Chapter 15
Improving Model Performance

We already explored several alternative machine learning (ML) methods for prediction, classification, clustering, and outcome forecasting. In many situations, we derive models by estimating model coefficients or parameters. The main question now is *How can we adopt the advantages of crowdsourcing and biosocial networking to aggregate different predictive analytics strategies?* Are there reasons to believe that such **ensembles** of forecasting methods may actually improve the performance (e.g., increase prediction accuracy) of the resulting consensus meta-algorithm? In this chapter, we are going to introduce ways that we can search for optimal parameters for a single ML method, as well as aggregate different methods into **ensembles** to enhance their collective performance relative to any of the individual methods part of the meta-aggregate.

After we summarize the core methods, we will present automated and customized parameter tuning, and show strategies for improving model performance based on meta-learning via bagging and boosting.

15.1 Improving Model Performance by Parameter Tuning

One of the methods for improving model performance relies on *tuning*, which is the process of searching for the best parameters for a specific method. Table 15.1 summarizes the parameters for each method we covered in previous chapters.

15.2 Using `caret` for Automated Parameter Tuning

In Chap. 7, we used KNN and plugged in random k parameters for the number of clusters. This time, we will test multiple k values simultaneously and pick the one with the highest accuracy. When using the `caret` package, we need to specify a

I. D. Dinov, *Data Science and Predictive Analytics*,
https://doi.org/10.1007/978-3-319-72347-1_15

Table 15.1 Synopsis of the basic prediction, classification and clustering methods and their core parameters

Model	Learning task	Method	Parameters
KNN	Classification	`class::knn`	`data, k`
K-Means	Classification	`stats::kmeans`	`data, k`
Naïve Bayes	Classification	`e1071::naiveBayes`	`train, class, laplace`
Decision Trees	Classification	`C50::C5.0`	`train, class, trials, costs`
OneR Rule Learner	Classification	`RWeka::OneR`	`class~predictors, data`
RIPPER Rule Learner	Classification	`RWeka::JRip`	`formula, data, subset, na.action, control, options`
Linear Regression	Regression	`stats::lm`	`formula, data, subset, weights, na.action, method`
Regression Trees	Regression	`rpart::rpart`	`dep_var ~ indep_var, data`
Model Trees	Regression	`RWeka::M5P`	`formula, data, subset, na.action, control`
Neural Networks	Dual use	`nnet::nnet`	`x, y, weights, size, Wts, mask, linout, entropy, softmax, censored, skip, rang, decay, maxit, Hess, trace, MaxNWts, abstol, reltol`
Support Vector Machines (Polynomial Kernel)	Dual use	`caret::train::svmLinear`	`C`
Support Vector Machines (Radial Basis Kernel)	Dual use	`caret::train::svmRadial`	`C, sigma`
Support Vector Machines (general)	Dual use	`kernlab::ksvm`	`formula, data, kernel`
Random Forests	Dual use	`randomForest::randomForest`	`formula, data`

class variable, a dataset containing a class variable, predicting features, and the method we will be using. In Chap. 7, we used the Boys Town Study of Youth Development dataset, normalized all the features, stored them in `boystown_n`, and formulated the outcome class variable first (`boystown$grade`).

```
str(boystown_n)

## 'data.frame':    200 obs. of  10 variables:
##  $ sex       : num  0 0 0 0 1 1 0 0 1 1 ...
##  $ gpa       : num  1 0 0.6 0.4 0.6 0.6 0.2 1 0.2 0.6 ...
##  $ Alcoholuse: num  0.182 0.364 0.182 0.182 0.545 ...
##  $ alcatt    : num  0.5 0.333 0.5 0.167 0.333 ...
##  $ dadjob    : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ momjob    : num  0 0 0 0 1 0 0 0 1 1 ...
##  $ dadclose  : num  0.143 0.429 0.286 0.143 0.286 ...
##  $ momclose  : num  0.143 0.571 0.286 0.286 0.143 ...
##  $ larceny   : num  0.25 0 0 0.75 0.25 0 0 0 0.25 0.25 ...
##  $ vandalism : num  0.429 0 0.286 0.286 0.286 ...

boystown_n<-cbind(boystown_n, boystown[, 11])
str(boystown_n)

## 'data.frame':    200 obs. of  11 variables:
##  $ sex           : num  0 0 0 0 1 1 0 0 1 1 ...
##  $ gpa           : num  1 0 0.6 0.4 0.6 0.6 0.2 1 0.2 0.6 ...
##  $ Alcoholuse    : num  0.182 0.364 0.182 0.182 0.545 ...
##  $ alcatt        : num  0.5 0.333 0.5 0.167 0.333 ...
##  $ dadjob        : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ momjob        : num  0 0 0 0 1 0 0 0 1 1 ...
##  $ dadclose      : num  0.143 0.429 0.286 0.143 0.286 ...
##  $ momclose      : num  0.143 0.571 0.286 0.286 0.143 ...
##  $ larceny       : num  0.25 0 0 0.75 0.25 0 0 0 0.25 0.25 ...
##  $ vandalism     : num  0.429 0 0.286 0.286 0.286 ...
##  $ boystown[, 11]: Factor w/ 2 levels "above_avg","avg_or_below": 2 1 2 1
2 2 1 2 1 2 ...

colnames(boystown_n)[11]<-"grade"
```

The dataset including a specific class variable and predictive features is now successfully created. We are using the KNN method as an example with the class variable `grade`. So, we plug this information into the `caret::train()` function. Note that `caret` is using the full dataset because it will automatically do the random sampling for us. To make the results reproducible, we utilize the `set.seed()` function that we previously used, see Chap. 14.

```
library(caret)

set.seed(123)
m<-train(grade~., data=boystown_n, method="knn")
m; summary(m)

## k-Nearest Neighbors
##
## 200 samples
##  10 predictor
##   2 classes: 'above_avg', 'avg_or_below'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...
## Resampling results across tuning parameters:
##
```

```
##   k  Accuracy   Kappa
##   5  0.7952617  0.5193402
##   7  0.8143626  0.5585191
##   9  0.8070520  0.5348281
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 7.

##             Length Class      Mode
## learn        2     -none-     list
## k            1     -none-     numeric
## theDots      0     -none-     list
## xNames      10     -none-     character
## problemType  1     -none-     character
## tuneValue    1     data.frame list
## obsLevels    2     -none-     character
```

In this case, using `str(m)` to summarize the object `m` may report out too much information. Instead, we can simply type the object name `m` to get more concise information about it.

1. Description about the dataset: number of samples, features, and classes.
2. Re-sampling process: here, we use 25 bootstrap samples with 200 observations (same size as the observed dataset) each to train the model.
3. Candidate models with different parameters that have been evaluated: by default, `caret` uses 3 different choices for each parameter, but for binary parameters, it only allows two choices, `TRUE` and `FALSE`). As KNN has only one parameter *k*, we have three candidate models reported in the output above.
4. Optimal model: the model with largest accuracy is the one corresponding to `k=5`.

Let's see how accurate this "optimal model" is in terms of the re-substitution error. Again, we will use the `predict()` function specifying the object `m` and the dataset `boystown_n`. Then, we can report the contingency table showing the agreement between the predictions and real class labels.

```
set.seed(1234)
p<-predict(m, boystown_n)
table(p, boystown_n$grade)

##
## p              above_avg avg_or_below
##   above_avg          132           17
##   avg_or_below         2           49
```

This model has (17 + 2)/200 = 0.09 re-substitution error (9%). This means that in the 200 observations that we used to train this model, 91% of them were correctly classified. Note that re-substitution error is different from accuracy. The accuracy of this model is 0.8, which is reported by a model summary call. As mentioned in Chap. 14, we can obtain prediction probabilities for each observation in the original `boystown_n` dataset.

```
head(predict(m, boystown_n, type = "prob"))

##   above_avg avg_or_below
## 1 0.0000000    1.0000000
## 2 1.0000000    0.0000000
## 3 0.7142857    0.2857143
## 4 0.8571429    0.1428571
## 5 0.2857143    0.7142857
## 6 0.5714286    0.4285714
```

15.2.1 Customizing the Tuning Process

The default setting of `train()` might not meet the specific needs for every study. In our case, the optimal *k* might be smaller than 5. The `caret` package allows us to customize the settings for `train()`.

`caret::trianControl()` can help us to customize re-sampling methods. There are 6 popular re-sampling methods that we might want to use in the following table (Table 15.2).

These methods are helping us find representative samples to train the model. Let's use *0.632 bootstrap* for example. Just specify `method="boot632"` in the `trainControl()` function. The number of different samples to include can be customized by `number=` option. Another option in `trainControl()` is about the model performance evaluation. We can change our preferred method of evaluation to select the optimal model. The `oneSE` method chooses the simplest model within one standard error of the best performance to be the optimal model. Other methods are also available in `caret` package. For detailed information, type `best` in R console.

We can also specify a list of *k* values we want to test by creating a matrix or a grid.

```
ctrl<-trainControl(method="boot632", number=25, selectionFunction="oneSE")
grid<-expand.grid(.k=c(1, 3, 5, 7, 9))
# Creates a data frame from all combinations of the supplied factors
```

Table 15.2 Six complementary methods for customizing the `caret::trainControl()` re-sampling

Resampling method	Method name	Additional options and default values
Holdout sampling	`LGOCV`	`p = 0.75` (training data proportion)
k-fold cross-validation	`cv`	`number = 10` (number of folds)
Repeated k-fold cross validation	`repeatedcv`	`number = 10` (number of folds), `repeats = 10` (number of iterations)
Bootstrap sampling	`boot`	`number = 25` (resampling iterations)
0.632 bootstrap	`boot632`	`number = 25` (resampling iterations)
Leave-one-out cross-validation	`LOOCV`	None

Usually, to avoid ties, we prefer to choose an odd number of clusters *k*. Now the constraints are all set. We can start to select models again using `train()`.

```
set.seed(123)
m<-train(grade~., data=boystown_n, method="knn",
         metric="Kappa",
         trControl=ctrl,
         tuneGrid=grid)
m

## k-Nearest Neighbors
##
## 200 samples
##  10 predictor
##   2 classes: 'above_avg', 'avg_or_below'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   1  0.8726660  0.7081751
##   3  0.8457584  0.6460742
##   5  0.8418226  0.6288675
##   7  0.8460327  0.6336463
##   9  0.8381961  0.6094088
##
## Kappa was used to select the optimal model using  the one SE rule.
## The final value used for the model was k = 1.
```

Here we added `metric="Kappa"` to include the *Kappa statistics* as one of the criteria to select the optimal model. We can see the output accuracy for all the candidate models are better than the default bootstrap sampling. The optimal model has *k=3*, a high accuracy 0.846, and a high Kappa statistic, which is much better than the model we had in Chap. 7. As you can see from the output, the SE rule no longer choses the model with the highest accuracy or Kappa statistic to be the "optimal model". It is a more comprehensive method than only looks at one statistic or a single quality measure.

15.2.2 Improving Model Performance with Meta-learning

Meta-learning involves building multiple learners (can be single or multiple learning algorithms) at the same time. It combines the output from these learners and generates more effective meta-classifiers.

To decrease the variance (bagging) or bias (boosting), *random forests* attempt in two steps to correct the general decision trees' trend to overfit the model to the training set:

1. Producing a distribution of simple ML models on subsets of the original data.
2. Combining the distribution into one "aggregated" model.

Before stepping into the details, let's briefly summarize:

- *Bagging* (stands for Bootstrap Aggregating) is a way to decrease the variance of your prediction by generating additional data for training from your original dataset. It generates multiple sets of the same cardinality/size as your original data, as combinations with repetitions. By increasing the size of your training set you can't improve the model predictive force, but just decrease the variance, narrowly tuning the prediction to the expected outcome.
- *Boosting* is a two-step approach, where one first uses subsets of the original data to produce a series of moderately performing models and then "boosts" their performance by combining them together using a particular cost function (e.g., Accuracy). Unlike bagging, in classical boosting, the subset creation is not random and depends upon the performance of the previous models: every new subset contains the elements that were (likely to be) misclassified by previous models. Usually, we prefer weaker classifiers in boosting. For example, a prevalent choice is to use stump (level-one decision tree) in AdaBoost (Adaptive Boosting).

15.2.3 Bagging

One of the most well-known meta-learning method is bootstrap aggregating or *bagging*. It builds multiple models with bootstrap samples using a single algorithm. The models' predictions are combined with voting (for classification) or averaging (for numeric prediction). Voting means that bagging model's prediction is based on the majority of learners' predictions for a class. Bagging is especially good with unstable learners like decision trees or SVM models.

To illustrate the Bagging method, we will again use the Quality of Life and chronic disease dataset in Chap. 9. Just like we did in the second practice problem in Chap. 11, we will use CHARLSONSCORE as the classes labels, which has 11 different class labels.

```
qol<-read.csv("https://umich.instructure.com/files/481332/download?download_
frd=1")
qol<-qol[!qol$CHARLSONSCORE==-9 , -c(1, 2)]
qol$CHARLSONSCORE<-as.factor(qol$CHARLSONSCORE)
```

To apply bagging(), we need to download the ipred package first. After loading the package, we build a bagging model with CHARLSONSCORE as class

label and all other variables in the dataset as predictors. We can specify the number of voters (decision tree models we want to have), the default number is 25.

```
# install.packages("ipred")
library(ipred)
set.seed(123)
mybag<-bagging(CHARLSONSCORE~., data=qol, nbagg=25)
```

Next, we shall use the `predict()` function to apply this model for prediction. For evaluation purposes, we create a table reporting the re-substitution error.

```
bt_pred<-predict(mybag, qol)
agreement<-bt_pred==qol$CHARLSONSCORE
prop.table(table(agreement))
```

```
## agreement
##       FALSE        TRUE
## 0.001718213 0.998281787
```

This model works very well with its training data. It labeled 99.8% of the cases correctly. To see its performances on feature data, we apply the `caret train()` function again with 10 repeated CV as re-sampling method. In `caret`, bagged trees method is called `treebag`.

```
library(caret)
set.seed(123)
ctrl<-trainControl(method="repeatedcv", number = 10, repeats = 10)
train(CHARLSONSCORE~., data=as.data.frame(qol), method="treebag", trControl=
ctrl)
```

```
## Bagged CART
##
## 2328 samples
##   38 predictor
##   11 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 2095, 2096, 2093, 2094, 2098, 2097, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.5234615  0.2173193
```

We got an accuracy of 52% and a fair Kappa statistics. This result is better than our previous prediction attempt in Chap. 11 using the `ksvm()` function alone (~50%). Here, we combined the prediction results of 38 decision trees to get this level of prediction accuracy.

In addition to decision tree classification, `caret` allows us to explore alternative `bag()` functions. For instance, instead of bagging based on decision trees, we can bag using an SVM model. `caret` provides a nice setting for SVM training, making

predictions and counting votes in a list object svmBag. We can examine these objects by using the str() function.

```
str(svmBag)

## List of 3
##  $ fit      :function (x, y, ...)
##  $ pred     :function (object, x)
##  $ aggregate:function (x, type = "class")
```

Clearly, fit provides the training functionality, pred the prediction and forecasting on new data, and aggregate is a way to combine many models and achieve voting-based consensus. Using the member operator, the $ sign, we can explore these three types of elements of the svmBag object. For instance, the fit element may be extracted from the SVM object by:

```
svmBag$fit

## function (x, y, ...)
## {
##     loadNamespace("kernlab")
##     out <- kernlab::ksvm(as.matrix(x), y, prob.model = is.factor(y),
##         ...)
##     out
## }
## <environment: namespace:caret>
```

fit relies on the ksvm() function in the kernlab package, which means this package needs to be loaded. The other two methods, pred and aggregate, may be explored in a similar way. They just follow the SVM model building and testing process we discussed in Chap. 11.

This svmBag object could be used as an optional setting in the train() function. However, this option requires that all features are linearly independent with trivial covariances, which may be rare in real world data.

15.2.4 Boosting

Bagging uses equal weights for all learners we included in the model. Boosting is quite different in terms of weights. Suppose we have the first learner correctly classifying 60% of the observations. This 60% of data may be less likely to be included in the training dataset for the next learner. So, we have more learners working on "hard-to-classify" observations.

Mathematically, we are using a weighted sum of functions to predict the outcome class labels. We can try to fit the true model using weighted additive modeling. We start with a random learner that can classify some of the observations correctly, possibly with some errors.

$$\hat{y}_1 = l_1.$$

This l_1 is our first learner and $\hat{y}_1$ denotes its predictions (this equation is in matrix form). Then, we can calculate the residuals of our first learner.

$$\epsilon_1 = y - v_1 \times \hat{y}_1,$$

where v_1 is a shrinkage parameter to avoid overfitting. Next, we fit the residual with another learner. This learner minimizes the following function $\sum_{i=1}^{N} \|y_i - L_{k-1} - l_k\|$, here `k=2`. Then we obtain a second model l_2 with:

$$\hat{y}_2 = l_2.$$

After that, we can update the residuals:

$$\epsilon_2 = \epsilon_1 - v_2 \times \hat{y}_2.$$

We repeat this residual fitting until adding another learner l_k results in an updated residual ϵ_k that is smaller than a small predefined threshold. At the end, we will have an additive model like:

$$L = v_1 \times l_1 + v_2 \times l_2 + \ldots + v_k \times l_K,$$

where we have k weak learners, but a very strong ensemble model.

Schapire and Freund found that although individual learners trained on the pilot observations might be very weak in predicting in isolation, boosting the collective power of all of them is expected to generate a model `no worse than the best of all individual constituent models` included in the boosting ensemble. Usually, the boosting results are quite a bit better than the best single model.

Boosting can be used for almost all models. Most commonly, it is applied to decision trees.

15.2.5 Random Forests

Random forests, or decision tree forests, represent a boosting method focusing on decision tree learners.

Training Random Forests

One approach to train and build random forests relies on using `randomForest()` under the `randomForest` package. It has the following components:

```
m<-randomForest(expression, data, ntree=500, mtry=sqrt(p))
```

- *expression*: the class variable and features we want to include in the model.
- *data*: training data containing class and features.
- *ntree*: number of voting decision trees.
- *mtry*: optional integer specifying the number of features to randomly select at each split. The p stands for number of features in the data.

Let's build a random forest using the Quality of Life dataset.

```
# install.packages("randomForest")
library(randomForest)

set.seed(123)
rf<-randomForest(CHARLSONSCORE~., data=qol)
rf

##
## Call:
##  randomForest(formula = CHARLSONSCORE ~ ., data = qol)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##         OOB estimate of  error rate: 46.13%
## Confusion matrix:
##      0   1 2 3 4 5 6 7 8 9 10 class.error
## 0  574 301 2 0 0 0 0 0 0 0  0   0.3454960
## 1  305 678 1 0 0 0 0 0 0 0  0   0.3109756
## 2   90 185 2 0 0 0 0 0 0 0  0   0.9927798
## 3   25 101 1 0 0 0 0 0 0 0  0   1.0000000
## 4    5  19 0 0 0 0 0 0 0 0  0   1.0000000
## 5    3   4 0 0 0 0 0 0 0 0  0   1.0000000
## 6    1   4 0 0 0 0 0 0 0 0  0   1.0000000
## 7    1   1 0 0 0 0 0 0 0 0  0   1.0000000
## 8    7   8 0 0 0 0 0 0 0 0  0   1.0000000
## 9    3   5 0 0 0 0 0 0 0 0  0   1.0000000
## 10   1   1 0 0 0 0 0 0 0 0  0   1.0000000
```

By default the model contains 500 decision trees and tried 6 variables at each split. Its OOB, or out-of-bag, error rate is about 46%, which corresponds to a poor accuracy rate (54%). Note that the OOB error rate is not re-substitution error. The confusion matrix next to it is reflecting OOB error rate for specific classes. All of these error rates are reasonable estimates of future performances with unseen data. We can see that this model is so far the best of all models, although it is still not good at predicting high CHARLSONSCORE.

Evaluating Random Forest Performance

The caret package also supports random forest model building and evaluation. It reports more detailed model performance evaluations. As usual, we need to specify a re-sampling method and a parameter grid. As an example, we use the 10-fold CV

re-sampling method. The grid for this model contains information about the `mtry` parameter (the only tuning parameter for random forest). Previously we tried the default value $\sqrt{38} = 6$ (38 is the number of features). This time we could compare multiple `mtry` parameters.

```
library(caret)
ctrl<-trainControl(method="cv", number=10)
grid_rf<-expand.grid(.mtry=c(2, 4, 8, 16))
```

Next, we apply the `train()` function with our `ctrl` and `grid_rf` settings.

```
set.seed(123)
m_rf <- train(CHARLSONSCORE ~ ., data = qol, method = "rf",
metric = "Kappa", trControl = ctrl,
tuneGrid = grid_rf)
m_rf
```

```
## Random Forest
##
## 2328 samples
##   38 predictor
##   11 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2095, 2096, 2093, 2094, 2098, 2097, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.5223871  0.1979731
##    4    0.5403799  0.2309963
##    8    0.5382674  0.2287595
##   16    0.5421562  0.2367477
##
## Kappa was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 16.
```

This call may take a while to complete. The result appears to be a good model, when `mtry=16` we reached a relatively high accuracy and good `kappa` statistic. This is a very good result for a learner with 11 classes.

15.2.6 Adaptive Boosting

We may achieve even higher accuracy using **AdaBoost**. Adaptive boosting (AdaBoost) can be used in conjunction with many other types of learning algorithms to improve their performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the

boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by the previous classifiers.

For binary classification, we could use `ada()` in package `ada` and for multiple classes (multinomial/polytomous outcomes) we can use the package `adabag`. The `boosting()` function allows us to select a type method by setting `coeflearn`. Two prevalent types of adaptive boosting methods can be used. One is AdaBoost. M1 algorithm including `Breiman` and `Freund`, and the other is `Zhu`'s `SAMME` algorithm. Let's see some examples:

```
set.seed(123)
qol<-read.csv("https://umich.instructure.com/files/481332/download?download_
frd=1")
qol<-qol[!qol$CHARLSONSCORE==-9 , -c(1, 2)]
qol$CHARLSONSCORE<-as.factor(qol$CHARLSONSCORE)
```

The key parameter in the `adabag::boosting()` method is *coeflearn*:

- *Breiman* (default), corresponding to $\alpha = \frac{1}{2} \times \ln\left(\frac{1-err}{err}\right)$, using the AdaBoost.M1 algorithm, where α is the weight updating coefficient
- *Freund*, corresponding to $\alpha = \ln\left(\frac{1-err}{err}\right)$, or
- *Zhu*, corresponding to $\alpha = \ln\left(\frac{1-err}{err}\right) + \ln\left(nclasses - 1\right)$.

The generalizations of AdaBoost for multiple classes ($\geq$2) include `AdaBoost.M1` (where individual trees are required to have an error $< \frac{1}{2}$) and `SAMME` (where individual trees are required to have an error $< 1 - \frac{1}{nclasses}$).

```
# install.packages("ada"); install.packages("adabag")
library("ada"); library("adabag")

qol_boost <- boosting(CHARLSONSCORE~.,data=qol,mfinal = 100,coeflearn =
'Breiman')
mean(qol_boost$class==qol$CHARLSONSCORE)

## [1] 0.5425258

qol_boost <- boosting(CHARLSONSCORE~.,data=qol,mfinal = 100,coeflearn =
'Freund')
mean(qol_boost$class==qol$CHARLSONSCORE)

## [1] 0.5524055

qol_boost <- boosting(CHARLSONSCORE~.,data=qol, mfinal = 100, coeflearn =
'Zhu')
mean(qol_boost$class==qol$CHARLSONSCORE)

## [1] 0.6542096
```

We observe that in this case, the `Zhu` approach achieves the best results. Notice that the default method is M1 `Breiman` and `mfinal` is the number of iterations for which boosting is run or the number of trees to use.

Try applying model improvement techniques using other data from the list of our Case-Studies (Fig. 15.1).

15.3 Assignment: 15. Improving Model Performance

Use some of the methods below to do classification, prediction, and model performance evaluation (Table 15.3).

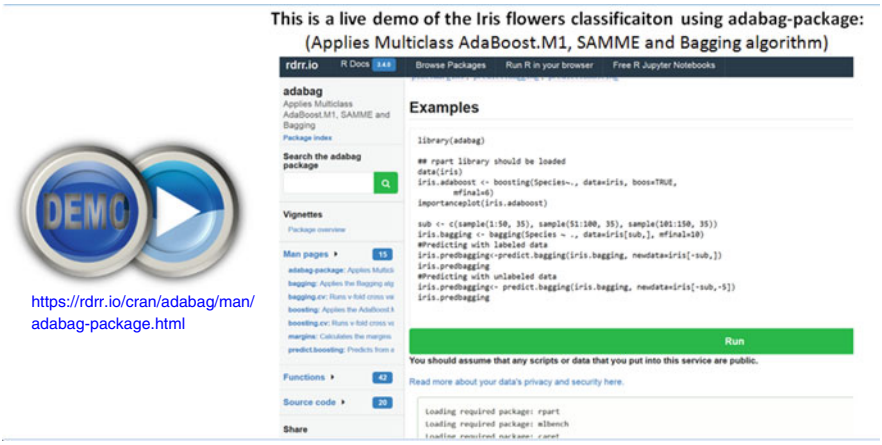

Fig. 15.1 Live demo: Iris flowers classification using adabag

Table 15.3 Performance evaluation for several classification, prediction, and clustering methods

Model	Learning task	Method	Parameters
KNN	Classification	`knn`	`k`
Naïve Bayes	Classification	`nb`	`fL, usekernel`
Decision Trees	Classification	`C5.0`	`model, trials, winnow`
OneR Rule Learner	Classification	`OneR`	None
RIPPER Rule Learner	Classification	`JRip`	`NumOpt`
Linear Regression	Regression	`lm`	None
Regression Trees	Regression	`rpart`	`cp`
Model Trees	Regression	`M5`	`pruned, smoothed, rules`
Neural Networks	Dual use	`nnet`	`size, decay`
Support Vector Machines (Linear Kernel)	Dual use	`svmLinear`	`C`
Support Vector Machines (Radial Basis Kernel)	Dual use	`svmRadial`	`C, sigma`
Random Forests	Dual use	`rf`	`mtry`

15.3.1 Model Improvement Case Study

From the course datasets, use the 05_PPMI_top_UPDRS_Integrated_LongFormat1.csv case-study data to perform a multi-class prediction.

Use `ResearchGroup` as response, which have "PD", "Control" and "SWEDD" three classes.

- Delete ID column, impute missing value with mean or median and justify your choice.
- Normalize the covariates.
- Implement automated parameter tuning process and report the optimal accuracy and κ.
- Set arguments and rerun the tuning, trying different `method` and `number` settings.
- Train a random forest, tune the parameters, report the result and output cross table.
- Use bagging algorithm and report the accuracy and κ.
- Perform randomForest and report the accuracy and κ.
- Report the accuracy by AdaBoost and make sure to try all three methods.
- Finally, give a brief summary about all the model improvement approaches.
- Try the procedure on other data in the list of Case-Studies, e.g., Traumatic Brain Injury Study and the corresponding dataset.

References

Zhu, J, Zou, H, Rosset, S, Hastie, T. (2009) *Multi-class AdaBoost*, Statistics and Its Interface, 2, 349–360.

Breiman, L. (1998): *Arcing classifiers*, The Annals of Statistics, 26(3), 801–849.

Freund, Y, Schapire, RE. (1996) *Experiments with a new boosting algorithm*, In Proceedings of the Thirteenth International Conference on Machine Learning, 148–156, Morgan Kaufmann