Revenue_Prediction

David Palazzo and Samantha Werdel

6/7/2019

First we'll load in the dataframes. Note that some initial work was done in Python to load the original data and make our train, test split.

We are using our own train, test split here opposed to the Kaggle competition, given the size of the data. Columns X (old index) and custom dimensions (index and duplicate country data) are dropped here.

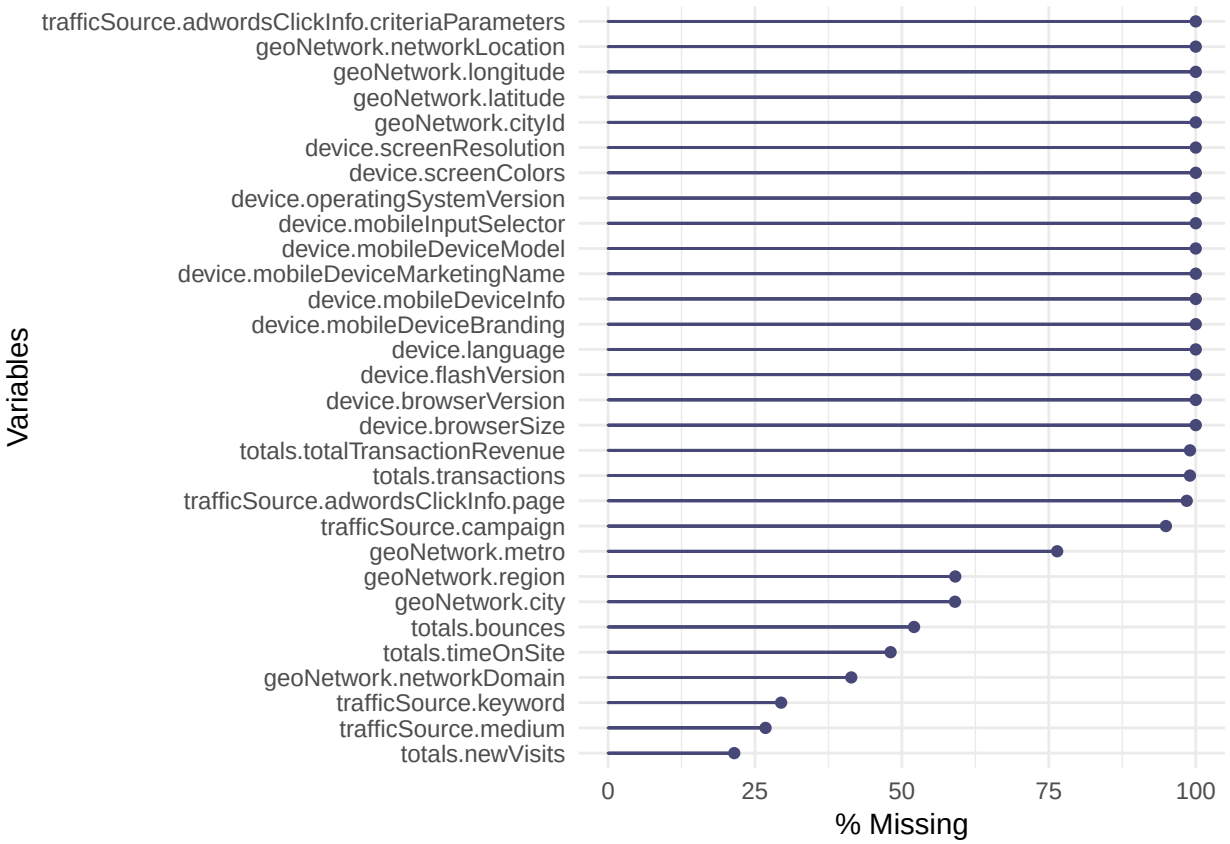
```
# Load data
test_data = read.csv('test_data.csv')
train_data = read.csv('train_data.csv')
# drop the first and second column of each dataframe (index)
train_data <- train_data[-c(1,3)]
test_data <- test_data[-c(1,3)]
```

Pull out the y variable transaction revenue.

```
# Split into X,y
y_train <- train_data$totals.transactionRevenue
X_train <- train_data[, !(colnames(train_data) %in% c('totals.transactionRevenue'))]
y_test <- test_data$totals.transactionRevenue
X_test <- test_data[, !(colnames(test_data) %in% c('totals.transactionRevenue'))]
# Split index for validation data
split_idx <- X_train$date < 20160808
```

We'll get an idea of missing values in the data.

```
is_na_val <- function(x) x %in% c("not available in demo dataset", "(not provided)",
                                  "(not set)", "<NA>", "unknown.unknown",  "(none)")

# change missing values to NA
X_train <- X_train %>% mutate_all(funs(ifelse(is_na_val(.), NA, .)))
X_test <- X_test %>% mutate_all(funs(ifelse(is_na_val(.), NA, .)))

# only show missing values greater than zero
missing <- X_train[,X_train %>% summarize_all(funs(sum(is.na(.))/length(.))) > 0.01]
gg_miss_var(missing, show_pct=TRUE)
```



Below, we drop columns with 100% missing values.

```
#change missing values to 0
X_train$totals.totalTransactionRevenue[is.na(X_train$totals.totalTransactionRevenue)] <- 0
X_train$totals.transactions[is.na(X_train$totals.transactions)] <- 0
X_test$totals.totalTransactionRevenue[is.na(X_test$totals.totalTransactionRevenue)] <- 0
X_test$totals.transactions[is.na(X_test$totals.transactions)] <- 0
#drop missing columns
drop_names <- c("trafficSource.adwordsClickInfo.criteriaParameters","geoNetwork.networkLocation","geoNe

X_train <- X_train[,!(names(X_train) %in% drop_names)]
X_test <- X_test[,!(names(X_test) %in% drop_names)]
```

Additionally, we'll drop the transaction data from the X variables, including date, and the unique identifiers.

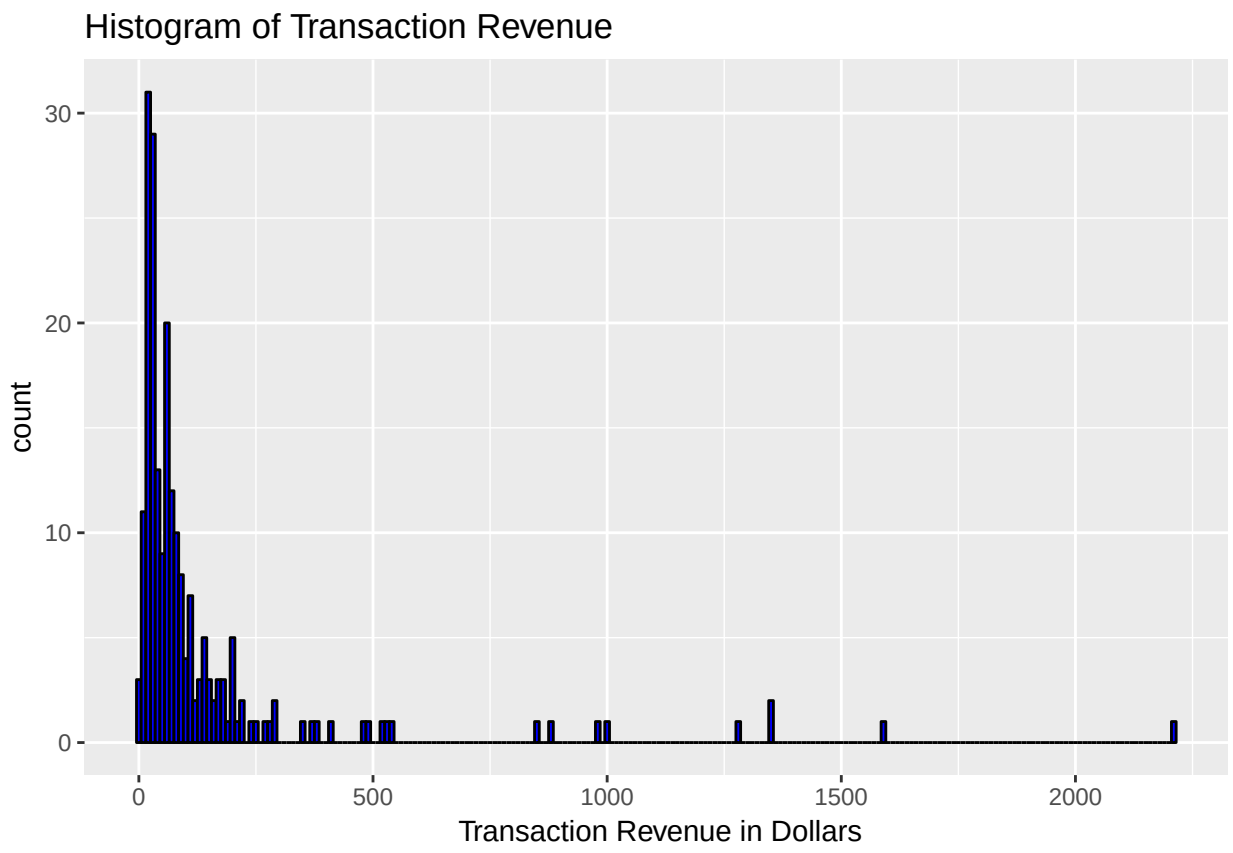
```
# Drop ID columns and total Rev
X_train <- X_train %<>%
  select(-date, -fullVisitorId, -visitId, -visitStartTime, -totals.totalTransactionRevenue, -totals.tra
  mutate_if(is.character, funs(factor(.) %>% as.integer))
X_test <- X_test  %<>%
  select(-date, -fullVisitorId, -visitId, -visitStartTime,  -totals.totalTransactionRevenue,-totals.tra
  mutate_if(is.character, funs(factor(.) %>% as.integer))
# Replace NAs with 0 for y variable revenue
y_train[is.na(y_train)]<- 0
y_test[is.na(y_test)] <- 0
```

For EDA purposes we will transform y variable to actual dollar value. Our data has transaction revenue multiplied by 10^6.

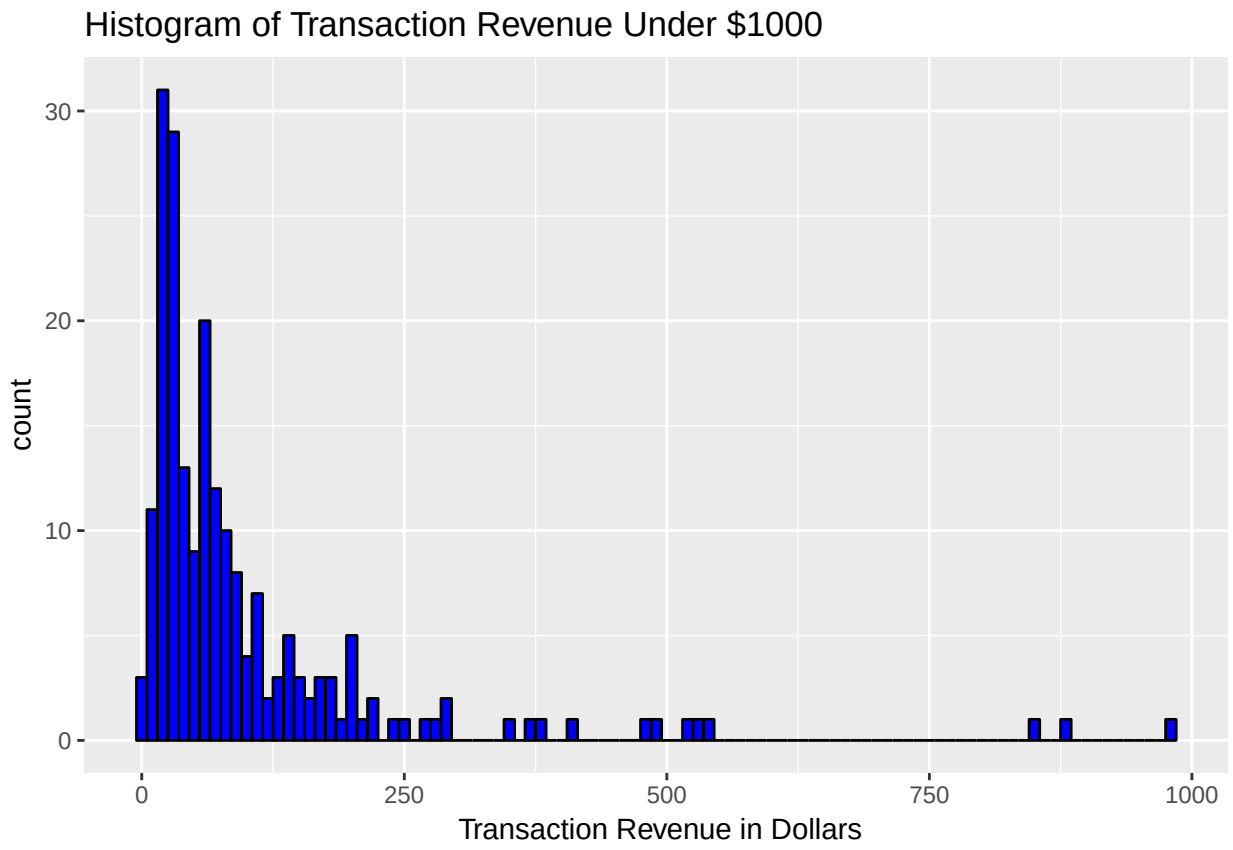
```
train_y_EDA<- y_train/1000000
summary(train_y_EDA)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##     0.000     0.000     0.000     1.365     0.000 2211.380
```

```
# Drop 0 instances
train_y_EDA<- train_y_EDA[!(train_y_EDA==0)]
# drop values over 1000
train_y_EDA1<- train_y_EDA[!(train_y_EDA>1000)]
g <- ggplot(data = data_frame(train_y_EDA),aes(x = train_y_EDA)) +
geom_histogram(binwidth = 10,color='black', fill='blue') + labs(title = "Histogram of Transaction Revenu
print(g)
```




```
gg <- ggplot(data = data_frame(train_y_EDA1),aes(x = train_y_EDA1)) +
geom_histogram(binwidth = 10,color='black', fill='blue') + labs(title = "Histogram of Transaction Revenu
print(gg)
```




```
(summary(train_y_EDA))
```

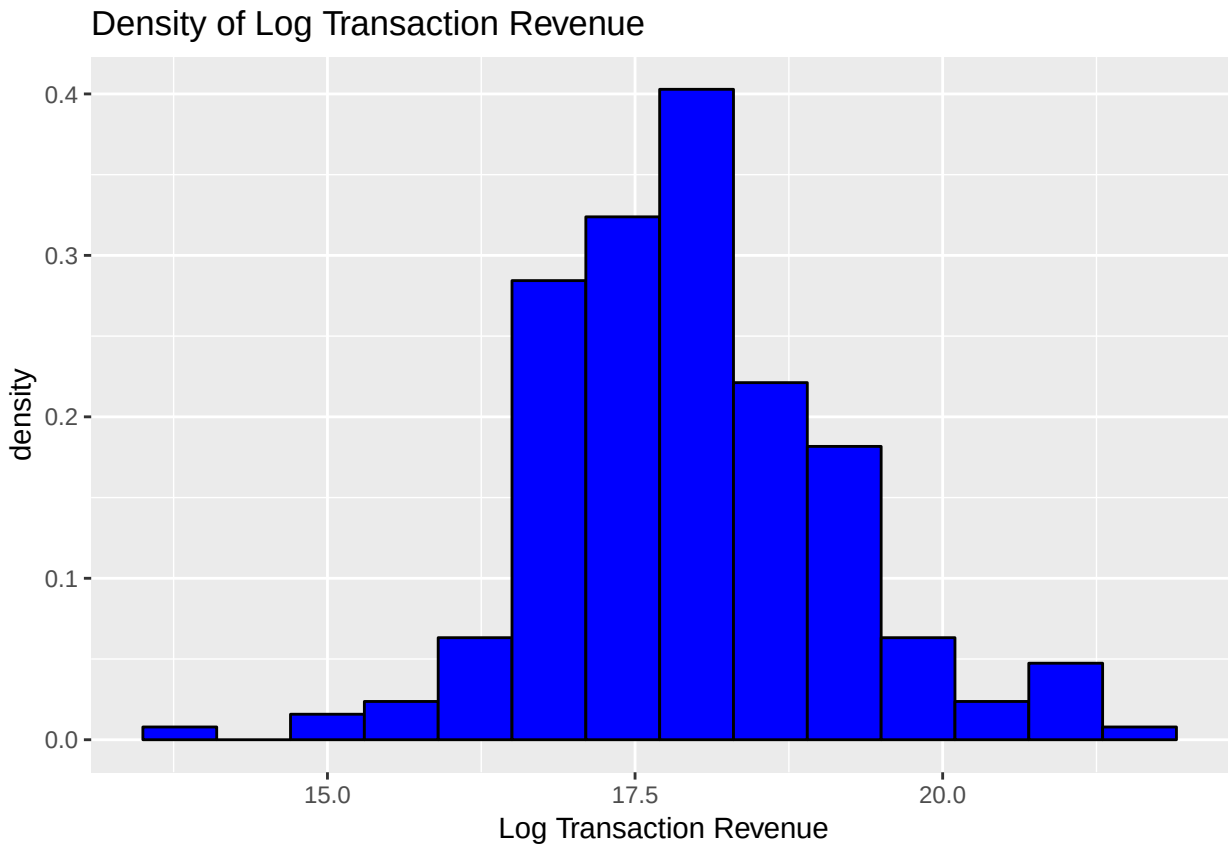
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.20   27.18   60.45  139.46  115.85 2211.38
```

We can see from the above plot that the majority of purchases at the store are low value items. The distribution shows a long positive tail, whose values greatly effect the overall mean of $139.

Let's take a look at the density plot of the natural log of Transaction revenue, which is the y value that we will predict.

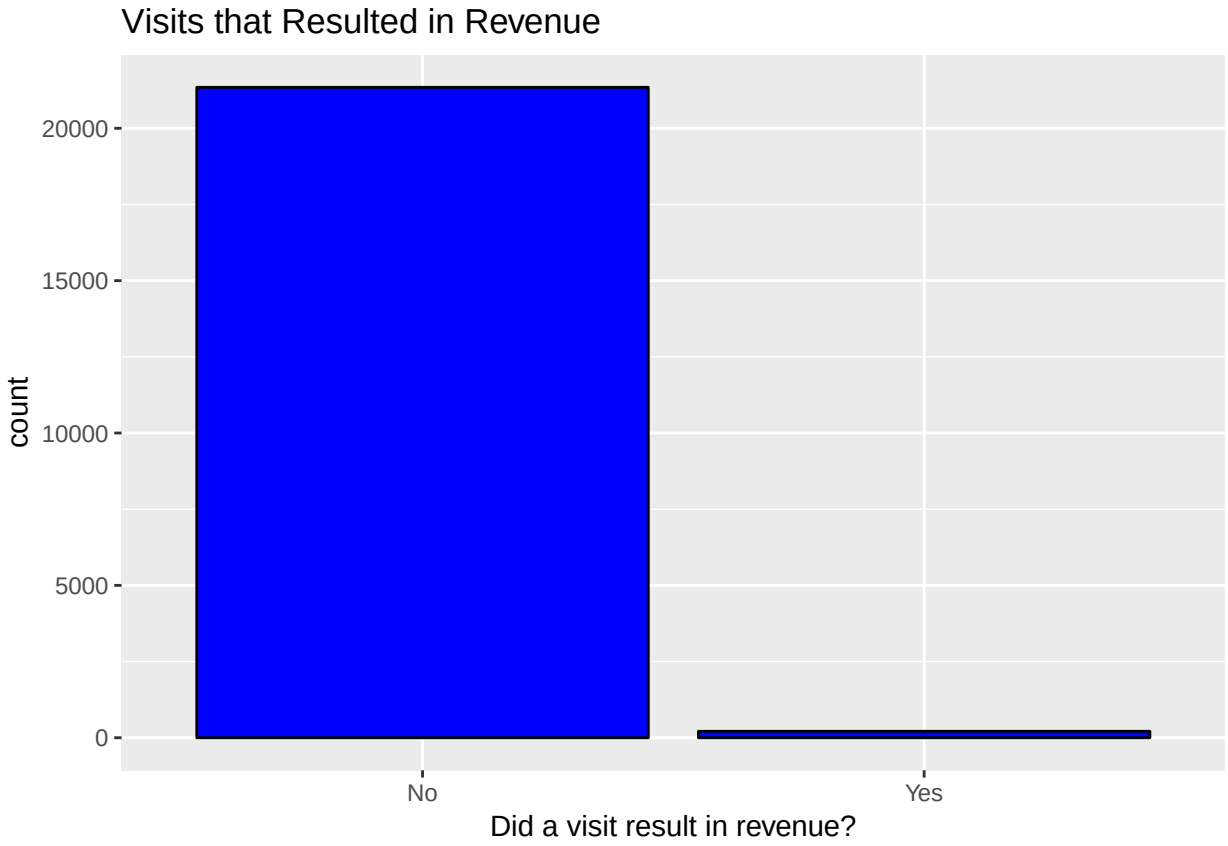
```
train_y_EDA2<- log(y_train)
g <- ggplot(data = data_frame(train_y_EDA2),aes(x=train_y_EDA2, y= ..density..)) +
geom_histogram(binwidth = .6, color='black',fill='blue') + labs(title = "Density of Log Transaction Rev
g
```

```
## Warning: Removed 21344 rows containing non-finite values (stat_bin).
```



We can see in the plot above that the log transformation normalized the transaction revenue distribution.

```
train_y_copy = y_train
bool=ifelse(y_train==0,'No','Yes')
g <- ggplot(data = data_frame(bool),aes(x=as.factor(bool))) +
geom_bar(color='black',fill='blue') + labs(title = "Visits that Resulted in Revenue") + labs(x="Did a v
g
```




```
bool1=ifelse(y_train==0,0,1)
print(sum(bool1)/length(bool1))
```

```
## [1] 0.009788912
```

We can see in the above bar plot, our data is heavily unbalanced. Approximately 1% of all visits to the site result in a purchase.

As we are predicting the log of Transaction revenue, let's transform our y variable.

```
y_test <- log1p(y_test)
y_train <- log1p(y_train)
```

The XGBoost library provides it's own class for input of X and y variables into the model.

```
dtrain <- xgb.DMatrix(data = data.matrix(X_train[split_idx, ]), label = data.matrix(y_train[split_idx ]
dval <- xgb.DMatrix(data = data.matrix(X_train[!split_idx, ]), label = data.matrix(y_train[!split_idx ]
dtest <- xgb.DMatrix(data = data.matrix(X_test),label=data.matrix(y_test))
```

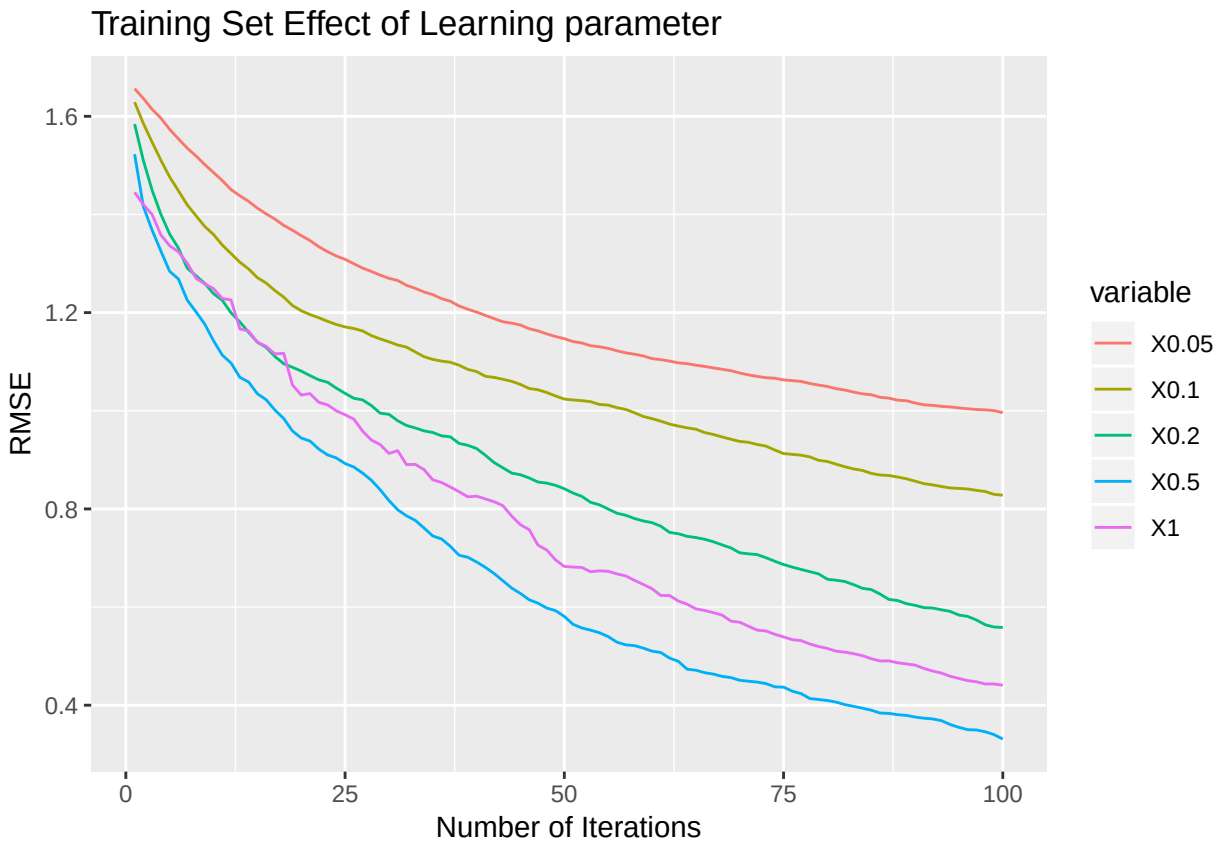
Before hyperparameter tuning, we'll first run some experiments some of the different parameters of the model to see their effects on both training and validation error. We'll keep all other parameters equal and run models on different values for eta, max_depth, and gamma assess the impact of each on training and validation set errors.

Note that this is not part of our hyperparameter tuning, it is only an experiment for learning purposes.

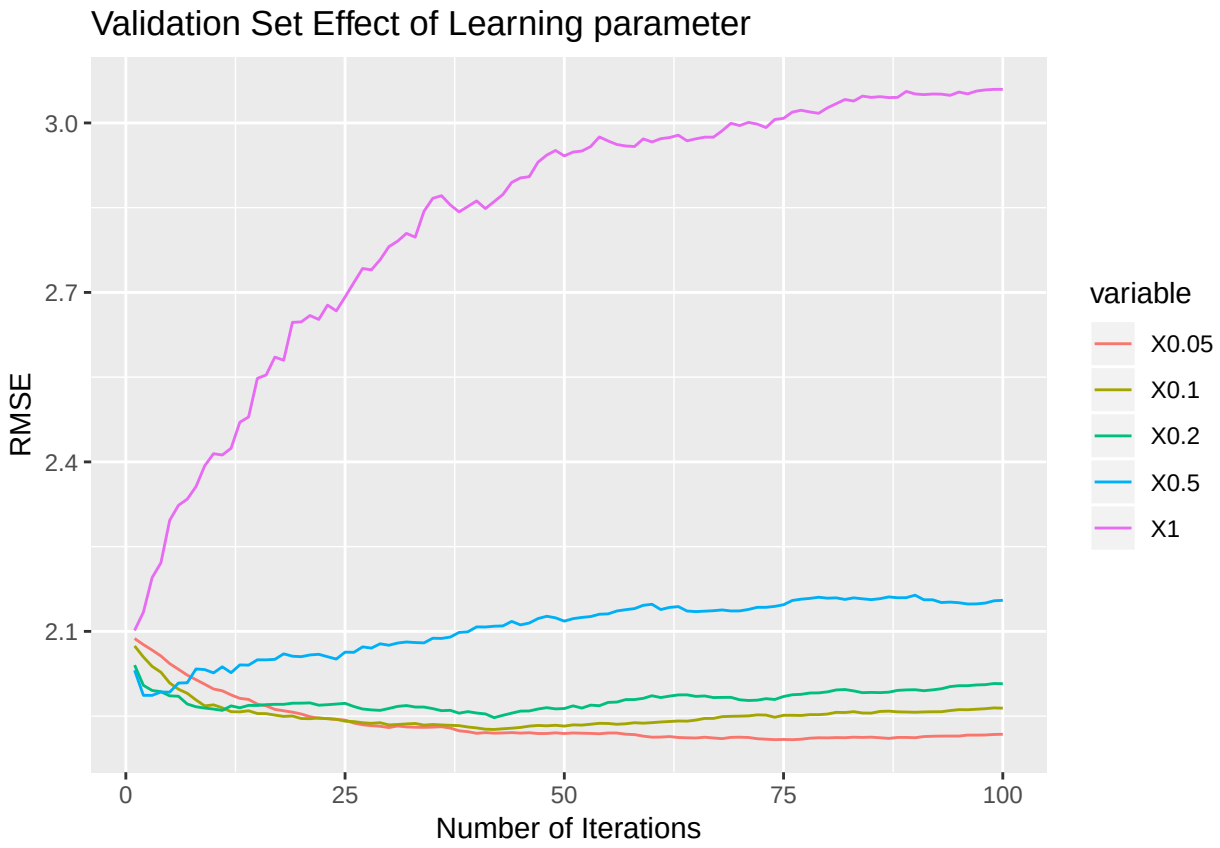
Source: https://insightr.wordpress.com/2018/05/17/tuning-xgboost-in-r-part-i/ - The above resource was used with slight modification to plot the validation set as well.

```
#ETA
set.seed(1)
```

```
eta=c(0.05,0.1,0.2,0.5,1)
train_eta = matrix(NA,100,length(eta))
val_eta = matrix(NA,100,length(eta))
colnames(train_eta) = colnames(val_eta) = eta

for(i in 1:length(eta)){
  params=list(eta = eta[i], colsample_bylevel=2/3,
              subsample = 0.5, max_depth = 6,
              min_child_weight = 1)
  xgb <- xgb.train(data = dtrain, nrounds = 100 , params = params, watchlist = list(val=dval, train=dtra
  train_eta[,i] = xgb$evaluation_log$train_rmse
  val_eta[,i] = xgb$evaluation_log$val_rmse
}

train_eta = data.frame(iter=1:100, train_eta)
train_eta = melt(train_eta,id.vars = "iter")

g<- ggplot(data = train_eta) + geom_line(aes(x = iter, y = value, color = variable)) + labs(title = "Tra
print(g)
```



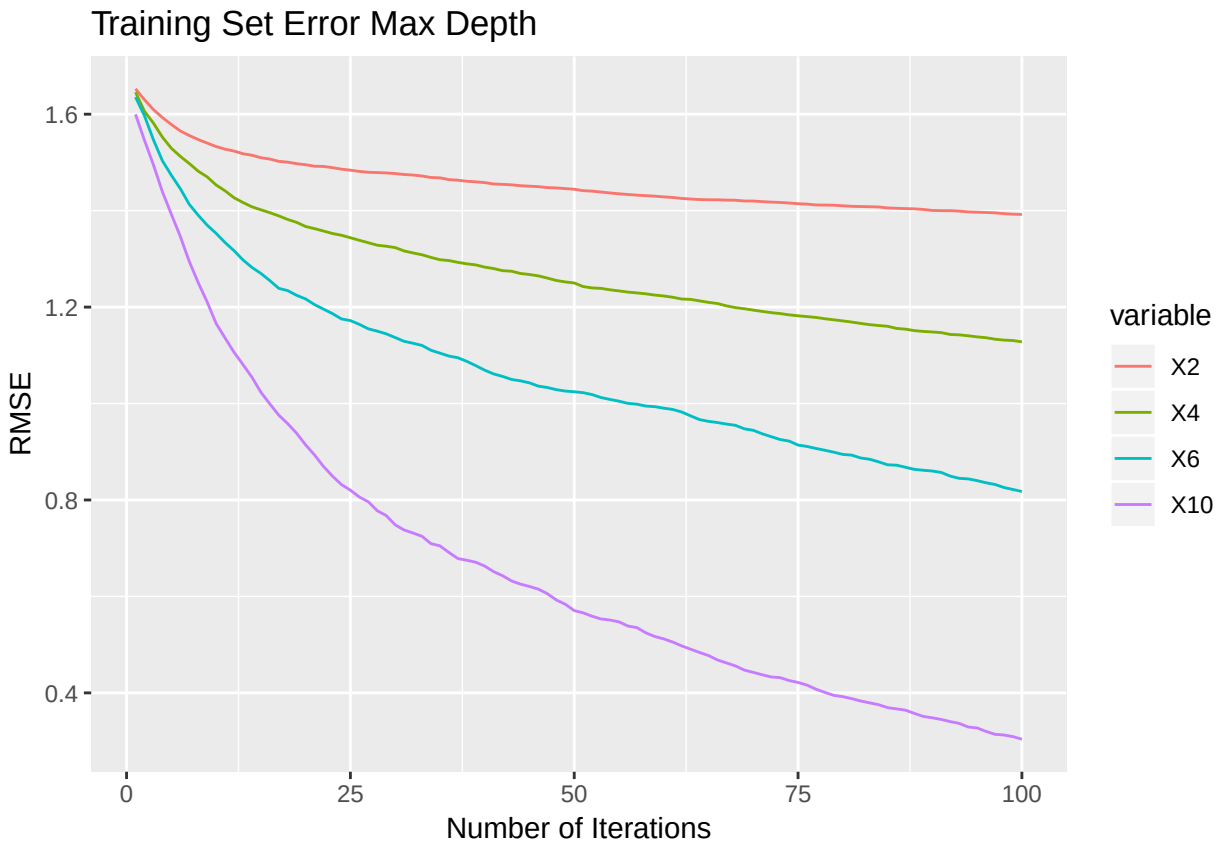

```
val_eta = data.frame(iter=1:100, val_eta)
val_eta = melt(val_eta,id.vars = "iter")

gg<- ggplot(data = val_eta) + geom_line(aes(x = iter, y = value, color = variable)) + labs(title = "Vali
print(gg)
```



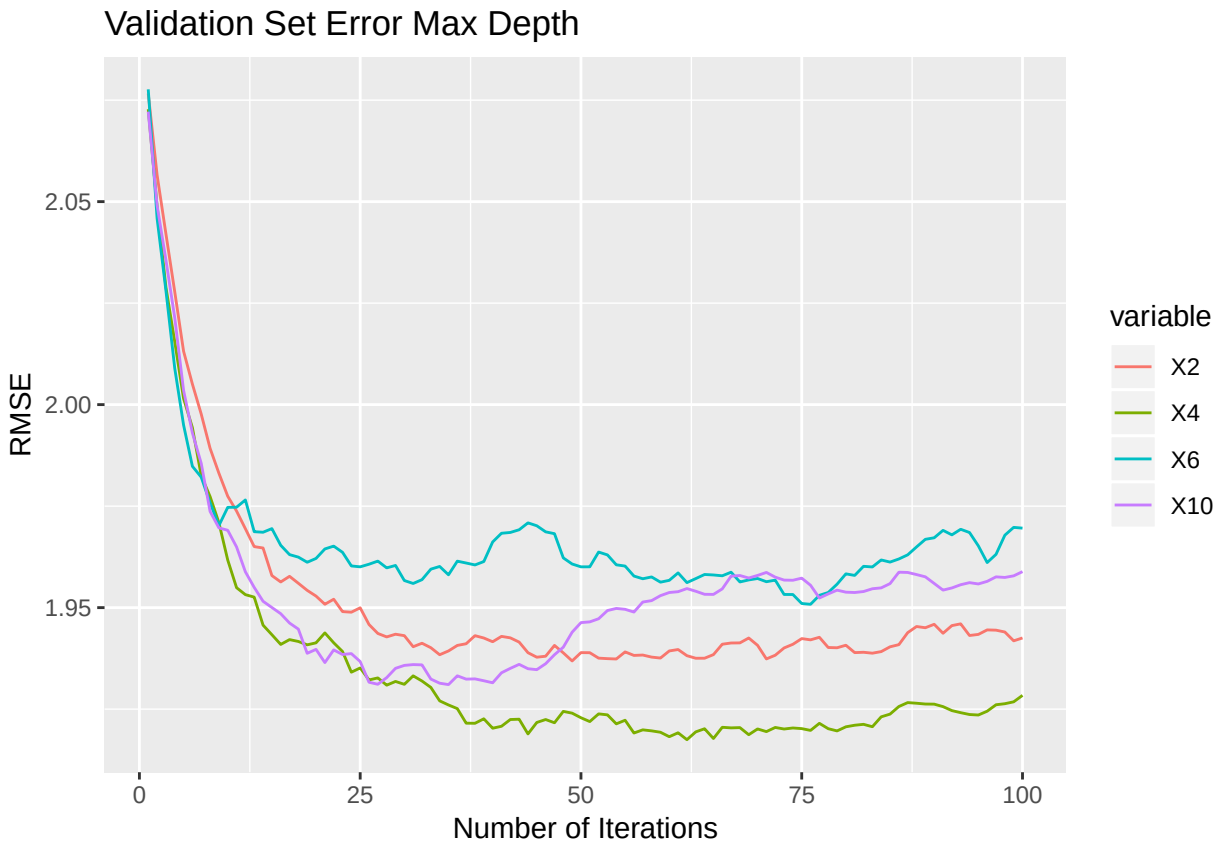
The first parameter that we'll assess is eta, or the learning parameter. This parameter controls the how much of the new model at each iteration is used in the updated model. Float values between 0 and 1 are valid inputs. We can see from the plots above that as the parameter increases toward one, the training set error decreases at a faster rate. However, when we look at the performance on the validation set we can see that the higher values of .5 and 1 perform poorly. This is a clear sign that using values of 1 and 0.5 severely overfit the training data.

```
# Max Depth
md=c(2,4,6,10)
train_md = matrix(NA,100,length(md))
val_md = matrix(NA,100,length(md))
colnames(train_md) = colnames(val_md) = md
for(i in 1:length(md)){
  params=list(eta=0.1,colsample_bylevel=2/3,
              subsample=0.5,max_depth=md[i],
              min_child_weight=1)
  xgb <- xgb.train(data = dtrain, nrounds = 100 , params = params, watchlist = list(val=dval, train=dtra
  train_md[,i] = xgb$evaluation_log$train_rmse
  val_md[,i] = xgb$evaluation_log$val_rmse
}

train_md = data.frame(iter=1:100, train_md)
train_md = melt(train_md, id.vars = "iter")
val_md = data.frame(iter=1:100, val_md)
val_md = melt(val_md, id.vars = "iter")

g <- ggplot(data = train_md) + geom_line(aes(x = iter, y = value, color = variable)) + labs(title = "Tra
```

```
print(g)
```

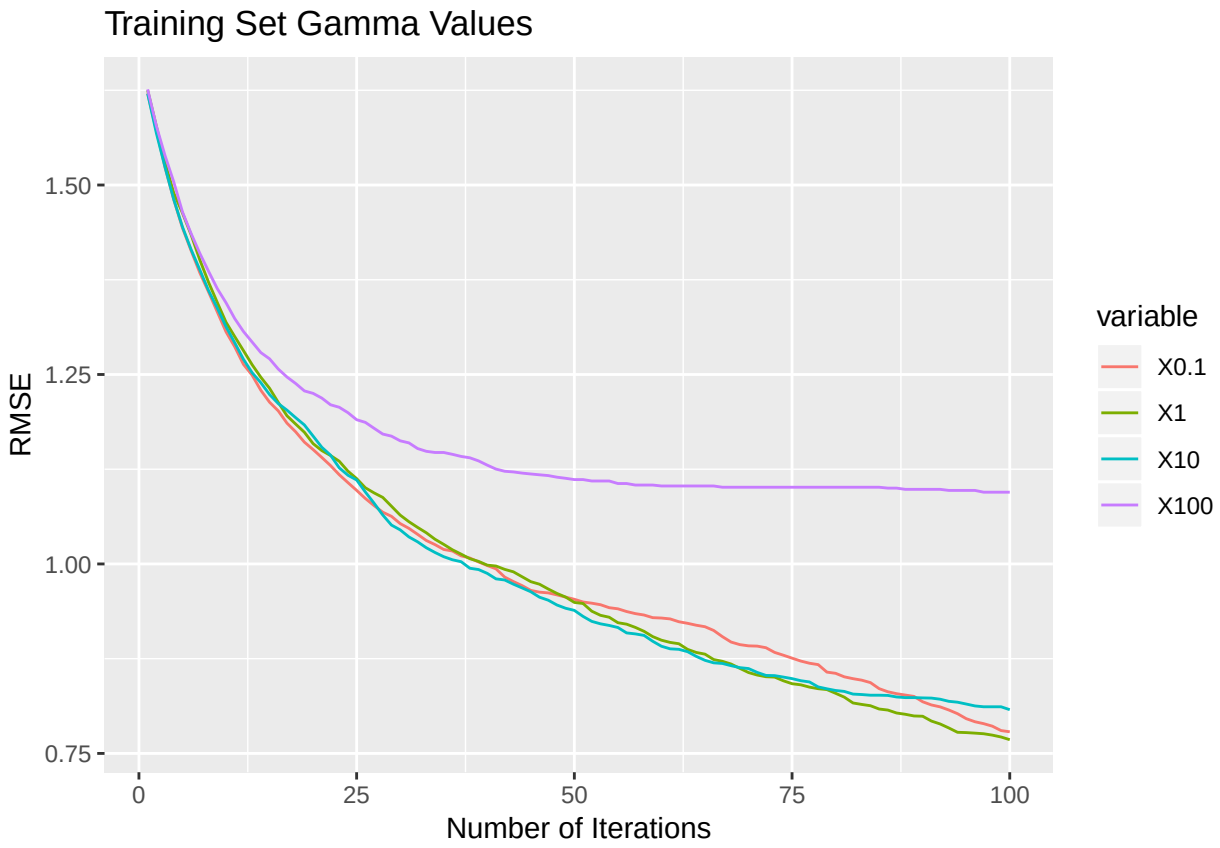



```
gg<- ggplot(data = val_md) + geom_line(aes(x = iter, y = value, color = variable)) + labs(title = "Vali
print(gg)
```

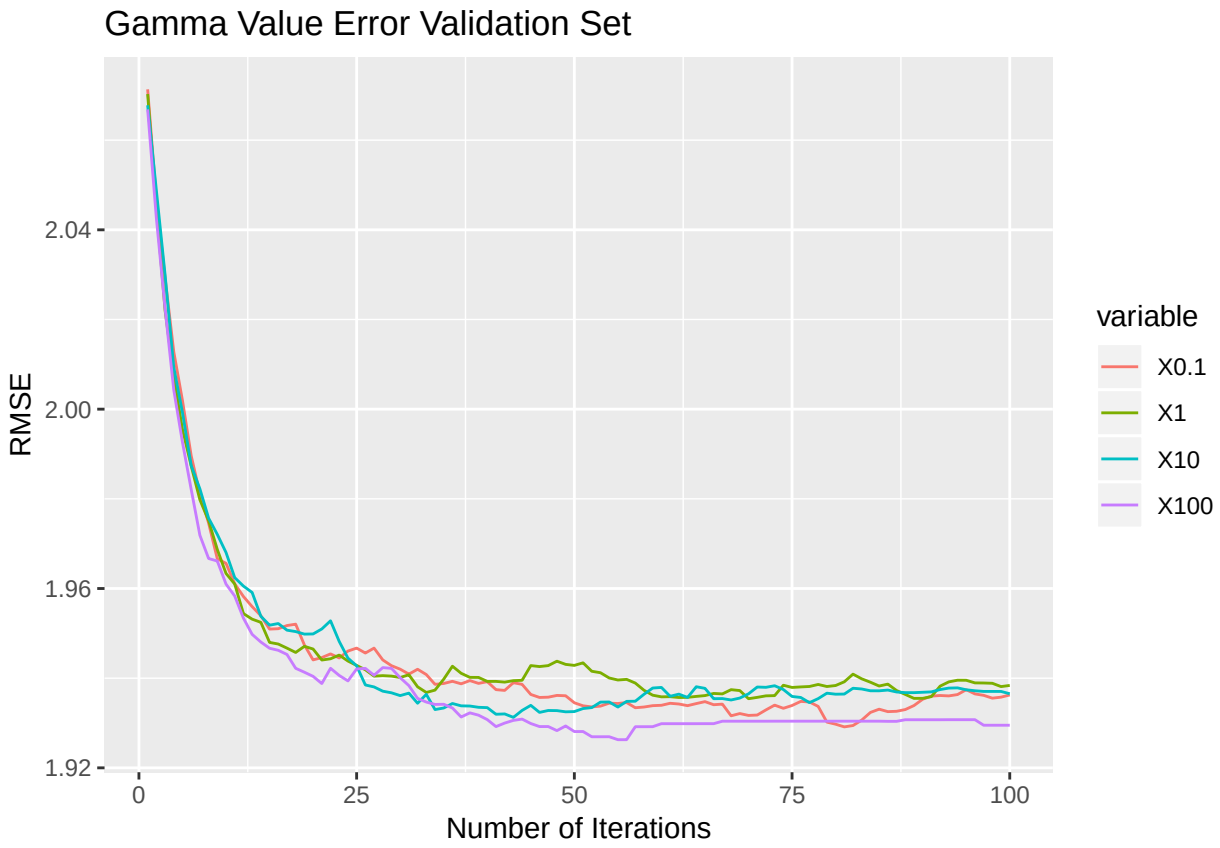


We've run the same experiment this time on the max_depth parameter. This parameter controls the maximum depth of the trees. Given the training dataset we can clearly see that the depth of the tree has a significant impact on the rate of decrease in the RMSE. Greater depth results in better RMSE scores. Again, however when we compare this to how the model performs on unseen data, we plot changes. Depths of 2 and 10 perform poorly, compared to depths of 4 and 6.

```
# Gamma
set.seed(1)
gamma=c(0.1,1,10,100)
train_gamma = matrix(NA,100,length(gamma))
val_gamma = matrix(NA,100,length(gamma))
colnames(train_gamma) = colnames(val_gamma) = gamma
for(i in 1:length(gamma)){
  params = list(eta = 0.1, colsample_bylevel=2/3,
                subsample = 1, max_depth = 6, min_child_weight = 1,
                gamma = gamma[i])
  xgb <- xgb.train(data = dtrain, nrounds = 100 , params = params, watchlist = list(val=dval, train=dtra
  train_gamma[,i] = xgb$evaluation_log$train_rmse
  val_gamma[,i] = xgb$evaluation_log$val_rmse
}

train_gamma = data.frame(iter=1:100, train_gamma)
train_gamma = melt(train_gamma, id.vars = "iter")
ggplot(data = train_gamma) + geom_line(aes(x = iter, y = value, color = variable))+ labs(title = "Train
```




```
val_gamma = data.frame(iter=1:100, val_gamma)
val_gamma = melt(val_gamma, id.vars = "iter")
ggplot(data = val_gamma) + geom_line(aes(x = iter, y = value, color = variable))+ labs(title = "Gamma Va
```



Above we can see the impact of different gamma values. Here we see much less variation in the training dataset with the exception of the extreme value of 100. Values 0.1 through 10 fit similarly on the training data. Looking at the performance on the validation set, here again we do not see as much variance in the performance as we have seen with eta and max_depth. Gamma values of 100 and 1 appear to perform the best.

We'll now use the mlr library to do hyperparameter tuning.

```
set.seed(3231)
params <- makeParamSet(makeDiscreteParam("booster",values = c("gbtree","gblinear")), makeIntegerParam("

X_train <- X_train %>%mutate_if(is.factor, as.integer)

trainDf <- X_train
trainDf$target = y_train

traintask <- makeRegrTask (data = data.frame(trainDf),target = 'target')

valDf <- X_train[!split_idx, ]
valDf$target = y_train[!split_idx ]

testtask <- makeRegrTask(data = data.frame(valDf),target = 'target')

lrn <- makeLearner("regr.xgboost")
```

```
## Warning in makeParam(id = id, type = "numeric", learner.param = TRUE, lower = lower, : NA used as a d
## ParamHelpers uses NA as a special value for dependent parameters.
```

```
lrn$par.vals <- list( objective='reg:linear', nrounds=100L, eta=0.05)
rdesc <- makeResampleDesc("CV",iters=5L)

ctrl <- makeTuneControlRandom(maxit = 10L)

mytune <- tuneParams(learner = lrn, task = traintask, resampling=rdesc, par.set = params, control = ctrl
```

Now that we have tuned the parameters to the model, let's train the final model and see how it performs on the test set.

```
lrn_tune <- setHyperPars(lrn,par.vals = mytune$x)
print(lrn_tune)
```

```
## Learner regr.xgboost from package xgboost
## Type: regr
## Name: eXtreme Gradient Boosting; Short name: xgboost
## Class: regr.xgboost
## Properties: numerics,weights,featimp,missings
## Predict-Type: response
## Hyperparameters: objective=reg:linear,nrounds=5,eta=0.372,booster=gbtree,max_depth=3,min_child_weigh
```

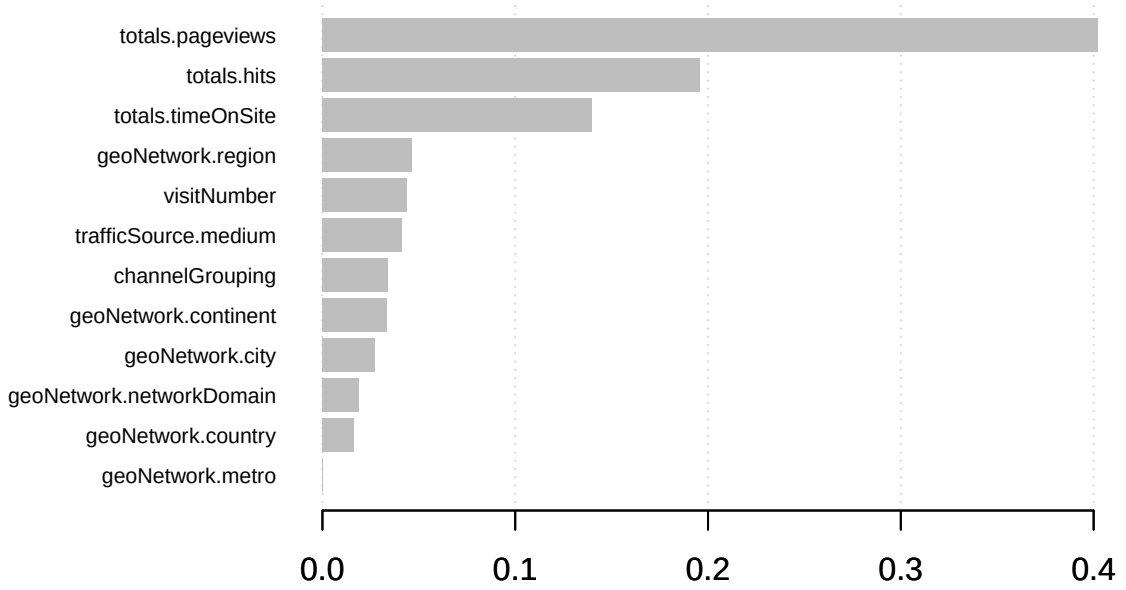
```
xgb_model <- train(lrn_tune, traintask)
```

```
## [1]  train-rmse:1.696682
## [2]  train-rmse:1.640082
## [3]  train-rmse:1.604133
## [4]  train-rmse:1.591608
## [5]  train-rmse:1.568483
```

```
result <- predict(xgb_model, testtask)
# Use both train and validation sets to train final model
dtrain <- xgb.DMatrix(data = data.matrix(X_train), label = data.matrix(y_train))
xgb <- xgb.train(data = dtrain, nrounds=5,eta=0.372,max_depth=3,min_child_weight=4.4,subsample=0.8,colsa
```

```
## [1]  val-rmse:2.204616   train-rmse:1.701702
## [2]  val-rmse:2.143973   train-rmse:1.647273
## [3]  val-rmse:2.108343   train-rmse:1.611315
## [4]  val-rmse:2.101781   train-rmse:1.597502
## [5]  val-rmse:2.077102   train-rmse:1.585193
```

```
X_names <- names(X_train)
importance_matrix <- xgb.importance(X_names, model = xgb)
xgb.plot.importance(importance_matrix)
```




```
importance_matrix
```

```
##                     Feature         Gain        Cover  Frequency
##  1:         totals.pageviews 4.023121e-01 0.4548732560 0.22857143
##  2:             totals.hits 1.955856e-01 0.3207091180 0.20000000
##  3:        totals.timeOnSite 1.400094e-01 0.1376386926 0.14285714
##  4:        geoNetwork.region 4.658029e-02 0.0002895462 0.02857143
##  5:             visitNumber 4.380252e-02 0.0146124327 0.08571429
##  6:     trafficSource.medium 4.127047e-02 0.0095743284 0.08571429
##  7:          channelGrouping 3.412231e-02 0.0009033842 0.02857143
##  8:     geoNetwork.continent 3.323641e-02 0.0022700424 0.02857143
##  9:          geoNetwork.city 2.740760e-02 0.0009805965 0.08571429
## 10: geoNetwork.networkDomain 1.907258e-02 0.0009535722 0.02857143
## 11:       geoNetwork.country 1.650378e-02 0.0004092253 0.02857143
## 12:         geoNetwork.metro 9.692577e-05 0.0567858053 0.02857143
##       Importance
##  1: 4.023121e-01
##  2: 1.955856e-01
##  3: 1.400094e-01
##  4: 4.658029e-02
##  5: 4.380252e-02
##  6: 4.127047e-02
##  7: 3.412231e-02
##  8: 3.323641e-02
##  9: 2.740760e-02
## 10: 1.907258e-02
## 11: 1.650378e-02
## 12: 9.692577e-05
```

We can see that page views, time on site and total hits are the most important predictors in our model. Intuitively, these predictors make sense, let's perform a Chi Squared test to ensure there is a statistical relationship with the y variable exists.

```
high_importance <- chisq.test(X_train$totals.pageviews, y_train)
```

```
## Warning in chisq.test(X_train$totals.pageviews, y_train): Chi-squared
## approximation may be incorrect
```

```
(high_importance)
```

```
##
##  Pearson's Chi-squared test
##
## data:  X_train$totals.pageviews and y_train
## X-squared = 420730, df = 21728, p-value < 2.2e-16
```

```
high_importance1 <- chisq.test(X_train$totals.timeOnSite, y_train)
```

```
## Warning in chisq.test(X_train$totals.timeOnSite, y_train): Chi-squared
## approximation may be incorrect
```

```
(high_importance1)
```

```
##
##  Pearson's Chi-squared test
##
## data:  X_train$totals.timeOnSite and y_train
## X-squared = 1118800, df = 294880, p-value < 2.2e-16
```

```
high_importance2 <- chisq.test(X_train$totals.hits, y_train)
```

```
## Warning in chisq.test(X_train$totals.hits, y_train): Chi-squared
## approximation may be incorrect
```

```
(high_importance2)
```

```
##
##  Pearson's Chi-squared test
##
## data:  X_train$totals.hits and y_train
## X-squared = 511440, df = 29682, p-value < 2.2e-16
```

```
low_importance <- chisq.test(X_train$trafficSource.keyword, y_train)
```

```
## Warning in chisq.test(X_train$trafficSource.keyword, y_train): Chi-squared
## approximation may be incorrect
```

```
(low_importance)
```

```
##
##  Pearson's Chi-squared test
##
## data:  X_train$trafficSource.keyword and y_train
## X-squared = 101.83, df = 13248, p-value = 1
```

Here we can see that the top three variables by importance give us significant p-values indicating that there is evidence to reject the null hypothesis that the variables are independent. Additionally, we have performed a Chi-squared test on the keyword variable, which has very low importance, here we do not find evidence to reject the null hypothesis that the variables are independent.