Boosting Machines and Stacking

In previous module we saw models based on bagging; random forests and extraTrees where each individual model was independent and eventual prediction of the ensemble of these models was a simple majority vote or avarage depending on whether the problem was of classification of regression.

In this model we look at the idea of boosting where each subsequent model improves upon the mistakes of previous model.

Model type which is being boosted in steps here is generally taken to be a weak learner so that individually they end up capturing most general trend [as per observation weightages] in the data; at the same time, avoiding overfitting of the data.

A very common choice of weak learner is a tree stump, which is a tree with only two leaf nodes or a single split. Although there is no hard rule in favor of choosing tree stumps of size 2, you can treat it as hyparameter and tune it accordingly as well.

You'll see that generally the discussion starts with AdaBoost in most boosting machines discussion, however its a rarely used technique and hence we have provided separate reading material on theoretical discussion for anyone who is interested. However formally we will directly talk about gradient boosting machines which are the prevalent algorithms in use now.

We'll start our discussion with gradient descent

Gradient Descent

Before we begin , here is a quick look back at one idea from our basic calculus:

Consider a function $y = f(x)$. In the picture below we are looking at very small segment of this function. For that triangle formed in the picture , we can write

$$tan(\theta) \ = \ \frac{\Delta y}{\Delta x} \qquad (1)$$

For very small Δx and Δy we can write $tan(\theta)$ in terms of gradient as follows:

$$tan(\theta) \ = \ \frac{\delta y}{\delta x} \qquad (2)$$

Equating this to (1) and after doing a little adjustment we get:

$$\Delta y \ = \ \frac{\delta y}{\delta x}\Delta x \qquad (3)$$

Idea in (3) can be generalized to higher dimensions also. Now consider cost function C dependent on parameters v_1 and v_2.

$$C \ = \ f(v_1, v_2)$$

Lets say we start with some default values of these parameters say 1 and 2. Now i would like to change these parameters to new value in such a way that change in my cost function is negative. Lets write change in cost function using the relation between change and gradient seen in (3) for higher dimension.:

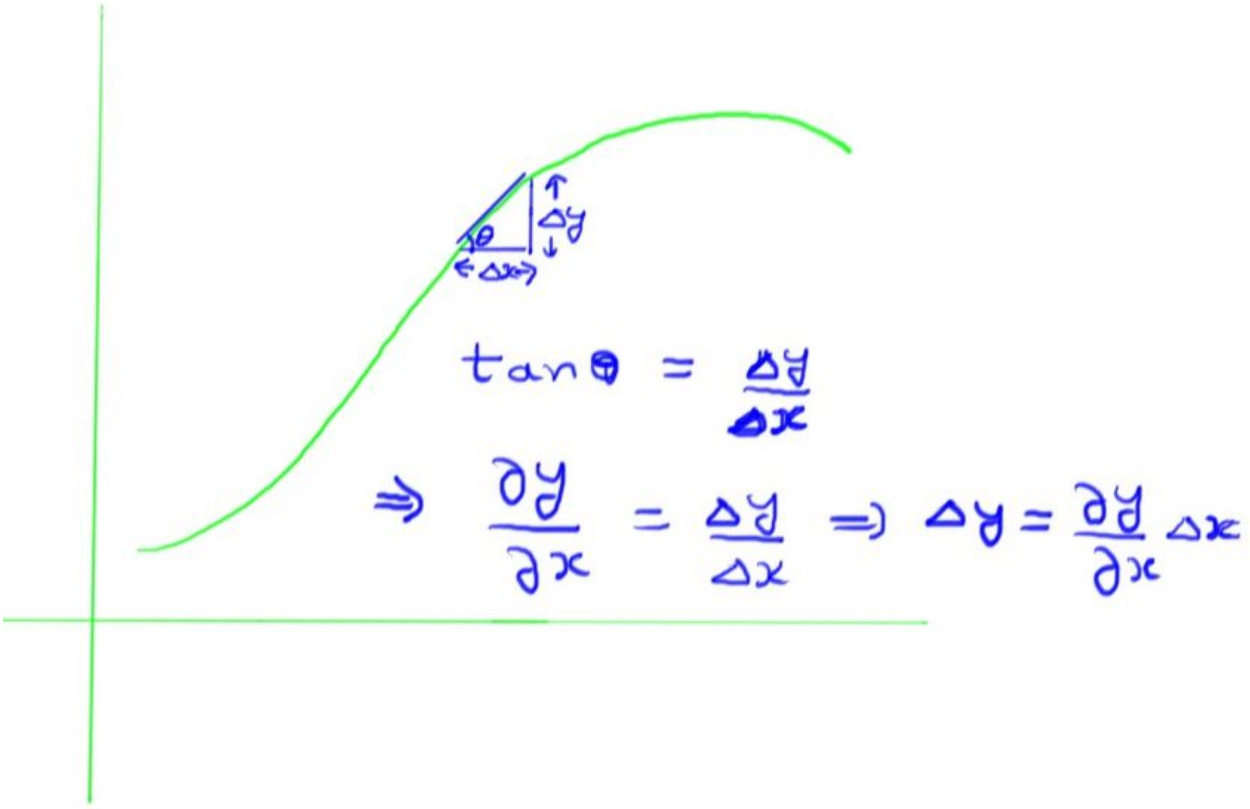

Figure 1: Gradient and Change in Function

$$\Delta C = \frac{\delta C}{\delta v_1}\Delta v_1 \ + \ \frac{\delta C}{\delta v_2}\Delta v_2 \tag{4}$$

If we consider gradient and change in parameters in vector formats as given below :

$$\nabla C \ = \ (\frac{\delta C}{\delta v_1}, \frac{\delta C}{\delta v_2}) \tag{5}$$

and

$$\Delta v \ = \ (\Delta v_1, \Delta v_2) \tag{6}$$

using (5) and (6) , we can rewrite (4) as follows:

$$\Delta C \ = \ \nabla C \bullet \Delta v \tag{7}$$

Now; we need to figure out how to change our parameters; that is, what should be the value of Δv so that change in cost function ΔC is always negative.

Consider :

$$\Delta v \ = \ -\eta \nabla C$$

if we put this back in the (7), we get :

$$\Delta C \ = \ -\eta \|\nabla C\|^2 \tag{8}$$

which is always negative as long as η is positive. Result in (8) simply means that we can change our parameters in the negative direction of the gradient of cost function and can always reduce our cost function. At the optimal point , gradient of the function will become zero and we wont be able to update parameters. Those values of the parameters will be the optimal values.

This idea is simply known as gradient descent. Where , given the cost function, we can start with some default value of parameters and change them in the manner shown above in order to obtain their optimal values which minimise cost function.

Gradient Boosting Machines

General idea behind boosting machines is that subsequent learners are built using gradient descent.

Lets revisit the particular idea at concept level and understand what do we mean by "subsequent model", learning from the mistakes of a previous model.

- Make a weak Learner $F(x_i)$
- Calculate residuals
- Fit h to residuals
- Update $F(x_i) \ := \ F(x_i) + h(x_i)$

Lets consider a general squared loss function:

$$L(y_i, F(x_i)) \ = \ (y_i \ - \ F(x_i))^2$$

We want to minimise overall loss $J = \sum_i L(y_i, F(x_i))$ *by adjusting* $F(x_1), F(x_2), ..., F(x_n)$.

Consider and idea where we treat $F(x_i)$ as parameters instead. We can write :

$$\frac{\delta J}{\delta F(x_i)} = \frac{\delta\left(\sum_i L(y_i, F(x_i))\right)}{\delta F(x_i)} = \frac{\delta L(y_i, F(x_i)}{\delta F(x_i)} = F(x_i) - y_i$$

This enables us to write

$$residuals = y_i - F(x_i) = -\frac{\delta J}{\delta F(x_i)} \tag{1}$$

Recall that in gradient descent we updates our parametres as follows

$$\theta_i := \theta_i - \rho\frac{\delta J}{\delta \theta_i}$$

Ofcourse Decision Trees here have no parameters to speak of, so how do we use gradient descent then? Think about it this way we started in our boosting model with very basic tree stump as our first model, for which the response was the original response. Now considering the overall combination of model itself as parameter , how do we go about adjusting it to reach towards the optimal model. For adjustment we add another tree stump to the model. But we need some response , which will work as adjustment factor to the overall ensemble of tree stump which taken together define our model. We take the functional gradient that we took of our cost function as the adjustment factor or response to the next tree stump in the queue.

Now see above how we were updating our $F(x_i)$,

$$F(x_i) := F(x_i) + h(x_i)$$

we can use (1) to rewrite this as follows:

$$F(x_i) := F(x_i) - \frac{\delta J}{\delta F(x_i)}$$

This basically means that , instead of fitting our weak learners h on residuals , we fit them on negative gradients of loss function. In case of simple squared loss, its one and the same thing. However this in theory, enables us to use any general loss function, especially the ones which are more robust to outliers in comparison to squared loss; such as abosulute loss or huber loss.

In case of classification, things are a little complex mathematically, but if you have enough patience to wade through the tedious mathematics , the eventual results are surprisingly simpler. Lets see.

We are going to contain the discussion to binary classification. If you recall in binary classfication we were tryiing to predict $P(y_i = 1|X_i)$ which we denoted as p_i. Also in order to ensure that , we can use any functional approximation (e.g. logistic regression) and link it to p_i, we wrote the model equation as follows

$$log(\frac{p_i}{1-p_i}) = F(x_i)$$

or

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

In case of logistic regression $F(x_i)$ was simple linear combination of predictors, in case of boosting machines , this will be a sequential ensembles of tree stumps (weak learners)

Further , following this , loss function was

$$\sum_{i=1}^{n} y_i * log(p_i) + (1 - y_i) * log(1 - p_i)$$

Lets rearrange things here a little.

$$= \sum_{i=1}^{n} log(1 - p_i) + y_i[log(p_i) - log(1 - p_i)]$$

$$= \sum_{i=1}^{n} log(1 - p_i) + y_i * log(\frac{p_i}{1 - p_i})$$

$$= \sum_{i=1}^{n} log(1 - p_i) + y_i * F(x_i)$$

$$= \sum_{i=1}^{n} log(1 - \frac{1}{1 + e^{-F(x_i)}}) + y_i * F(x_i)$$

$$= \sum_{i=1}^{n} log(\frac{e^{-F(x_i)}}{1 + e^{-F(x_i)}}) + y_i * F(x_i)$$

$$= \sum_{i=1}^{n} log(\frac{1}{1 + e^{F(x_i)}}) + y_i * F(x_i)$$

$$= \sum_{i=1}^{n} -log(1 + e^{F(x_i)}) + y_i * F(x_i)$$

so now we have a rather simplified loss function in form of the model itself , instead of the p_i

Lets calculate a functional gradient on this

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

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

$$\frac{\delta J}{\delta F(x_i)} = y_i - p_i$$

which is nothing but kind of pseudo residuals for classfication, simple difference between expected and predicted probabilities; which is again pretty simple to calculate operationally .

Before we close this discussion, one more point; instead of subsequent tree stumps being fitted on the raw functional gradient that we saw earlier , it is fitted on a fraction of the residual/gradient in order to further facilitate avoiding overfitting. This fractional factor in context of boosting machines is known as shrinkage.

This theoretical discussion for new learners , at least at first glance is scary to say the least. Dont let the incomplete comprehension of mathematics here deter you from using the boosting machines in practice. You will find the R codes to be very similar to random Forest. You can always revisit the theoretical discussion couple of times to get a better grasp of underlying mechanism.

Regression with gbm

In this case study we'll be predicting bike sharing numbers based on weather and other factors . Again, I am not going to walk you through the train test combining exercise , thats implied.

```
bs=read.csv("/Users/lalitsachan/Dropbox/0.0 Data/bike_sharing_hours.csv",stringsAsFactors = F)

CreateDummies=function(data,var,freq_cutoff=0){
  t=table(data[,var])
  t=t[t>freq_cutoff]
  t=sort(t)
  categories=names(t)[-1]

  for( cat in categories){
    name=paste(var,cat,sep="_")
    name=gsub(" ","",name)
    name=gsub("-","_",name)
    name=gsub("\\?","Q",name)
    name=gsub("<","LT_",name)
    name=gsub("\\+","",name)
    name=gsub("\\/","_",name)
    name=gsub(">","GT_",name)
    name=gsub("=","EQ_",name)
    name=gsub(",","",name)

    data[,name]=as.numeric(data[,var]==cat)
  }

  data[,var]=NULL
  return(data)
}
```

here cnt is nothing but simple sum of casual and registered. cnt is our response, we'll be dropping the date and two other columns.

```
library(dplyr)
bs=bs %>% select(-dteday,-casual,-registered)
```

next we'll make dummy vars for appropriate columns

```
char_cols=c("season","mnth","hr","holiday","weekday","workingday","weathersit")
for(col in char_cols){
  bs=CreateDummies(bs,col,500)
}
```

you'll see that we'll follow same procedure here for parameter tuning as randomForest, just that name and usual values of the paramaters to be tried are going to be different.

```
library(gbm)
library(cvTools)
```

```
param=list(interaction.depth=c(1:7),
           n.trees=c(50,100,200,500,700),
           shrinkage=c(.1,.01,.001),
           n.minobsinnode=c(1,2,5,10))
```

Here interaction.depth, controls how weak is our learner. value 1 means weak learners are decision trees with single split . very high numbers here will lead to overfit.

n.trees means the same thing as ntree in randomForest, and n.minobsinnode is similar to nodesize in randomForest. Shrinkage is the fraction that we talked about earlier , ideal values to try out here should be ideally less than 1. Very small values generally tend to result in high values of n.trees.

```
subset_paras=function(full_list_para,n=10){

  all_comb=expand.grid(full_list_para)

  s=sample(1:nrow(all_comb),n)

  subset_para=all_comb[s,]

  return(subset_para)
}

num_trials=10
my_params=subset_paras(param,num_trials)
# Note: A good value for num_trials is around 10-20% of total possible
# combination. It doesnt have to be always 10
```

```
myerror=9999999

for(i in 1:num_trials){
  print(paste0('starting iteration:',i))
  # uncomment the line above to keep track of progress
  params=my_params[i,]

  k=cvTuning(gbm,cnt~.,
             data =bs,
             tuning =params,
             args = list(distribution="gaussian"),
             folds = cvFolds(nrow(bs), K=10, type = "random"),
             seed =2,
             predictArgs = list(n.trees=params$n.trees)
             )
  score.this=k$cv[,2]

  if(score.this<myerror){
    print(params)
    # uncomment the line above to keep track of progress
    myerror=score.this
    print(myerror)
    # uncomment the line above to keep track of progress
    best_params=params
  }

  print('DONE')
```

```
  # uncomment the line above to keep track of progress
}
```

```
myerror
```

```
## [1] 52.29379
```

This is tentative measure of performance .

```
best_params
```

```
##   interaction.depth n.trees shrinkage n.minobsnode
## 1                 6     500       0.1           10
```

This is the best combination of paramter values as per cv errors. Lets build our final model using these values

```
bs.gbm.final=gbm(cnt~.,data=bs,
                 n.trees = best_params$n.trees,
                 n.minobsinnode = best_params$n.minobsnode,
                 shrinkage = best_params$shrinkage,
                 interaction.depth = best_params$interaction.depth,
                 distribution = "gaussian")
```

We can now use this model to make prediction on test data if we are given one.

```
test.pred=predict(bs.gbm.final,newdata=bs_test,n.trees = best_params$n.trees)
write.csv(test.pred,"mysubmission.csv",row.names = F)
```

Classfication with gbm

We'll be solving the same problem as earlier module with gbm.

```
ci_train=read.csv("~/Dropbox/0.0 Data/census_income.csv",
                  stringsAsFactors =F)
glimpse(ci_train)
```

```
## Observations: 32,561
## Variables: 15
## $ age            <int> 39, 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30,...
## $ workclass      <chr> " State-gov", " Self-emp-not-inc", " Private", ...
## $ fnlwgt         <int> 77516, 83311, 215646, 234721, 338409, 284582, 1...
## $ education      <chr> " Bachelors", " Bachelors", " HS-grad", " 11th"...
## $ education.num  <int> 13, 13, 9, 7, 13, 14, 5, 9, 14, 13, 10, 13, 13,...
## $ marital.status <chr> " Never-married", " Married-civ-spouse", " Divo...
## $ occupation     <chr> " Adm-clerical", " Exec-managerial", " Handlers...
## $ relationship   <chr> " Not-in-family", " Husband", " Not-in-family",...
## $ race           <chr> " White", " White", " White", " Black", " Black...
## $ sex            <chr> " Male", " Male", " Male", " Male", " Female", ...
## $ capital.gain   <int> 2174, 0, 0, 0, 0, 0, 0, 0, 14084, 5178, 0, 0, 0...
## $ capital.loss   <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ hours.per.week <int> 40, 13, 40, 40, 40, 40, 16, 45, 50, 40, 80, 40,...
## $ native.country <chr> " United-States", " United-States", " United-St...
## $ Y              <chr> " <=50K", " <=50K", " <=50K", " <=50K", " <=50K...
```

```
ci_train=ci_train %>% select(-education)
ci_train$Y=as.numeric(ci_train$Y==" >50K")
```

```
cat_var=names(ci_train)[sapply(ci_train,is.character)]


for(var in cat_var){
  ci_train=CreateDummies(ci_train,var,500)
}

## For classification no need tk convert to factor type for gbm
## we'll just change distribution to "bernoulli"
```

Again, i have not expliclty involved test data here for avoiding repetition, you should.

```
param=list(interaction.depth=c(1:7),
           n.trees=c(50,100,200,500,700),
           shrinkage=c(.1,.01,.001),
           n.minobsinnode=c(1,2,5,10))

num_trials=10
my_params=subset_paras(param,num_trials)

mycost_auc=function(y,yhat){
  roccurve=pROC::roc(y,yhat)
  score=pROC::auc(roccurve)
  return(score)
}
# Note: A good value for num_trials is around 10-20% of total possible
# combination. It doesnt have to be always 10
```

```
myauc=0

## Cvtuning
## This code will take couple hours to finish
## Dont execute in the class
for(i in 1:num_trials){
  # print(paste('starting iteration :',i))
  # uncomment the line above to keep track of progress
  params=my_params[i,]

  k=cvTuning(gbm,Y~.,
             data =ci_train,
             tuning =params,
             args=list(distribution="bernoulli"),
             folds = cvFolds(nrow(ci_train), K=10, type ="random"),
             cost =mycost_auc, seed =2,
             predictArgs = list(type="response",n.trees=params$n.trees)
             )
  score.this=k$cv[,2]

  if(score.this>myauc){
    # print(params)
    # uncomment the line above to keep track of progress
    myauc=score.this
    # print(myauc)
    # uncomment the line above to keep track of progress
```

```
    best_params=params
  }

  # print('DONE')
  # uncomment the line above to keep track of progress
}
```

```
myauc
```

```
## [1] 0.9217784
```

this is the tentative performance measure.

```
best_params
```

```
##   interaction.depth n.trees shrinkage n.minobsinnode
## 1                 7     700      0.01              5
```

these are the best parameter choices as par the cv performance.

Lets use these to build our final model.

```
ci.gbm.final=gbm(Y~.,data=ci_train,
                 n.trees = best_params$n.trees,
                 n.minobsinnode = best_params$n.minobsinnode,
                 shrinkage = best_params$shrinkage,
                 interaction.depth = best_params$interaction.depth,
                 distribution = "bernoulli")
```

If we want to make prediction on test data to submit simple probability scores then we can do this

```
test.score=predict(ci.gbm.final,newdata=ci_test,type='response',
                   n.trees = best_params$n.trees)
write.csv(test.score,"mysubmission.csv",row.names=F)
```

if the requirement is to submit hardclasses we can make prediction on train data itself and detrmine cutoff in usual manner as discussed earlier and then use the cutoff to convert test.score to hardclasses.

Stacking

So far we saw many algortihms and the idea is to eventually use the one which performs best for our given data. But that limits us to use , at the end just one algorithm.

Lets ask ourselves a question that why did the non-linear algorithm performed better than linear models in most of the cases. This happens because in most of the cases there is some bit of complex nonlinear pattern in the data which linear models are not designed to capture. Be it randomForest, Extra Trees or GBM, they capture some kind of non-linear pattern and hence perform better.

That brings us to ur second question, why these algorithms perform differently than each other then. Reason is that they end up capturing some what different even if slightly, non-linear pattern in the data. Even the same algorithm might capture different patterns given different hyper parameter choices. That tells me that when i eventually select just one algorithm to go ahead with, I am essentially not making use of non-linear patterns captured by the other algorithm.

So what we are saying is that $ Y = F(X_{i})$ works because F here captures complex transfomrations of our X_i such that $F(X_i)$ is linearly related with Y. Can we not use these algorithms, many of them at once to extract these transformations instead and use their prediction as variables for our linear model. This idea will let us make use of all the algorithms and thus patterns extracted by them, instead of having to use just one and discard others.

Now, all of this boils down to using the predictions from complex algorithms as variables in second layer of linear models. Idea is suspiciously simple, ofcourse the devil is in details.

Catch which makes the implementation difficult is that, predictions can be used as variables for modeling your response only when they are out of sample prediction. To ensure that we'll be using an idea very much like cross validation.

A rough sketch of the implementation looks like this

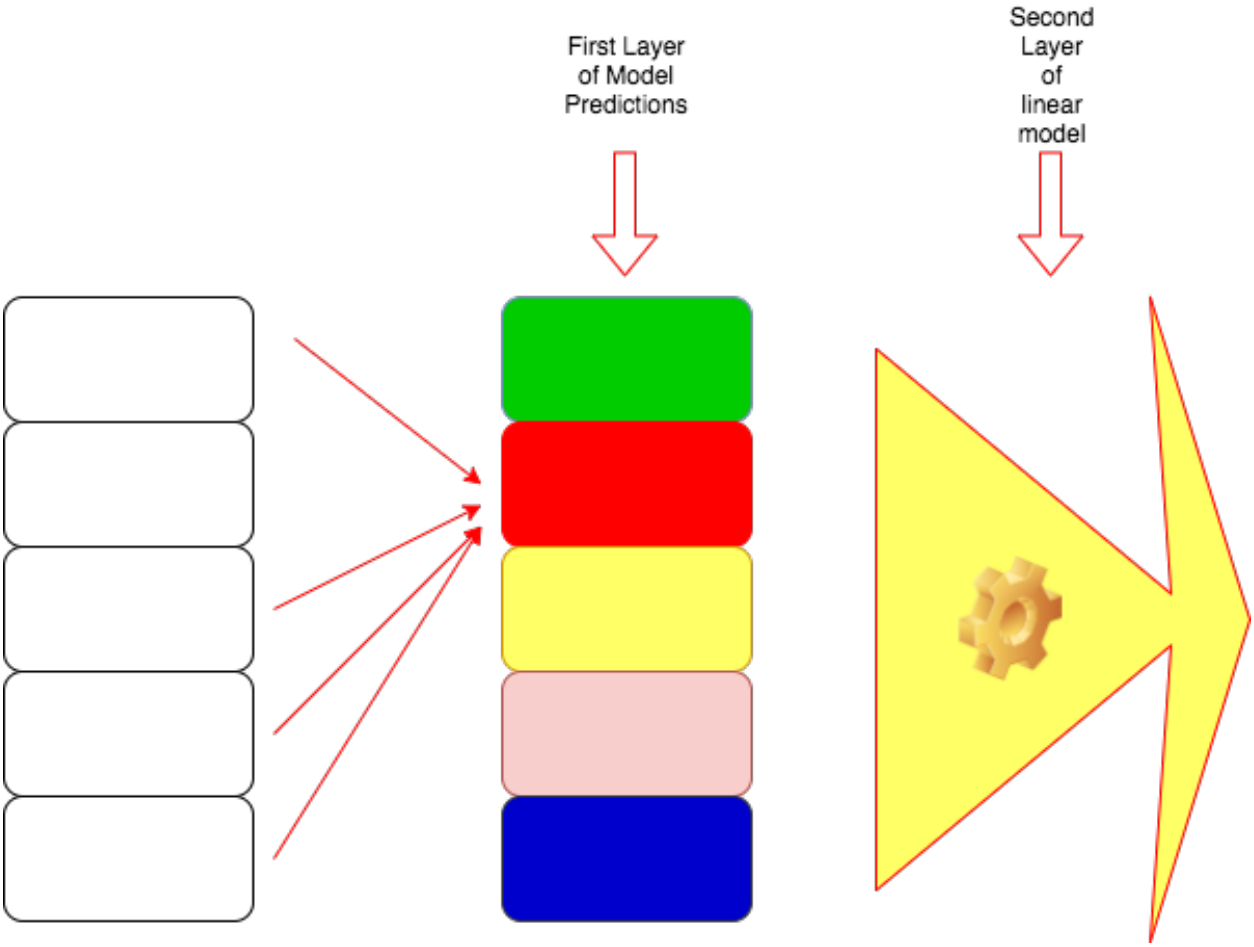

Figure 2: Stacking

1. Instead of building the first layer of non-linear models on the entire data , they are built on subsets.
2. The complementary subset which is not involved in the modeling process , predictions are made on that .
3. This process is repeated for each subset and each algorithm
4. Next layer of linear model is built on these out of sample predictions
5. In order to make use of this complex ensemble of models on test/production data, we need prredictions for test data as well. For this the same model is built on the entire training data and predictions are made on test data [which is by design out of sample anyway]

Lets look at the implementation for a classfication case. Note that we are making use of tuned parameter from earlier. In the example , we are combining only randomForest , theoretically there is no limit on how many and what kind of algorithms you can make use of.

We'll be making use of a custom function for folds

```
mykfolds=function(nobs,nfold=5){

  t=cvFolds(nobs,K=nfold,type='random')

  folds=list()

  for(i in 1:nfold){

    test=t$subsets[t$which==i]
    train=t$subsets[t$which!=i]

    folds[[i]]=list('train'=train,'test'=test)
  }

  return(folds)
}
```

We'll be using the revenue grid data. Data prep process is simply copied from earlier.

```
rg_train=read.csv("~/Dropbox/0.0 Data/rg_train.csv",stringsAsFactors = FALSE)
rg_test=read.csv("~/Dropbox/0.0 Data/rg_test.csv",stringsAsFactors = FALSE)

rg_test$Revenue.Grid=NA

rg_train$data='train'
rg_test$data='test'

rg=rbind(rg_train,rg_test)

rg = rg %>%
  mutate(children=ifelse(children=="Zero",0,substr(children,1,1)),
         children=as.numeric(children))
```

```
## Warning: package 'bindrcpp' was built under R version 3.3.2
```

```
rg=rg %>%
mutate(a1=as.numeric(substr(age_band,1,2)),
       a2=as.numeric(substr(age_band,4,5)),
      age=ifelse(substr(age_band,1,2)=="71",71,ifelse(age_band=="Unknown",NA,0.5*(a1+a2)))
       ) %>%
  select(-a1,-a2,-age_band)
```

```
## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion
```

```
## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion
```

```
cat_cols=c("status","occupation","occupation_partner","home_status",
           "family_income","self_employed",
           "self_employed_partner","TVarea","gender","region")

for(cat in cat_cols){
  rg=CreateDummies(rg,cat,50)
}
```

```
rg=rg %>%
  select(-post_code,-post_area)

rg$Revenue.Grid=as.numeric(rg$Revenue.Grid==1)

for(col in names(rg)){

  if(sum(is.na(rg[,col]))>0 & !(col %in% c("data","Revenue.Grid"))){

    rg[is.na(rg[,col]),col]=mean(rg[rg$data=='train',col],na.rm=T)
  }

}

rg$Revenue.Grid=as.numeric(rg$Revenue.Grid==1)

rg_train=rg %>% filter(data=='train') %>% select(-data)
rg_test=rg %>% filter(data=='test') %>% select (-data,-Revenue.Grid)
```

```
myfolds=mykfolds(nrow(rg_train),10)
```

We'll start with an empty data frame which we'll fill with out of sample predictions using a for loop.

```
rg_train_layer=data.frame(rf_var=numeric(nrow(rg_train)),
                          gbm_var=numeric(nrow(rg_train)))
```

```
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
for(i in 1:10){
  print(c(i))
  fold=myfolds[[i]]

  train_data=rg_train[fold$train,]
  # for this iteration model will be built on this chunk of the data
  test_data=rg_train[fold$test,]
  # predicitons will be made on this chunk which is not being
  # used in the modeling process

  print('rf')

  rf.fit=randomForest(factor(Revenue.Grid)~.-REF_NO,
                      mtry=10,
                      ntree=500,
                      maxnodes=100,
                      nodesize=10,
```

```
                        data=train_data
                        )
  ## these value of parameters have been chosen randomly here
  ## but separately tuned parameter choices will be better
  rf_score=predict(rf.fit,newdata=test_data,type='prob')[,1]

  print('gbm')
  gbm.fit=gbm(Revenue.Grid~.-REF_NO,data=train_data,
                 interaction.depth=7,
                       n.trees=700,
                       shrinkage=0.01,
                       n.minobsinnode=5,
                 distribution = "bernoulli")
   ## these value of parameters have been chosen randomly here
  ## but separately tuned parameter choices will be better
  gbm_score=predict(gbm.fit,newdata=test_data,
                    n.trees=700,type='response')

  rg_train_layer$rf_var[fold$test]=rf_score

  rg_train_layer$gbm_var[fold$test]=gbm_score
}
```

```
## [1] 1
## [1] "rf"
## [1] "gbm"
## [1] 2
## [1] "rf"
## [1] "gbm"
## [1] 3
## [1] "rf"
## [1] "gbm"
## [1] 4
## [1] "rf"
## [1] "gbm"
## [1] 5
## [1] "rf"
## [1] "gbm"
## [1] 6
## [1] "rf"
## [1] "gbm"
## [1] 7
## [1] "rf"
## [1] "gbm"
## [1] 8
## [1] "rf"
## [1] "gbm"
## [1] 9
## [1] "rf"
## [1] "gbm"
## [1] 10
## [1] "rf"
## [1] "gbm"
```

Now we'll build prediction data for test data, by building model on entire training data.

```
rg_test_layer=data.frame(rf_var=numeric(nrow(rg_test)),
                         gbm_var=numeric(nrow(rg_test)))

full.rf=randomForest(factor(Revenue.Grid)~.-REF_NO,
                         mtry=10,
                         ntree=500,
                         maxnodes=100,
                         nodesize=10,
                         data=rg_train
                         )
full.gbm=gbm(Revenue.Grid~.-REF_NO,data=rg_train,
                  interaction.depth=7,
                        n.trees=700,
                        shrinkage=0.01,
                        n.minobsinnode=5,
                  distribution = "bernoulli")
# note that paramater choices are exactly same as earlier

rg_test_layer$rf_var=predict(full.rf,newdata=rg_test,type='prob')[,1]
rg_test_layer$gbm_var=predict(full.gbm,newdata=rg_test,
                              n.trees=700,type='response')
```

We'll add response to training layer that we created earlier and build a linear model on top.

```
rg_train_layer$Revenue.Grid=rg_train$Revenue.Grid

log.mod=glm(Revenue.Grid~.,data=rg_train_layer,family = "binomial")
```

We can use this to make prediction on test data using the test layer that we built earlier .

```
test.score=predict(log.mod,newdata=rg_test_layer,type='response')
write.csv(test.score,"mysubmission.csv",row.names = F)
```

??

It seems like over the duration of the course we have been gradually moving towards , more and more complex algorithms. And all the while claiming every time that the latest one does better.

Well then , why dont we directly start with the best algorithm in the first place. Why bother learning the simple ones. Simple answer is that , its all about balance between interpretability and complexity.

Simpler models , at times might not give great results in terms of predictions , but they are much more easy to interpret and hence wit hresults more actionable . So these are useful when focus is not just on prediction, but utilising the results eventually to overhaul business process as well.

Whereas, if correct prediciton is of utmost important, complexity can be tolerated.

We'll stop our discussion here. In case of any clarification, please take to QA forum.

Prepared by : Lalit Sachan

Contact : lalit.sachan@edvancer.in