First, let’s start by installing and loading icarus and nnet, the two packages needed in this tutorial, from CRAN (if necessary):

install.packages(c("icarus","nnet"))

library(icarus)

library(nnet)

Then load the data:

load("data/weighting\_ML\_part1.RData")

The RData file contains two dataframes, one for the training set and one for the test set. They contain results of some international soccer games, from 01/2008 to 12/2016 for the training set, and from 01/2017 to 11/2017 for the test. Along with the team names and goals scored for each side, a few descriptive variables that we’re going to use as features of our ML models:

> head(train\_soccer)

Date team opponent\_team home\_field elo\_team

1 2010-10-12 Belarus Albania 1 554

2 2010-10-08 Bosnia and Herzegovina Albania 0 544

3 2011-06-07 Bosnia and Herzegovina Albania 0 594

4 2011-06-20 Argentina Albania 1 1267

5 2011-08-10 Montenegro Albania 0 915

6 2011-09-02 France Albania 0 918

opponent\_elo importance goals\_for goals\_against outcome year

1 502 1 2 0 WIN 2010

2 502 1 1 1 DRAW 2010

3 564 1 2 0 WIN 2011

4 564 1 4 0 WIN 2011

5 524 1 2 3 LOSS 2011

6 546 1 2 1 WIN 2011

*elo\_team* and *opponent\_elo* are quantitative variables indicative of the level of the team at the date of the game ; *importance* is a measure of high-profile the game played was (a friendly match rates 1 while a World Cup game rates 4). The other variables are imo self-descriptive.

Then we can train a multinomial logistic regression, with *outcome* being the predicted variable, and compute the predictions from the model:

outcome\_model\_unw <- multinom(outcome ~ elo\_team + opponent\_elo + home\_field + importance,

data = train\_soccer)

test\_soccer$pred\_outcome\_unw <- predict(outcome\_model\_unw, newdata = test\_soccer)

The sheer accuracy of this predictor is kinda good:

> ## Accuracy

> sum(test\_soccer$pred\_outcome\_unw == test\_soccer$outcome) / nrow(test\_soccer)

[1] 0.5526316

but it has a problem: it never predicts draws!

> summary(test\_soccer$pred\_outcome\_unw)

DRAW LOSS WIN

0 208 210

And indeed, draws being less common than other results, it seems more profitable for the algorithm that optimizes accuracy never to predict them. As a consequence, the probabilities of the game being a draw is always lesser than the probability of one team winning it.

A common solution to this problem is to use reweighting to correct the imbalances in the sample, which we’ll now tackle. It is important to note that the weighting trick has to happen in the training set to avoid “data leaks”.

**Subsampling Techniques**

To illustrate these methods, let’s simulate some data with a class imbalance using this method. We will simulate a training and test set where each contains 10000 samples and a minority class rate of about 5.9%:

**library**(caret)

**set.seed**(2969)

imbal\_train <- **twoClassSim**(10000, intercept = -20, linearVars = 20)

imbal\_test <- **twoClassSim**(10000, intercept = -20, linearVars = 20)

**table**(imbal\_train$Class)

##

## Class1 Class2

## 9411 589

Let’s create different versions of the training set prior to model tuning:

**set.seed**(9560)

down\_train <- **downSample**(x = imbal\_train[, -**ncol**(imbal\_train)],

y = imbal\_train$Class)

**table**(down\_train$Class)

##

## Class1 Class2

## 589 589

**set.seed**(9560)

up\_train <- **upSample**(x = imbal\_train[, -**ncol**(imbal\_train)],

y = imbal\_train$Class)

**table**(up\_train$Class)

##

## Class1 Class2

## 9411 9411

**library**(DMwR)

**set.seed**(9560)

smote\_train <- **SMOTE**(Class ~ ., data = imbal\_train)

**table**(smote\_train$Class)

##

## Class1 Class2

## 2356 1767

**library**(ROSE)

**set.seed**(9560)

rose\_train <- **ROSE**(Class ~ ., data = imbal\_train)$data

**table**(rose\_train$Class)

##

## Class1 Class2

## 4939 5061

For these data, we’ll use a bagged classification and estimate the area under the ROC curve using five repeats of 10-fold CV.

ctrl <- **trainControl**(method = "repeatedcv", repeats = 5,

classProbs = TRUE,

summaryFunction = twoClassSummary)

**set.seed**(5627)

orig\_fit <- **train**(Class ~ ., data = imbal\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

**set.seed**(5627)

down\_outside <- **train**(Class ~ ., data = down\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

**set.seed**(5627)

up\_outside <- **train**(Class ~ ., data = up\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

**set.seed**(5627)

rose\_outside <- **train**(Class ~ ., data = rose\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

**set.seed**(5627)

smote\_outside <- **train**(Class ~ ., data = smote\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

We will collate the resampling results and create a wrapper to estimate the test set performance:

outside\_models <- **list**(original = orig\_fit,

down = down\_outside,

up = up\_outside,

SMOTE = smote\_outside,

ROSE = rose\_outside)

outside\_resampling <- **resamples**(outside\_models)

test\_roc <- **function**(model, data) {

**library**(pROC)

roc\_obj <- **roc**(data$Class,

**predict**(model, data, type = "prob")[, "Class1"],

levels = **c**("Class2", "Class1"))

**ci**(roc\_obj)

}

outside\_test <- **lapply**(outside\_models, test\_roc, data = imbal\_test)

outside\_test <- **lapply**(outside\_test, as.vector)

outside\_test <- **do.call**("rbind", outside\_test)

**colnames**(outside\_test) <- **c**("lower", "ROC", "upper")

outside\_test <- **as.data.frame**(outside\_test)

**summary**(outside\_resampling, metric = "ROC")

##

## Call:

## summary.resamples(object = outside\_resampling, metric = "ROC")

##

## Models: original, down, up, SMOTE, ROSE

## Number of resamples: 50

##

## ROC

## Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

## original 0.9098237 0.9298348 0.9386021 0.9394130 0.9493394 0.9685873 0

## down 0.9095558 0.9282175 0.9453907 0.9438384 0.9596021 0.9836254 0

## up 0.9989350 0.9999980 1.0000000 0.9998402 1.0000000 1.0000000 0

## SMOTE 0.9697171 0.9782214 0.9834234 0.9817476 0.9857071 0.9928255 0

## ROSE 0.8782985 0.8941488 0.8980313 0.8993135 0.9056404 0.9203092 0

outside\_test

## lower ROC upper

## original 0.9130010 0.9247957 0.9365905

## down 0.9286964 0.9368361 0.9449758

## up 0.9244128 0.9338499 0.9432869

## SMOTE 0.9429536 0.9490585 0.9551634

## ROSE 0.9383809 0.9459729 0.9535649

The training and test set estimates for the area under the ROC curve do not appear to correlate. Based on the resampling results, one would infer that up-sampling is nearly perfect and that ROSE does relatively poorly. The reason that up-sampling appears to perform so well is that the samples in the majority class are replicated and have a large potential to be in both the model building and hold-out sets. In essence, the hold-outs here are not truly independent samples.

In reality, all of the sampling methods do about the same (based on the test set). The statistics for the basic model fit with no sampling are fairly in-line with one another (0.939 via resampling and 0.925 for the test set).

## 11.2 Subsampling During Resampling

Recent versions of **caret** allow the user to specify subsampling when using train so that it is conducted inside of resampling. All four methods shown above can be accessed with the basic package using simple syntax. If you want to use your own technique, or want to change some of the parameters for SMOTE or ROSE, the last section below shows how to use custom subsampling.

The way to enable subsampling is to use yet another option in trainControl called sampling. The most basic syntax is to use a character string with the name of the sampling method, either "down", "up", "smote", or "rose". Note that you will need to have the **DMwR** and **ROSE** packages installed to use SMOTE and ROSE, respectively.

One complication is related to pre-processing. Should the subsampling occur before or after the pre-processing? For example, if you down-sample the data and using PCA for signal extraction, should the loadings be estimated from the entire training set? The estimate is potentially better since the entire training set is being used but the subsample may happen to capture a small potion of the PCA space. There isn’t any obvious answer.

The default behavior is to subsample the data prior to pre-processing. This can be easily changed and an example is given below.

Now let’s re-run our bagged tree models while sampling inside of cross-validation:

ctrl <- **trainControl**(method = "repeatedcv", repeats = 5,

classProbs = TRUE,

summaryFunction = twoClassSummary,

## new option here:

sampling = "down")

**set.seed**(5627)

down\_inside <- **train**(Class ~ ., data = imbal\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

## now just change that option

ctrl$sampling <- "up"

**set.seed**(5627)

up\_inside <- **train**(Class ~ ., data = imbal\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

ctrl$sampling <- "rose"

**set.seed**(5627)

rose\_inside <- **train**(Class ~ ., data = imbal\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

ctrl$sampling <- "smote"

**set.seed**(5627)

smote\_inside <- **train**(Class ~ ., data = imbal\_train,

method = "treebag",

nbagg = 50,

metric = "ROC",

trControl = ctrl)

Here are the resampling and test set results:

inside\_models <- **list**(original = orig\_fit,

down = down\_inside,

up = up\_inside,

SMOTE = smote\_inside,

ROSE = rose\_inside)

inside\_resampling <- **resamples**(inside\_models)

inside\_test <- **lapply**(inside\_models, test\_roc, data = imbal\_test)

inside\_test <- **lapply**(inside\_test, as.vector)

inside\_test <- **do.call**("rbind", inside\_test)

**colnames**(inside\_test) <- **c**("lower", "ROC", "upper")

inside\_test <- **as.data.frame**(inside\_test)

**summary**(inside\_resampling, metric = "ROC")

##

## Call:

## summary.resamples(object = inside\_resampling, metric = "ROC")

##

## Models: original, down, up, SMOTE, ROSE

## Number of resamples: 50

##

## ROC

## Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

## original 0.9098237 0.9298348 0.9386021 0.9394130 0.9493394 0.9685873 0

## down 0.9140294 0.9381766 0.9453610 0.9438490 0.9492917 0.9684522 0

## up 0.8887678 0.9308075 0.9393226 0.9392084 0.9517913 0.9679569 0

## SMOTE 0.9203876 0.9453453 0.9520074 0.9508721 0.9596354 0.9746933 0

## ROSE 0.9305013 0.9442821 0.9489859 0.9511117 0.9572416 0.9756750 0

inside\_test

## lower ROC upper

## original 0.9130010 0.9247957 0.9365905

## down 0.9354534 0.9419704 0.9484875

## up 0.9353945 0.9431074 0.9508202

## SMOTE 0.9465262 0.9524213 0.9583164

## ROSE 0.9369170 0.9448367 0.9527563

The figure below shows the difference in the area under the ROC curve and the test set results for the approaches shown here. Repeating the subsampling procedures for every resample produces results that are more consistent with the test set.

## 11.3 Complications

The user should be aware that there are a few things that can happening when subsampling that can cause issues in their code. As previously mentioned, when sampling occurs in relation to pre-processing is one such issue. Others are:

* Sparsely represented categories in factor variables may turn into zero-variance predictors or may be completely sampled out of the model.
* The underlying functions that do the sampling (e.g. SMOTE, downSample, etc) operate in very different ways and this can affect your results. For example, SMOTE and ROSE will convert your predictor input argument into a data frame (even if you start with a matrix).
* Currently, sample weights are not supported with sub-sampling.
* If you use tuneLength to specify the search grid, understand that the data that is used to determine the grid has not been sampled. In most cases, this will not matter but if the grid creation process is affected by the sample size, you may end up using a sub-optimal tuning grid.
* For some models that require more samples than parameters, a reduction in the sample size may prevent you from being able to fit the model.

## 11.4 Using Custom Subsampling Techniques

Users have the ability to create their own type of subsampling procedure. To do this, alternative syntax is used with the sampling argument of the trainControl. Previously, we used a simple string as the value of this argument. Another way to specify the argument is to use a list with three (named) elements:

* The name value is a character string used when the train object is printed. It can be any string.
* The func element is a function that does the subsampling. It should have arguments called x and y that will contain the predictors and outcome data, respectively. The function should return a list with elements of the same name.
* The first element is a single logical value that indicates whether the subsampling should occur first relative to pre-process. A value of FALSE means that the subsampling function will receive the sampled versions of x and y.

For example, here is what the list version of the sampling argument looks like when simple down-sampling is used:

down\_inside$control$sampling

## $name

## [1] "down"

##

## $func

## function(x, y)

## downSample(x, y, list = TRUE)

##

## $first

## [1] TRUE

As another example, suppose we want to use SMOTE but use 10 nearest neighbors instead of the default of 5. To do this, we can create a simple wrapper around the SMOTE function and call this instead:

smotest <- **list**(name = "SMOTE with more neighbors!",

func = **function** (x, y) {

**library**(DMwR)

dat <- **if** (**is.data.frame**(x)) x **else** **as.data.frame**(x)

dat$.y <- y

dat <- **SMOTE**(.y ~ ., data = dat, k = 10)

**list**(x = dat[, !**grepl**(".y", **colnames**(dat), fixed = TRUE)],

y = dat$.y)

},

first = TRUE)

The control object would then be:

ctrl <- **trainControl**(method = "repeatedcv", repeats = 5,

classProbs = TRUE,

summaryFunction = twoClassSummary,

sampling = smotest)

[](https://topepo.github.io/caret/subsampling-for-class-imbalances.html)https://topepo.github.io/caret/

Commonly, you would do:

train\_soccer$weight <- 1

train\_soccer[train\_soccer$outcome == "DRAW",]$weight <- (nrow(train\_soccer)/table(train\_soccer$outcome)[1]) \* 1/3

train\_soccer[train\_soccer$outcome == "LOSS",]$weight <- (nrow(train\_soccer)/table(train\_soccer$outcome)[2]) \* 1/3

train\_soccer[train\_soccer$outcome == "WIN",]$weight <- (nrow(train\_soccer)/table(train\_soccer$outcome)[3]) \* 1/3

> table(train\_soccer$weight)

0.916067146282974 1.22435897435897

3336 1248

The draws are reweighted with a factor greater than 1 and the other games with a factor lesser than 1. This balances the predicted outcomes and thus improves the quality of the probabilities …

outcome\_model <- multinom(outcome ~ elo\_team + opponent\_elo + home\_field + importance,

data = train\_soccer,

weights = train\_soccer$weight)

test\_soccer$pred\_outcome <- predict(outcome\_model, newdata = test\_soccer)

> summary(test\_soccer$pred\_outcome)

DRAW LOSS WIN

96 167 155

… though at a loss in accuracy:

> ## Accuracy

> sum(test\_soccer$pred\_outcome == test\_soccer$outcome) / nrow(test\_soccer)

[1] 0.5263158

Now let’s look at the balance of our training sample on other variables:

> round(table(test\_soccer$importance) / nrow(test\_soccer),2)

1 2 3 4

0.26 0.08 0.54 0.12

> round(table(train\_soccer$importance) / nrow(train\_soccer),2)

1 2 3 4

0.56 0.08 0.23 0.12

It seems that the test set features a lot more important matches than the training set. Let’s look further, in particular at the dates the matches of the training set were played:

> round(table(train\_soccer$year) / nrow(train\_soccer),2)

2008 2009 2010 2011 2012 2013 2014 2015 2016

0.10 0.11 0.11 0.10 0.11 0.13 0.11 0.11 0.12

Thus the matches of each year between 2008 and 2016 have the same influence on the final predictor. A better idea would be to give the most recent games a slightly higher influence, for example by increasing their weight and thus reducing the weights of the older games:

nyears <- length(unique(train\_soccer$year))

year\_tweak <- rep(1/nyears,nyears) \* 1:nyears

year\_tweak <- year\_tweak \* 1/sum(year\_tweak) ## Normalization

> year\_tweak

[1] 0.02222222 0.04444444 0.06666667 0.08888889 0.11111111 0.13333333

[7] 0.15555556 0.17777778 0.20000000

We determine it is thus a good idea to balance on these two additional variables (*year* and *importance*). Now how should we do this? A solution could be to create an indicator variable containing all the values of the cross product between the variables *outcome*, *year* and *importance*, and use the same reweighting technique as before. But this would not be very practical and more importantly, some of the sub-categories would be nearly empty, making the procedure not very robust. A better solution is to use survey sampling calibration and Icarus 

train\_soccer$weight\_cal <- 1

importance\_pct\_test <- unname(

table(test\_soccer$importance) / nrow(test\_soccer),

)

marginMatrix <- matrix(, nrow = 0, ncol = 1) %>% ## Will be replaced by newMarginMatrix() in icarus 0.3.2

addMargin("outcome", c(0.333,0.333,0.333)) %>%

addMargin("importance", importance\_pct\_test) %>%

addMargin("year", year\_tweak)

train\_soccer$weight\_cal <- calibration(data=train\_soccer, marginMatrix=marginMatrix,

colWeights="weight\_cal", pct=TRUE, description=TRUE,

popTotal = nrow(train\_soccer), method="raking")

outcome\_model\_cal <- multinom(outcome ~ elo\_team + opponent\_elo + home\_field + importance,

data = train\_soccer,

weights = train\_soccer$weight\_cal)

test\_soccer$pred\_outcome\_cal <- predict(outcome\_model\_cal, newdata = test\_soccer)

icarus gives a summary of the calibration procedure in the log (too long to reproduce here). We then observe a slight improvement in accuracy compared to the previous reweighting technique:

> sum(test\_soccer$pred\_outcome\_cal == test\_soccer$outcome) / nrow(test\_soccer)

[1] 0.5478469

But more importantly we have reason to believe that the we improved the quality of the probabilities assigned to each event (we could check this using metrics such as the Brier score or calibration plots) 

It is also worth noting that some algorithms (especially those who rely on bagging, boosting, or more generally on ensemble methods) naturally do a good job at balancing samples. You could for example rerun the whole code and replace the logit regressions by boosted algorithms. You would then observe fewer differences between the unweighted algorithm and its weighted counterparts.

Stay tuned for the part 2, where we’ll show a trick to craft better probabilities (particularly for simulations) using external knowledge on probabilities.