoost algorithm (6)

Without loss of generality, let's split last tree leaf  $q_T$  into two nodes  $q(L)$  and  $q(R)$ , and thus  $I_T = I(L) \cup I(R)$  (Here  $I(L) := \{i : q(\mathbf{x}_i) = q(L)\}$ ,  $I(R) := \{i : q(\mathbf{x}_i) = q(R)\}$ ). The optimal loss for the tree after we split the leaf becomes

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

## The XGBoost algorithm (7)

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

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

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

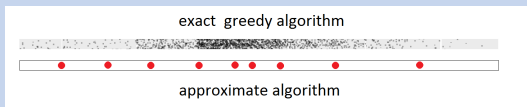
## The XGBoost algorithm (8)

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 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).
- 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 with python

There're two ways to build XGBoost predictive models.

### Approach 1. Convert data into DMatrix (recommended)

```
import xgboost as xgb
import pandas as pd

train = pd.read_csv('train.csv')
y_train = train['prediction']
del train['prediction']
DM_train = xgb.DMatrix(data=train, label=y_train)

model = xgb.train(dtrain=DM_train, params=parameters)
# 'parameters' is a dictionary containing parameters of the model
# prediction
test = pd.read_csv('test.csv')
DM_test = xgb.DMatrix(data=test)
y_test = model.predict(DM_test)
```

(DMatrix is an internal data structure used by XGBoost which is optimized for both memory efficiency and training speed)

## Getting started with python

### Approach 2. Scikit-Learn wrapper interface

```
import xgboost as xgb
import pandas as pd

train = pd.read_csv('train.csv')
test = pd.read_csv('train.csv')
y_train = train['prediction']
del train['prediction']

# For regression:
reg = xgb.XGBRegressor(learning_rate=0.05, gamma=0)
                                # There're more parameter options
reg.fit(train, y_train)
y_test = reg.predict(test)

# For classification:
clf = xgb.XGBClassifier(learning_rate=0.05, gamma=0)
```

Advantage: you can use it like using other sklearn predictors

## Category of parameters

From now on I'll focus on Approach 1 (The parameter names for two approaches are slightly different). The major parameters are classified 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

## Tree parameters

- 1 **num\_boost\_round** (int, value  $\in \{1, 2, 3, \dots\}$ ):  
number of boosted trees to be used (number of iterations).
- 2 **max\_depth** (int, value  $\in \{1, 2, 3, \dots\}$ ): the maximum depth for each tree.  
Usage: To avoid overfitting.
- 3 **min\_child\_weight** (int, value  $\in \{1, 2, 3, \dots\}$ ):  
minimum weight for every leaf.  
Usage: To avoid overfitting.

## Learning parameters

- 1 eta** (float, value  $\in (0, 1]$ ): Shrinkage constant  $\eta$ , or learning rate.  $Model \leftarrow Model + \eta \cdot F^{(t)}$   
Usage: pick smaller  $\eta$  to avoid overfitting.
- 2 base\_score**: The initial prediction score  $\hat{y}^{(0)}$ .  
Usage: To reduce training time.



## Regularization parameters

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

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

- 1 gamma** (float, value  $\in [0, \infty)$ ):  $\gamma$  in eq (7)

Usage: To avoid overfitting by reducing the complexity of trees built.

- 2 alpha**: (float, value  $\in [0, \infty)$ ):  $\alpha$  in eq (7).

Usage: To avoid overfitting.

- 3 lambda** (float, value  $\in [0, \infty)$ ):  $\lambda$  in eq (7).

Usage: To avoid overfitting.

## Randomness parameters

- 1 **seed** (int, value  $\in \{1, 2, 3, \dots\}$ ): Random seed.
- 2 **subsample** (float, value  $\in (0, 1]$ ): Proportion of training data used to build prediction models.  
Usage: To avoid overfitting. Suggested values: 0.6-1.
- 3 **colsample\_bytree** (float, value  $\in (0, 1]$ ): subsample ratio of columns when constructing each tree.  
Usage: To avoid overfitting. Suggested values: 0.6-1.
- 4 **colsample\_bylevel** (float, value  $\in (0, 1]$ ): subsample ratio of columns for each split, in each level.  
Usage: To avoid overfitting. Suggested values: 0.6-1.

## Some advice for parameter tuning

- 1 Set eta to 0.005-0.2. Pick an appropriate value for base\_score. (0.5 for binary classification, average value for regression)
- 2 Tune max\_depth. First try 3-10.
- 3 Tune regularization parameters and other tree parameters.
- 4 Tune randomness parameters.
- 5 An automatic XGBoost parameter tuner can be found in my github:  
[https://github.com/Shiutang-Li/Auto\\_XGBoost/](https://github.com/Shiutang-Li/Auto_XGBoost/)

# Applications

## Build an XGBoost predictive model

In this section, I'll demonstrate the key steps to build a predictive model with XGBoost.

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

# Applications

## Example data set: Kaggle competition - Allstate Claims Severity

- 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.

# Applications

## Build an XGBoost predictive model

- 1 Feature selection / Creating new features / Outlier removal / Data cleaning (omitted)
- 2 Convert all categorical features to numerical features
- 3 Parameter Tuning and Modeling building
- 4 Model Ensemble (omitted)

# Applications

## Convert all categorical features to numerical features

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**.

**lexical encoding:** 'rainy', 'sunny', 'cloudy'  $\rightarrow$  1, 2, 3  
(pandas.factorize)

**one hot encoding:** 'A', 'B', 'C'  $\rightarrow$  [1,0,0], [0,1,0], [0,0,1]  
(pandas.get\_dummies)

# Applications

## My suggestions on categorical features conversion (1)

- If the categorical feature has orders, we better encode this feature with numerical levels. (Say Perfect = 1, Good = 2, Soso = 3, Bad = 4).
- Why? Because every time XGBoost splits a node, it's looking for a feature  $F$  and a threshold value  $k$  and the data are split into two parts:  $\{F > k\}$  and  $\{F \leq k\}$ . Numerical level encoding ensures data points with higher levels go to one cluster and the other low level data points go to the other cluster.



## My suggestions on categorical features conversion (2)

- If the categorical feature does not have any orders, but it contains very few distinct values (Say only 2 or 3 distinct values), then one hot encoding is appropriate. It's a bad idea to use one hot encoding if there're too many distinct values for that feature.
- Why? Every time XGBoost splits a node, only one feature is selected. If a categorical feature has 30 distinct values, then a single split can only separate one value from the other 29 values, which is very inefficient, and each one hot encoded feature is usually not informative. As a result, XGBoost tends not to select these features for node splitting.

## My suggestions on categorical features conversion (3)

- In an NLP problem, if we're using word importance as a feature, then it is a good idea to use tf-idf encoding.
- the idea of tf-idf is counting the key word frequency in current document multiplied by the log inverse frequency that this key word appears in other documents or not.

## My suggestions on categorical features conversion (4)

- For a regression problem, group mean is another choice.
- Say a categorical feature  $F$  has 3 distinct values  $A$ ,  $B$ ,  $C$ , the average values of the training labels with  $F = A$ ,  $F = B$ ,  $F = C$  are 10.3, 8.2, 9.7, then we can replace  $A$  with 10.3,  $B$  with 8.2,  $C$  with 9.7.
- This is not a perfect approach but it's better than just randomly encoding categorical feature as 1, 2, 3, ...

# Applications

## Two choices to tune parameters

For XGBoost, there're two choices to tune parameters. The first approach is applying `xgb.cv()` to do cross validations, and then use `xgb.train()` to build models with the best parameters found in previous step.

The second approach is separating the training set into a training set plus a validation set, to build a watchlist and decide when to stop training.

I'll later demonstrate the details in the demo file.

# Applications

## GPU accelerated XGBoost

- GPU accelerated XGBoost is now available in [https://github.com/dmlc/xgboost/tree/master/plugin/updater\\_gpu](https://github.com/dmlc/xgboost/tree/master/plugin/updater_gpu), and the development team are currently adding more features to GPU accelerated XGBoost.
- Advantage: 2X - 10X faster than ordinary XGBoost
- Drawback: may still contain bugs because this is pretty new tool. Prediction results are not reproducible (therefore not a perfect choice for kaggle competitions).
- I'll later demonstrate how to use it in the demo file.

# Applications

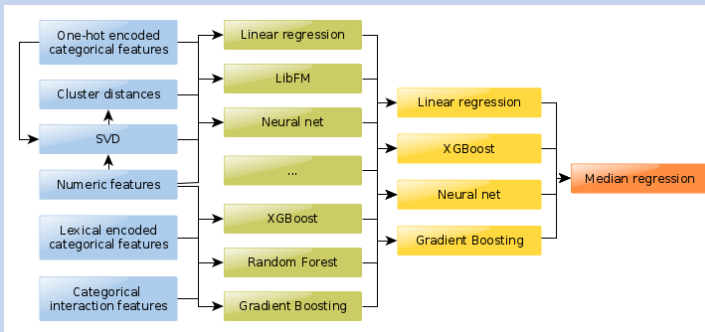
## 2nd place solution

- Feature selection
- Define a new loss function instead of 'mae'
- Applied  $\log(y + 200)$  transformation to labels, to make the distribution like normal distribution
- Combined various predictors (XGBoost, Neural Networks, and others) with 3-level stacking (figure in the next page)

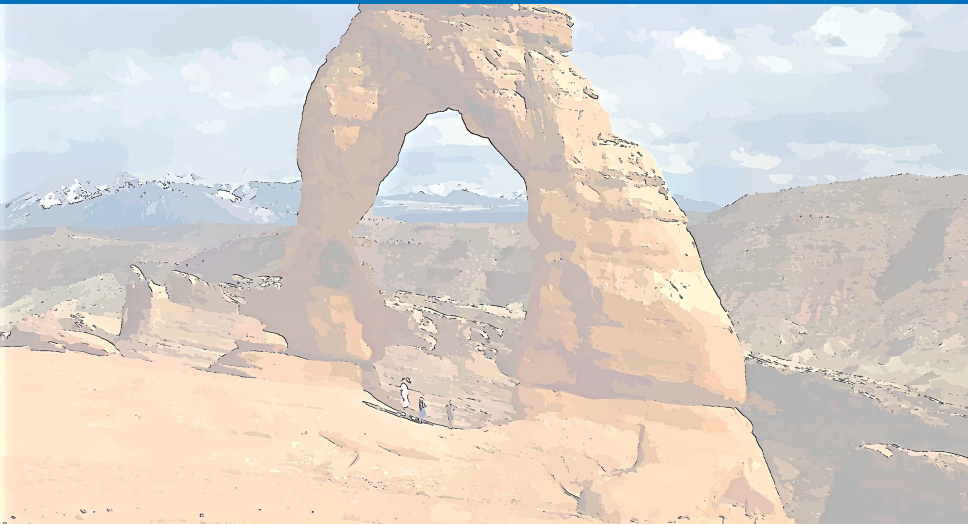
# Applications

## 2nd place solution

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



# Demo





## 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/>

# Job wanted :)

- Currently looking for full-time position /internship opportunities as a data scientist /data analyst /statistician in Utah
- Connect me on linkedin or email me:  
<https://www.linkedin.com/in/shiutang-li-64b1a885/>  
stazlee (at) hotmail.com
- I need a stage to rock n roll with your team