Assignment 3: CMTH642

Christopher Graham

November 27, 2015

Overview NEED TO REWRITE

This assignment develops a model to predict a subjective quality rating on Portuguese red vinho verde wines, and to compare our results with those obtained by the original paper to work with the data.

We instigated two main changes over the original paper's approach: applying an ensemble learning model to the data, and using an undersampling/SMOTE oversamling algorithm to deal with class imbalance in the data.

. The data is provided at https://archive.ics.uci.edu/ml/datasets/Wine+Quality and was originally used for the paper "Modeling wine preferences by data mining from physicochemical properties" available at http://repositorium.sdum.uminho.pt/bitstream/1822/10029/1/wine5.pdf

The data set provides provides 11 chemical measurements of Portuguese red wines, and one quality score assigned by human testers. The data is complete and clean. Pre-processing is described in the original paper.

Our goal is to develop a model that will predict the quality rating of a wine based on these chemical measurements.

1. Import and Split Data

The data is provided at https://archive.ics.uci.edu/ml/datasets/Wine+Quality and was originally used for the paper "Modeling wine preferences by data mining from physicochemical properties" available at http://repositorium.sdum.uminho.pt/bitstream/1822/10029/1/wine5.pdf

The data set provides provides 11 chemical measurements of Portuguese red wines, and one quality score assigned by human testers. The data is complete and clean. Pre-processing is described in the original paper.

```
# load all libraries required in analysis
require(caret)
require(corrplot)
require(FNN)

# load functions that are used later in analysis
source('ensemble_eval.R')
source('over_under_wrapper.R')

red <- read.csv('winequality-red.csv', sep = ';')
# Treating as classification problem so convert quality to factor variable
red$quality <- as.factor(red$quality)

# Divide off a testing set for final validation
# All work will be done only based on the training set
set.seed(25678)
# Note that createDataPartition attempts to balance classes
train_idx <- createDataPartition(y=red$quality, p=0.8, list = FALSE)
training <- red[train_idx,]
testing <- red[-train_idx,]
```

2. Check Data Characteristics

Completeness

```
if (sum(complete.cases(testing)) == nrow(testing)) {
    print('All complete cases')
}
```

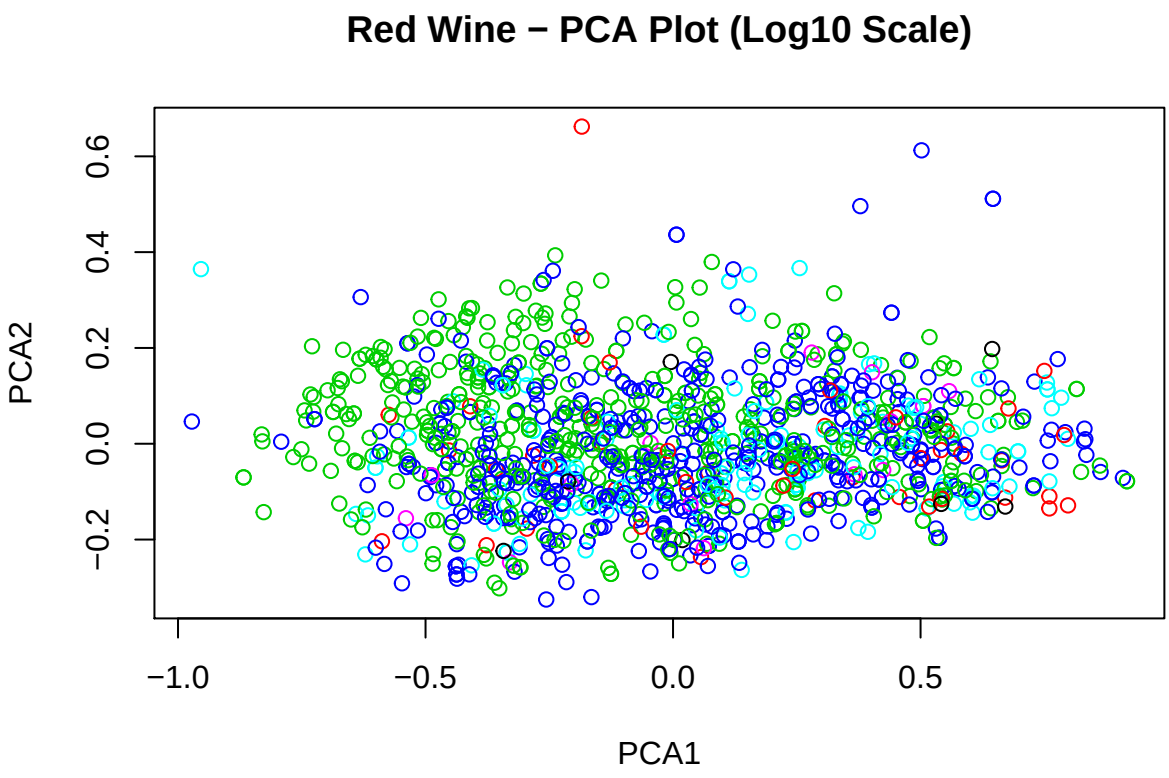
```
## [1] "All complete cases"
```

The data is complete in all variables, and as such there is no need to impute values.

2-D Visualization

Applying Principal Component Analysis to the data set, we can capture over 99% of the variance with 2 Principal Components, giving us a pretty good visualization of the data:

```
pc_comp <- prcomp(log10(training[,-12]+1))
plot(pc_comp$x[,1], pc_comp$x[,2],
     col = training$quality,
     main = 'Red Wine - PCA Plot (Log10 Scale)',
     xlab = 'PCA1',
     ylab = 'PCA2')
```

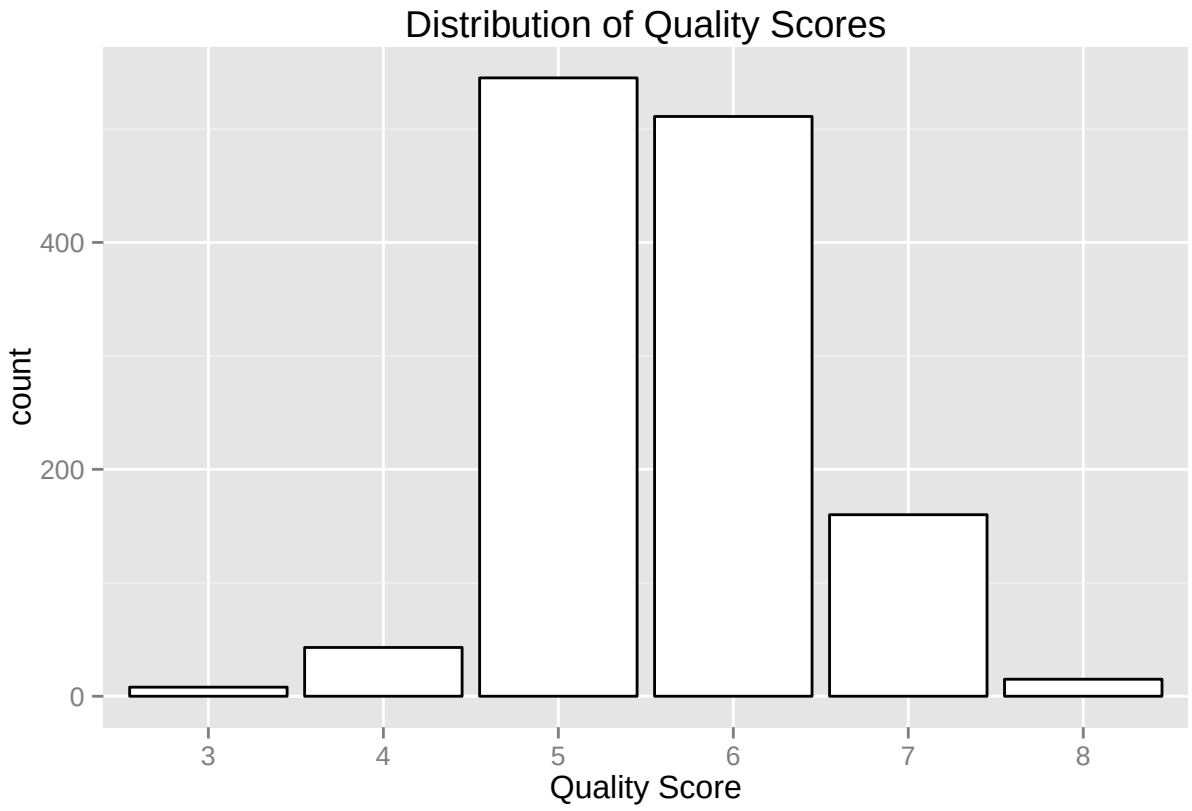


Essentially, it's a mess, with no readily apparent trends. There seems to be a bit of a division between some of the major classes, but nothing particularly clear. We'll see if we can make sense of it with some machine learning.

Class Balance

But the data isn't perfect. The biggest problem is with the distribution of quality scores. Specifically, over 80% of the wines have average ratings (5 or 6), and very few wines get ratings at the more extreme ends of the rating spectrum. (3 and 8 are the extreme values awarded, even though wines were rated on a 10-point scale). Class imbalance has been shown to be a significant problem in building an effective Machine Learning model. We provide a strategy for dealing with this below, but first will look at some other characteristics of the data.

```
ggplot(training, aes(x=quality)) +
    geom_histogram(color='black', fill='white') +
    ggtitle('Distribution of Quality Scores') +
    xlab('Quality Score')
```



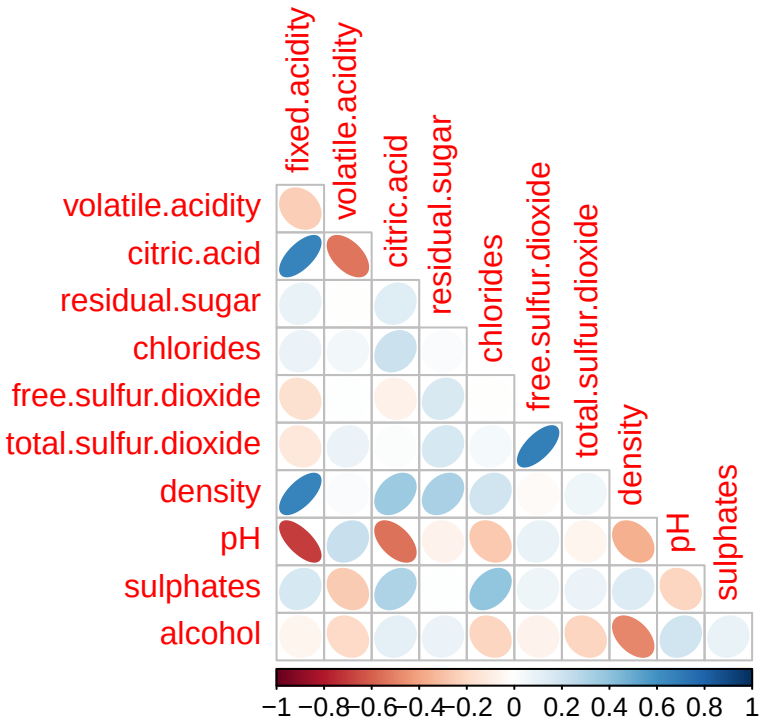
Variable Relationships

```
# Check for variables with near zero variability
nearZeroVar(training, saveMetrics = TRUE)
```

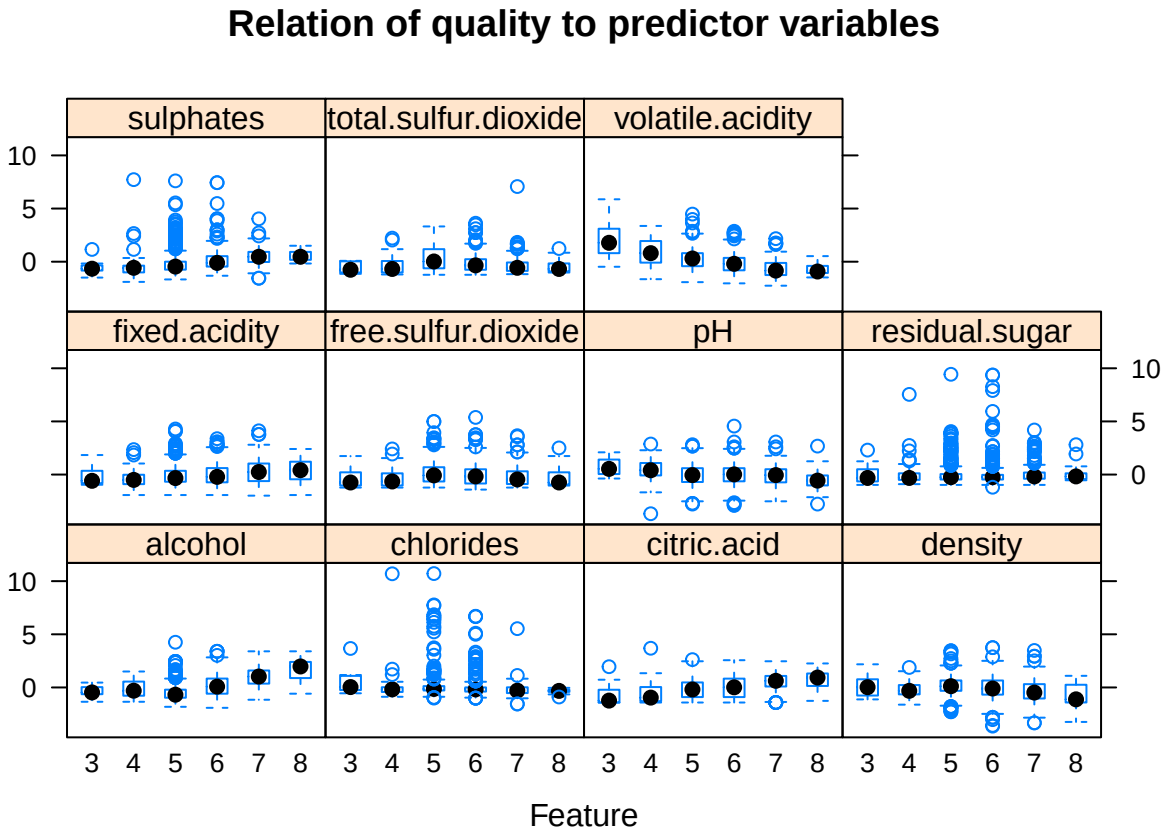
```
##                     freqRatio percentUnique zeroVar   nzv
## fixed.acidity        1.086957     7.2542902   FALSE FALSE
## volatile.acidity     1.055556    10.6864275   FALSE FALSE
## citric.acid          1.824561     6.2402496   FALSE FALSE
## residual.sugar       1.070796     6.6302652   FALSE FALSE
## chlorides            1.255814    10.9984399   FALSE FALSE
```

```
## free.sulfur.dioxide   1.337209     4.6021841   FALSE FALSE
## total.sulfur.dioxide  1.166667    10.9984399   FALSE FALSE
## density               1.033333    30.9672387   FALSE FALSE
## pH                    1.119048     6.7082683   FALSE FALSE
## sulphates             1.000000     7.3322933   FALSE FALSE
## alcohol               1.425000     4.6801872   FALSE FALSE
## quality               1.066536     0.4680187   FALSE FALSE
```

```
# look at correlation between predictor variables
corrplot(cor(training[-12]), method = 'ellipse', type = 'lower',
         title = 'Correlation of predictor variables', diag=FALSE,
         mar=c(0,0,1,0))
```




```
# Compare relationsip of quality to predictor variables
# first normalize the variables so we can compare in one graph
norm_obj <- preProcess(training[,-12], method=c('center', 'scale'))
train_norm <- predict(norm_obj, training)
featurePlot(x=train_norm[,-12], y=train_norm$quality, plot='box',
            main = 'Relation of quality to predictor variables')
```



What this preliminary exploration tells us is:

- None of the variables show near zero variability, meaning there is no *prima facie* reason to exclude any at the start of our analysis
- For the most part, there does not seem to be a huge amount of collinearity between variables. There are some instances of collinearity, where it might be expected (citric.acid, fixed.acidity and pH), (free.sulfur.dioxide and total.sulfur.dioxide). So there may be some room to reduce dimensionality.
- There are some clear relationships between some of the predictor variables and wine quality. volatile.acidity, alcohol and density seem particularly important.

Strategies for dealing with Class Imbalance in Provided Data

Since one of the goals for a prediction system would be to help vintners identify wines that are likely to be either very good or very bad, the lack of wine specimens with extreme scores could be problematic for our model development.

Dealing with class imbalance is a common problem in machine learning. The standard approach in this situation is a combination of under-sampling the majority class and over-sampling the minority class to create a distribution that will help to develop an adequate predictive model. However, there is a persistent concern about the extent to which this under & oversampling should occur, and the specific techniques to implement.

While there are different pre-made solutions to this problem, many of them are designed around a binary response variable. Specifically, the R package 'unbalanced' provides a number of pre-made approaches to dealing with an unbalanced data set. But, unfortunately, it requires a binary response variable.

In this case, we are approaching this as a multiple-level classification, relying on the classes provided in the original data (reasons for this are discussed below).

I have identified two different approaches to effectively determining an over-/ under-sampling approach for multi-class response variables. The first is the wrapper approach proposed by Chawla et al[1], and the resampling ensemble algorithm proposed by Qian et al[2].

Both of these approaches rely on a combination of SMOTE oversampling and random undersampling. However, the Chawla paper offers a more coherent (and KDD-Cup proven!) approach to determining sample levels, and so we will use that approach here.

Note, however that the algorithms used in Chawla et al require us to identify a classifier algorithm that will be used to determine stop points for sampling.

So, we need to turn briefly to model selection.

3. Model Selection

Regression vs. Classification

As noted on the source page for this data set, the data lends itself to either a regression or classification approach. The original paper to use this data set adopted a regression approach to the data, arguing that regression allows us to better evaluate "near miss" predictions (e.g. if the true value is 3, we can up-score the model if it predicts 4, rather than 7). The paper used both Neural Network (NN) and Support Vector Machine (SVM) models, and obtained an accuracy rate of 64% with a 0.5 error tolerance in quality prediction, and 89% accuracy with a 1.0 error tolerance.[3]

Furthermore, given the somewhat arbitrary way in which quality scores were derived (the median value from 3 or more judges), it's safe to say that these are not absolute values, but rather approximations. As such, regression probably makes most sense.

But Assignment 1 was based around regression, and I'd like to make this one a bit more challenging, so I'm going to frame it as a classification problem. This means the fairest comparison for our results is with the 0.5 tolerance level in the original paper as a difference of less than 0.5 would round to the correct value.

Performance Criteria

In order to ensure easy comparability with the original paper, and for simplicity of understanding, our primary judge of performance will be overall classification accuracy.

However, overall accuracy does not indicate how effective our model is at identifying minority class examples.

Note that because we are implementing the Chawla et al algorithm, our over/under sampling function will evaluate performance based on the model's f-statistic (also known as F1 score). The closer this is to 1, the better the performance:

$$f - measure = \frac{2 * precision * recall}{recall + precision}$$

The f-statistic is a good overall performance measure, balancing precision and recall. We will therefore look at the f-stat for non-average-rated wines. This will give us a better understanding of how well our model identifies bad or excellent wines. For the purposes of this analysis, we define non-average-rated wines as having a quality score of 3, 4, 7 or 8.

In order to minimize the impact of the relative overweight in class 7, we will be using the mean of the f-stats for these four classes, rather than calculating an overall f-stat for all non-average-rated classes combined.

[1] Chawla et al "Automatically countering imbalance..."

[2] Qian et al. "A resampling ensemble algorithm..."

[3] Cortez P, Cerdeira A, Almeida F, Matos T, Reis J. Modeling wine preferences by data mining from physicochemical properties. Decision Support Systems. 2009;47(4):547-53.

Model Selection

Given the relatively weak predictive accuracy (at T=0.5) for the models in the first paper, it appears that a single model may not be all that great at classification. In these instances, one of the more interesting approaches is *ensemble learning* - essentially combining predictions from multiple models into one super-model.[4]

In terms of specific ensemble learning implementation, we are going to use the approach suggested by Jeff Leek of John's Hopkins in the Coursera course "Practical Machine Learning"[5]. This is essentially a stacking algorithm.

In order to build an ensemble model, we first need to build a number of individual models to combine into the final approach. In an attempt to bring together all we've done in the course (plus a little bit more!), we're going to use:

- multinomial logistic regression (multinom or mn)
- Naïve Bayes (nb)
- Random Forest (rf)
- Boosting; specifically boosting with trees (gbm)
- Linear Discriminant Analysis (lda) - to see if there are any interesting differences with Naive Bayes if we don't assume variable independence

We are specifically excluding the two models of the original paper to see if we can do better without the models used there.

Also, that because the SMOTE oversampling algorithm relies on a knn methodology, we will not be including knn in the base models in order to avoid potentially introducing erroneous re-classification.

Finally, note that for the final model, the model we will use is a simple plurality vote among the model responses (the modal value).

The code for the specific model being run is in Appendix A.

k-fold Cross-validation

Note that our model evaluation is set using 10 repetitions of 10-fold cross-validation, and that the input data is normalized on each iteration of the model. This is set through the trainControl function in the caret library.

Normalizing Data

As per the original paper, we will be evaluating the feed-in models based on standardized data. This is done through the preProcess function in caret.

Implementation of Ensemble Model

Model baseline testing

As a baseline, before dealing with the class imbalance, let's see how our model performs:

[4]Numerous articles have shown this, but see specifically: Rokach L. Ensemble-based classifiers. Artificial Intelligence Review. 2010;33(1):1-39.

[5]Course notes available at: http://sux13.github.io/DataScienceSpCourseNotes/8_PREDMACHLEARN/Practical_Machine_Learning_Course_Notes.pdf, starting on p. 67

```r
# First, get the predictions for the training set
base_results <- ensemble_eval(training)
```

```r
# complete confusion matrix for training data
cm <- confusionMatrix(base_results[[1]], training$quality)
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   3   4   5   6   7   8
##          3   4   0   1   0   0   0
##          4   0  15   1   1   0   0
##          5   3  16 436 126   3   0
##          6   1  11 102 357  55   5
##          7   0   1   5  27 102   4
##          8   0   0   0   0   0   6
##
## Overall Statistics
##
##                Accuracy : 0.7176
##                  95% CI : (0.6921, 0.7421)
##     No Information Rate : 0.4251
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5498
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 3 Class: 4 Class: 5 Class: 6 Class: 7 Class: 8
## Sensitivity           0.50000  0.34884   0.8000   0.6986  0.63750  0.40000
## Specificity           0.99922  0.99839   0.7992   0.7743  0.96702  1.00000
## Pos Pred Value        0.80000  0.88235   0.7466   0.6723  0.73381  1.00000
## Neg Pred Value        0.99687  0.97787   0.8438   0.7949  0.94926  0.99295
## Prevalence            0.00624  0.03354   0.4251   0.3986  0.12480  0.01170
## Detection Rate        0.00312  0.01170   0.3401   0.2785  0.07956  0.00468
## Detection Prevalence  0.00390  0.01326   0.4555   0.4142  0.10842  0.00468
## Balanced Accuracy     0.74961  0.67361   0.7996   0.7365  0.80226  0.70000
```

The model gives us a pretty good result overall, with a predicted 71% accuracy on the training data, better than the 64% accuracy (at a 0.5 threshold) from the original paper.

The f-stats for the minority classes (3,4,7 and 8), however, are noticeably better for our base model than for the original paper:[6]

```r
# Average non-average-quality f-stat for our model
f_stat <- NULL
for (i in c(1,2,5,6)) {
    f_stat<- c(f_stat,
               (2 * cm$byClass[i,1] * cm$byClass[i,3]) /
```

[6]Note that because the original paper predicted 0 true positives for class 3 and class 8, the f-stat for each of those is technically NaN. I have substituted with zero to compare.

```
                    (cm$byClass[i,1] + cm$byClass[i,3]))
}
f_stat <- mean(f_stat)

# Average non-average-quality f-stat for original paper
paper_f <- mean(c(0, 0, (2*(1/53)*(1/5))/(1/53+1/5),
                  (2*(82/199)*(82/140))/(82/199+82/140)))

f_stat; paper_f
```

```
## [1] 0.5922719
```

```
## [1] 0.1295646
```

```
# Store results for various models in a data frame for easy comparison
model_summary <- data.frame(model = c('paper', 'base_train'),
                            accuracy = c(0.64, cm$overall[1]),
                            f_stat = c(paper_f, f_stat),
                            stringsAsFactors = FALSE)
```

Baseline against test set

As a test, we'll apply this model to the test data and see if our results are legitimate or due to overfitting.

```
base_test <- ensemble_eval(training, testing)
```

```
cm2 <- confusionMatrix(base_test$final, testing$quality)
cm2
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   3   4   5   6   7   8
##          3   0   0   1   0   0   0
##          4   0   0   0   0   0   0
##          5   1   8 103  38   4   0
##          6   1   2  31  82  20   2
##          7   0   0   1   7  15   1
##          8   0   0   0   0   0   0
##
## Overall Statistics
##
##                Accuracy : 0.6309
##                  95% CI : (0.5752, 0.6842)
##     No Information Rate : 0.429
##     P-Value [Acc > NIR] : 3.848e-13
##
##                   Kappa : 0.3928
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
```

```
##
##                      Class: 3 Class: 4 Class: 5 Class: 6 Class: 7 Class: 8
## Sensitivity          0.000000  0.00000   0.7574   0.6457  0.38462 0.000000
## Specificity          0.996825  1.00000   0.7182   0.7053  0.96763 1.000000
## Pos Pred Value       0.000000      NaN   0.6688   0.5942  0.62500      NaN
## Neg Pred Value       0.993671  0.96845   0.7975   0.7486  0.91809 0.990536
## Prevalence           0.006309  0.03155   0.4290   0.4006  0.12303 0.009464
## Detection Rate       0.000000  0.00000   0.3249   0.2587  0.04732 0.000000
## Detection Prevalence 0.003155  0.00000   0.4858   0.4353  0.07571 0.000000
## Balanced Accuracy    0.498413  0.50000   0.7378   0.6755  0.67612 0.500000
```

```
# Average non-average-quality f-stat for our model
test_f <- mean(c(0,0,0,(2*cm2$byClass[5,1] * cm2$byClass[5,3])/ (cm2$byClass[5,1] + cm2$byClass[5,3])))
test_f
```

```
## [1] 0.1190476
```

```
# Update results table
model_summary <- rbind(model_summary, rep(NA, 3))
model_summary[3,1] <-'base_test'
model_summary[3,2] <- cm2$overall[2]
model_summary[3,3] <- test_f
```

When we apply our model to the testing set, we see a relatively significant degradation in performance - essentially to the level of the original paper, likely indicating some overfitting in the test results.

Importantly, our model is doing a very poor job at predicting minority classes in the test set – especially quality ratings of 3, 4, and 8, which have the weakest representation in our data.

So, in an effort to improve on this, we need to turn to our function for undersampling and SMOTE oversampling.

Undersampling/SMOTE oversampling implementation

As discussed earlier, I have decided to implement the "Wrapper Undersample SMOTE Algorithm" developed by Chawla et al as a way to re-balance the provided data.

Simplified, the algorithm works as follows:

1) Establish a baseline performance level for each majority and minority class
2) For each Majority class, undersample by 10% until either minority class predictions stop improving, or until majority class predictions degrade by 5%.
3) For each Minority class, SMOTE oversample by 100%, and continue this until that minority class performance is increased by less than 5%.

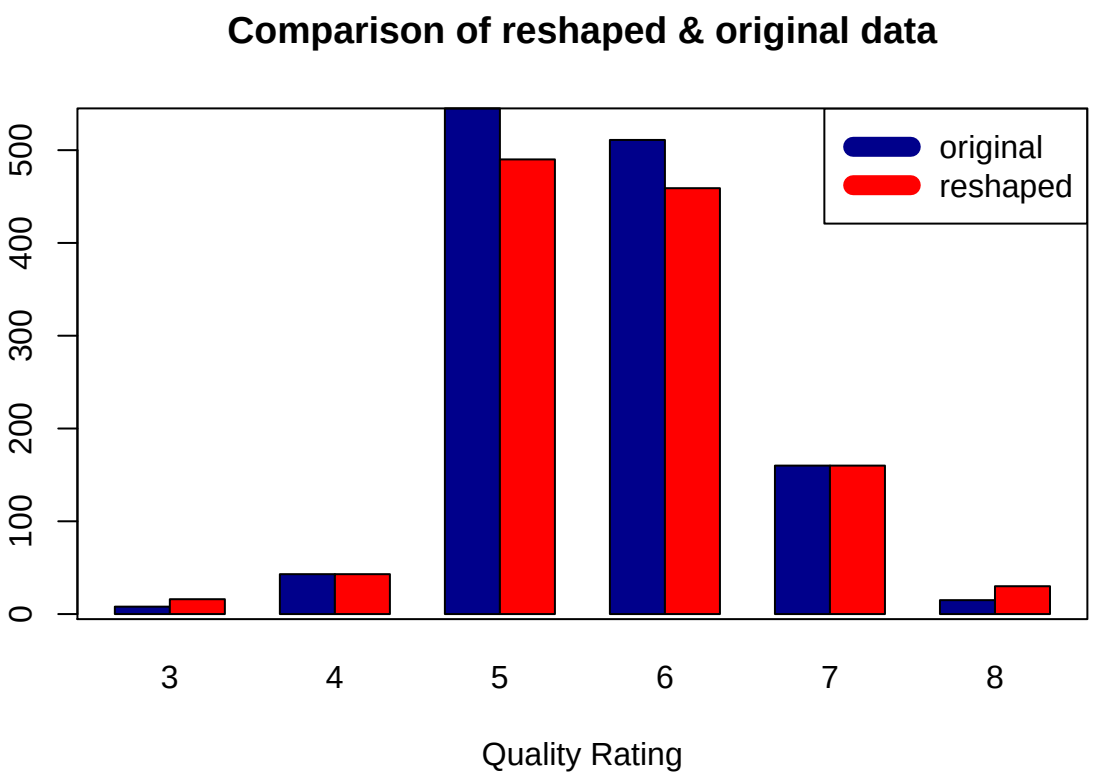
The specific R code for my implementation is shown in Appendix 2. The algorithm returns a reshaped data set that will be used to generate our final predictive model.[7]

[7]Because I don't have a great grasp of how R memory allocation works, I was unable to run this through the RMD file, as my laptop ran out of memory in executing the algorithm. Instead I ran the code through R's console, saved the final data set to disk and loaded it into the RMD file for further manipulation.

```
# algorithm needs a named vector of class categories
#       with -1 = minority; 1 = majority; 0 = neutral
class_list <- c(-1,-1,1,1,0,-1)
names(class_list) <- c(3,4,5,6,7,8)
new_training <- wrapper(class_list = class_list, train_set = training,
                        eval_fun = ensemble_eval, seed=TRUE)
```

First, let's see how our new distribution compares to the original training data:

```
barplot(rbind(table(training$quality),
              table(new_training$quality)),
        main = 'Comparison of reshaped & original data',
        xlab = 'Quality Rating',
        col = c('darkblue', 'red'),
        beside=TRUE)
legend('topright', c('original', 'reshaped'),
       col = c('darkblue', 'red'),
       lwd=10)
box()
```



The changes appear to have been fairly minor, with no oversampling at all for a quality rating of four. That mean we would have seen maximum impact on our f-statistics after one or two iterations of undersamling or oversampling.

But more importantly, let's see if our reshaped data gives us a better model. We will look at how the model works for both our original training data, and how well it applies to the testing data set.

```
new_train_results <- ensemble_eval(new_training, training)
```

```
new_test_results <- ensemble_eval(new_training, testing)
```

```
cm3 <- confusionMatrix(new_train_results$final, training$quality)
cm4 <- confusionMatrix(new_test_results$final, testing$quality)
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(new_test_results$final, testing
## $quality): Levels are not in the same order for reference and data.
## Refactoring data to match.
```

```
model_summary <- rbind(model_summary, rep(NA, 3), rep(NA,3))
model_summary[4,1] <- 'reshaped_train'
model_summary[5,1] <- 'reshaped_test'
model_summary[4,2] <- cm3$overall[1]
model_summary[5,2] <- cm4$overall[2]
f_stat <- NULL
for (i in c(1,2,5,6)) {
    f_stat<- c(f_stat,
               (2 * cm3$byClass[i,1] * cm3$byClass[i,3]) /
                   (cm3$byClass[i,1] + cm3$byClass[i,3]))
}
model_summary[4,3] <- mean(f_stat)
f_stat <- NULL
for (i in c(1,2,5,6)) {
    f_stat<- c(f_stat,
               (2 * cm4$byClass[i,1] * cm4$byClass[i,3]) /
                   (cm4$byClass[i,1] + cm4$byClass[i,3]))
}
model_summary[5,3] <- mean(f_stat)
model_summary
```

```
##                   model  accuracy    f_stat
##                   paper 0.6400000 0.1295646
## Accuracy     base_train 0.7176287 0.5922719
## 3             base_test 0.3927899 0.1190476
## 4        reshaped_train 0.7152886 0.6353565
## 5         reshaped_test 0.4035412       NaN
```

Appendix 1 - Model code

This function has been coded **only** to run and evaluate the classification algorithm. It doesn't actually return the models that are developed - it only returns the predictions for the final model and each of the intermediate models. If the actual models are needed, it's a fairly trivial change to actually return them.

```
# Helper function to get mode of a vector
getMode
```

```
## function (vect)
## {
##     unq_vect <- unique(vect)
##     return(unq_vect[which.max(tabulate(match(vect, unq_vect)))])
## }
```

```
# Main code, including test set evaluation algorithm
ensemble_eval
```

```
## function (train_set, test_set = NULL)
## {
##     evaluate_test <- function(model_list, test_set) {
##         test_set <- predict(model_list[[7]], test_set)
##         ensemble_data <- data.frame(matrix(NA, nrow = nrow(test_set),
##             ncol = 0))
##         for (i in 2:6) {
##             new_pred <- predict(model_list[[i]], test_set)
##             ensemble_data <- cbind(ensemble_data, new_pred)
##         }
##         colnames(ensemble_data) <- c("mn", "nb", "rf", "gbm",
##             "lda")
##         test_pred <- apply(ensemble_data, 1, function(x) getMode(x))
##         ensemble_data <- cbind(final = test_pred, ensemble_data)
##         return(ensemble_data)
##     }
##     fit_control <- trainControl(method = "repeatedcv", number = 10,
##         repeats = 10)
##     norm_obj <- preProcess(train_set[, -12], method = c("center",
##         "scale"))
##     train_set <- predict(norm_obj, train_set)
##     set.seed(15954)
##     mn_model <- train(quality ~ ., data = train_set, method = "multinom",
##         trControl = fit_control, trace = FALSE)
##     mn_pred <- predict(mn_model, train_set)
##     set.seed(15954)
##     nb_model <- train(quality ~ ., data = train_set, method = "nb",
##         trControl = fit_control)
##     nb_pred <- predict(nb_model, train_set)
##     set.seed(15954)
##     rf_model <- train(quality ~ ., data = train_set, method = "rf",
##         trControl = fit_control)
##     rf_pred <- predict(rf_model, train_set)
##     set.seed(15954)
##     gbm_model <- train(quality ~ ., data = train_set, method = "gbm",
```

```
##         trControl = fit_control, verbose = FALSE)
##     gbm_pred <- predict(gbm_model, train_set)
##     set.seed(15954)
##     lda_model <- train(quality ~ ., data = train_set, method = "lda",
##         trControl = fit_control)
##     lda_pred <- predict(lda_model, train_set)
##     ensemble_data <- data.frame(mn = mn_pred, nb = nb_pred, rf = rf_pred,
##         gbm = gbm_pred, lda = lda_pred)
##     final_pred <- apply(ensemble_data, 1, function(x) getMode(x))
##     if (missing(test_set)) {
##         return(cbind(final = final_pred, ensemble_data))
##     }
##     else {
##         return(evaluate_test(list(final_model = final_pred, mn_model = mn_model,
##             nb_model = nb_model, rf_model = rf_model, gbm_model = gbm_model,
##             lda_model = lda_model, norm_obj = norm_obj), test_set))
##     }
## }
```

Appendix 2 - Over/Under Sample and Multiclass SMOTE Algorithm code

Pseudocode for the wrapper, undersample & undersample algorithms can be found in the Chawla et al "Automatically Countering Imbalance".

Pseudocode for the SMOTE algorithm can be found in Chawla et al "SMOTE Synthetic Minority Oversampling"

```
wrapper
```

```
## function (class_list, train_set, eval_fun, seed = FALSE)
## {
##     require(caret)
##     require(FNN)
##     class_data <- data.frame(rbind(orig = class_list))
##     colnames(class_data) <- names(class_list)
##     class_data <- rbind(class_data, maj = class_list == 1)
##     class_data <- rbind(class_data, min = class_list == -1)
##     class_data <- rbind(class_data, under = as.numeric(class_data[1,
##         ] == 1))
##     class_data <- rbind(class_data, smote = rep(0, ncol(class_data)))
##     class_data <- rbind(class_data, base_f = evaluate_model(eval_fun,
##         train_set))
##     class_data <- rbind(class_data, best_f = class_data["base_f",
##         ])
##     original_data <- train_set
##     if (sum(class_data["maj", ]) > 0) {
##         new_list <- wrap_undersample(class_data, train_set, eval_fun,
##             seed)
##         train_set <- new_list[[1]]
##         class_data <- new_list[[2]][rownames(new_list[[2]]) !=
##             "stop_sampling", ]
##     }
##     if (sum(class_data["min", ]) > 0) {
##         new_list <- wrap_smote(class_data, train_set, original_data,
##             eval_fun, seed)
##         train_set <- new_list[[1]]
##         class_data <- new_list[[2]]
##     }
##     return(train_set)
## }
```

```
wrap_undersample
```

```
## function (class_data, train_data, eval_fun, seed = FALSE)
## {
##     original <- train_data
##     class_data <- rbind(class_data, stop_sampling = !class_data["maj",
##         ])
##     while (sum(class_data["stop_sampling", ]) < ncol(class_data)) {
##         print("classes to go")
##         print(sum(class_data["stop_sampling", ]))
```

```
##         for (i in 1:ncol(class_data)) {
##             print("i")
##             print(i)
##             if (class_data["stop_sampling", i] == FALSE) {
##                 train_len = nrow(train_data)
##                 print("train len")
##                 print(train_len)
##                 train_data <- undersample(train_data, original,
##                   colnames(class_data)[i], class_data, eval_fun,
##                   seed)
##                 if (nrow(train_data) == train_len) {
##                   class_data["stop_sampling", i] <- TRUE
##                 }
##             }
##         }
##         print("new df")
##         print(nrow(train_data))
##         new_f <- evaluate_model(eval_fun, train_data)
##         class_data["best_f", ] <- pmax(as.numeric(class_data["best_f",
##             ]), new_f)
##     }
##     return(list(train_data, class_data))
## }
```

```
undersample
```

```
## function (train_data, original, class_val, class_data, eval_fun,
##     seed = FALSE)
## {
##     sample_decrement <- 0.1
##     increment_minimum <- 0.05
##     other_classes <- train_data[train_data[, ncol(train_data)] !=
##         class_val, ]
##     curr_class <- train_data[train_data[, ncol(train_data)] ==
##         class_val, ]
##     if (seed) {
##         set.seed(2292)
##     }
##     curr_class <- curr_class[sample(nrow(curr_class), floor(nrow(curr_class) *
##         (1 - sample_decrement))), ]
##     new_train <- rbind(other_classes, curr_class)
##     new_f <- evaluate_model(eval_fun, new_train, original)
##     for (i in 1:ncol(class_data)) {
##         if (class_data["orig", i] != -1 & ((class_data["best_f",
##             i] - new_f[i])/class_data["best_f", i] > increment_minimum)) {
##             return(list(train_data, class_data))
##         }
##         else if (class_data["min", i] & (class_data["best_f",
##             i] > new_f[i])) {
##             return(train_data)
##         }
##         else {
##             return(new_train)
##         }
```

```
##     }
## }
```

```
wrap_smote
```

```
## function (class_data, train_data, original, eval_fun, seed = FALSE)
## {
##     class_data <- rbind(class_data, stop_sampling = !class_data["min",
##         ])
##     while (sum(class_data["stop_sampling", ]) < ncol(class_data)) {
##         print("classes to go")
##         print(sum(class_data["stop_sampling", ]))
##         for (i in 1:ncol(class_data)) {
##             print("i")
##             print(i)
##             if (class_data["stop_sampling", i] == FALSE) {
##                 train_len = nrow(train_data)
##                 print("train len")
##                 print(train_len)
##                 train_data <- smote_sample(train_data, original,
##                   colnames(class_data)[i], class_data, eval_fun,
##                   seed)
##                 if (nrow(train_data) == train_len) {
##                   class_data["stop_sampling", i] <- TRUE
##                 }
##                 print("new df")
##                 print(nrow(train_data))
##             }
##         }
##         new_f <- evaluate_model(eval_fun, train_data)
##         class_data["best_f", ] <- pmax(as.numeric(class_data["best_f",
##             ]), new_f)
##     }
##     return(list(train_data, class_data))
## }
```

```
smote_sample
```

```
## function (train_data, original, class_val, class_data, eval_fun,
##     seed = FALSE)
## {
##     increment_minimum <- 0.05
##     other_classes <- train_data[train_data[, ncol(train_data)] !=
##         class_val, ]
##     curr_class <- train_data[train_data[, ncol(train_data)] ==
##         class_val, ]
##     synth_class <- smote(curr_class, train_data, seed)
##     new_f <- evaluate_model(eval_fun, rbind(train_data, synth_class),
##         original)
##     print("new_f")
##     print(new_f)
##     class_col <- paste("Class:", class_val)
##     if ((new_f[class_col] - class_data["best_f", as.character(class_val)])/class_data["best_f",
```

```
##         as.character(class_val)] < increment_minimum) {
##         return(train_data)
##     }
##     else {
##         return(rbind(train_data, synth_class))
##     }
## }
```

```
smote
```

```
## function (minor_class, train_data, seed = FALSE)
## {
##     k <- 6
##     synth_data <- data.frame(matrix(NA, nrow = 0, ncol = ncol(minor_class)))
##     colnames(synth_data) <- colnames(minor_class)
##     if (seed) {
##         set.seed(2292)
##     }
##     nbr_idxs <- get.knnx(train_data[, 1:(ncol(train_data) - 1)],
##         minor_class[, 1:(ncol(train_data) - 1)], k = k)[[1]]
##     for (i in 1:nrow(minor_class)) {
##         near_nbrs <- train_data[nbr_idxs[i, 2:6], ]
##         class_rep <- minor_class[i, ]
##         synth_data <- rbind(synth_data, populate(class_rep, near_nbrs,
##             k - 1))
##     }
##     return(synth_data)
## }
```

```
populate
```

```
## function (class_rep, near_nbrs, k)
## {
##     os_rate <- 1
##     new_samples <- data.frame(matrix(NA, nrow = 0, ncol = ncol(near_nbrs)))
##     colnames(new_samples) <- colnames(near_nbrs)
##     while (os_rate > 0) {
##         nn <- sample(1:k, 1)
##         synthetic <- class_rep[1, ]
##         for (i in 1:(ncol(near_nbrs) - 1)) {
##             dif <- near_nbrs[nn, i] - class_rep[1, i]
##             gap <- runif(1, 0, 1)
##             synthetic[1, i] <- class_rep[1, i] + gap * dif
##         }
##         new_samples <- rbind(new_samples, synthetic)
##         os_rate <- os_rate - 1
##     }
##     return(new_samples)
## }
```

```
# Custom evaluation function
evaluate_model
```

```
## function (eval_fun, train_set, test_set = NULL)
## {
##     if (missing(test_set)) {
##         cm <- confusionMatrix(eval_fun(train_set)$final, train_set[,
##             ncol(train_set)])
##     }
##     else {
##         cm <- confusionMatrix(eval_fun(train_set, test_set)$final,
##             test_set[, ncol(test_set)])
##     }
##     f_vals <- (2 * cm$byClass[, 1] * cm$byClass[, 3])/(cm$byClass[,
##         1] + cm$byClass[, 3])
##     return(f_vals)
## }
```

Bibliography

Chawla NV, Cieslak DA, Bowyer KW, Kegelmeyer WP SMOTE Synthetic Minority Over-sampling TEchnique Journal of Artificial Intelligence Research. 2002;16:341-78

Chawla NV, Cieslak DA, Hall LO, Joshi A. Automatically countering imbalance and its empirical relationship to cost. Data Mining and Knowledge Discovery. 2008;17(2):225-52.

Cortez P, Cerdeira A, Almeida F, Matos T, Reis J. Modeling wine preferences by data mining from physicochemical properties. Decision Support Systems. 2009;47(4):547-53.

Qian Y et al., A resampling ensemble algorithm for classification of imbalance problems, Neurocomputing (2014), http://dx.doi.org/10.1016/j.neucom.2014.06.021i

Rokach L. Ensemble-based classifiers. Artificial Intelligence Review. 2010;33(1):1-39.