COMPM041:Group Assignment
Click Through Rate
The Centzontotochtin Principle

Jonathan Bourne
John Goodacre
David Twomey

April 2016

Abstract

In this project we developed several approaches to predicting click through rate for real time bidding on online advertising. Our initial approach took us down a dead end in terms of predictive ability although we developed some novel approaches to solve the issue. Our final solution used feature hashing and a sparse matrix implementation of the XGboost algorithm with feature engineering to reduce over-fitting on rare occurrences. Our final score on the public leaderboard was 0.80815, successfully beating the baseline.

1 Introduction

In this report we outline methods that can be used to predict click through on on line advertising. This is very useful to know as it allows companies to model how effective their campaigns will be for a given amount of money and bidding strategy. Online click through data is in general very sparse with thousands of impressions before a click, this provides many challenges for the prospective modeller as so few positive observations can lead to always predicting no. Although it is possible to re-balance such data sets by weighting the importance of the positive and negative classes, in this project we will use the Area Under the ROC Curve or AUC, to ensure that negative and positives are both included more equally.

1.1 Team Roles

This project was highly collaborative, with all members active in discussing techniques and solutions to problems that arose underway, as the language was R, Jonathan Bourne did most of the implementation.

2 Statistical Analysis of the training data and feature engineering

Table 1 shows a break down of the data types and unique values of each column, as can be seen the distribution of values is highly skewed with a few variables having only a single value, whilst others (UserID) are almost entirely composed of unique identifiers. Data sets such as these need to be viewed carefully as simply putting in all the data without cleaning can lead to under performance of models, or massively over fitting of the model. In total there were 2847802 observations and 2057 psotive click examples. This created a highly skewed data set which makes modelling particularly challenging.

	variable	uniques	classType
1	Click	2	integer
2	Weekday	7	integer
3	Hour	24	integer
4	Timestamp	1862073	numeric
5	Logtype	1	integer
6	Userid	2641292	character
7	UserAgent	36	character
8	IP	489690	character
9	Region	35	integer
10	City	370	integer
11	AdExchange	3	integer

12	Domain	15146	character
13	URL	715316	character
14	AnonymousURLID	1	character
15	AdSlotId	51529	character
16	AdSlotWidth	11	integer
17	AdSlotHeight	6	integer
18	AdslotVisability	4	integer
19	AdSlotformat	3	integer
20	AdSlotFloorPrice	197	integer
21	CreativeID	11	character
22	KeypageURL	2	character
23	AdvertiserID	1	integer
24	UserTags	720665	character

Table 1: details of the variables in the data set

2.1 Full Matrix approach

Initial feature engineering was performed in order to try and create a data set that provided a balance between tractability and maximisation of useful data.

1. User agent was split from being both operating system and browser to being to separate variables: operating system and browser.
2. The tags variable was split into one hot vector encoding for each tag type
3. There were an extremely large number of URLs and Domain groups, these were changed into URLgroup and DOMgroup, which were 1 if the Domain or URL have ever had a click and 0 otherwise.
4. Selected vfactor variables were expanded to one hot vector encoding, this included Usertags, AdExchange, CreativeID, AdSlotformat, AdslotVisability.
5. a total number of tags variable was included that showed how many tags each row had.

2.2 Sparse Matrix approach

1. Remove variables UserID, Logtype, AdvertiserID,
2. URL, IP, AdSlotId and Domain are remade so that all ID's that occur less than 30 times (IP less than 100) are renames as a single "smallgroup" class, this reduces the number of expanded variables by approximately 99%, the cut off were based on an analysis ranking the total number of clicks per count group.
3. The data is split into a training and test set for trailing and tuning of the algorithms; the full data set will be used for the final model.
4. The train and test data sets are converted to a feature hashed sparse matrix and weighted to speed the boosting process.
5. Train, tune and compare models
6. Rebuild final model with correct parameters on full data set of 2847802 observations.

2.3 Data set Engineering Full matrix only

Due to memory requirements it wasn't possible to use the entire data sets, and hence, as a result, sub sampling was needed as clicks are so sparse random sampling could result in the data set not containing any positive examples or so few positive examples that training wasn't possible. In order to get around this problem, stratified sampling was used combined with k-fold cross validation. However it was soon discovered that as the data set was so large, a reasonable k-folds score may not reflect the resultant AUC score that would be obtained in the test set, in other words the model was still overtrained even though cross-validation suggested it was reasonable. To try and avoid taking samples that did not reflect the true structure of the data, a method for creating ensemble models was developed following the steps below:

1 Separate positive and negative clicks into two separate data sets

2 Create a hold-out set of both the negative and positive data sets

3 Bootstrap re-sample the positive training data set.

4 Randomly sample the negative train data set for 20,000 data points

5 Combine the two sampled data sets and repeat steps 3 and 4 x times.

The above process was used to create multiple data sets which contained essentially all the same positive examples of click-through (small variations due to the bootstrapping) and random samples of the negative click-through. This collection of data sets could then be used to create an ensemble of models. the hold-out data was used to test the quality of the models. As some of the one-hot-vectors had very few positives, when the data was sampled they could be constant: this can create problems in the prediction and as a result these variables were removed from each data set individually.

3 Model Implementation and Hyper-parameters

As was implied in section 2.3 the extreme imbalance in the data set was a challenge when it came to predictive modelling and the modelling pipeline was quite an iterative process.

Core to our modelling process was the modelList function shown in code listing 1. This function took the list of data sets we had generated and a model training object (see listing 2 for an example) and outputted a list of trained models. In order to correct for the class imbalances, even after the down sampling we had performed, the data was re-weighted. This re-weighting was calculated for every data set the code can be seen on lines 18 to 21 of 1

The tuning of hyper-parameters of the model is generally a key and important part of the model making process, however, some models do not need to go through such optimization. In our model-making process, our model generation framework, bagged trees, logistic regression did not need hyper-parameter tuning, however the SVM and random forest did. We used the grid search settings from the `caret` package and built a switching mechanism into our model list generator so that models which didn't implement grid search weren't calculated multiple times, this can be seen in line 4 of code listing 1.

Due to the time to fit the list of models, parallel processing was needed, however the memory requirements are so considerable that, the fitting process was done on the cloud using high-performance multi-cored processors running a linux system.

Listing 1: The Model List function, this was core to out multiply re-sampled modelling technique

```
1
2  modellist<- function(trainmodel, trainsets, GridOn = TRUE){
3      ctrlAUCnone <- trainControl(
4                          method = ifelse(GridOn,"cv","none"),
5                          number = 5,
6                          returnData =FALSE,
7                          ## Estimate class probabilities
8                          classProbs = TRUE,
9                          summaryFunction = twoClassSummary,
10                         #describe the steps on each iteration
11                         verboseIter= TRUE,
12                         #trim model to reduce size
13                         trim = TRUE)
14
15     Logsemble<- lapply(trainsets, function(n){
16       dataNone <- n
17
18       pos <- sum(dataNone$Click ==1)
19       posweight <- (nrow(dataNone)-pos)/pos #
20       weightvect <- rep(1, nrow(dataNone))
21       weightvect[dataNone$Click ==1] <- posweight
22
23       modLogNone <- eval(trainmodel)
24     }
```

```
  )
}
```

Listing 2: The train Object was used as input to the Model List function

```
Logtrain <- substitute(train(dataNone[,-1],
                        make.names(dataNone$Click),
                        method = 'glm',
                        family = 'quasibinomial',
                        trControl =ctrlAUCnone,
                        metric='ROC',
                        weights = weightvect))
```

4 Results

Although normally tree models are very effective methods for prediction, problems due to the sparseness of the data set meant the initial tree models scored well on k-fold cross validation and hold out data, but terribly in the test set getting results as low as 0.5. This was partially to do with the way the hold-out data was designed, initially hold out data was not in the proportions of the dataset as whole, this meant that the models were not being evaluated using the correct weighting with positive click prediction being much more heavily represented that in the true dataset. With the tree models being fairly unsuccessful, a simple logistic regression was used using 1% of the total data set, this proved to be the best model for a long time, holding the maximum predictive power at 0.73271 as the team struggled with managing the imbalance issues.

Analysis as to why the results were not meeting baseline was conducted with a plot of the correlation of predicted values between the two most successful model groups: Logistic Regression and Bagged Trees. Figure 1 shows how Logistic regression outputs models that are quite stable as the three predictions are all highly correlated with each other. Conversely, the Bagged tree model has much weaker correlation indicating lower levels of model stability which is possibly why it seldom outperforms the Logistic regression on tests of this data set. Within-group correlation is also low, this is usually a good thing as it means that the models can be successfully ensembled to produce better overall results than one model alone, as the models can fill in for each others weak spots. In this case however it made no difference to model performance, which is probably related to the lack of consistency within the tree model.

An attempt at ensembling was made and hence the models were checked for correlation as shown in table 2. The models chosen for ensembling were GLM, treebag and black boost as they had all performed similarly (GLM being the best), however ensembling returned results that were in fact worse than a single Logistic regression implementation of the GLM.

	glm	rpart	blackboost	svmRadial	svmRadialWeights	treebag	xgbTree
glm	1.00	0.37	0.87	0.75	0.74	0.91	0.91
rpart	0.37	1.00	0.73	-0.04	-0.03	0.28	0.63
blackboost	0.87	0.73	1.00	0.61	0.61	0.67	0.87
svmRadial	0.75	-0.04	0.61	1.00	1.00	0.52	0.46
svmRadialWeights	0.74	-0.03	0.61	1.00	1.00	0.50	0.44
treebag	0.91	0.28	0.67	0.52	0.50	1.00	0.92
xgbTree	0.91	0.63	0.87	0.46	0.44	0.92	1.00

Table 2: A correlation plot of the different model types for the Full matrix approach

4.1 Sparse Matrix

As mentioned in the introduction after failing to make progress using traditional methods the team took a step back and fundamentally re-analysed the problem approach. As a result the sparse matrix approach was implemented and proved much more successful as it dramatically reduced the memory constraints that were affecting model build quality. This is a particular weakness to the R language which can make it inefficient

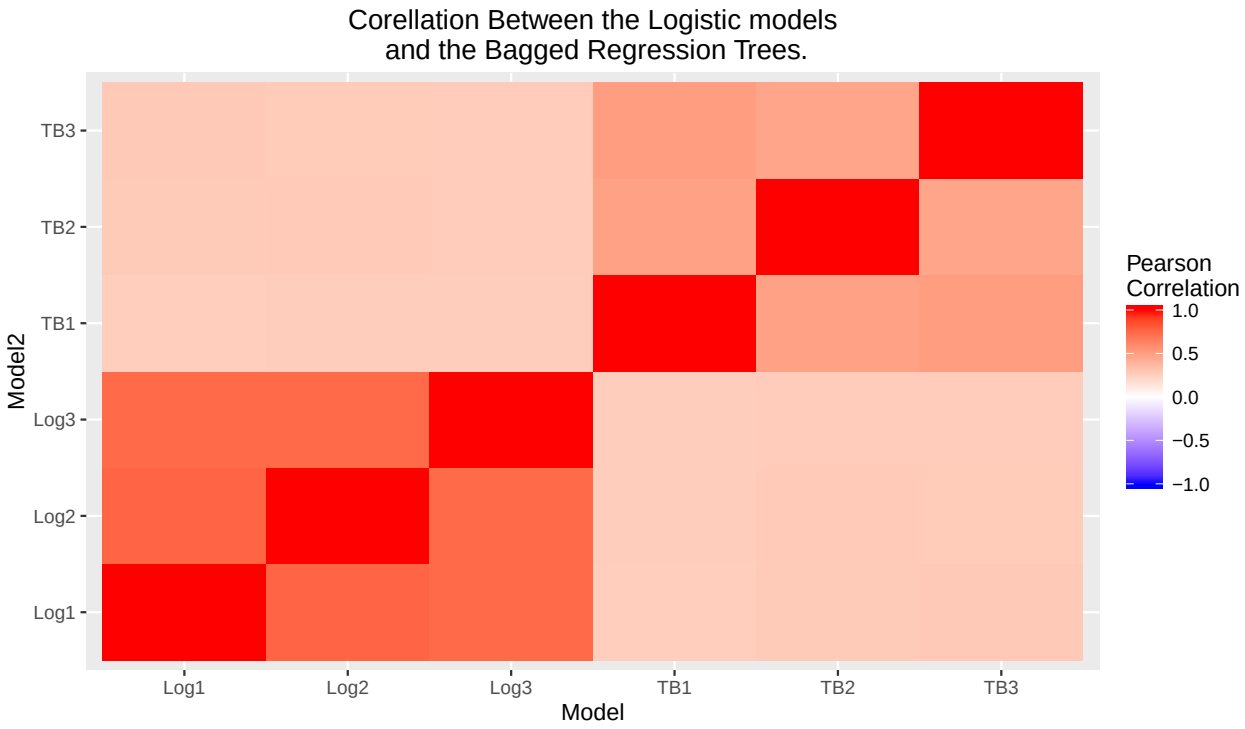

Figure 1: There is strong correlation within the Logistic regression but weaker correlation within the Bagged tree models.

for modelling on very large data sets. The final model performance for the XGboost model was 0.80815, other models were trained using the sparse matrix format such as an glm with elastic-net regularisation but this resulted in considerably poorer performance than XGboost and trained much slower. We also trained an XGboost model using full expansion i.e not group rarely occuring variables, however this, too, underperformed. It's possible that this is due to over-fitting on clicks with certain rarely-occurring variables like 'IP address'.

We performed a grid search across several hyper-parameters and found that there was a strong effect on over-fitting with both the Tree depth and percentage of columns sampled hyper-parameters. Figur 2 increasing tree depth came a clear increase in training auc and a slight decrease in testing auc. Decreasing the the column sampling percentage by from 70% to 50% reduced the over-fitting by almost a percentage point as shown un figure 3. An analysis of the std deviations revealed only a random cloud with no clustering see figure 4. The effects of changing the parameters were so strong that AUC could be predicted with an $R^2 > 0.8$ using a two variable linear regression, with such strong indicators that over fitting was occurring but that the cross-validation auc was relatively predictable we changed our strategy to produce the final model.

A plot of the training and hold out test error for a cross-validation of the XGboost model can be seen in Figure 5. The model shows the growing difference between the train AUC and the test AUC during per iteration cross-validation of the model. As can be seen the more boositing iterations that occur, the larger the difference between the two states becomes: this is due to the training model over-fitting on it's data. It can also be seen that the performance on the test set levels off after approximately 80 iterations. A table of the first 100 iterations can be seen in Appendix A

4.2 Final Model

The final model was an XGboost tree, trained on the full data set, with the hyper parameters, learning rate a=0.1, tree depth = 6, column sampling percentage = 10%, 200 boosting iterations and weights balancing positive and negative classes. Due to sparse matrix expansion there were 10's of thousands of variables.

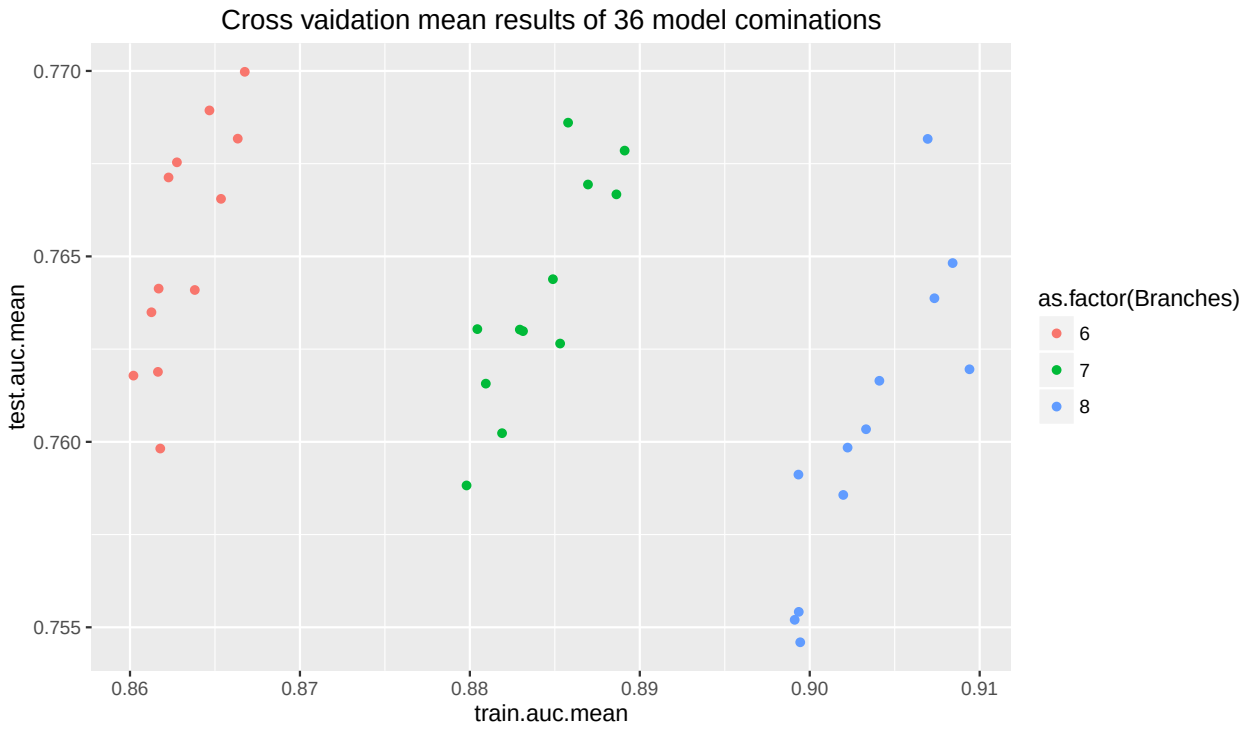

Figure 2: There is a clear clustering by number of branches

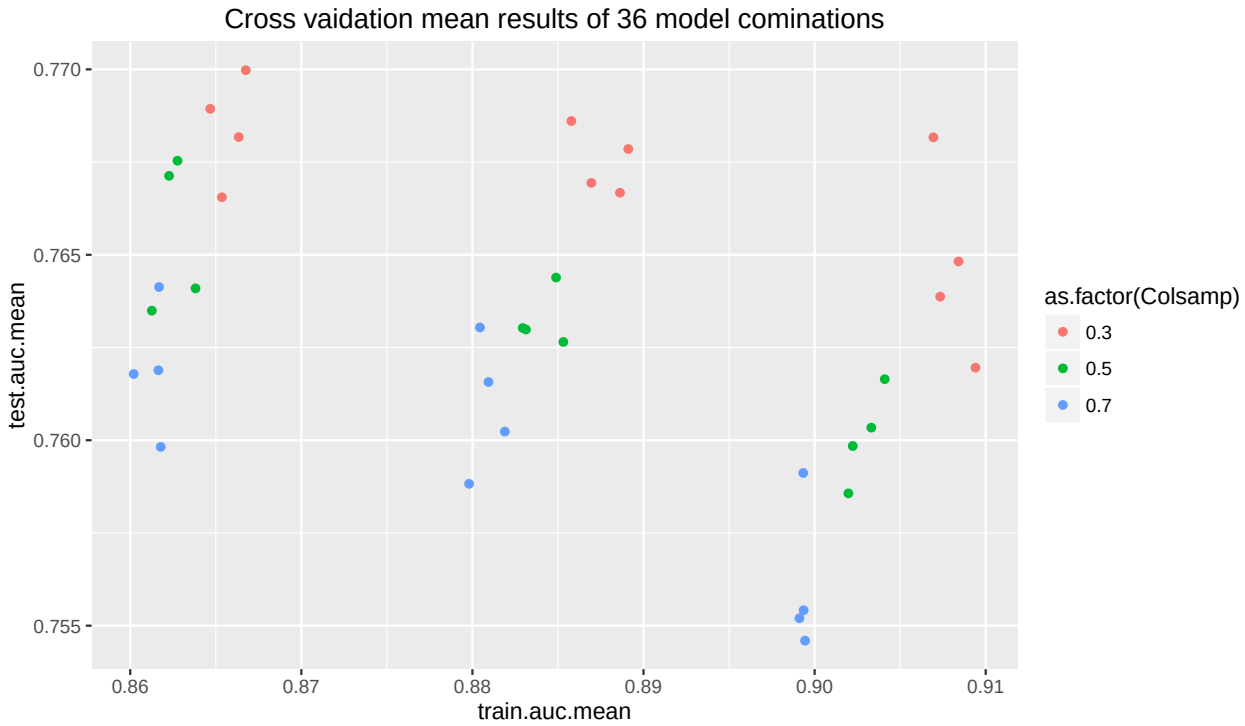

Figure 3: The amount of sampling of the columns also revealed clear clustering.

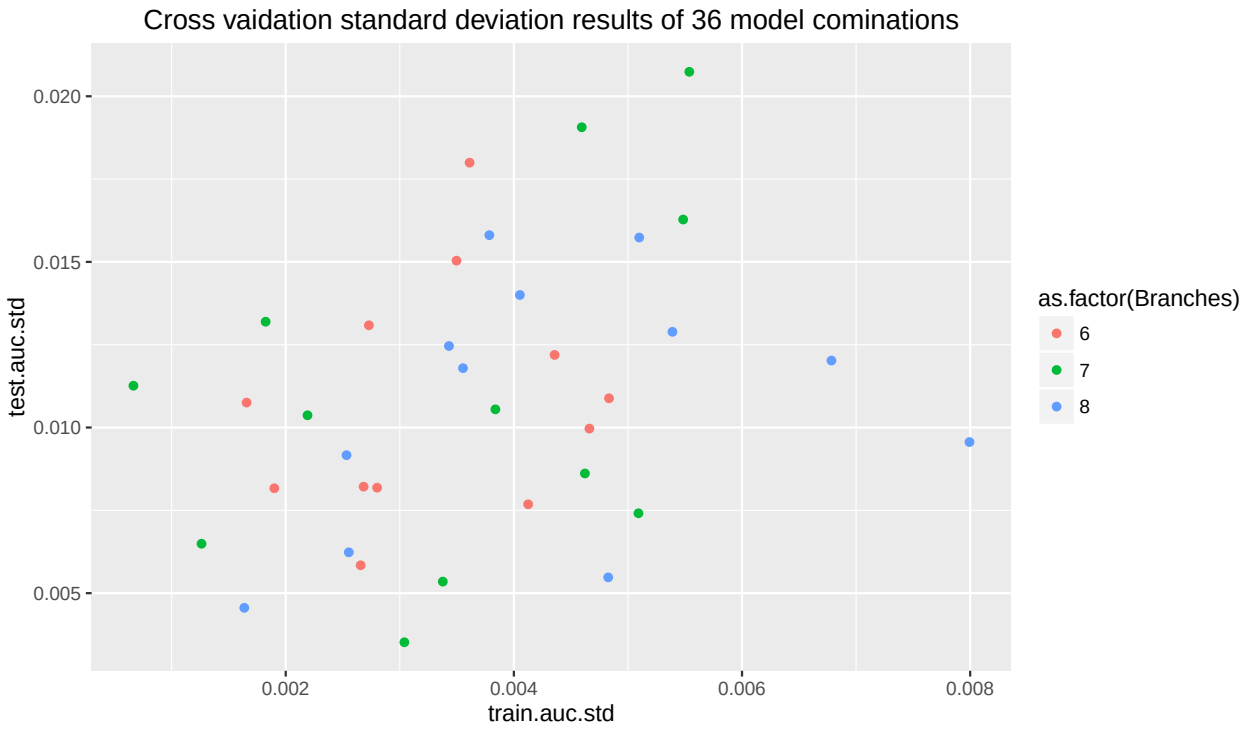

Figure 4: An analysis of the standard deviations reveals only random noise.

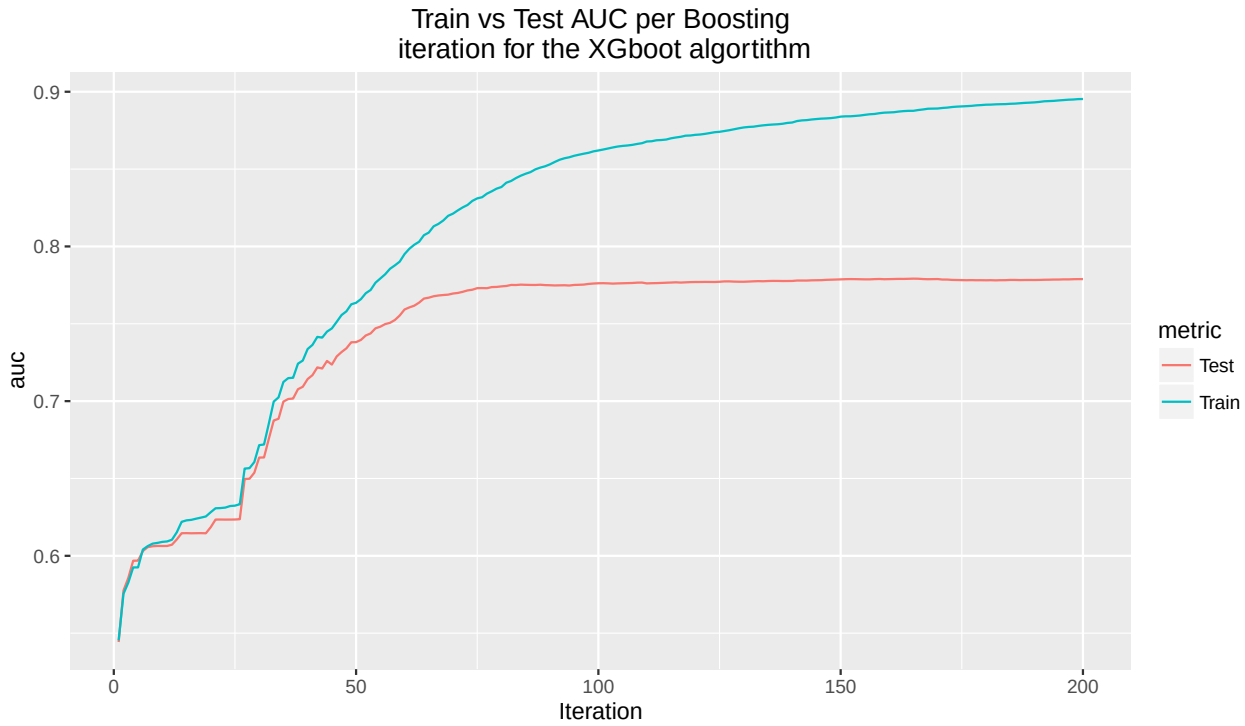

Figure 5: The figure shows the difference between train and test error.

5 Conclusion

There were many areas of learning in this project from simple to more complicated. Particular areas of interest were that, on re-balancing, the the glm became meaningless as a binomial classifier, and as a result we had to use a quasi-binomial method (see code listing 2). More complex efforts such as the multiple data sets method to deal with the over-fitting as a result of down sampling, were also very interesting to implement. However, it must be said that for all the innovation required to get around the problems, a major point of learning was how little many techniques help if the overriding method is lacking a key component. Although this proved to be disheartening at times, it does force the modeller to take a step back and try to get a higher-level picture before diving into the more granular details of model tuning and optimisation.

Our final point is that reviewing the whole process at a fundamental level was (eventually) the key to our success: once all avenues had been exhausted and conventional methods had reached memory limits even on high-powered computers, we realised that our approach must be fundamentally inefficient. With the implementation of sparse matrices, we could vastly increase the number of observations that were processable from tens of thousands to millions, this allowed us to train on a truly representative data set, without the need for ensembling multiple small data sets. This breakthrough was required in order for us to produce results above the baseline and go beyond the 0.8 AUC barrier.

It is clear that in very large highly-imbalanced datasets weighting, and down/up sampling methods are not necessarily appropriate tools for prediction, and instead stripping back the modelling and feature engineering process and reverting to more efficient method incorporating a simple sparse matrix is actually a better starting point.

5.1 Future Work

- **Affect of multiple re-sampling on classification accuracy**: The multiple modelling technique was interesting and did improve model model performance and allowed a greater variation and combination of models. How would model performance be affected on a less imbalanced data set? Is there are point at which such a technique stops being useful, and feature hashing becomes necessary?

- **Ensemble modelling of sparse data**: when there are very few examples of a class ensemble modelling can be a challenge as all data point are needed. Is it possible to break up a training set in order to build an ensemble model even when there are large class imbalances? If the two models are not similar in performance, can a non-linear ensemble still be used to increase performance?

A Xgboost performance per boosting iteration

Iteration	train.auc.mean	train.auc.std	test.auc.mean	test.auc.std
1	0.55	0.04	0.54	0.05
2	0.58	0.01	0.58	0.03
3	0.58	0.02	0.59	0.02
4	0.59	0.02	0.60	0.03
5	0.59	0.02	0.60	0.03
6	0.60	0.00	0.60	0.02
7	0.61	0.00	0.61	0.01
8	0.61	0.00	0.61	0.01
9	0.61	0.00	0.61	0.01
10	0.61	0.00	0.61	0.01
11	0.61	0.01	0.61	0.01
12	0.61	0.01	0.61	0.01
13	0.62	0.01	0.61	0.02
14	0.62	0.01	0.61	0.02
15	0.62	0.01	0.61	0.02
16	0.62	0.01	0.61	0.02
17	0.62	0.01	0.61	0.02
18	0.62	0.01	0.61	0.02
19	0.63	0.01	0.61	0.02
20	0.63	0.01	0.62	0.02
21	0.63	0.00	0.62	0.01
22	0.63	0.00	0.62	0.01
23	0.63	0.00	0.62	0.01
24	0.63	0.00	0.62	0.01
25	0.63	0.00	0.62	0.01
26	0.63	0.00	0.62	0.01
27	0.66	0.03	0.65	0.05
28	0.66	0.03	0.65	0.05
29	0.66	0.04	0.65	0.05
30	0.67	0.04	0.66	0.05
31	0.67	0.04	0.66	0.05
32	0.69	0.03	0.68	0.04
33	0.70	0.02	0.69	0.03
34	0.70	0.02	0.69	0.02
35	0.71	0.02	0.70	0.03
36	0.71	0.01	0.70	0.03
37	0.72	0.01	0.70	0.03
38	0.72	0.02	0.71	0.02
39	0.73	0.02	0.71	0.02
40	0.73	0.01	0.71	0.03
41	0.74	0.01	0.72	0.03
42	0.74	0.01	0.72	0.03
43	0.74	0.01	0.72	0.03
44	0.74	0.01	0.73	0.03
45	0.75	0.01	0.72	0.03
46	0.75	0.01	0.73	0.02
47	0.76	0.00	0.73	0.02
48	0.76	0.00	0.73	0.02
49	0.76	0.00	0.74	0.02
50	0.76	0.00	0.74	0.02
51	0.77	0.00	0.74	0.02
52	0.77	0.00	0.74	0.02
53	0.77	0.00	0.74	0.02
54	0.78	0.00	0.75	0.02
55	0.78	0.01	0.75	0.02
56	0.78	0.01	0.75	0.02
57	0.79	0.00	0.75	0.03

Iteration	train.auc.mean	train.auc.std	test.auc.mean	test.auc.std
58	0.79	0.00	0.75	0.02
59	0.79	0.00	0.76	0.02
60	0.80	0.01	0.76	0.02
61	0.80	0.01	0.76	0.02
62	0.80	0.01	0.76	0.02
63	0.80	0.00	0.76	0.02
64	0.81	0.00	0.77	0.02
65	0.81	0.00	0.77	0.02
66	0.81	0.00	0.77	0.02
67	0.81	0.00	0.77	0.02
68	0.82	0.00	0.77	0.02
69	0.82	0.00	0.77	0.02
70	0.82	0.00	0.77	0.02
71	0.82	0.01	0.77	0.02
72	0.83	0.01	0.77	0.02
73	0.83	0.01	0.77	0.02
74	0.83	0.00	0.77	0.02
75	0.83	0.00	0.77	0.02
76	0.83	0.00	0.77	0.02
77	0.83	0.00	0.77	0.02
78	0.84	0.00	0.77	0.02
79	0.84	0.01	0.77	0.02
80	0.84	0.00	0.77	0.02
81	0.84	0.01	0.77	0.02
82	0.84	0.00	0.78	0.02
83	0.84	0.01	0.78	0.02
84	0.85	0.00	0.78	0.02
85	0.85	0.00	0.78	0.02
86	0.85	0.00	0.78	0.02
87	0.85	0.00	0.78	0.02
88	0.85	0.00	0.78	0.02
89	0.85	0.00	0.78	0.02
90	0.85	0.00	0.77	0.02
91	0.85	0.01	0.77	0.02
92	0.86	0.01	0.77	0.02
93	0.86	0.01	0.77	0.02
94	0.86	0.01	0.77	0.02
95	0.86	0.01	0.78	0.02
96	0.86	0.01	0.78	0.02
97	0.86	0.01	0.78	0.02
98	0.86	0.01	0.78	0.02
99	0.86	0.01	0.78	0.02

Table 3: cross-validation training data per round for the submitted XGboost tree algorithm