A Monte Carlo study on methods for handling class imbalance

Mark H. White II / markhwhiteii@gmail.com

Many applications of classification problems in machine learning involve class imbalance—a situation where the class of interest (the "minority" or "positive" class) makes up a very small percentage of the total data (i.e., 5% or less of cases are in this class). Algorithms that try to maximize the overall accuracy of prediction bias in favor of the majority (or "negative") class. For example, if 1% of cases are classified as "positive," then an algorithm can ensure 99% accuracy by simply predicting *all* cases as "negative." This obviously misses the whole point of training a model in the first place, as often we are training the model explicitly because we want to be able to predict this class or event with a low base rate. Class imbalance problems arise in important applications: Will a contact lead to a donation to our campaign? Does this person have cancer? Is this credit card transation fraudulent?

There are a number of ways to address class imbalance in two-class classification problems (Branco, Torgo, & Ribeiro, 2015; Longadge, Dongre, & Malik, 2013). When researchers propose new methods or review a number of them, they generally use real-world, pre-existing data sets to see how their methods perform (Galar, Fernández, Barrenechea, Bustince, & Herrera, 2012; Weiss, McCarthy, & Zabar, 2007). While this adds the benefit of ecological validity, it does not allow us to see systematic relationships between characteristics of the data and performance. Monte Carlo simulation methods (Johnson, 2013) can be used to generate sets of data that are identical in every way, save for characteristics we are interested in studying. Because these simulation methods control aspects of the data while holding everything else constant, simulation studies can yield conclusions like, "Algorithm X outperforms Algorithm Y, but only in small samples." We would know that this is due to the sample size (and not some other aspect of the data), because we can control the sample size in setting the parameters to our simulations.

The present paper takes a Monte Carlo approach, systematically varying certain aspects about the data across 500 simulations (while holding other aspects constant). I will analyze all combinations of four algorithmic approaches and four sampling approaches to addressing class imbalance. I will create one data generating process that only allows the following aspects to vary: sample size, amount of class imbalance, number of predictors, and proportion of predictors that are noise (i.e., do not predict the outcome).

Method

Data Generating Process

I generated data with a dichotomous outcome. The average class imbalance was 97% in the negative and 3% in the positive clases. These data were simulated by adapting the `twoClassSim` function from the `caret` R package (Kuhn, 2008). I generated 500 data sets using the following steps:

1. The number of cases in each data set were randomly drawn from a distribution, $N(40000, 5000)$.
2. Two multivariate normal predictors (A and B) were generated, correlating with one another at $r = .65$. These two variables contributed to the log-odds by: $4A + 4B + 2AB$.
3. Another variable, $J \sim U(-1, 1)$, was generated. This variable further added to the log-odds by: $J^3 + 2 \times \exp(-6 \times (J - 0.3)^2)$.
4. Two more variables, $K \sim U(0, 1)$ and $L \sim U(0, 1)$, were generated and contributed to the log-odds by: $2 \times \sin(K \times L)$.

5. A number X was drawn from a distribution, $N(50, 7)$. Another number Y was drawn from a distribution, $N(.15, .033)$. $Z = X - (X \times Y)$ variables were drawn from a distribution, $N(0, 1)$. Each of these Z variables further added to the log-odds in an additive fashion, where coefficients were (a) of alternating signs and (b) evenly spaced from 2.50 to 0.25.
6. $\frac{Y}{2}$ variables were drawn from a distribution, $N(0, 1)$, and did not contribute to the log-odds. However, they were still used as inputs in each of the models.
7. The log-odds for each case were converted to probabilities. For each data set, a positive (i.e., minority) class proportion, M, was sampled from a distribution, $N(.03, .007)$. Probabilities were sorted from lowest to highest. The difference between the probability for the $1 - M$th highest probability and M was calculated, and this constant was added to the probability for each case. This was done to ensure the minority class proportion was about M.

Models

For each of the 500 data sets, I randomly divided the data into training and testing sets along a .70/.30 split. I trained 16 models on the training cases for each data set. These models were created by every iteration of 4 sampling techniques and 4 algorithms. I discuss each in turn.

Sampling Techniques

Data were preprocessed using four different techniques (using the `ubUnder`, `ubOver`, and `ubSMOTE` functions from the `unbalanced` R package, respectively; Pozzolo, Caelen, & Bontempi, 2015):

- **Undersampling.** Cases were randomly dropped from the majority (i.e., negative) class until it was the same size as the minority (i.e., positive) class. For example, if there were i cases in the minority and j cases in the majority class, $j - i$ cases were randomly discarded from the majority class.
- **Oversampling.** Cases were randomly replicated, with replacement, from the minority class until it was the same size as the majority class. For example, if there were i cases in the minority and j cases in the majority class, $j - i$ random replications, with replacement, were made of cases in the minority class.
- **Synthetic minority over-sampling technique (SMOTE).** "Synthetic" cases were made from the minority class as a way to "oversample" it, while certain cases from the majority class were randomly dropped from the data set. There are many variants of SMOTE, but I employed the version described by Chawla, Bowyer, Hall, & Kegelmeyer (2002). Synthetic cases were created along the following process: for each case in the minority class, (a) find its k nearest neighbors, (b) randomly selecting 2 of these nearest neighbors, (c) calculate a line connecting the original case to each of these two randomly-selected nearest neighbors, (d) choosing a random point on these lines, (e) saving these points as a new case, and (d) labeling these new cases as part of the minority class. For each synthetically-generated minority case, 2 of the cases from the majority class were included in the "SMOTEd" data set. For example, if an original data set contained 90 cases in the negative class and 10 in the positive class, the "SMOTEd" data set would include 40 negative cases and 30 positive cases.
- **None.** This was a control condition, and no preprocessing was done to the data. The class imbalance was not adjusted for in the preparation of the data.

Algorithms

Predictions were made using four different algorithms:

- **Random forest.** I employed Breiman's (2001) random forest approach using the `randomForest` function from the R package of the same name (Liaw & Wiener, 2002). This involves training t

number of decision trees on a reduced data set with a subset of p predictors and n resampled—with replacement—cases from the inputted data set. Predictions are made by letting each of these t trees predict the outcome for any given case, and a simple majority vote is taken on how to classify the case. For example, if 80 trees predicted "A" and 20 trees predicted "B," the case would be predicted as class "A." In the event of a tie, the prediction is randomly made. For the current study, I set $t = 100$, p equal to the square root of the number of predictors in the inputted data set, and n equal to the number of cases in the inputted data set. I did not limit how many nodes each tree was allowed to form, and the minimum number of cases required to create a node was set at one. Given `X` as the inputs and `y` as the class label, the code for the random forest was: `randomForest(X, y, ntree = 100)`.

- **AdaBoost.** I employed Freund and Schapire's (1996) AdaBoost.M1 approach using the `adaboost` function from the `fastAdaboost` R package (Chatterjee, 2016). This algorithm also involves training t number of decision trees (note that this method can be used with any learning algorithm, but the function used here employs decision trees). Instead of generating a number of trees independently of one another based on a subset of cases and predictors (like the random forest), AdaBoost works in serial. Once the the first tree ($t = 1$) is fit, cases in the second tree ($t = 2$) are weighted based on the error of the first tree. The third tree $t = 3$, in turn, is trained on cases weighted by the error of the second tree. That is, the weights for each decision tree t are determined by the error of $t - 1$. In the case of the first tree, every case has the same weight, $\frac{1}{n}$, where n is the sample size of the data set. Weights are calculated such that cases that $t - 1$ predicts *incorrectly* are weighted *more* in t. In this way, AdaBoost focuses in on the "mistakes" of the previous trees. AdaBoost labels one class -1 and the other +1, and the final prediction is based on taking the sign of the weighted sum of predictions and their weights: $\text{sign}(\alpha_1 h_1(x) + ... + \alpha_t h_t(x))$ where t is the total number of trees, α are weights, and $h_t(x)$ are predictions from a tree. For the current study, I set $t = 10$. Given `y` as the class label and `data` as the name of the data frame, the code for AdaBoost was: `adaboost(y ~ ., data, nIter = 10)`.
- **Gradient boosting.** I employed the other most popular form of boosting, gradient boosting, as well. I used the "extreme gradient boosting," or "XGBoost" variant (Chen & Guestrin, 2016) of gradient boosting, using the `xgboost` function from the R package of the same name (Chen, He, Benesty, Khotilovich, & Yang, 2017). While AdaBoost learns from the "mistakes" of previous trees (again, other classifiers can be used here, but decision trees are employed in the current study) by calculating weights for the cases, gradient boosting learns from the "mistakes" by training trees t that are trained on the residuals of $t - 1$. Residuals are treated as negative gradients, and the model is updated with new trees t using gradient descent. XGBoost is one of the most popular implementations, providing a number of optimizations of gradient boosting in both terms of efficiency and accuracy. Like with AdaBoost, I set the number of trees to fit at $t = 10$; the rest of the hyperparameters were left to the package defaults. The code for XGBoost was: `xgboost(X, y, nrounds = 10, objective = "binary:logistic")`.
- **Decision tree.** I also employed one decision tree as a "control" non-ensemble learner. I used Ross Quinlan's C5.0 algorithm (see Kuhn & Johnson, 2013 for a description) using the `C5.0` function from the `C50` R package (Kuhn, Weston, Coulter, & Culp, 2015). The code for this decision tree was: `C5.0(X, y)`.

Performance Assessment

These 16 models then made predictions on the holdout cases. For each model, the confusion matrix was recorded and a number of performance metrics were used to compare the models.

The overall accuracy of a learner is not a useful metric for assessing performance in the presence of class imbalance. For example, if the positive class is only 2% of the data set, then we could achieve 98% accuracy simply by labeling every class in the negative. This is essentially useless, given that the whole reason of undertaking the machine learning task in this case is to be able to predict the positive class. For this reason, I will focus on four other metrics:

- **Recall.** The proportion of positive cases that were predicted correctly. This is the number of true positives (TP) over the sum of TP and false negatives (FN), $\frac{TP}{TP+FN}$. This quantifies what proportion

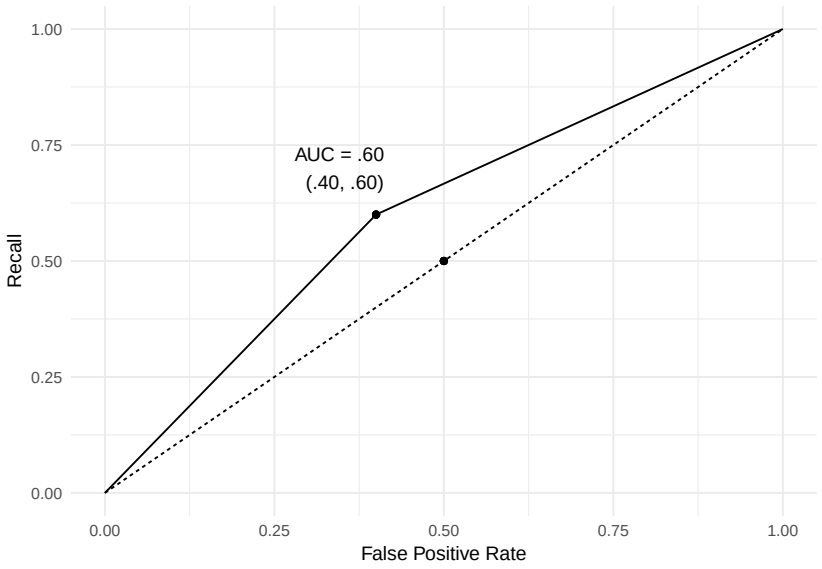

Figure 1: An example of an ROC plot showing AUC. Random guessing is denoted by the dotted line, where AUC is 0.5.

of the cases we were interested in predicting correctly were actually recovered by our model.

- **Precision.** The proportion of *correct* positive predictions. This is TP over the sum of TP and false positives (FP), $\frac{TP}{TP+FP}$. This quantifies what proportion of the cases we predicted as a positive class were *actually* of the positive class. This is also known as "positive predictive value," because it measures how "valuable" a positive prediction is.
- **F1 score.** This is the harmonic mean of recall and precision, $F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$. It is a weighted mean where both recall and precision contribute to the score equally. It is one way to assess model performance when classes are imbalanced (Wallace, Small, Brodley, & Trikalinos, 2011), as it balances the benefit of predicting a minority class correctly with the cost of predicting a majority class incorrectly. Scores range from 0 (worst) to 1 (perfect).
- **AUC(ROC).** This is short for the "area under the receiver operating characteristic curve." In the present analyses, each model will have a curve determined by one point in two-dimensional space. The model's false-positive rate, $\frac{FP}{FP+TN}$, is plotted on the x-axis, while the recall is plotted on the y-axis. One point is plotted for each model. A straight line then connects the point to the bottom-left and top-right corners of the figure. The area underneath this curve is the AUC. It is calculated as $\frac{1+R-FPR}{2}$, were R is the recll and FPR is the false-positive rate. I used this as my primary measure of model performance, following Galar et al.'s (2012) review. See Figure 1 for an example.

Results

The metrics I want to use to compare various approaches require that *some* positive predictions are made. However, some models might simply label all (or nearly all) cases in the negative to achieve, on average, 97% accuracy. Table 1 shows what proportion of the time (out of the 500 simulations) that the combination of sampling technique and algorithm made no positive class predictions.

Table 1: Proportion of simulations where each model made zero positive class predictions.

Model	Proportion P = 0
C5.0, None	1.000
Random Forest, Over	1.000
Random Forest, None	0.976
XGBoost, None	0.964
AdaBoost, Over	0.510
AdaBoost, None	0.230
AdaBoost, SMOTE	0.000
AdaBoost, Under	0.000
C5.0, Over	0.000
C5.0, SMOTE	0.000
C5.0, Under	0.000
Random Forest, SMOTE	0.000
Random Forest, Under	0.000
XGBoost, Over	0.000
XGBoost, SMOTE	0.000
XGBoost, Under	0.000

First, this illustrates the relative importance of sampling techniques over the algorithm used. Doing no sampling technique yielded $> .96$ models making no preeictions for all algorithms—save for AdaBoost, which was able to reduce this to .23.

However, making at least *one* positive prediction is an incredibly low bar. Remember that the denominator of the F_1 score is precision + recall. If the model does not make enough positive predictions to make this sum equal > 0, the F_1 score cannot be calculated. Table 2 shows what proportion of the time the combination of sampling technique and algorithm yielded so few positive predictions that an F_1 score was undefined.

Table 2: Proportion of simulations where each model made so few positive class predictions that an F1 score could not be calculated.

Model	Proportion F1 is N/A
C5.0, None	1.000
Random Forest, None	1.000
Random Forest, Over	1.000
XGBoost, None	0.998
AdaBoost, Over	0.972
AdaBoost, None	0.922
C5.0, Over	0.046
XGBoost, Over	0.004
AdaBoost, SMOTE	0.000
AdaBoost, Under	0.000
C5.0, SMOTE	0.000
C5.0, Under	0.000
Random Forest, SMOTE	0.000
Random Forest, Under	0.000
XGBoost, SMOTE	0.000
XGBoost, Under	0.000

Again, the sampling techniques discriminated on this proportion more than did the algorithms employed. All

of the oversampling and no preprocessing models had at least one instance where the F_1 score was undefined, Although C5.0 and XGBoost, both using oversampling, were much closer to 0 than to 1, for the following analyses, I focus on the models using SMOTE and undersampling.

Comparing Mean Model Performance

Tabe 3 lists the means for each metric on the models employing SMOTE and undersampling as data preprocessing techniques. These models are sorted in decreasing fashion from best AUC to worst AUC. What discriminates these models the most is the effect of using undersampling versus SMOTE on recall: The highest recall using undersampling was .676, while the best using SMOTE was .335.

It is not only important to get good performance on average, however; the variance should also be examined. Figure 2 displays density plots from the 500 simulations for each of the 8 models using SMOTE and undersampling and each of the 4 metrics.

Table 3: Mean performance on each metric for models using SMOTE and undersampling.

Model	Precision	Recall	F1	AUC(ROC)
Random Forest, Under	0.022	0.606	0.042	0.600
XGBoost, Under	0.020	0.585	0.039	0.581
AdaBoost, Under	0.018	0.676	0.036	0.570
XGBoost, SMOTE	0.022	0.286	0.041	0.550
AdaBoost, SMOTE	0.021	0.301	0.039	0.545
C5.0, Under	0.017	0.570	0.033	0.540
C5.0, SMOTE	0.018	0.335	0.033	0.528
Random Forest, SMOTE	0.026	0.069	0.037	0.516

While using a single C5.0 decision tree employed after undersampling appeared to separate itself above the models using SMOTE in terms of recall, it is clear that this model displays far more variance than the rest of the models using undersampling (as shown by the wider-tailed density than the ensemble methods).

Table 3 and Figure 2 show that, on the primary metric of interest (AUC), the ensemble methods (i.e., random forest, XGBoost, AdaBoost) that employed undersampling performed the best. I compared the means on all four metrics between these three models. To do this, I ran four separate Bayesian models, using Stan. First, I estimated the mean for each of the 8 models of interest. Then, I calculated pairwise differences between each of the 3 undersampled ensemble models. The priors for each of the 8 means were normally-distributed, with a mean that was equal to the grand mean of all 8 models on the relevant metric. The standard deviations for the priors of group means were .10 for recall and AUC, while they were .005 for precision and F_1 scores. The prior for the error was an uninformative half-cauchy prior with a mean of 0 and standard deviation of 1. Results for these comparisons are shown in Table 4.

Table 4: Pairwise mean differences and 95% credible intervals between all three ensemble models using undersampling.

Outcome	Pairwise Comparison	Difference	2.5%	97.5%
AUC(ROC)	Random Forest - XGBoost	0.019	0.017	0.022
	Random Forest - AdaBoost	0.030	0.028	0.033
	XGBoost - AdaBoost	0.011	0.009	0.014
F1	Random Forest - XGBoost	0.003	0.002	0.004
	Random Forest - AdaBoost	0.006	0.005	0.007
	XGBoost - AdaBoost	0.003	0.002	0.004
Recall	Random Forest - XGBoost	0.022	0.015	0.028

Outcome	Pairwise Comparison	Difference	2.5%	97.5%
	Random Forest - AdaBoost	-0.069	-0.076	-0.063
	XGBoost - AdaBoost	-0.091	-0.098	-0.084
Precision	Random Forest - XGBoost	0.002	0.001	0.002
	Random Forest - AdaBoost	0.003	0.003	0.004
	XGBoost - AdaBoost	0.002	0.001	0.003

It is important to remember that these models are only those using undersampling. Random forest outperformed the other two ensembles on every metric, except for recall. A random forest was able to cover 1.9% and 3.0% more AUC than XGBoost and AdaBoost, respectively. Considering recall, AdaBoost recovered 6.9% and 9.1% more of the positive cases than the random forest and XGBoost models, respectively.

Data Characteristics and Performance

Lastly, I looked at the relationship between characteristics of the data (i.e., minority size, sample size, noise variable, and predictors) and the performance of the model in terms of AUC. The bivariate relationships between the data characteristics and the AUC, split by model, are presented in Figure 3. The primary interest here was to see if there were any interactions between data characteristics and model; however, we can see that no meaningful interactions are present: Even though data characteristics were predictors of performance, the main effect of model held across the range of minority size, sample size, and number of noise and predictor variables.

Curiously, it appears that *more* input variables led to *worse* performance. Since the data generation procedure defined noise variables as a proportion of predictor variables, the two are positively-related. A better model, then, is to collapse across models and predict AUC from all four characteristics of the data in one model. I did this in another Bayesian model in Stan. Uninformative priors, $N(0, 10)$, were set for all regression coefficients. A half-cauchy prior with a mean of 0 and standard deviation of 10 was set for the error term. As can be seen in Table 5, less class imbalance led to greater AUC.

Table 5: Regression coefficients and 95% credible intervals predicting AUC(ROC).

	Coefficient	2.5%	97.5%
(Intercept)	0.568	0.556	0.580
N	0.000	0.000	0.000
Noise Variables	0.000	-0.001	0.001
Minority Size	0.552	0.291	0.819
Predictor Variables	-0.001	-0.001	-0.001

Discussion

References

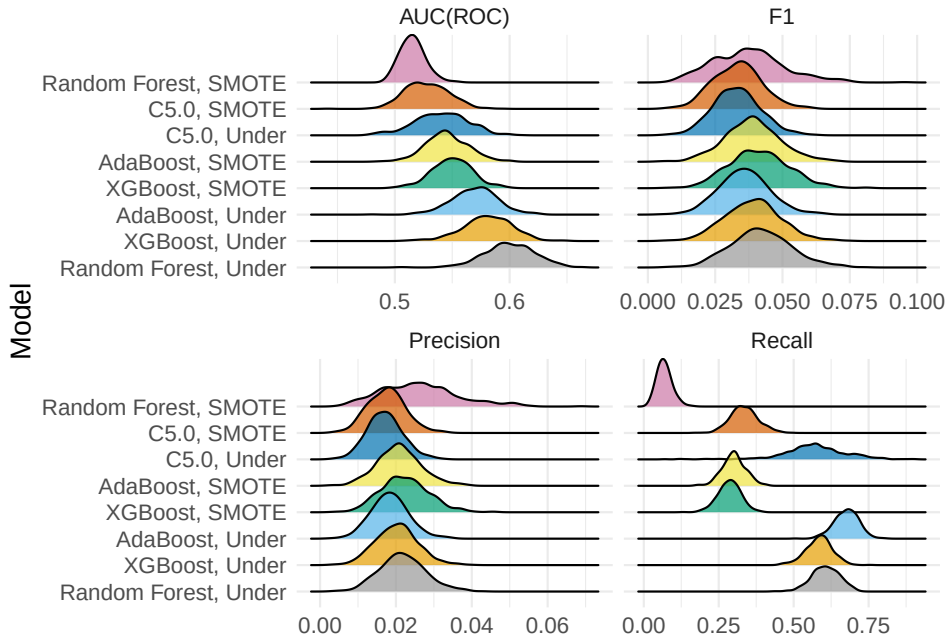

Figure 2: Density plots for all 4 performance metrics for each of the 8 models using SMOTE and undersampling.

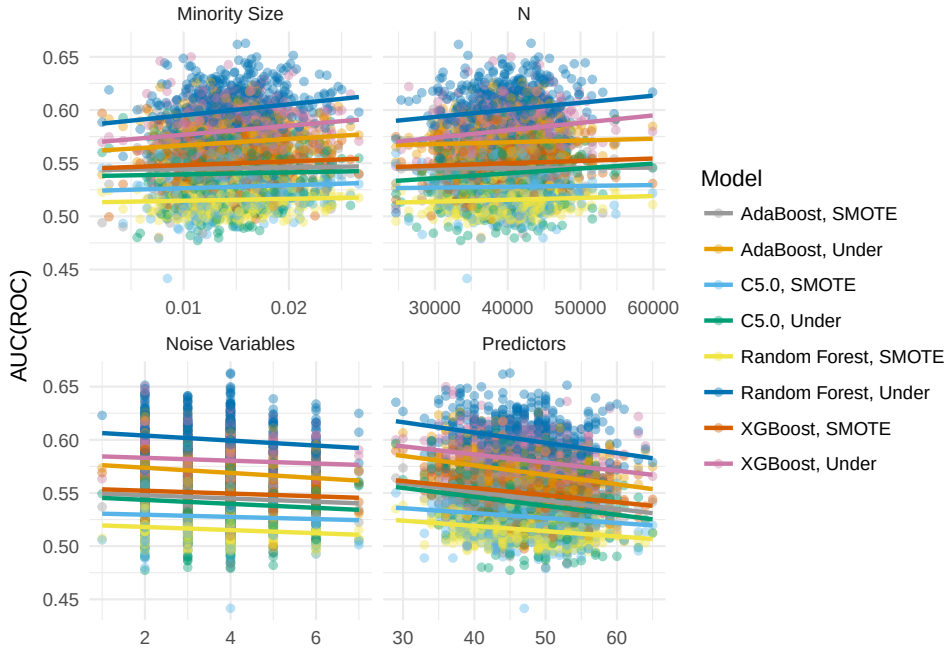

Figure 3: Relationships between data characteristics and AUC, by model.