Kaggle Contest Write Up

Data Science 2014 - Display Advertisement Challenge

Team 4: Roger Rabbit

Members
- Carlos Pereira
- Hilal Shaath
- Lirong Dai
- Vineet Abraham

Instructor: Roger S. Barga

Table of Contents

Introduction

This document describes the steps performed through data analysis and experimentation, as well as the submission results obtained by the team Roger Rabbit for the Display Advertisement Challenge. This challenge was taken on as the final project for the UW Data Science 2014 certification program.

Contest Data Files

The available data consist of a portion of Criteo's traffic over a period of 7 days. There are 2 CSV files containing training and test data. Each row corresponds to an ad displayed by Criteo. There are a total of 13 numerical features and 26 categorical features. The train file has a label for clicked (1) and non-clicked (0) ads.

When looking at the file sizes (11GB train + 1GB test), it is clear that the main problem is how to build a scalable model that can fit as much data as possible while running within acceptable execution times.

The train data was loaded into a relational table allowing us to quickly get stats and samples from the original train set.

```
select count(1) from criteo_train;
 Count(1)
 --------
 45840617

select top 5 * from criteo_train order by Id;
 Id       Label I1   I2  I3   I4   I5   I6   I7 I8 I9  I10  I11 I12  I12  C1       C2
 -------- ----- ---- --- ---- ---- ---- ---- -- -- --- ---- --- ---- ---- -------- --------
 10000000     0    1   1    5    0 1382    4 15  2 181    1   2 null null 68fd1e64 80e26c9b
 10000001     0    2   0   44    1  102    8  2  2   4    1   1 null null 68fd1e64 f0cf0024
 10000002     0    2   0    1   14  767   89  4  2 245    1   3    3    3 287e684f 0a519c5c
 10000003     0 null 893 null null 4392 null  0  0   0 null   0 null null 68fd1e64 2c16a946
 10000004     0    3  -1 null    0    2    0  3  0   0    1   1 null null 8cf07265 ae46a29d
```

Once the train data file was loaded, we generated 2 equally distributed sample files corresponding to 0.01% and 0.001% of the complete train dataset.

Data Analysis

The initial data analysis revealed that numerical features have a large amount of missing data; seven of the features have 20% or more missing values.

```
select (count(I) / 45840617) from criteo_train where I IS NULL;
------------------
I1     0.4536055
```

```
I2     0
I3     0.214644733
I4     0.216780874
I5     0.02580936
I6     0.223651614
I7     0.043255657
I8     0.000496787
I9     0.043255657
I10    0.4536055
I11    0.043255657
I12    0.765078097
I13    0.216780874
```

For the categorical columns, most of the values are present; hence only 5 features have a considerable amount of missing data.

```
select (count(C) / 45840617) from criteo_train where C IS NULL;
-----------------
// omitting other C columns...

C19    0.44006515
C20    0.44006515
C22    0.762534959
C25    0.44006515
C26    0.44006515
```

Notice that C19:C20 and C25:C26 have the exact same amount of missing rows, which can be evidence of a strong correlation between those columns.

First Model: Random Forest

The main reason why we chose to start off with random forests is the fact that decision tree algorithm will automatically select the best set of features from the input data, making it easier to decide what features should be investigated more deeply and what features can be ignored.

Development Environment

The first model was created using R implementation called “RevoScaleR”. The advantage of using this library instead of Weka or the standard R software is that RevoScaleR has built-in functions for classification and regression trees on very large data sets. Their improved data set structures load chunks of the data at each iteration and uses histograms to train the trees instead of the raw data itself.

Data Preparation

Clean Missing Data

The first step was to remove from the training input any “I” column with more than 20% of missing data. The second step was to replace missing data with mean values of the remaining numerical columns. We decided to filter/replace the missing data because that would skew the DT to aggregate samples with NA in the same tree nodes.

```
train$I3[is.na(train$I3)] <- median(train$I3, na.rm=TRUE)
train$I5[is.na(train$I5)] <- median(train$I5, na.rm=TRUE)
...
train$I11[is.na(train$I11)] <- median(train$I11, na.rm=TRUE)
```

Pre-Process Factor Levels

Random Forests in R can only digest factors with up to 32 levels, so we only included C cols having less numerosity. It’s likely that C cols in the train and test data sets will have different factor levels, so we combined both data sets and re-calculated the factor levels.

```
############ Combine data sets and fix factor levels ############
test$Label <- 0
combi <- rbind(train, test)

combi$C6 <- factor(combi$C6)
combi$C9 <- factor(combi$C9)
combi$C14 <- factor(combi$C14)
combi$C17 <- factor(combi$C17)
combi$C20 <- factor(combi$C20)
combi$C22 <- factor(combi$C22)
combi$C23 <- factor(combi$C23)
```

Random Forest Training and Result Evaluation

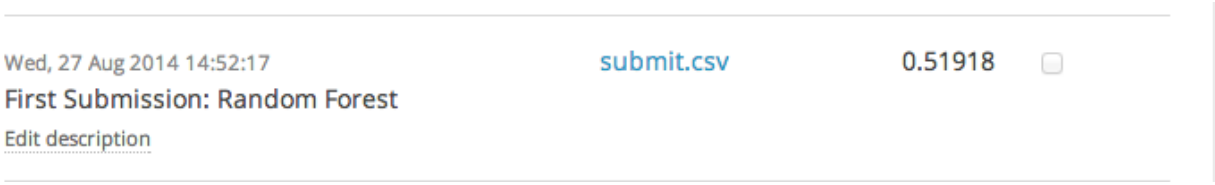
After a few attempts, we found out that using a training file with 1% of the original train data (continuous, not randomly distributed) would be representative enough without consuming excessive amounts of memory and CPU to run.

Thus, we split back the train/test data sets and called the training method ‘rxDForest’, which is the RevoScaleR implementation of the randomForest algorithm.

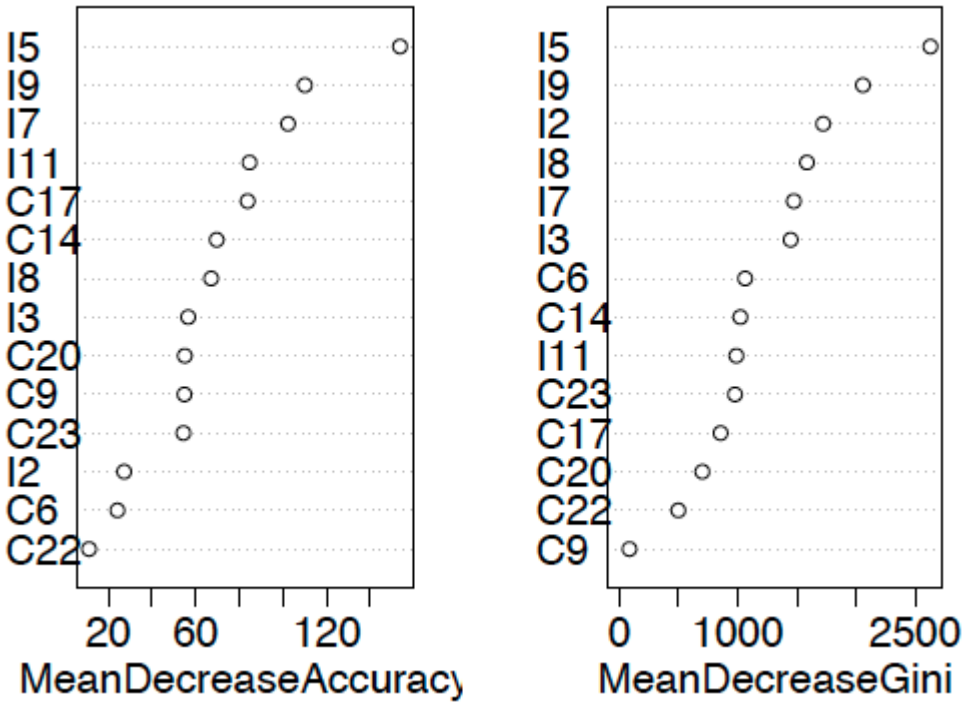
```
train <- combi[1:458406,]
test <- combi[458407:6500541,]

############ Create Random Forest ############
rxForest <- rxDForest(Label ~ I2 + I3 + I5 + I7 + I8 + I9 + I11 +
                        C6 + C9 + C14 + C17 + C20 + C22 + C23
                      , data=train, importance=TRUE, nTree=2000)

############ Make Submission File ############
Prediction <- rxPredict(rxForest, data=test, type="prob", overwrite=TRUE)

# Backup predictions into binary file
```

```
rxDataStep(inData=Prediction, outFile="PredictionBkp", overwrite=TRUE)

# Create CSV file with probabilities
submit <- data.frame(Id=test$Id, Predicted=Prediction$Label_Pred)
write.csv(submit, file="submit.csv", row.names=FALSE)
```

After uploading the first submission to Kaggle, the calculated score was: 0.51918



We also collected the average importance (varImpPlot) of the variables used in our random forest.



Second Model: Boosted Trees and Feature Selection

In our next try, we looked into Gradient Boosted Trees, which is an ensemble of decision trees that incrementally boosts the accuracy based on a loss function of residuals. The motivation for using GBMs came after the class assignment #6: Kaggle blog research. We saw that many

winning models were created using boosted trees (among other algorithms) which gave us the idea to compare its performance against the previous random forest.

In addition to that, we also decided to apply a two-step approach. The first boosting model would use every column in the dataset as a predictor and then use the variable importance and gbm performance output, to fit a final optimized model.

Data Preparation

The train and test files will be the same from the last submission, i.e., let's train the GBM ensemble using the first 1% rows from the train file.

The GBM library in R also has a limit on the number of factors allowed on each categorical feature and we need to select which columns won't be used in the model. A better approach here would be not removing any "C" col, and then applying binning techniques to reduce numerosity and check if there is any valuable signal in those features.

```
## check levels; gbm has a limit of 1024 per factor;
dim(train); dim(test)
sapply(train[,16:41],function(x) length(levels(x)))

# 16    17    18    19    20    21    22    23    24   25    26    27    28    29    30
# C1    C2    C3    C4    C5    C6    C7    C8    C9   C10   C11   C12   C13   C14   C15
# 1002  527 166431  72578 230   14  10042   475   3  22493   4459 144344 3027  26   7739

# In the future we should create bins for those features
c_excluded <- c(18, 19, 22, 25, 26, 27, 28, 30, 31, 33, 34, 36, 39, 41)
train <- train[,-c_excluded]
```

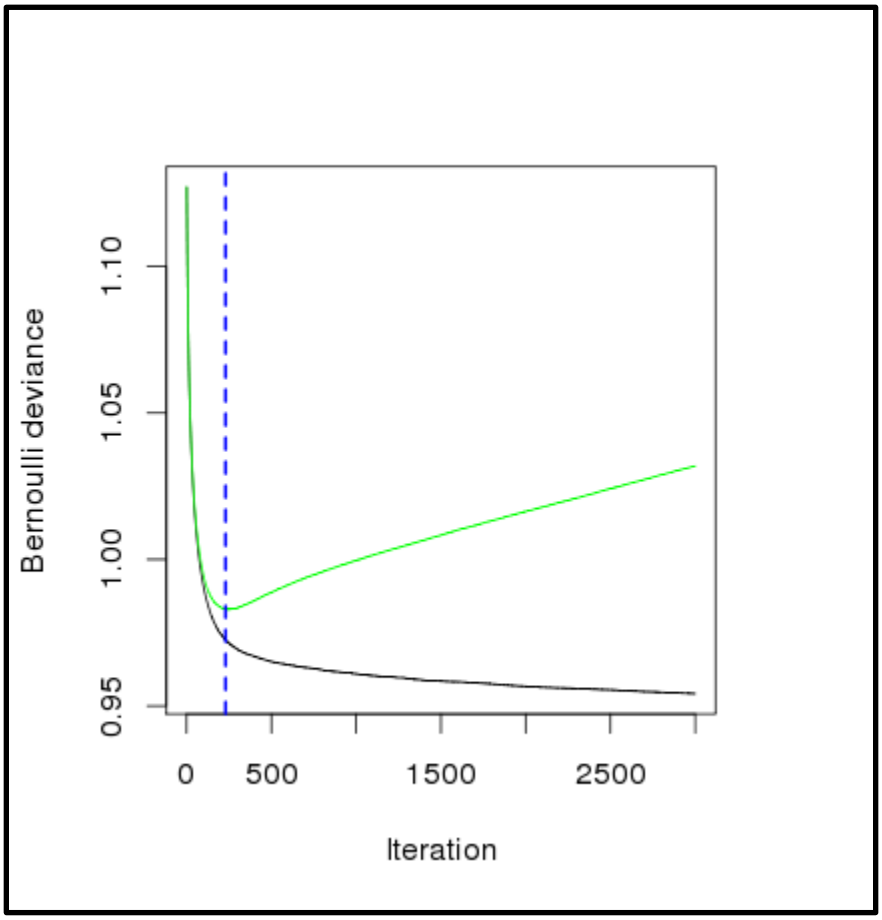
Feature Selection

The next section will fit a model with all non-filtered features using 3K trees and 10 fold cross-validation. Once the model is complete, we will plot the optimal number of trees using the 'gbm.perf' method and the most important variables.

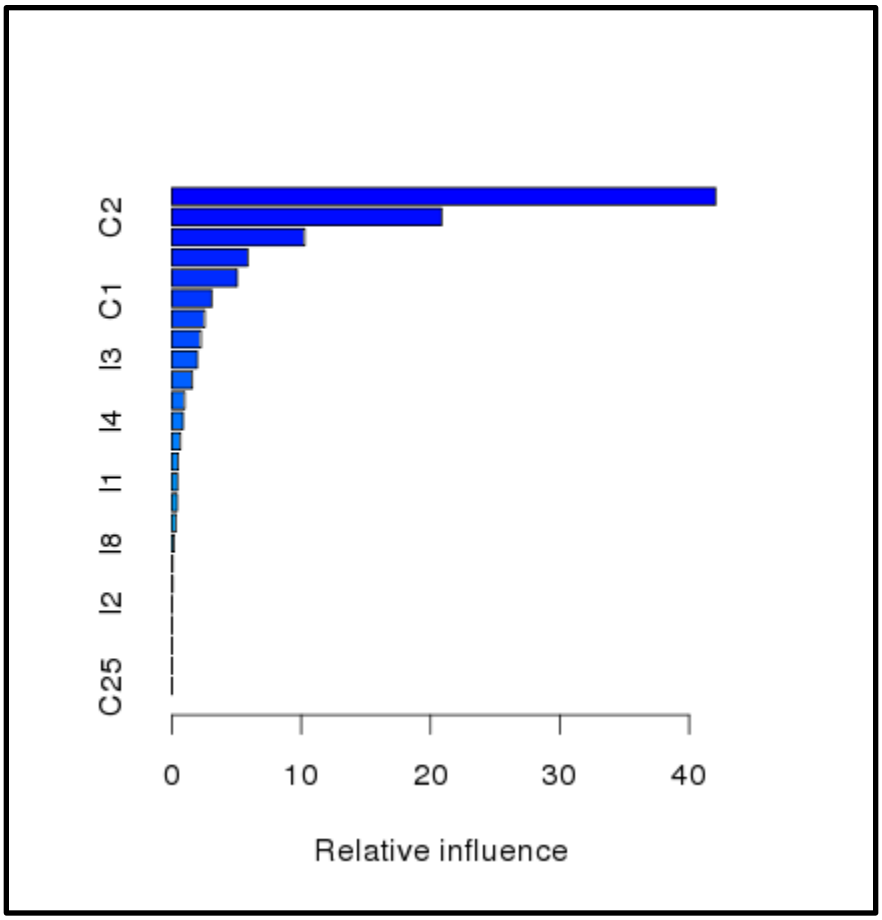
```
## Fit a GBM model to search the best features
library(gbm)
GBM_NTREES = 3000

feature_model <- gbm(Label~. ,data=train[,-c(1)]
                     ,var.monotone=NULL # which vars go up or down with target
                     ,distribution="bernoulli"
                     ,n.trees=GBM_NTREES
                     ,shrinkage=0.05
                     ,interaction.depth=3
                     ,bag.fraction = 0.5
                     ,train.fraction = 1
                     ,n.minobsinnode = 10
                     ,cv.folds = 10
                     ,keep.data=TRUE
                     ,verbose=TRUE)
```

```
best.iter <- gbm.perf(feature_model, method="cv") ##the best iteration number
print(pretty.gbm.tree(feature_model, best.iter))
summary(feature_model, n.trees=best.iter)
```

GBM performance plot:



Feature Summary:




```
# var     rel.inf
# I6   I6 42.04094383
# C2   C2 20.85789215
# I11 I11 10.27174442
# I13 I13  5.86261973
# C14 C14  5.04273909
# C1   C1  3.07837031
# C17 C17  2.54729484
# I5   I5  2.24711525
# I3   I3  1.95978243
# C23 C23  1.58229260
# C20 C20  0.99134385
# I4   I4  0.86641050
# I7   I7  0.66188380
# C8   C8  0.48895777
# I1   I1  0.43074617
# I12 I12  0.39816170
# I9   I9  0.33358313
# I8   I8  0.18087074
# C6   C6  0.05885133
# C22 C22  0.05508166
# I2   I2  0.02170433
# I10 I10  0.02161038
# C5   C5  0.00000000
# C9   C9  0.00000000
```

```
# C25 C25  0.00000000
```

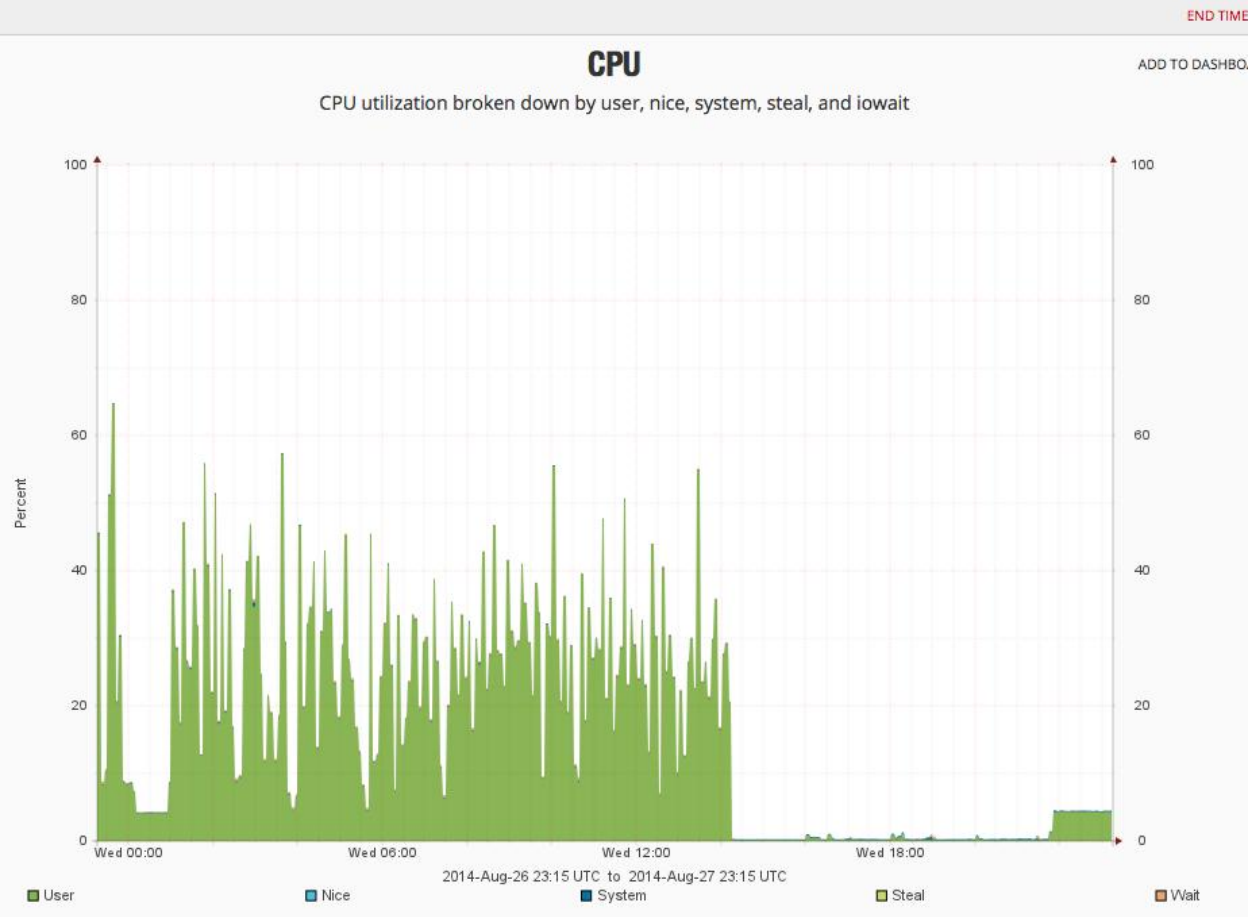
Optimized Boosted Model

Now we will fit a new model using all features with importance greater than zero and run our predictions using the optimal number of trees returned by the gbm.perf function earlier.

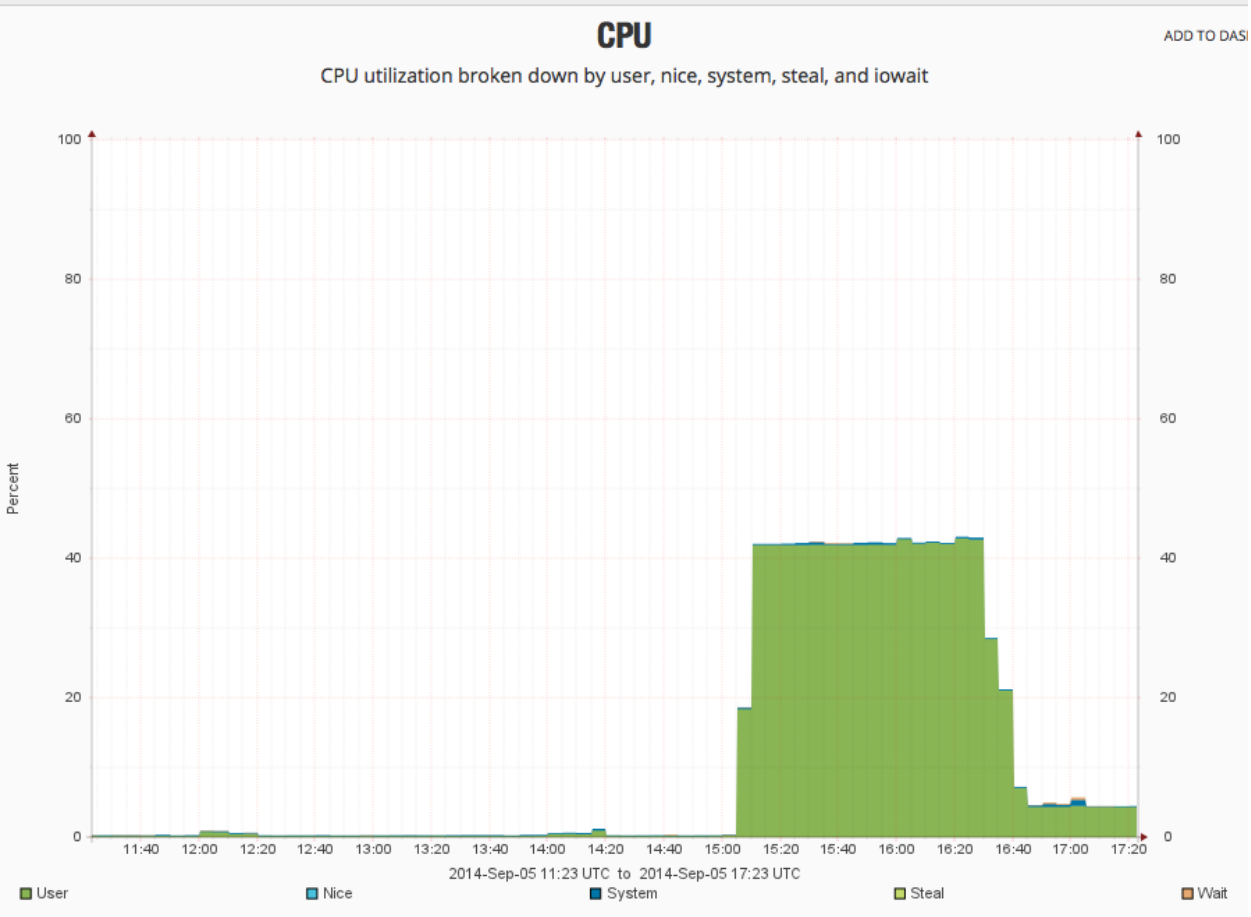
```
top_features <- c(2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20,22,23,24,25,26)

## Build final GBM model
gbm_model <- gbm(Label ~ .
                 ,data=train[,top_features]
                 ,var.monotone=NULL # which vars go up or down with target
                 ,distribution="bernoulli"
                 ,n.trees=GBM_NTREES
                 ,shrinkage=0.05
                 ,interaction.depth=3
                 ,bag.fraction = 0.5
                 ,train.fraction = 1
                 ,n.minobsinnode = 10
                 ,cv.folds = 10
                 ,keep.data=TRUE
                 ,verbose=TRUE)

## Predict probabilities
Prediction <- predict(gbm_model, test, best.iter, type="response")
summary(Prediction)

# Create submission file
submit <- data.frame(Id=test$Id, Predicted=Prediction)
write.csv(submit, file="submit.csv", row.names=FALSE
```

GBM Submission and Forest Comparison

The first visible improvement we noticed using GBMs was performance. Random forests took 7x more time to run in our local environment and they also consumed much more CPU in the meantime. The following charts show the CPU usage during the training of both Random Forests and GBM models.



Random Forest training duration: 15h



Gradient Boosted Trees training duration: 2h

And finally, the Kaggle result obtained from our new model was: 0.50587.

Submission	Files	Public Score	Selected?
Fri, 05 Sep 2014 20:16:48 gbm feature selection, best iter number Edit description	submit_gbm_best_iter.csv	0.50587	☐

This new result is an improvement over the baseline naive random forest (0.51918), but still far from beating the benchmark regression present in the Leaderboard (0.48427), which indicates that maybe the C columns left out due the factor limitation constraints do carry an important signal that's being missed.

Best Model: Vowpal Wabbit

Baseline Model

Since many of us were having performance issues due to the large size of both the train and test datasets, we took to the forums to see what other tools people were using for analysis. We found Vowpal Wabbit (VW), a fast and scalable machine learning language originally created by Yahoo! Research. The VW input format was very different from the csv files we were given from Criteo, so the first step was to convert the data into the format below using Python.

```
-1 '10001610 |i I9:17 I8:17 I3:4 I2:19 I5:20355 I4:17 I7:0 I11:0 I13:17 I12:0 |c 74ef3502
174e4cac 9.17E+83 9a422971 2ed68727 e5ba7672 9900b0a7 18231224 07d13a8f a73ee510 0b153874
7ecf3c0b 2c16a946 05db9164 cdbfd303 fe6b92e5 3.09E+78 64712dc5 3a171ecb 27cf34b3 9117a34a
```

The “-1” indicates “no click”, the “‘10001610” is the record number, the “|i” indicates numerical values, and the “|c” indicates categorical values. This entire process (both train and test sets) took less than an hour. Once the data was converted, we were able to begin creating the initial model.

VW learns by iterating through training samples one at a time and optimizing a loss function. One of the loss functions it can optimize is logarithmic loss for logistic regression, which is also the same method by which this competition is scored. We ran the following code to create the model and predict the outputs for the test dataset.

```
# Training VW:

./vw click.train.new.vw -f click.model.vw --loss_function logistic

# Testing VW:

./vw click.test.new.vw -t -i click.model.vw -p click.preds.txt
```

After uploading this submission to Kaggle, the calculated score was: 0.47973, which was an overall improvement from our previous entries.

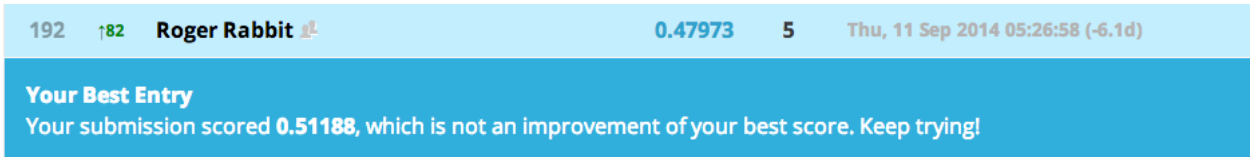
178	↓36	Chris & Tom	0.47973	7	Thu, 21 Aug 2014 21:38:34 (-8.1d)
179	↓36	Anne	0.47973	4	Thu, 28 Aug 2014 04:42:51 (-3.5h)
180	**↑76**	**Roger Rabbit**	**0.47973**	**4**	**Fri, 05 Sep 2014 20:16:48 (-16.6h)**
181	↓37	perdo	0.47973	1	Sat, 19 Jul 2014 22:58:20
182	↓37	jay_Mtl	0.47973	3	Mon, 18 Aug 2014 21:06:47 (-0.7h)

Feature Selection

The baseline model in VW used all the training data to create itself. VW does have some options for data preparation but we did not get well familiarized enough with the program and decided to try and use Weka for that step instead. Using the 0.01% random sample (larger datasets in Weka kept running into memory issues) from the train set and the InfoGain attribute evaluator, the following 9 attributes had an InfoGain value of at least 0.10: C3, C21, C16, C4, I5,

C10, C7, C24, and C11. After removing all the columns that had an InfoGain score of less than 0.10 from both the train and test sets, and running the VW model again, we created a new set of predictions.

Unfortunately when submitting these predictions to Kaggle, they did not result in a better score. We achieved 0.5188 vs our baseline VW score of 0.47973.



The reasons for this could be many. The train set used to perform the InfoGain evaluation was only 0.01% of the total data, this sample may not have been a good representation of the overall data. Obviously there was some useful signal within some of the features that we removed. Not being as familiar with VW and its capabilities may also be preventing us from optimizing the data preparation portion of this challenge.

Conclusion

Through this competition, we have learned that the first important thing for any data science project is to properly understand the datasets. For this particular project, the very first challenge was to build a scalable model that can fit as much data as possible while running within acceptable execution times. It was also beneficial to select models that can help us understand the data first. For example, our first model, Random Forest helped us with feature selection for our subsequent models. Another valuable lesson we learned with this project is no one model is good for all datasets. Therefore, no matter which model we ended up using, data preparation was always an important first step. Dataset needs to be customized and processed for each model. Only in doing so can we achieve the best gains for our models.

The following are our insights of using R and VW:

Some notes from using VW:

- Great at modeling large datasets when resources are constrained.
- Difficult low-level language to use, therefore data-preparation is best suited for another tool.
- For the VW baseline model, the entire train dataset was used to create the model. This allowed the model to find even minimal signal in every feature. While this is great in theory, it makes it very hard to improve the model by using less data.
- It may have been more fruitful to do some initial feature selection, then create a baseline model, and then do data preparation or parameter tweaking to improve it further.
- While definitely a good challenge to use a new tool like VW, we will probably go back to using more familiar tools like R, Python, and Weka in the future.

More notes from experience using R:

- Flexible and easy to run on small to medium models, not so good on large models since it always tries to load the entire data set in memory before running the calculations.
- Can be extended to handle big data using third party software (free for academic only, enterprise is expensive)
- Very good plotting libraries to produce nice visual representations of the models.
- Well documented and easy to find implementations of the major machine learning algorithms: decision trees, linear regression, SVMs, etc.
- Most of the ensembles have very low limits on the number of factors allowed on each categorical feature. Leaving nominal features with high numerosity out of the model is not a good idea. Extra work with feature engineering is necessary (discretization, binning, etc.).