Extreme Gradient Boosting [Scalable tree based ensemble approach] in R

Kamal Mishra
Data Science and AI Practitioner
Aug-2019

Agenda

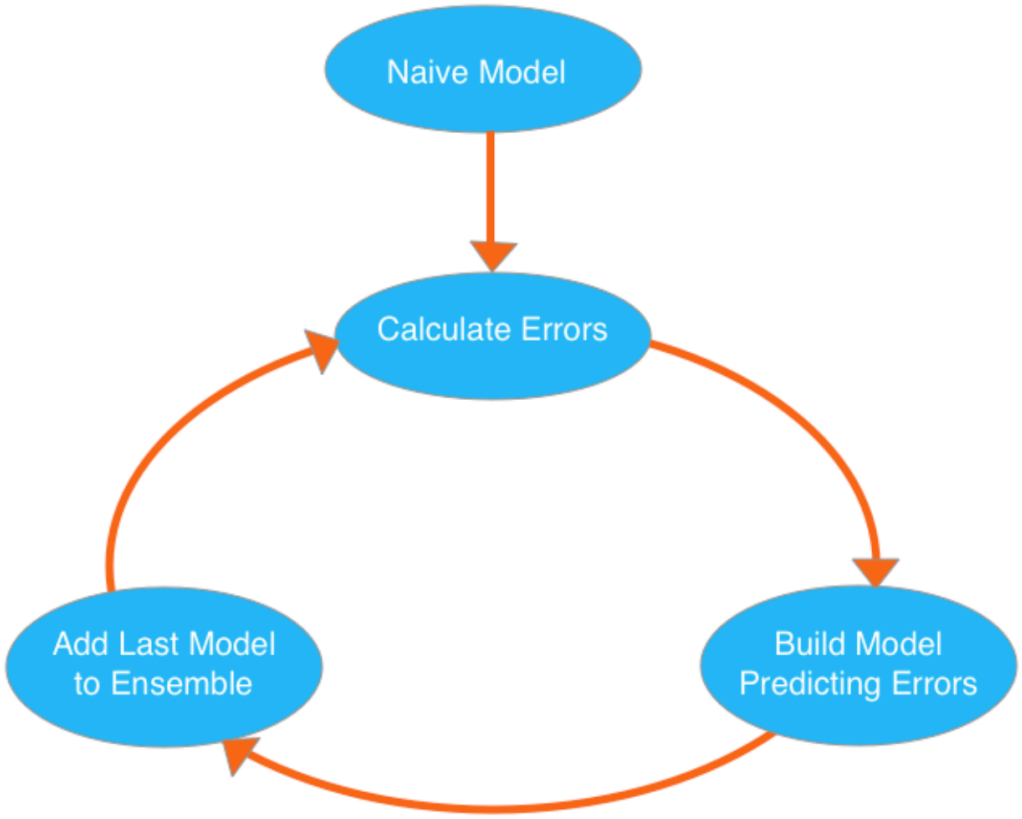
- Background and Introduction
- Key concepts around Supervised Learning
- Regression Tree and Ensemble
- Gradient Boosting based Learning
- Unique features of XGBoost
- References
- Appendix: Sample Code snippet of using XGBoost in R using simple IRIS dataset

Background and Introduction

Introduction

- Introduced in 2014
- e**X**treme **G**radient **Boost**ing (XGBoost)
- From it's inception, it has been used in many machine learning competitions, hackathons etc. to win from measurement perspective and has been the holy grail in most scenarios for most users. It has been proving it's mettle in terms of speed and performance both.
- Tianqi Chen and Carlos came up with the approach in 2016 as part of Univ of Washington research

What is XGBoost – how it works



- Works with standard tabular data, as opposed to more exotic types of data like images and videos.
- To reach peak accuracy, XGBoost models require more knowledge and model tuning than techniques like Random Forest.
- It goes through cycles that repeatedly builds new models and combines them into an ensemble model.
- Starts the cycle by calculating the errors for each observation in the dataset. We then build a new model to predict those. We add predictions from this error-predicting model to the "ensemble of models."
- To make a prediction, we add the predictions from all previous models. We can use these predictions to calculate new errors, build the next model, and add it to the ensemble.
- There's one piece outside that cycle. We need some base prediction to start the cycle. In practice, the initial predictions can be pretty naive. Even if it's predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.

Key Concepts around Supervised Learning

Elements used in Supervised Learning

- Notations: $x_i \in \mathbf{R}^d$ i-th training example
- **Model**: how to make prediction $\hat{y}_i$ given x_i
  - Linear model: $\hat{y}_i = \sum_j w_j x_{ij}$ (include linear/logistic regression)
  - The prediction score $\hat{y}_i$ can have different interpretations depending on the task
    - Linear regression: $\hat{y}_i$ is the predicted score
    - Logistic regression: $1/(1 + exp(-\hat{y}_i))$ is predicted the probability of the instance being positive
    - Others… for example in ranking $\hat{y}_i$ can be the rank score
- **Parameters**: the things we need to learn from data
  - Linear model: $\Theta = \{w_j | j = 1, \cdots, d\}$

Objective Function in the context

- Objective function that is everywhere

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

Training Loss measures how well model fit on training data

Regularization, measures complexity of model

- Loss on training data: $L = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$
  - Square loss: $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
  - Logistic loss: $l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})$
- Regularization: how complicated the model is?
  - L2 norm: $\Omega(w) = \lambda \|w\|^2$
  - L1 norm (lasso): $\Omega(w) = \lambda \|w\|_1$

Different flavors for regularizing loss

- Ridge regression: $\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\|w\|^2$
  - Linear model, square loss, L2 regularization
- Lasso: $\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\|w\|_1$
  - Linear model, square loss, L1 regularization
- Logistic regression:

  $$\sum_{i=1}^{n}[y_i \ln(1 + e^{-w^T x_i}) + (1 - y_i)\ln(1 + e^{w^T x_i})] + \lambda\|w\|^2$$

  - Linear model, logistic loss, L2 regularization
- The conceptual separation between model, parameter, objective also gives you **engineering benefits**.
  - Think of how you can implement SGD for both ridge regression and logistic regression

Bias and Variance tradeoff

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

Training Loss measures how well model fit on training data

Regularization, measures complexity of model

- Why do we want to contain two component in the objective?
- Optimizing training loss encourages **predictive** models
  - Fitting well in training data at least get you close to training data which is hopefully close to the underlying distribution
- Optimizing regularization encourages **simple** models
  - Simpler models tends to have smaller variance in future predictions, making prediction **stable**

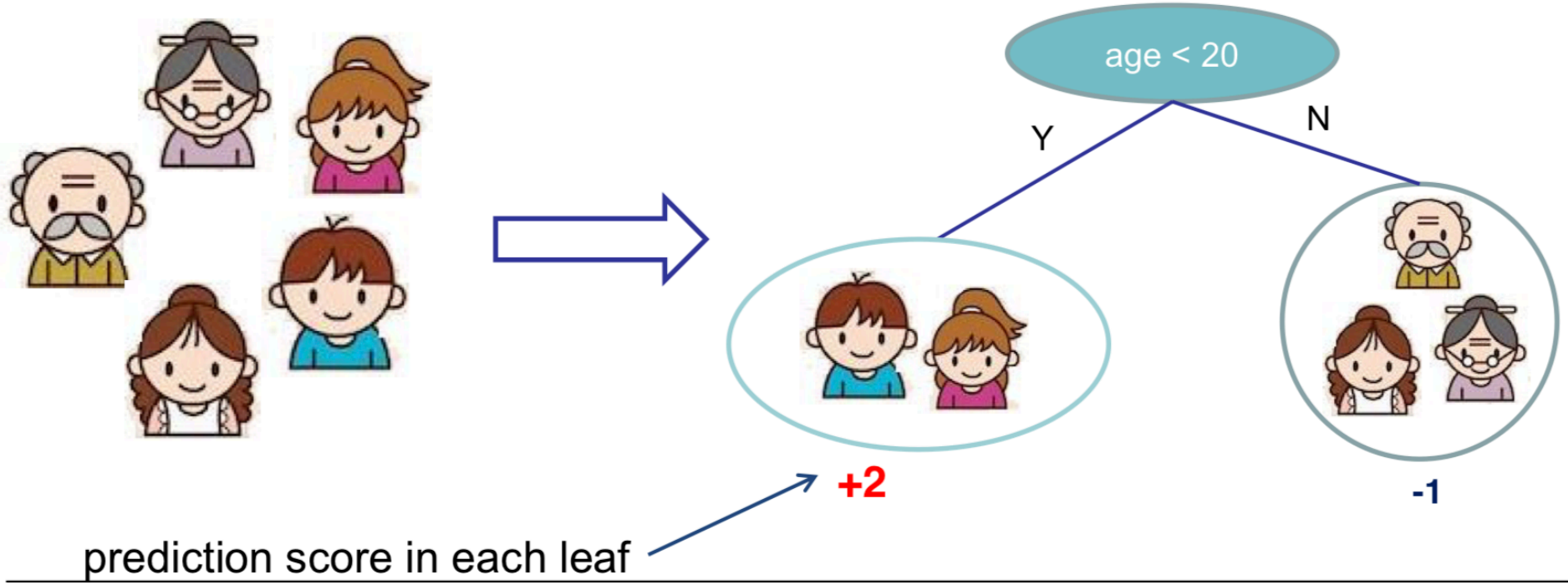
Regression Tree and Ensemble

CART – Classification and Regression Tree

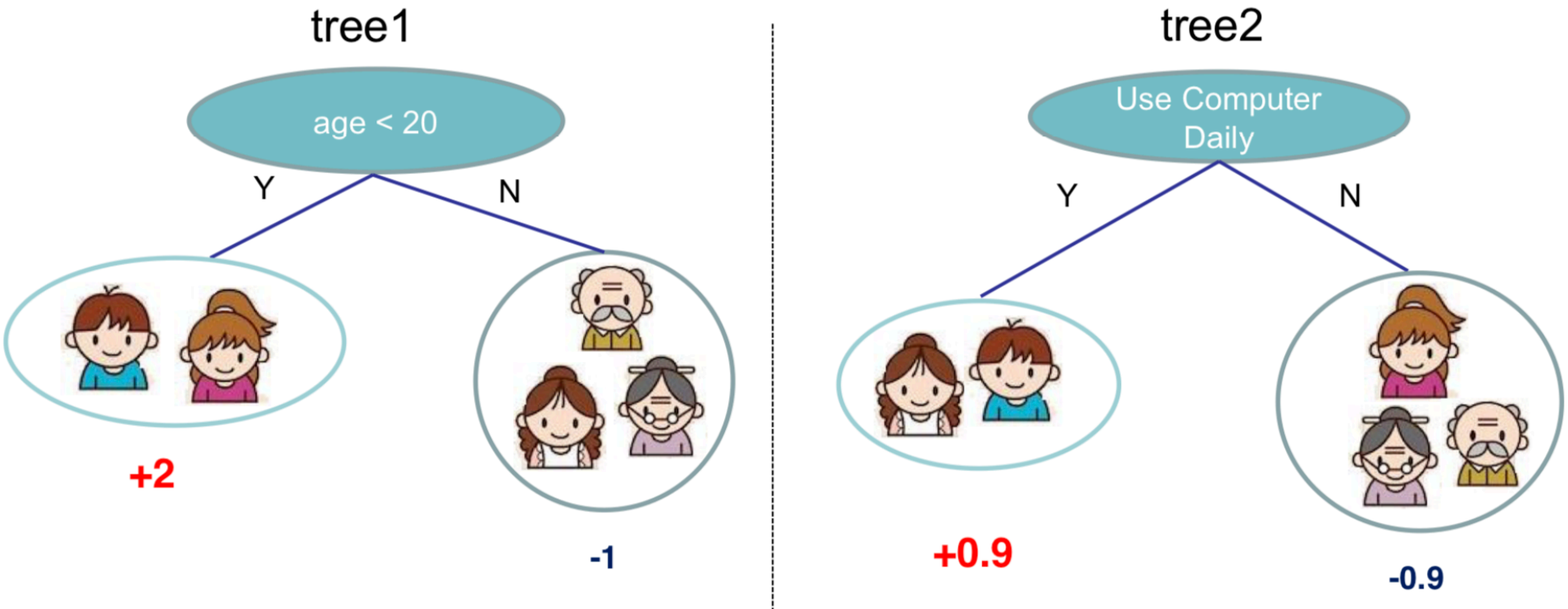
- regression tree (also known as classification and regression tree):
  - Decision rules same as in decision tree
  - Contains one score in each leaf value

Input: age, gender, occupation, …

Like the computer game X



Regression Tree Ensemble



f( ) = 2 + 0.9= 2.9 f( )= -1 - 0.9= -1.9

Prediction of is sum of scores predicted by each of the tree

Tree Ensemble Approaches

- Very widely used, look for GBM, random forest…
  - Almost half of data mining competition are won by using some variants of tree ensemble methods
- Invariant to scaling of inputs, so you do not need to do careful features normalization.
- Learn higher order interaction between features.
- Can be scalable, and are used in Industry

Model and Parameters

- Model: assuming we have K trees

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$$

Space of functions containing all Regression trees

Think: regression tree is a function that maps the attributes to the score

- Parameters
  - Including structure of each tree, and the score in the leaf
  - Or simply use function as parameters

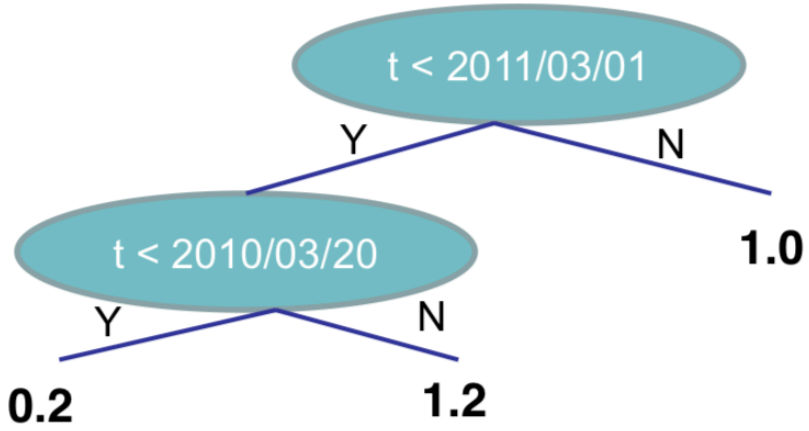
$$\Theta = \{f_1, f_2, \cdots, f_K\}$$

  - Instead learning weights in $\mathbf{R}^d$, we are learning functions(trees)

Learning Tree on a single variable

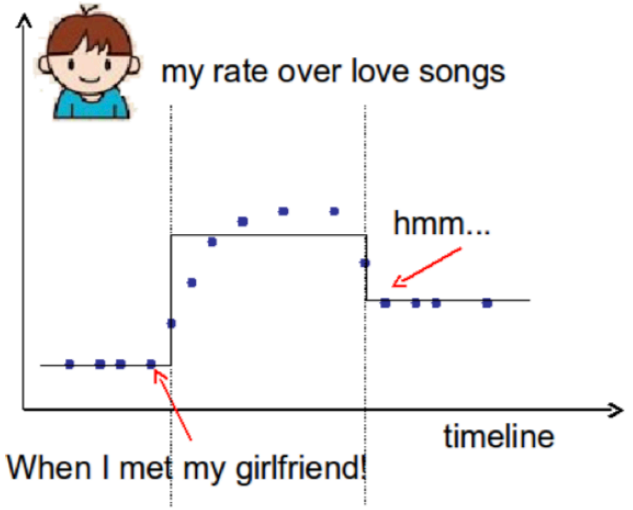
- How can we learn functions?
- Define objective (loss, regularization), and optimize it!!
- Example:
  - Consider regression tree on single input t (time)
  - I want to predict whether I like romantic music at time t

The model is regression tree that splits on time



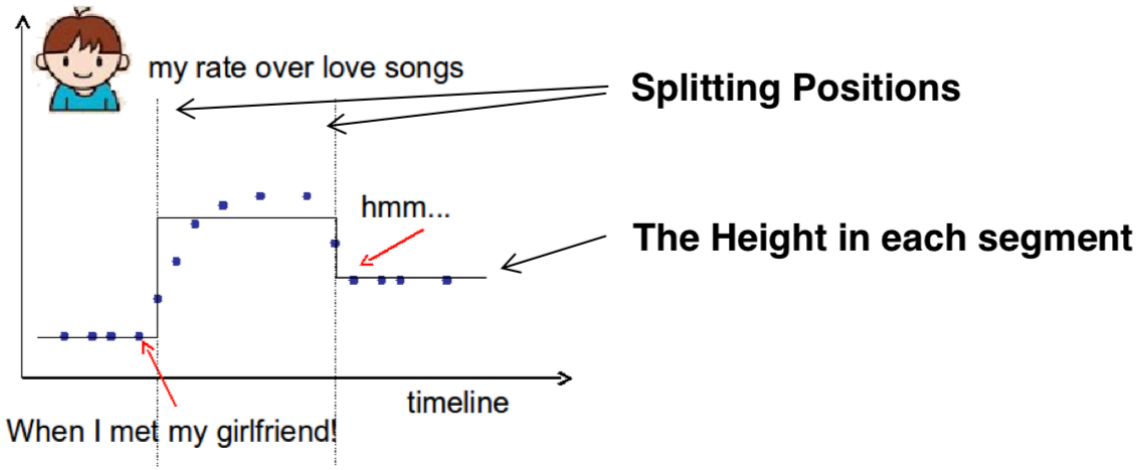
Equivalently

Piecewise step function over time



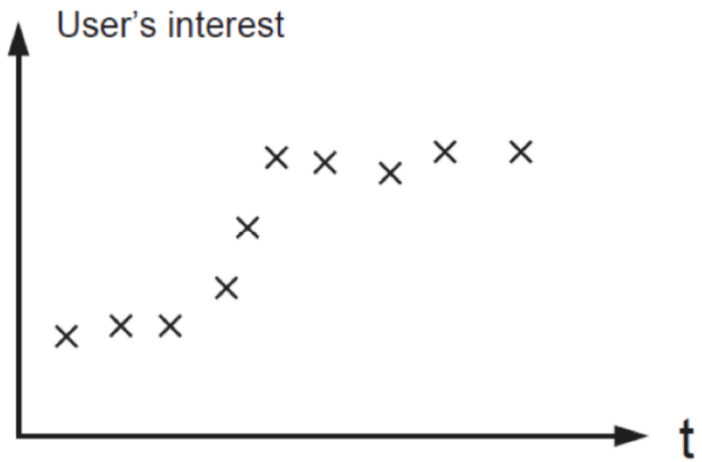
How to interprete the step function in a timeseries context

- Things we need to learn

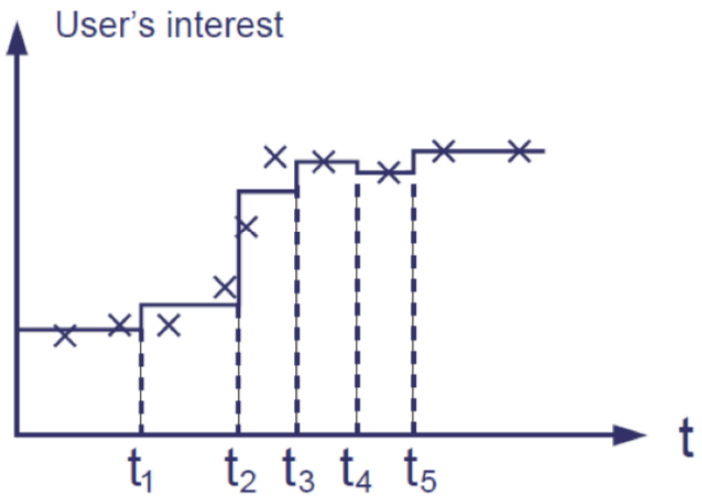


- Objective for single variable regression tree(step functions)
  - Training Loss: How will the function fit on the points?
  - Regularization: How do we define complexity of the function?
    - Number of splitting points, l2 norm of the height in each segment?

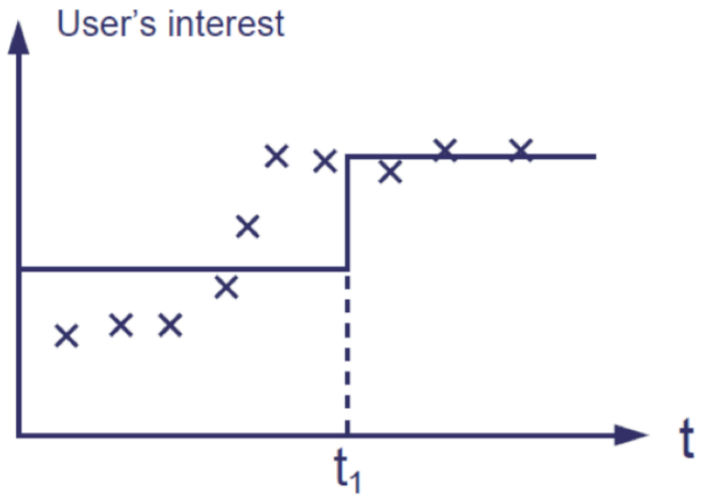
Learning the step function



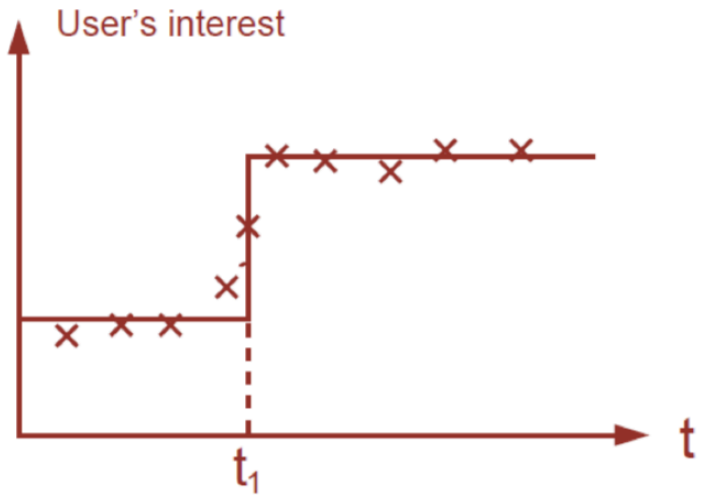
Observed user's interest on topic k against time t



☒ Too many splits, $\Omega(f)$ is high



☒ Wrong split point, $L(f)$ is high



☑ Good balance of $\Omega(f)$ and $L(f)$

Now, intent for tree ensemble..

- Model: assuming we have K trees

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$$

- Objective

$$Obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

Training loss

Complexity of the Trees

- Possible ways to define Ω ?
  - Number of nodes in the tree, depth
  - L2 norm of the leaf weights

Objective vs Heuristic way to look at it

- When you talk about (decision) trees, it is usually heuristics
  - Split by information gain
  - Prune the tree
  - Maximum depth
  - Smooth the leaf values
- Most heuristics maps well to objectives, taking the formal (objective) view let us know what we are learning
  - Information gain -> training loss
  - Pruning -> regularization defined by #nodes
  - Max depth -> constraint on the function space
  - Smoothing leaf values -> L2 regularization on leaf weights

Objective vs Heuristic way to look at it

- When you talk about (decision) trees, it is usually heuristics
  - Split by information gain
  - Prune the tree
  - Maximum depth
  - Smooth the leaf values
- Most heuristics maps well to objectives, taking the formal (objective) view let us know what we are learning
  - Information gain -> training loss
  - Pruning -> regularization defined by #nodes
  - Max depth -> constraint on the function space
  - Smoothing leaf values -> L2 regularization on leaf weights

Gradient Boosting based Learning

Overall message to look at..

- Bias-variance tradeoff is everywhere
- The loss + regularization objective pattern applies for regression tree learning (function learning)
- We want **predictive** and **simple** functions
- This defines what we want to learn (objective, model).
- But how do we learn it?

Adaptive Training based Learning

- Objective: $\sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_k \Omega(f_k), f_k \in \mathcal{F}$
- We can not use methods such as SGD, to find f (since they are trees, instead of just numerical vectors)
- Solution: **Additive Training (Boosting)**
  - Start from constant prediction, add a new function each time

$$
\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{aligned}
$$

New function

Model at training round t

Keep functions added in previous round

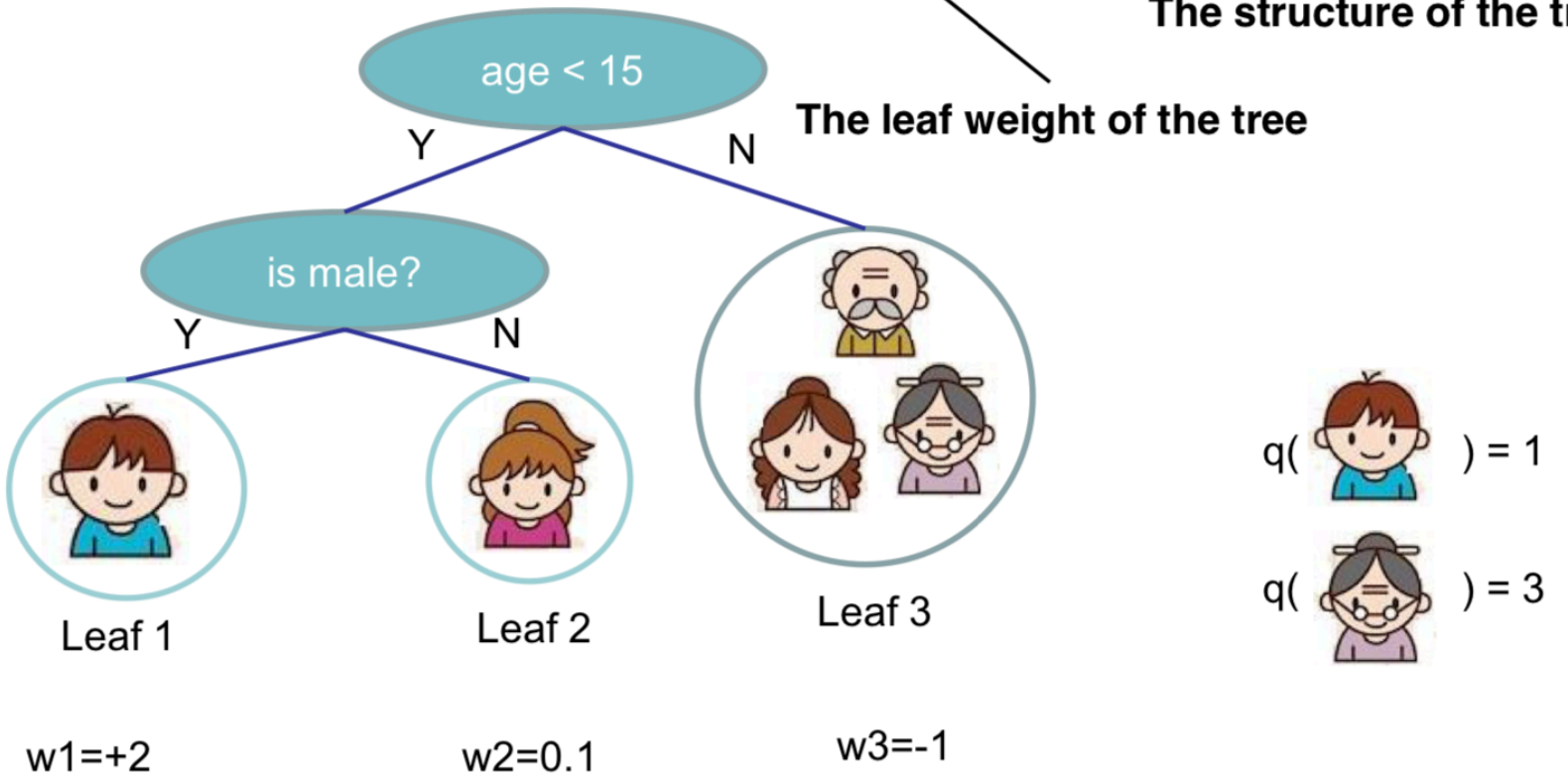
Refine the definition of tree

- We define tree by a vector of scores in leafs, and a leaf index mapping function that maps an instance to a leaf

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

The structure of the tree

The leaf weight of the tree

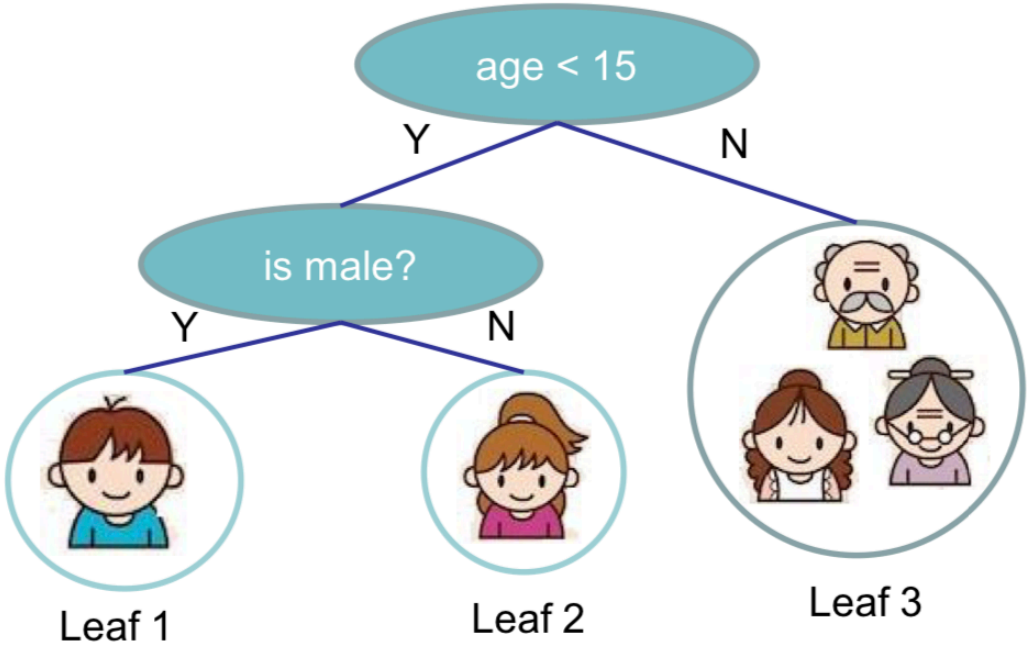


Complexity of tree - weights

- Define complexity as (this is not the only possible definition)

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

Number of leaves **L2 norm of leaf scores**



$$\Omega = \gamma 3 + \frac{1}{2}\lambda(4 + 0.01 + 1)$$

Unique features of XGBoost

Regularization

Option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting

Handling Sparse Data

Missing values or data processing steps like one-hot encoding make data sparse. XGBoost incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data

Weighted Quantile Sketch

Finding split points when the data points are of equal weights (using quantile sketch algorithm) is known. However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data

Block Structure

For faster computing, XGBoost can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted in blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling

Cache Awareness

In XGBoost, non-continuous memory access is required to get the gradient statistics by row index. Hence, XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored

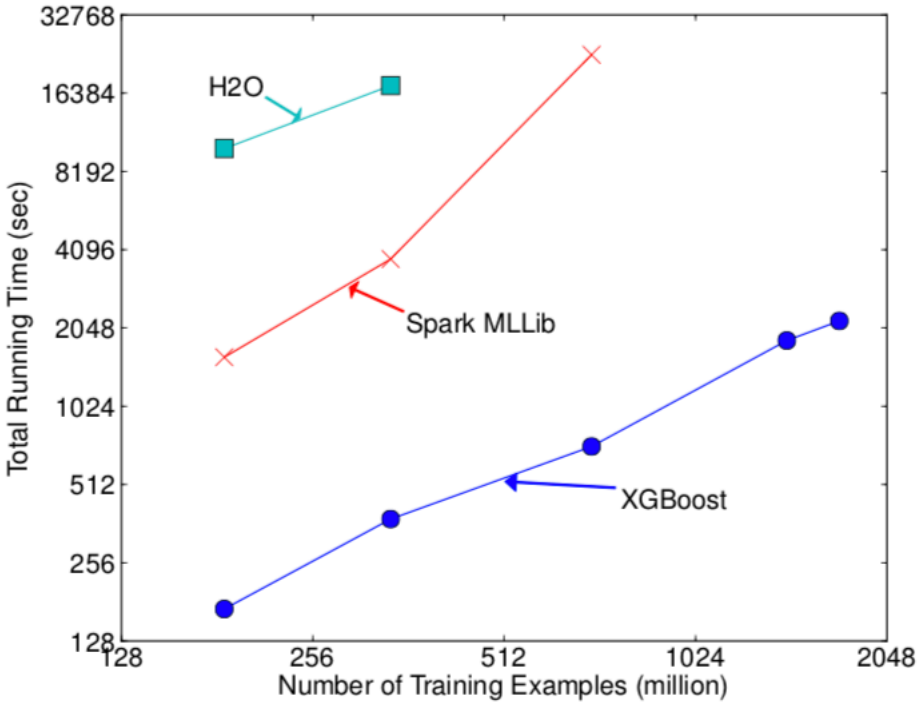
Out of Core Computing

This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

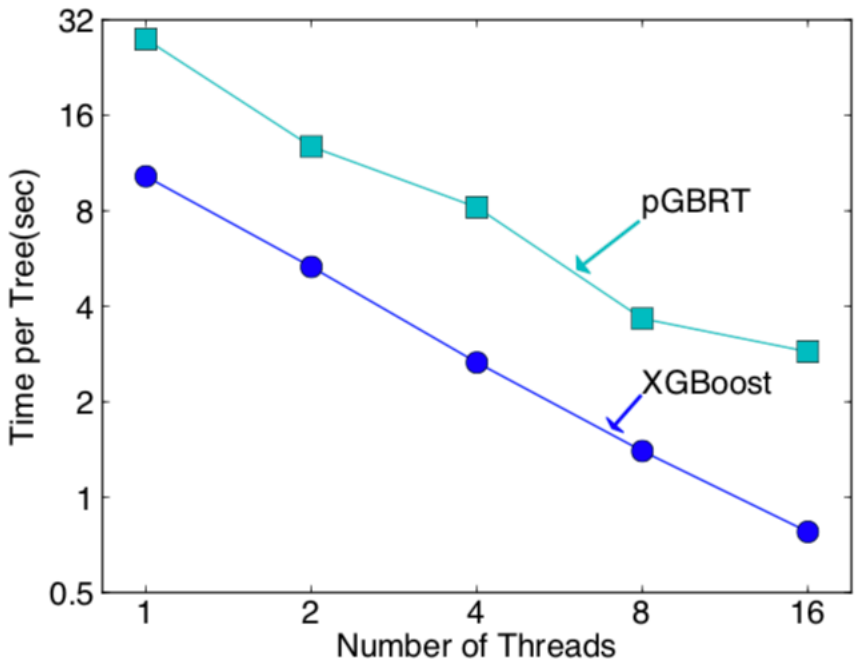
Comparison of tree boosting methods

Method	Exact Greedy	Approximate Global	Approximate Local	Out of Core	Sparsity Aware	Parallel
XGBoost	Yes	Yes	Yes	Yes	Yes	Yes
pGBRT	No	No	Yes	No	No	Yes
Spark MLlib	No	Yes	No	No	Partially	Yes
H2O	No	Yes	No	No	Partially	Yes
Scikit Learn	Yes	No	No	No	No	No
R GBM	Yes	No	No	No	Partially	No

Analysis and experiment results by researchers



End to end time cost including data loading



Comparison using Yahoo LTRC dataset

References

- AV - https://www.analyticsvidhya.com/blog/2016/01/xgboost-algorithm-easy-steps/
- What is XGBoost from Kaggle - https://www.kaggle.com/dansbecker/xgboost
- ML with XGBoost - https://www.kaggle.com/rtatman/machine-learning-with-xgboost-in-r
- XGBoost documentation - https://xgboost.readthedocs.io/en/latest/R-package/xgboostPresentation.html
- Understanding math behind XGBoost - https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/

Appendix :

IRIS data analysis

Load libraries and data

1. Load all required libraries / packages

In [11]:
```
#install.packages("xgboost") - if not installed earlier
library(xgboost)
library(caret)
library(dplyr)
```

2. Load the dataset

In [13]:
```
set.seed(1000)
data(iris)
species = iris$Species
label = as.integer(iris$Species)-1
iris$Species = NULL
```

Data preparation

3. Data Preparation: Split the data for training and testing

In [14]:
```
n=nrow(iris)
train.index = sample(n,floor(0.74*n))
train.data = as.matrix(iris[train.index,])
train.label = label[train.index]
test.data = as.matrix(iris[-train.index,])
test.label = label[-train.index]
```

4. Data Preparation: Create xgb Dmatrix objects

In [15]:
```
xgb.train.object = xgb.DMatrix(data = train.data,label=train.label)
xgb.test.object = xgb.DMatrix(data = test.data,label=test.label)
```

5. Data Preparation: Define parameters

In [16]:
```
# The multi:softprob objective tells the algorithm to calculate probabilities for every
# possible outcome (in this case, a probability for each of the three flower species), for
# every observation.

num_class = length(levels(species))
params = list(
  booster = "gbtree",
  eta = 0.001,
  max_depth = 5,
  gamma = 3,
  subsample = 0.74,
  colsample_bytree = 1,
  objective = "multi:softprob",
  eval_metric = "mlogloss",
  num_class = num_class
)
```

Model development

6. Model Development: Train the model

In [17]:
```
# Train the XGBoost classifer
xgb.fit=xgb.train(
  params=params,
  data=xgb.train.object,
  nrounds=10000,
  nthreads=1,
  early_stopping_rounds=15,
  watchlist=list(val1=xgb.train.object,val2=xgb.test.object),
  verbose=0
)
```

In [18]:
```
# Review the final model and results
xgb.fit
```

```
##### xgb.Booster
raw: 3 Mb
call:
  xgb.train(params = params, data = xgb.train.object, nrounds = 10000,
    watchlist = list(val1 = xgb.train.object, val2 = xgb.test.object),
    verbose = 0, early_stopping_rounds = 15, nthreads = 1)
params (as set within xgb.train):
  booster = "gbtree", eta = "0.001", max_depth = "5", gamma = "3", subsample = "0.74", colsample_b
ytree = "1", objective = "multi:softprob", eval_metric = "mlogloss", num_class = "3", nthreads = "
1", silent = "1"
xgb.attributes:
  best_iteration, best_msg, best_ntreelimit, best_score, niter
callbacks:
  cb.evaluation.log()
  cb.early.stop(stopping_rounds = early_stopping_rounds, maximize = maximize,
    verbose = verbose)
# of features: 4
niter: 3017
best_iteration : 3002
best_ntreelimit : 3002
best_score : 0.256246
nfeatures : 4
evaluation_log:
    iter val1_mlogloss val2_mlogloss
       1      1.097309      1.097386
       2      1.095949      1.096094
---
    3016      0.164426      0.256262
    3017      0.164426      0.256265
```

Predict new outcomes

7. Model Development: Predict new outcomes

In [19]:
```
# We can predict new outcomes given the testing data set that we set aside earlier.
# We use the predict function to predict the likelihood of each observation in test.data
# of being each flower species.

# Predict outcomes with the test data
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = levels(species)
```

Evaluation

8. Evaluation: Identify the class with highest probability for each prediction

In [20]:
```
# Iterate over the predictions and identify the label (class) with the highest probability.
# This allows us to evaluate the true performance of the model by comparing the actual
# labels with the predicted labels.
# Use the predicted label with the highest probability

# Please don't forget to re-convert your labels back to the names of the species
# by adding 1 back to the integer values

xgb.pred$prediction = apply(xgb.pred,1,function(x) colnames(xgb.pred)[which.max(x)])
xgb.pred$label = levels(species)[test.label+1]
```

9. Evaluation: Check Accuracy of predictions

In [21]:
```
# Calculate the accuracy of the predictions. This compares the true labels from the test data
# set with the predicted labels (with the highest probability), and it represents the percent
# of flower species that were accuracy predicted using the XGBoost model. My results suggest
# that XGBoost can consistently achieve an accuracy of at least 90%!

result = sum(xgb.pred$prediction == xgb.pred$label)/nrow(xgb.pred)
print(paste("Final prediction Accuracy is ",sprintf("%1.2f%%",100*result)))
```

```
[1] "Final prediction Accuracy is  94.87%"
```