University of Oklahoma

Intelligent Data Analytics

DSA 5103

Final Project

Daniel Brumley
dbrumley90@gmail.com

December 16, 2016

Contents

Executive Summary

From March to May of 2016, Santander Bank hosted a challenge on the data science competitions website, Kaggle. The goal of the competition was to correctly identify a customer as satisfied or unsatisfied with Santander's services. More than $5,000$ teams competed, and the winning models for the competition were selected based on maximal AUC.

In the current project, the predictive modeling aspect of the Santander Customer Satisfaction challenge took something of a backseat. Instead, the data was used as a playground with the primary aim of exploring an advanced technique known as model stacking, which has historically lead to winning Kaggle models. Particular focus was given to developing and implementing an independent model stacking algorithm in the `R` programming language.

Beyond this, working with the Santander Customer Satisfaction data presented additional challenges of its own. The training dataset, an anonymized set of over $75,000$ observations in more than 300 variables, was especially imbalanced with a mere 4% of the observations belonging to the target class. To ameliorate the class imbalance problem, an over-sampling routine was introduced into the cross validation procedure. Additionally, the dimensions of the data proved to be unmanageable given the limited processing power and the computational costs associated with the validation scheme. To cope with this, the data went through a hefty preprocessing phase that included a backwards feature selection routine. Consequently, the predictor space was reduced to 7% its original size.

In total, eight models were developed, four of which were stacked models. The stacked models were compared to the individual, non-stacked models and also benchmarked against the winning model in the competition. While the stacked models performed better than any of the non-stacked models, the models did not fare as well against the benchmark. However, such a discrepancy can be ascribed to factors outside the mechanics of the model stacking algorithm, such as limiting the available data.

Given the project's emphasis on model stacking, it is recommended that the associated algorithm undergo futher fine tuning. Particular note should be given to developing preliminary or intermediate diagnostics that might better assist in constructing the final model. The algorithm should also be compared against others that utilize different validation schemes.

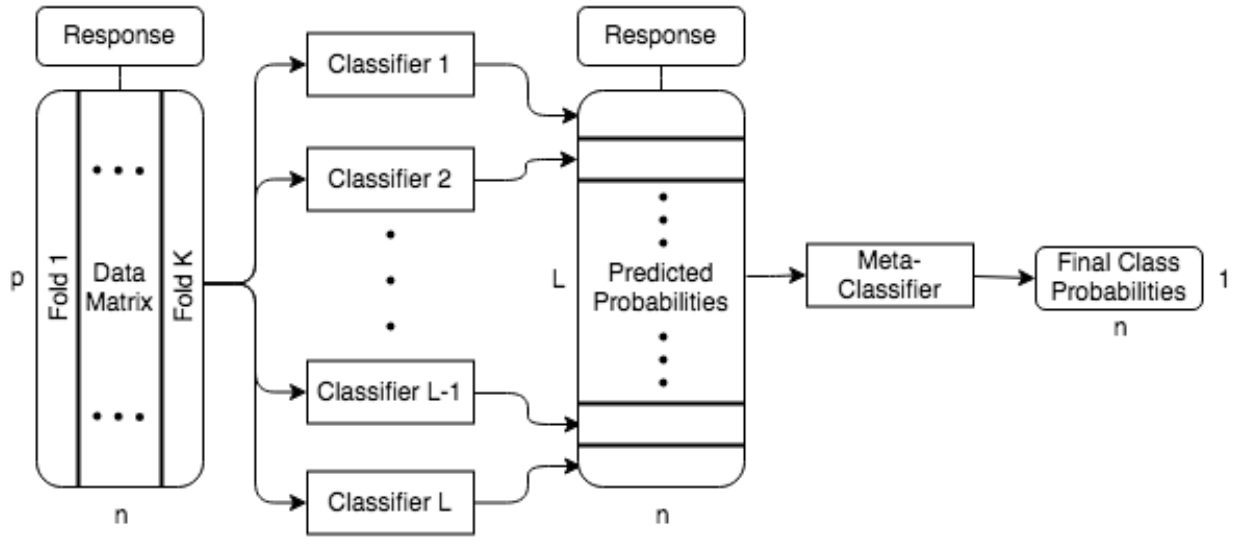

Figure 1: Visualizing the training of a stacked model. (Figure generated using *draw.io*, a free diagramming software application.)

Background

Problem Description

Santander Bank, a subsidiary of the Santander Group, has a large presence in the Northeastern region of the United States [1]. The success of Santander depends intimately on the company's ability to satisfy its customers' financial needs. However, this is no easy task, and disgruntled customers are likely to leave without much warning. As such, Santander would like to identify potentially unhappy customers so as to preempt their departure.

To this end, the bank enlisted the help of determined Kagglers from around the world. The Santander Customer Satisfaction competition was launched in March 2016 on Kaggle and ran for two months, ending the following May. With a $60,000 prize at stake, more than $5,000$ submissions were filed. Using anonymized data, the goal of the competition was to develop a model that could classify customers into one of two categories: satisfied or unsatisfied. Models were evaluated on an unlabeled test set using the area under the ROC curve, or AUC. As is standard in Kaggle competitions, the test set was partitioned into two smaller sets leading to scores in a public and private leaderboard with the latter determining results of the competition [2, 3, 4].

Model Stacking

Like most Kaggle competitions, the top competitors in the Santander Customer Satisfaction challenge utilized a model ensembling technique known as model stacking, or stacked generalization [5, 6]. Though the concept of model stacking has been around since the early 90s [7], the available literature is patchy and very often conflicting. As such, implementations differ in many key details, and model stacking very much remains a "black box" method in practice. The primary aim of this project, therefore, was to implement a model stacking procedure. The key details of this procedure, given in the context of a classification problem, are outlined below.

The motivation for employing stacked generalization is similar to any other ensembling procedure: There is power in numbers. However, unlike boosting and bagging, which pool together a family of classifiers of

Data: An $(n + m) \times p$ matrix of predictors with response y
Result: A two-tiered classification model
split the data into an $n \times p$ training set and $m \times p$ validation set
select L base learners
for i = *1..L* **do**
  using the training set, perform K-fold CV on model i
end
generate K stratified folds on the training set
for i = *1..K* **do**
  for j = *1..L* **do**
    perform $(K-1)$-fold CV for model j, holding out fold i
    validate on fold i and store the predicted probabilities
  end
end
train a meta-learner using the $n \times L$ matrix of probabilities and the $n \times 1$ response vector y_1
for j = *1..L* **do**
  using model j and the validation set, generate predicted probabilities
end
validate the meta-learner on the $m \times L$ matrix of probabilities and the $m \times 1$ response vector y_2

Algorithm 1: A model stacking algorithm.

the same type, model stacking brings together classifiers of many different types. This is accomplished by training a so-called "meta-classifier" on the predicted probabilities of the underlying classifers. The final result is a two-tiered model whose base level consists of an assortment of classifiers of differing types that produce probabilities from the data and whose top level consists of an additional classifier that maps these probabilities to a vector of final probabilites (see **Figure 1**).

While the general idea is fairly simple, the exact mechanics of the procedure are unfortunately muddied as practitioners disagree on how to best embed stacking into a validation plan. The major point of contention revolves around the issue of model overfitting. The implementation developed for this project was an amalgamation of strategies detailed in [8, 9].

Essentially, the modeling stacking algorithm is composed of three parts. In the first part, the L base learners are tuned on an $n \times p$ training set using K-fold cross validation. In the second part, the training set is decomposed into K folds, which are distinct from the ones used in the previous cross validation step. One of the folds is selected as a holdout set while the remaining $K - 1$ folds form a new training set on which $(K-1)$-fold cross validation is performed for each base learner. The resulting models are then used to predict on the holdout set. This process is repeated for each fold. The proabiliites generated from each fold are stored in an $n \times L$ matrix that is column-bound to the train response y. The meta-learner is trained on this new data matrix. Finally, the entire model is run against an $m \times p$ validation set.

The resulting algorithm is summarized in **Algorithm 1**. (See Appendix D for the actual implementation used for this project.) A couple of points are worth noting. The validation procedure contains natural breakpoints following each of its three phases that allow for running of model diagnostics. Such diagnostics are utilized to guide decisions regarding model selection and parameters in the sections below. Additionally, the procedure can be easily extended to accommodate models with more than two layers though this is not undertaken in the current project.

Analysis

Initial Impressions and Preprocessing

As noted above, the Santander Customer Satisfaction competition consisted of train and test datasets. The training dataset consisted of $76,020$ observations in 369 numeric variables with a binary response; the test dataset consisted of $75,818$ observations using the same 369 numeric variables. The data was completely anonymized, and variable names were uninformative (and in Spanish). There were no missing values.

One of the defining characteristics of the training data was its severe class imbalance: less than 4% of the training set observations belonged to the target class. Because such imbalance risks overfitting to the dominant class, it was decided an over-sampling routine would be introduced into the model training routine. The particular sampling algorithm used is called the *Synthetic Minority Over-sampling TEchique*, or *SMOTE* for short. The SMOTE algorithm works by down-sampling the majority while over-sampling the minority class. The over-sampling is accomplished by creating new, synthetic samples along the lines that join the original sample to its k nearest neighbors in the minority class. This method has been shown to lead to better classifier performance in ROC space than either under- or over-sampling alone [10]. The implementation used for this project came from the **DMwR** package.

Following **Algorithm 1**, the training set was split into smaller train and validation sets using stratified random sampling. The resulting sets had $60,817$ (80%) and $15,203$ (20%) observations, respectively. Both sets contained an ID column that was removed.

Because of the high computational costs associated with the model training and sampling algorithms, the data was further pared. The new, smaller training set underwent a series of transformations, which included a Box-Cox transform and removal of zero variance predictors - see Appendix A for details. Additionally, predictor pairs with absolute correlations greater than 0.95 were removed from the set. This initial preprocessing reduced the number of predictors from 370 to 137 - a 63% reduction in the predictor space. The same transformations were subsequently applied to the validation and test sets.

Feature selection

Though the preprocessing above signifantly reduced the size of the data, it was still thought to be too unwieldy. The data was made more manageable by employing *Recursive Feature Elimination* (RFE) via the **caret** package. The algorithm is a cross-validated form of backwards selection whereby smaller and smaller subsets of the top performing predictors are retained so as to maximize the given performance metric [11].

In the present case, random forest RFE was performed on the training set using 5-fold cross validation so as to maximize model AUC. The results are captured in **Figure 2**. It can be seen an optimal AUC is obtained with only 25 of the 137 predictors. In particular, the predictor thought to be most important to the classification task was *var15*. As observed by [12], the distribution of the variable suggests it contains each customer's age. It is interesting to note the skew in the conditional distributions (see **Figure 3**) reflects the fact that younger customers are generally more satisfied with their service - or, perhaps, just less discerning of poor service.

The results of the RFE were cross-referenced with the results of a univariate feature selection procedure optimizing the column-wise AUC, and the two were found to be in accordance (see Appendix B.1). The selected variables were also found to be relatively uncorrelated (see Appendix B.2). Ultimately, the optimal set of 25 predictors was used as the final set of predictors in the training, validation, and test sets - a reduction of 93% from the original set of predictors.

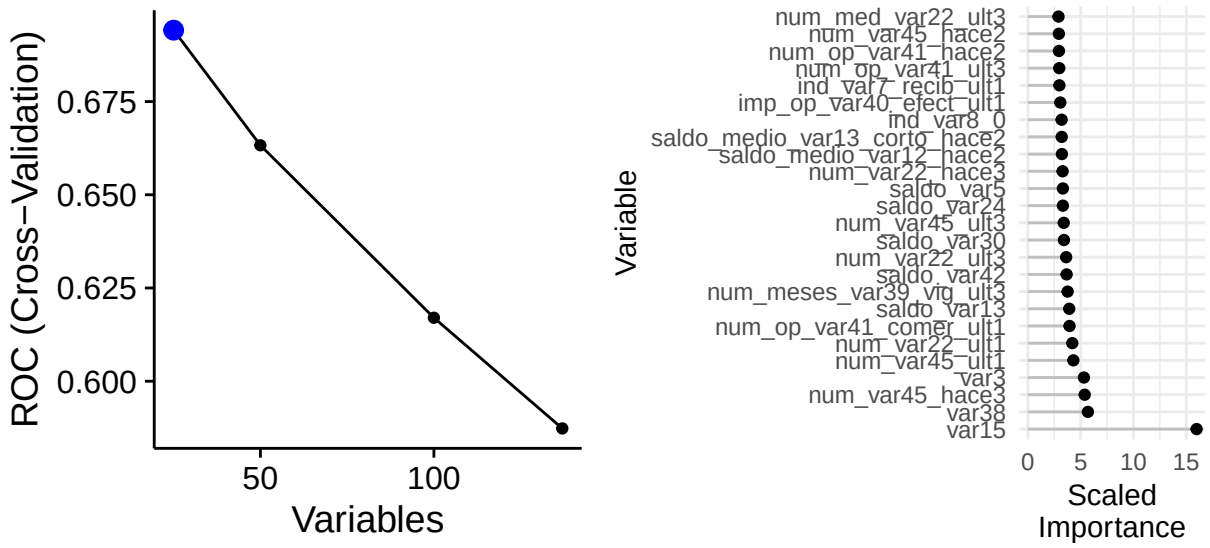

Figure 2: The results of random forest recursive feature elimination on the training set. **Left:** The performance profile across different subset sizes. **Right:** The scaled variable importance scores of the optimal subset predictors.

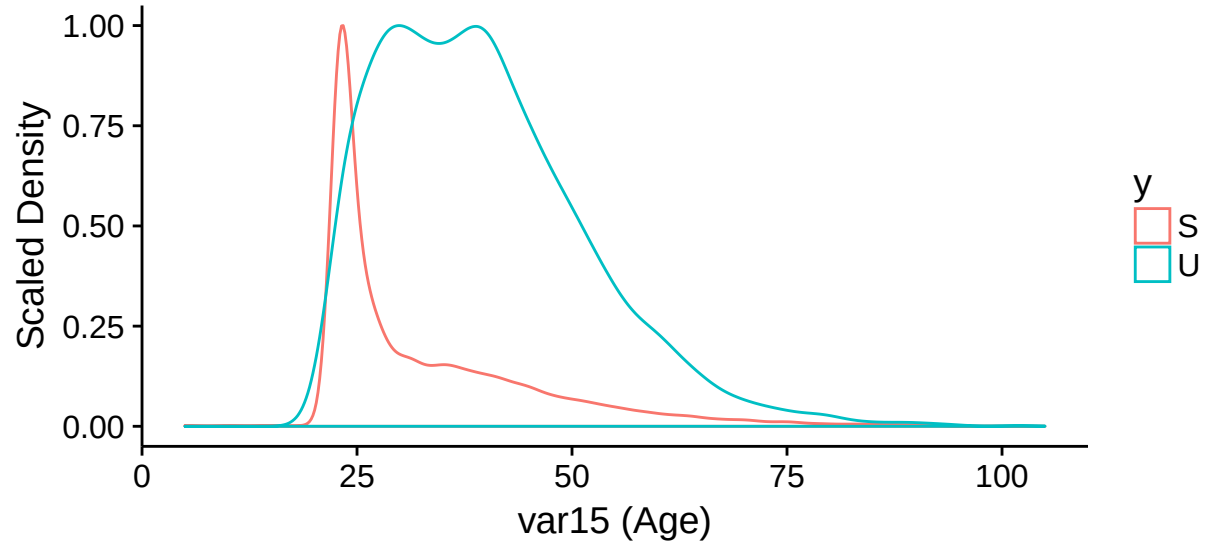

Figure 3: Plots of var15 conditioned on the response y, where S represents the satisified customers and U the unsatisfied customers.

Model Selection and Validation

As noted in [7], the foundation upon which an optimal stacked model is built should consist of models that excel at different aspects of the task at hand: shallow learners, deep learners, and everything in between. With this criteria, a tentative set of four classifiers was selected. This included a random forest (a "mid-to-deep" bagging method) implemented via the **ranger** package; an XGBoost model (a "mid-to-deep" predictive boosted trees method) implemented via the **xgboost** package; a GBM model (another "mid-to-deep" variant of the previous boosted trees method) implemented via the **gbm** package; and an elastic net logistic regression (a "shallow" regularization method) implemented via the **glmnet** package.

In accordance with **Algorithm 1**, each model was tuned on the entire training dataset using 5-fold cross-

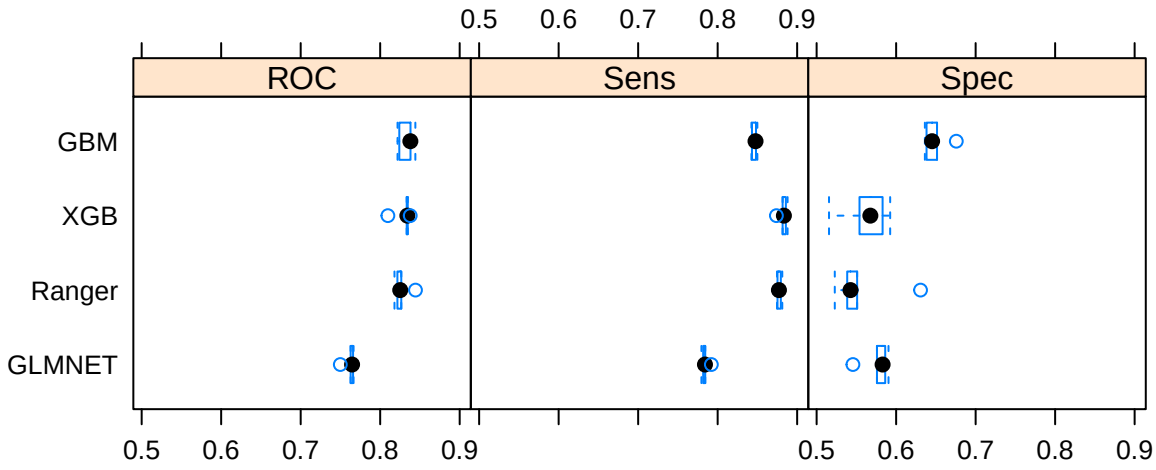

Figure 4: Resampling statistics for the tentative set of base learners. Note that satisfied customers represent the positive class.

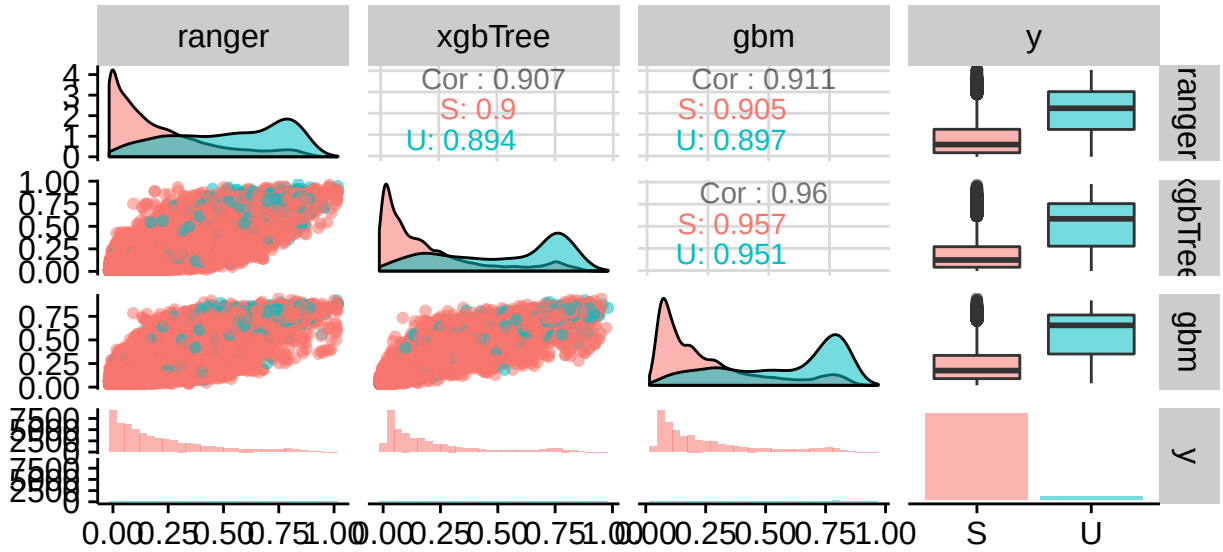

Figure 5: A pairs plot of the out-of-sample probabilities and response, where S and U represent satisfied and unsatisfied customers, respectively. The plot shows histograms of the probabilties conditioned on the response, a boxplot of the response, scatterplots of the probabilities plotted against eachother, densitiy plots of the probabilities conditioned on the response, correlation statistics, and box-and-whisker plots of the probabilities against the response.

validation to maximize AUC with the SMOTE sampling algorithm run inside each fold. The resampling results are given in **Figure 4**. As expected, the models performed with varying degrees of success on different aspects of the classification task. The random forest (Ranger) and XGB models, though performing well overall, appeared to overfit to the majority class - evident from the high sensitivity and low specificty of the predictions. The opposite case was true for the logistic regression model (GLMNET), which performed poorly overall but partially redeemed itself with its higher specificity. The GBM model outperformed all the other models in terms of AUC and general fit. Due to these results, the GLMNET model was dropped from the set of base learners. A more fine-tuned comparison of the models is given in Appendix C.

As outlined in *Algorithm 1*, the the predicted probabilities from the remaining base learners were used to tune the second layer of the model. As an intermediary step, a pairs plot of the response and the predicted

probabilities resulting from the cross-validation procedure was generated (see **Figure 5**). From this, it can be seen the probabilities are highly correlated, especially those stemming from the boosted models. In this setting, high correlations are not only expected but tolerated - after all, the models were trained on the same data and perform comparably under the same performance metric. The point is that predicted probabilities between models differ enough to be useful when pooled together.

A simple, non-regularized logistic model was chosen as a meta-learner to avoid the overhead associated with an additional cross-validation stage. After training this model, it was realized that more stacked models could be developed by taking different linear combinations of the probabilities from the base models developed in the first stage of the model stacking procedure. An additional three models were constructed. The first, AVG, was formed by taking a simple average of the Ranger, XGB, and GBM predicted probabilities; the second, AVG_WEIGHTED, by taking a weighted average of the same probabilites with a weight ratio of $2:1:2$, respectively, which was chosen by analysing the resampling results of the base models (see Appendix C); and a third, AVG_ALL, by taking a simple average of the preceding probabilities, plus the probabilities produced from the GLMNET model.

After constructing the models on the training set, the models were run against the validation set. **Table 1** provides a summary of the final models. The stacked models performed better than any of the individual models as expected. Interestingly, the model with the best performance was the AVG_ALL in spite of the fact it contained the weakest classifier.

Model	Description	AUC
RANGER	Random forest	0.8195221
XGB	Extreme gradient boosting decision trees	0.8246625
GBM	Gradient boosting decision trees	0.8245484
GLMNET	Elastic net logistic regression	0.7658138
STACKED	Base models: RANGER, XGB, GBM Meta-classifier: Logistic regression	0.8281551
AVG	Base models: RANGER, XGB, GBM Meta-classifier: (RANGER + XGB + GBM) / 3	0.8290044
AVG_WEIGHTED	Base models: RANGER, XGB, GBM Meta-classifier: (2×RANGER + XGB + 2×GBM) / 5	0.8288215
AVG_ALL	**Base models: RANGER, XGB, GBM, GLMNET** **Meta-classifier: (RANGER + XGB + GBM + GLMNET) / 4**	**0.8304236**

Table 1: Summary of the validiation set results.

Results

The models developed in the preceding section were used to generate predictions on the test set, which were subsequently submitted to Kaggle for scoring. The final results are displayed in **Table 2**. All the models performed far better than a trivial model simply predicting each customer satisfied with the quality of Santander's services. It can also be seen that the stacked models performed better than any of the individual models. Between the stacked models, the AVG model performed best while the AVG_ALL fared the worst and, incidentally, took the biggest hit in AUC in the transition from validation to test set.

For comparison, the winning model in the Santander Customer Satisifaction comptetition had a final AUC score of 0.829072 on the private leaderboard. Though this is only a difference of 0.009193, it would have been significant in terms of leaderboard placement: Had any of the models actually been submitted into the competition, none would have placed below even the $3,000^{th}$ position. This difference in performance is most likely a result of reducing the available predictor space. Though necessary given the limited processing power of the machine the models were trained on, such a decision would be anathema to most Kagglers, who would instead opt for more computing power (for a cost).

Model	Public Leaderboard AUC	Private Leaderboard AUC
RANGER	0.821544	0.810702
XGB	0.827493	0.812445
GBM	0.829226	0.814785
GLMNET	0.764603	0.752537
STACKED	0.832911	0.819248
AVG	**0.833189**	**0.819879**
AVG_WEIGHTED	0.832937	0.819854
AVG_ALL	0.831765	0.817196

Table 2: Kaggle public and private leaderboard scores for each model.

Conclusion

The main goal of this project was to develop and implement a functional model stacking procedure to gain insight into stacked models. Though the models stemming from this procedure are unlikely to win any competititons, their development was informative and in line with the aims of the project.

With regard to the predictive modeling aspect of the project, it was found that the decision to scale back the data lead to a decrease in performance with respect to the benchmark winning model. Whether this decrease is characterized as fatal or miniscule depends entirely on context. In a Kaggle competition, it is almost certainly to be the former. As noted above, the models developed for this project would not have even broken into the top $3,000$. Outside the Kaggle arena, however, it would be difficult to justify the additional computing time needed to close such a negligible performance differential. With that said, a more efficient algorithm might better balance all of these factors.

Future work would focus on fine tuning the model stacking algorithm. To this end, it would be interesting to see if base model probabilities could be decorrelated by training the models according to distinct performance metrics and, if so, whether such a step would significantly increase the performance of the overall model. The development of more precise model selection heuristics based on prelimary and/or intermediary diagnostics would be another avenue worth exploring. Alternative algorithms employing different validation schemes should also be investigated and compared.

References

[1] Wikipedia. Santander bank. https://en.wikipedia.org/wiki/Santander_Bank. Accessed: 11-6-2016.

[2] Kaggle. Santander customer satisfaction. https://www.kaggle.com/c/santander-customer-satisfaction. Accessed: 11-6-2016.

[3] Kaggle. Kaggle faq. https://www.kaggle.com/wiki/KaggleMemberFAQ. Accessed: 11-6-2016.

[4] Kaggle. Santander customer satisfaction private leaderboard. https://www.kaggle.com/c/santander-customer-satisfaction/leaderboard. Accessed: 11-6-2016.

[5] MLWave. Kaggle ensembling guide. http://mlwave.com/kaggle-ensembling-guide/. Accessed: 11-13-2016.

[6] dnc1994. How to rank 10% in your first kaggle competition. https://dnc1994.com/2016/05/rank-10-percent-in-first-kaggle-competition-en/. Accessed: 11-15-2016.

[7] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[8] Erin LeDell. Ensembles in h20. https://github.com/h2oai/h2o-tutorials/blob/master/tutorials/ensembles-stacking/H2O_World_2015_Ensembles.pdf. Accessed: 11-15-2016.

[9] J. Thompson. Ensemble modeling: Stack model example. https://www.kaggle.com/jimthompson/house-prices-advanced-regression-techniques/ensemble-model-stacked-model-example, 9 2016. Accessed: 11-21-2016.

[10] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *CoRR*, abs/1106.1813, 2011.

[11] Max Kuhn. The caret package. http://topepo.github.io/caret/recursive-feature-elimination.html, 11 2016. Accessed: 11-25-2016.

[12] Andre Muta. Santander customer satisfaction forum. https://www.kaggle.com/c/santander-customer-satisfaction/forums/t/19291/data-dictionary?forumMessageId=110414#post110414. Accessed: 11-21-2016.

Appendix

Appendix A

The printout below provides a summary of the transformations described in the Initial Impressions and Preprocessing section.

```
## Created from 60817 samples and 43 variables
##
## Pre-processing:
##   - Box-Cox transformation (2)
##   - ignored (0)
##   - removed (41)
##
## Lambda estimates for Box-Cox transformation:
## -0.6, -0.1
```

The Box-Cox transformed variables were *var15* ($\lambda = -.01$) and *var38* ($\lambda = -0.6$). Histograms of the variables before and after the transformation are given in **Figure 6**.

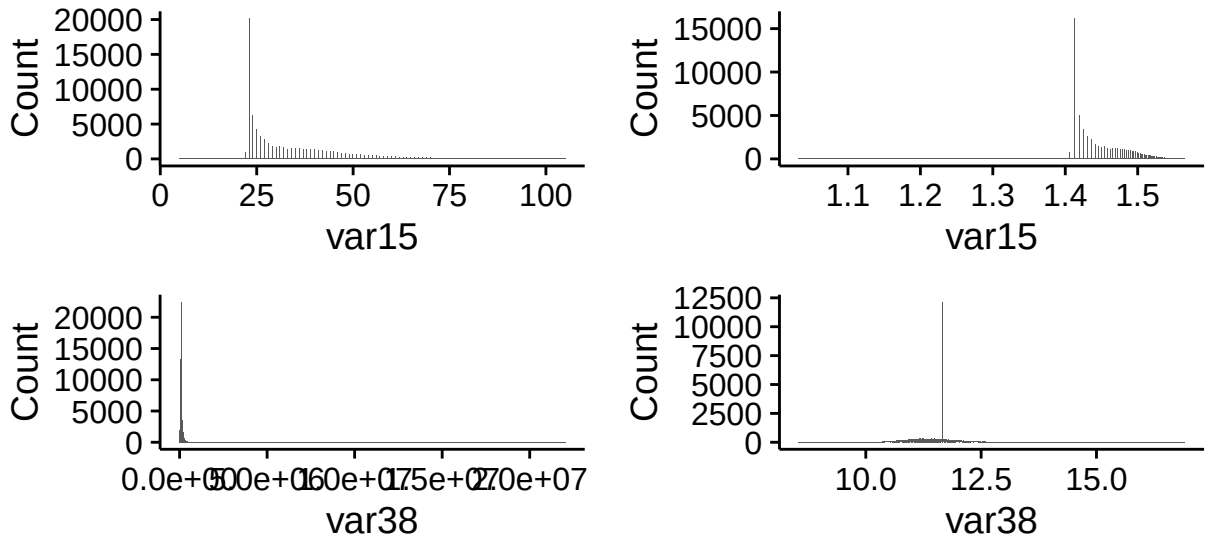

Figure 6: Histograms of the Box-Cox transformed variables before (**left**) and after (**right**) the transformation.

Appendix B

Appendix B.1

As noted in the Feature selection section, the results of the random forest RFE were cross-referenced with a univariate feature selection procedure that worked as follows. First, simple, single predictor models were constructed for each column of the training set. For each model, the resulting AUC was computed using the Wilcoxon Rank Sum Test. This was done via the `colAUC` function in the **caTools** package. The models were then ranked according to AUC to gauge the importance of each predictor.

The results of this procedure for the top 25 variables are displayed in **Figure 7**. Noticeably, the predictors selected are identical to the ones picked in the RFE procedure.

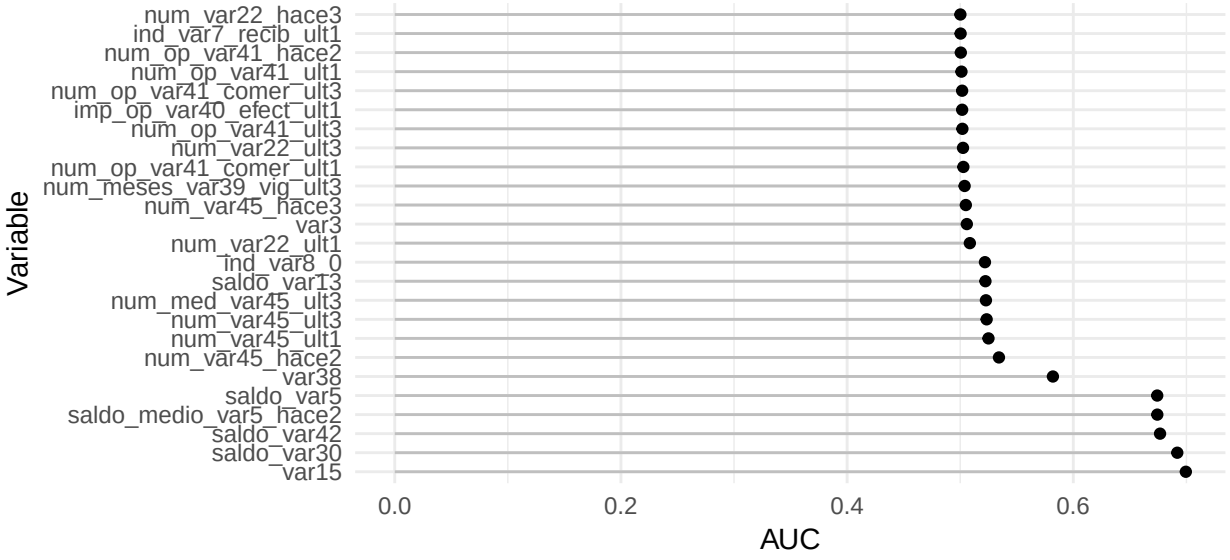

Figure 7: The results of the univarate feature selection procedure.

Appendix B.2

The optimal set of 25 predictors chosen by the random forest RFE were found to be relatively uncorrelated. **Figure 8** shows a correlation plot of the optimal predictors, where correlation was determined using Spearman's rank correlation coefficient.

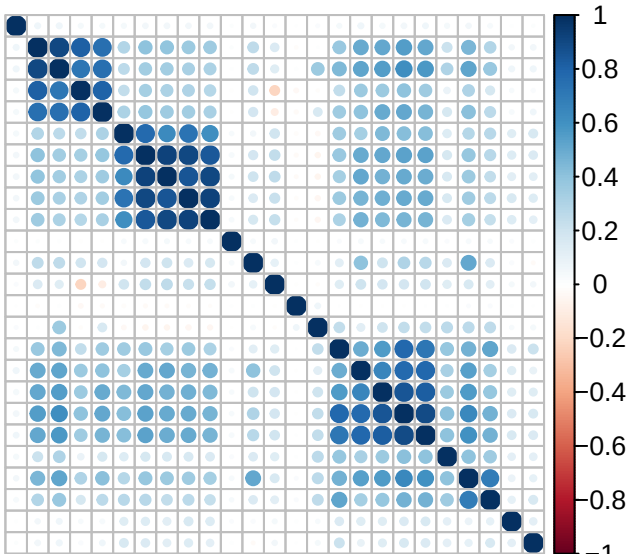

Figure 8: Correlation plot of the optimal predictors from resulting from the random forrest RFE.

Appendix C

The printout below gives resampling statistics between the base models trained in the Model Selection and Validation section.

```
##
## Call:
## summary.diff.resamples(object = diff(results))
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## ROC
##        Ranger   XGB       GBM       GLMNET
## Ranger          -0.002785 -0.006078  0.065221
## XGB    1.000000           -0.003293  0.068006
## GBM    1.000000 1.000000             0.071299
## GLMNET 0.001260 5.961e-05 0.002046
##
## Sens
##        Ranger    XGB       GBM       GLMNET
## Ranger           -0.004982  0.031262  0.093186
## XGB    0.4876295            0.036244  0.098168
## GBM    0.0009586 0.0004266            0.061924
## GLMNET 5.789e-06 1.181e-06 1.966e-05
##
## Spec
##        Ranger  XGB       GBM       GLMNET
## Ranger         -0.005412 -0.092248 -0.019137
## XGB    1.00000           -0.086836 -0.013725
## GBM    0.01032 0.05163             0.073111
## GLMNET 1.00000 0.57222   0.02400
```

From the top table, it can be seen that the GLMNET model is an outlier in terms of AUC, which was the central reason it was dropped from the set of base models ultimately used in the STACKED model. It can also be seen from the second and third tables that there is strong evidence supporting the claim that the Ranger and GBM models differ in their specificity and sensitivity - that is, their ability to correctly predict satisfied and unsatisfied customers, respectively. This suggests that the two models, though equal in the terms of their overall predictive abilties (see the top table), cater to different aspects of the classification task. This observation guided the selection of the weights used for the AVG_WEIGHTED model.

Appendix D

The *R* code used to implement the model stacking algorithm - note that what was called the validation set in the report above is referenced as the test set in the code below. The cross-validation steps were executed in parallel on a 2012 MacBook Pro utilizing a quad core 2.9 GHz Intel Core i7 processor. The entire procedure took around ten hours to execute.

```
# setup the train control
my.train <- trainControl(method = "cv", number = 5, classProbs = TRUE,
                         summaryFunction = twoClassSummary, sampling = "smote")

# setup a cluster for parallel computing
cl <- makeCluster(detectCores())
registerDoParallel(cl)

# train the models
fit.ranger <- train(trainX, trainY, metric = "ROC", method = "ranger",
                    trControl = my.train, tuneLength = 5)

fit.xgb <- train(trainX, trainY, metric = "ROC", method = "xgbTree",
                 trControl = my.train, tuneLength = 5)

fit.gbm <- train(trainX, trainY, metric = "ROC", method = "gbm",
                 trControl = my.train, tuneLength = 5)

fit.glmnet <- train(trainX, trainY, metric = "ROC", method = "glmnet",
                 trControl = my.train, tuneLength = 5)

# terminate workers and register a sequential backend
stopCluster(cl); registerDoSEQ()

# collect resampling results
results <- resamples(list(Ranger = fit.ranger,
                          XGB = fit.xgb,
                          GBM = fit.gbm,
                          GLMNET = fit.glmnet))

# plot resampling results
bwplot(results)

# select the models to be used
models <- c("ranger", "xgbTree", "gbm")

# set the positive and negative classes
# note: classes are set this way to accomodate the glm model, which
# automatically sets the positive class to the second level and the
# negative class to the first
positive <- levels(trainY)[2]; negative <- levels(trainY)[1]

# generate folds
set.seed(54321)
folds <- createFolds(trainY, k = 5)
```

```r
# adjust the train control
my.train$number <- 4

# setup a matrix to store out of sample probabilities
oos.probs <- matrix(numeric(dim(trainX)[1] * length(models)),
                    nrow = dim(trainX)[1],
                    ncol = length(models))
colnames(oos.probs) <- models

# setup a cluster for parallel computing
cl <- makeCluster(detectCores())
registerDoParallel(cl)

for (i in 1:length(folds)) {

  # set the current fold
  current.fold <- unlist(folds[i])

  for (j in 1:length(models)) {

    # fit the model on training data, minus fold i
    fit <- train(trainX[-current.fold, ],
                 trainY[-current.fold],
                 metric = "ROC",
                 method = models[j],
                 trControl = my.train,
                 tuneLength = 5)

    # predict on fold i and retrieve the predicted probabilities
    fit.probs <- extractProb(list(fit),
                             testX = trainX[current.fold, ],
                             testY = trainY[current.fold])
    fit.probs <- fit.probs %>% dplyr::filter(dataType == "Test")

    # store the predicted probabilities
    oos.probs[current.fold, j] <- fit.probs[, positive]

  }

}

# terminate workers and register a sequential backend
stopCluster(cl); registerDoSEQ()

# fit a logistic regression model on predicted probabilities
fit.glm <- train(oos.probs,
                 trainY,
                 metric = "ROC",
                 method = "glm",
                 trControl = trainControl(method = "none",
                                          classProbs = TRUE,
                                          summaryFunction = twoClassSummary),
                 tuneLength = 5)
```

```
# create a list of models to predict on
test.fits <- list(RANGER = fit.ranger,
                  XGB = fit.xgb,
                  GBM = fit.gbm,
                  GLMNET = fit.glmnet)

# predict probabilities for each model
test.probs <- test.fits %>%
  map(function(x) { caret::predict(x, newdata = testX, type = "prob")[, positive] } )

# predict remaining probabilities
test.probs$STACKED <- caret::predict(fit.glm,
                                     newdata = data.frame(ranger = test.probs$RANGER,
                                                          xgbTree = test.probs$XGB,
                                                          gbm = test.probs$GBM),
                                     type = "prob")[, positive]
test.probs$AVG <- (test.probs$RANGER +
                     test.probs$XGB +
                     test.probs$GBM) / 3
test.probs$AVG_WEIGHTED <- (2 * test.probs$RANGER +
                              test.probs$XGB +
                              2 * test.probs$GBM) / 5
test.probs$AVG_ALL <- (test.probs$RANGER +
                         test.probs$XGB +
                         test.probs$GBM +
                         test.probs$GLMNET) / 4

# print the test set results
test.probs %>% map(function(x) { AUC::auc(AUC::roc(x, testY)) } )
```