Classification Kaggle

Chris, Mike, Nathan

December 5, 2016

The first step in any categorization problem is to clean up the data set. This one was particularly rough, with Monthly Income, Debt Ratio, and Number of Dependents having missing or seemingly innaccurate data. We began by loading in all of the data, and removing the redundant indexing provided by Kaggle. We then removed the response variable, Defaulting, from the data frame to simplify the code for all of our models.

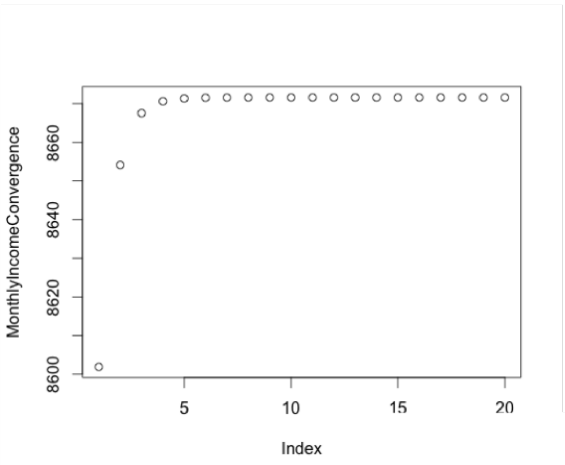
```
#Imputation Code
setwd("~/Dropbox/Pomona/Math/Math 154/Kaggle/Final Write Up") # Chris's data directory

Train <- read.csv('cs-training.csv', header=TRUE,comment.char="") #Loads training data
Train$X <- NULL # Removes redundant indexing of training data
SeriousDlqin2yrs <- Train$SeriousDlqin2yrs #Stores response variable
Train$SeriousDlqin2yrs <- NULL #Removes response variable from training data frame

N_var <- length(colnames(Train)) # Number of variables
N_obs <- length(Train$age) # Number of observations

Test <- read.csv('cs-test.csv', header=TRUE, comment.char="") #Loads test data
Test$X <- NULL # Removes redundant indexing
Test$SeriousDlqin2yrs <- NULL #Removes response variable from test data frame
```

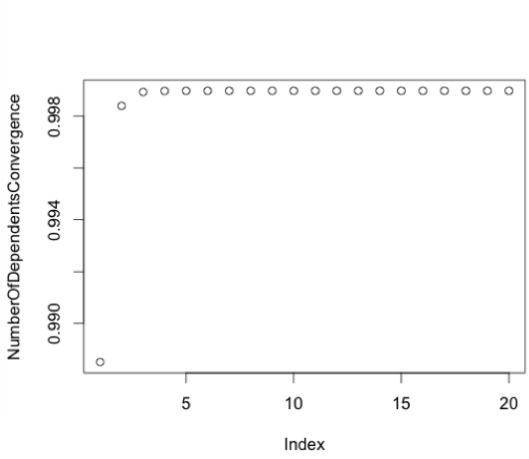
The following three functions all work on the same idea. We take the data frame, do linear regression to solve for the variable in question, and then update the values in the data frame. Much like a Gibbs sampler, we hope that running this repeatedly will cause us to converge to a better value for Monthly Income, Debt Ratio, and Number of Dependents than, say, the mean of the data set. It should be noted that we tried using a SVM to do the predictions, but the number of observatons made it too cumbersome for our computers to run even once, let alone repeatedly.

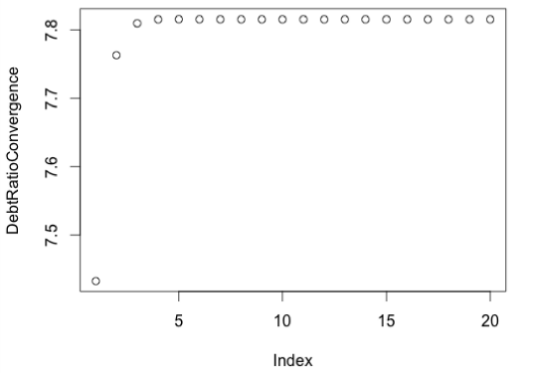
```
Update_MI <- function(df,index) {
  # Updates 0 and NA values of Monthly Income
  #
  # Args:
  #   df: data frame of observations
  #
  # Returns:
  #   dataframe w/ updated values
  LinReg <- lm(MonthlyIncome ~ .,df)
  # Performs linear regression for Monthly Income using all other variables
  df$MonthlyIncome[index] <- predict(LinReg,df[index,]) # Imputes for Monthly Income
  print(paste("MI: ",df$MonthlyIncome[index[1]]),sep="") # Check for convergence
  return(df)
}

Update_NoD <- function(df,index) {
  # Updates NA values of Number of Dependents
  #
  # Args:
  #   df: data frame of observations
```

```
  #
  # Returns:
  #   dataframe w/ updated values
  LinReg <- lm(NumberOfDependents ~ .,df)
  # Performs linear regression for Monthly Income using all other variables
  df$NumberOfDependents[index] <- predict(LinReg,df[index,]) # Imputes for Number of Dependents
  print(paste("NoD: ",df$NumberOfDependents[index[1]]),sep="") # Check for convergence
  return(df)
}

Update_DR <- function(df,index) {
  # Updates Debt Ratio for Monthly Income = 0 or NA
  #
  # Args:
  #   df: data frame of observations
  #
  # Returns:
  #   dataframe w/ updated values
  LinReg <- lm(DebtRatio ~ .,df)
  # Performs linear regression for Debt Ratio using all other variables
  df$DebtRatio[index] <- predict(LinReg,df[index,]) # Imputes for Debt Ratio
  print(paste("DR: ",df$DebtRatio[index[1]]),sep="") # Check for convergence
  return(df)
}
```

Here we are, it's finally time to impute the data. Impute will run each of our 3 prediction functions 20 times, which seems to be a good number for letting them converge. Here's a graph or 3 of the convergence:







We intialize our damaged data with the mean of the three variables respectively, but we could have just initilialized it with some other constant. We then store the imputed training and test data in a csv file to avoid having to impute ever again.

```
Impute <- function(df) {
  # Imputes NA values in the data frame using linear regression
  #
  # Args:
  #   df: data frame of observations
  #
  # Returns:
  #   data frame with imputed values for Monthly Income, Debt Ratio, and Number of Dependents
  na.NoD.index <- which(is.na(df$NumberOfDependents))
  # Observations with Number of Dependents = NA
  na.MI.index <- which(is.na(df$MonthlyIncome))
  # Observations with Monthly Income = NA
  zero.MI.index <- which(df$MonthlyIncome == 0)
  # Observations with Monthly Income = 0
  zero.na.MI.index <- sort(c(na.MI.index,zero.MI.index))
  # Observations with Monthly Income = 0 or NA

  Debt_Hold <- df$DebtRatio[zero.na.MI.index]
  # Stores Debt Values where Debt Ratio could not be calculated
  df$DebtRatio[zero.na.MI.index] <- NA # Replaces Debt Values with NA

  mean_NoD <- mean(df$NumberOfDependents,na.rm=TRUE) # Mean Number of Dependents
```

```
  mean_DR <- mean(df$DebtRatio,na.rm=TRUE) # Mean Debt Ratio
  mean_MI <- mean(df$MonthlyIncome,na.rm=TRUE) # Mean Monthly Income

  df$DebtRatio[zero.na.MI.index] <- mean_DR # Insert Mean Number of Dependents as initial
  # for Observations with Monthly Income = 0 or NA
  df$NumberOfDependents[na.NoD.index] <- mean_NoD # Insert Mean Number of Dependents as initial
  #for Observations with NoD = NA

  # Imputation
  for(i in 1:20) {
    df <- Update_MI(df,zero.na.MI.index)
    df <- Update_NoD(df,na.NoD.index)
    df <- Update_DR(df,zero.na.MI.index)
  }

  df$DebtRatio[zero.na.MI.index] <- Debt_Hold / df$MonthlyIncome[zero.na.MI.index]
  # Replace DR with original debt divided by imputed MI
  df$MonthlyIncome[zero.MI.index] <- 0 # Reset MI for those with original MI = 0
  df$NumberOfDependents[df$NumberOfDependents < 0] <- 0 # Set NoD = 0 if imputed NoD < 0

  return(df)
}

Train <- Impute(Train)
Train$SeriousDlqin2yrs <- SeriousDlqin2yrs #Returns response variable
Test <- Impute(Test)

write.csv(Train,"training_imputed.csv") #saves imputed data
write.csv(Test,"test_imputed.csv")
```

We now begin the process of creating an ensemble learner. We upload our data and remove the response variable and indexing from the data frame for ease of prediction. We will use 3 models: a gradient boosting machine, a random forest, and an XGBoost. We had coded a support vector machine, but it did not improve our AUC when we submitted to Kaggle, so we removed it.

```
#Ensemble Predictions
library(dplyr)
library(e1071)
library(randomForest)
library(gbm)
library(xgboost)

Train <- read.csv("training_imputed.csv") #uploads imputed training
Test <- read.csv("test_imputed.csv") #uploads imputed test
Train$X <- NULL #removes redundant indexing
Test$X <- NULL #removes redundant indexing
Test$SeriousDlqin2yrs <- NULL #removes response variable from test data frame
```

Below is our Gradient Boosting Machine. This works by creating a weak prediction model, usually just able to predict the mean. It then improves upon the model, or boosts it, a set number of times. If stage 1 is F_M, the initial model, then $F_{M+1} = h(x) + F_M$, where $h(x)$ is our new correction. $h(x)$ varies depending on the problem. This GBM uses CART trees. Shrinkage, bag.fraction, and interaction.depth are GBM parameters that control the size of the trees, the amount of data to use in the next iteration, and the number of variables

to consider in the model. Obviously, we want to consider all variables so we set interaction.depth to 10. The other two parameters were found by examining R vignette's online.

```
GBMModel <- function(dfTrain) {
  # Creates a Gradient Boosting Machine
  #
  # Args:
  #   dfTrain: data frame of training observations
  #
  # Returns:
  #   GBM
  GB <- gbm(dfTrain$SeriousDlqin2yrs ~ ., data=dfTrain, n.trees=5000,
            keep.data=FALSE, shrinkage=0.01, bag.fraction=0.3,
            interaction.depth=10)
  GB
}
```

Below is our random forest model. After some testing, we settled on creating 2000 trees. The random forest had a bad habit of overfitting the data set, but it was definitely the simplest to code and still gave good results.

```
forestModel <- function(dfTrain) {
  # Creates a random forest
  #
  # Args:
  #   dfTrain: data frame of training observations
  #
  # Returns:
  #   Random forest
  train.Forest <- randomForest(formula = as.factor(SeriousDlqin2yrs) ~ .,
                               data = Train, ntree = 2000, importance = TRUE,
                               na.action=na.omit)
  train.Forest
}
```

Below is our extreme gradient boost model. It was the most complicated to code but provided the best predictions: it uses trees and linear predictions to run up to 10x faster than the gradient booster. We begin by creating a data matrix specific to the package, and call it training. This ensures none of the data is stored as a factor. Param is a list of parameters that XGB will use in predictions. Most of them were chosen by viewing R vignettes. Objective indicates which kind of learners we will use, booster tells us we will use a tree booster, a low value of eta prevents overfitting, max depth is the depth of a tree, letting subsample be .5 only lets trees grow from half the data, which prevents overfitting, and colsample_bytree must be equal to subsample. Finally, when we call the model function, we set some more parameters. nrounds is set to 500, which allows the model to pass through the data 500 times. Letting verbose be 1 prints our evaluation metric: the auc.

```
xgbModel <- function(dfTrain, dfTest) {
  # Creates a XGBoost categorization model
  #
  # Args:
  #   dfTrain: data frame of training observations
  #   dfTest: data frame of test observations
  #
```

```
  # Returns:
  #   XGB prediction model
  test_names <- names(dfTest) #stores the data labels
  train_names <- names(dfTrain) #stores the data labels
  intersect_names <- intersect(test_names, train_names) #finds the data labels in common

  training <- xgb.DMatrix(data = data.matrix(dfTrain[intersect_names]),
                          label=(dfTrain[,c("SeriousDlqin2yrs")]))


  # colsample_bytree - subsample ratio of columns when constructing each tree.
  param <- list(objective = "binary:logistic", booster = "gbtree", eta = 0.01, max_depth = 50,
                subsample = 0.5, colsample_bytree = 0.5)

  train.xgb <- xgboost(data = training, params = param, nrounds = 500, verbose = 1, eval_metric="auc")

  train.xgb
}
```

Ensemble will create our 3 models based on the training data, and output them as a list of models. Predictions then takes the test data, runs predicitions on it, and outputs a single list of probabilities for Kaggle to read. The single probability is a weighted avergae of the 3 models, with more weight being given to models that perform better individually.

```
ensemble <- function(dfTrain, dfTest) {
  # A homemade ensemble learner
  #
  # Args:
  #   dfTrain: data frame of training observations
  #   dfTest: data frame of test observations
  #
  # Returns:
  #   a list of models


  models <- list(rfs=forestModel(dfTrain),
                          gbs=GBMModel(dfTrain),
                          xgb=xgbModel(dfTrain, dfTest))
  models
}

ensemblePredictions <- function(model, train, test) {
  # Predicts from our ensemble learner
  #
  # Args:
  #   model: our ensemble learner
  #   train: data frame of training observations
  #   test: data frame of test observations
  #
  # Returns:
  #   a csv file of probabilitites for each observation
  pred <- predict(model$rfs, test, type = 'prob')[,2] #predict using the random forest
  pred2 <- predict(model$gbs, test, n.trees = 1000) #predict using the Gradient Boost
```

```
  test_names <- names(test)
  train_names <- names(train)
  intersect_names <- intersect(test_names, train_names)

  testDataXGB <- xgb.DMatrix(data = data.matrix(test[,intersect_names]))
  pred3 <- predict(xgb1,testDataXGB) #predict using the XGBoost

  pred <- data.frame(Id = seq.int(nrow(test)), ProbabilityRF=pred,
                     ProbabilityGBM=pred2, ProbabilityXGB=pred3)
  #stores predicted probabilities into data frame
  pred$ProbabilityGBM <- 1/(1+exp(-pred$ProbabilityGBM)) #GBM outputs not the actual probabilities
  # so this sigmoid function gives us them

  # After submitting each model individually, this upweights models with greater individual AUC
  pred$Probability <- ((pred$ProbabilityGBM)*4 + pred$ProbabilityRF + (pred$ProbabilityXGB)*20)/25

  pred <- pred %>% dplyr::select(Id, Probability) #This is just a way to get probabilities into a
  #format for the data frame that Kaggle reads
  write.csv(pred, file="probabilities4-1-10.csv", row.names=FALSE)
}

ensemble.models <- ensemble(Train, Test) #creates the learner
ensemblePredictions(ensemble.models, Train, Test) #predicts
```

In the end, Kaggle gave as an AUC of 0.86643 which would have put us in the 150th place of the competition.