Lab

Some of the ideas for this lab appear at:

Ref: http://machinelearningmastery.com/machine-learning-ensembles-with-r/ [accessed 05/11/2024]

Aim

To learn about ensemble prediction: boosting and bagging and stacking.

Before you start

Load the following packages (download them if needed):

- mlbench
- caret
- caretEnsemble
- gbm
- adabag (optional)

You might get a warning about gbm. Ignore it for now.

The library adabag is needed if you want to train the Adaboost.M1 model, but it takes quite a while to train and so is optional for this lab.

The Ionosphere dataset

For this lab we will use the Ionosphere dataset. It contains 351 instances of radar data described using 34 numeric values plus the class, which is nominal with values *good* and *bad*. The second variable, *V2*, is always 0 and, therefore, provides no information.

16 high-frequency antennas tried to sense free electrons in the ionosphere. 17 pulse numbers were used, and data was described by 2 attributes per pulse number. The class is *good* for those radar returns showing some type of structure in the ionosphere. The class is *bad* for those radar returns which did not show any structures, i.e. the signals passed through.

Load and inspect this dataset.

```
data(Ionosphere)
```

```
view(Ionosphere)
```

Make a copy as you will be removing/transforming some attributes

```
ionos <- Ionosphere
```

Remove the second attribute as it is always zero.

```
ionos <- ionos[,-2]
```

The first variable can take the value of zero or one. To ensure it is taken as a numeric value

```
ionos$V1 <- as.numeric(as.character(ionos$V1))
```

The class should be a factor.

```
ionos$Class <- factor(ionos$Class)
```

Inspect the data

```
View(ionos)
```

Boosting

First, we set the evaluation method. We would want 3 repeats of 10-fold cross validation but because the algorithms take a long time to run, for this lab we will use 1 repeat. Change the number of repeats to a higher figure (e.g. 3) for "real" experiments.

```
control <- trainControl(method="repeatedcv", number=10, repeats=1)
```

Compare the performance of C5.0 with and without boosting. To get boosted C5.0, set the trials (number of boosting iterations) to a number higher than 1, e.g. 9.

```
# no boosting
tunec50Grid <- expand.grid(trials = 1,
                        model = "tree",
                        winnow = FALSE)


# boosting
tunec50Boost <- expand.grid(trials = 9,
                        model = "tree",
                        winnow = FALSE)
C5.0.mod <- train(Class~., data=ionos, method="C5.0", metric="Accuracy",
trControl=control, tuneGrid=tunec50Grid)


C5.0.modBoost <- train(Class~., data=ionos, method="C5.0",
metric="Accuracy", trControl=control, tuneGrid=tunec50Boost)
```

Has boosting improved the accuracy?

Exercise

1. Compare the performance of C5.0 with and without boosting. Comment on whether any difference in results is **statistically significant.**
2. *Warning: This exercise includes algorithm AdaBoost.M1, which takes a really long time to run – more than 5 minutes but less than 15. You may omit this algorithm from this exercise.*
   Creating several models, one for C5.0, one for CART (i.e. rpart) and the rest for the following boosted algorithms: AdaBoost, Extreme Gradient Boosting and Stochastic Gradient Boosting (gbm). Note that C5.0 does include boosting. I.e. create a model for each of the following methods:
   - `"C5.0"`
   - `"rpart"`
   - `"Adaboost,M1"`
   - `"xgbTree"`
   - `"gbm"`

   Run each algorithm in its own .rmd code cell. Compare the results. Which algorithm works best? Are the differences in results statistically significant?

Variable importance with *varImp*

The `varImp()` function within caret can be used to ascertain the importance of variables in models obtained using *train*.

A table of variables in descending order of importance for each algorithm can be produced as follows (note that the adaBoost one is commented out as you may not have run it and that **you will need to replace the names of the models with the ones you have used yourself**):

```
C5.0 <- row.names(varImp(C5.0.mod)$importance)

CART <- row.names(varImp(rpart.mod)$importance)

# adaBoost <- row.names(varImp(adabo.mod)$importance)

xgb <- row.names(varImp(xgb.mod)$importance)

gbm <- row.names(varImp(gbm.mod)$importance)

imp <- data.frame(C5.0, CART, adaBoost, xgb, gbm)

knitr::kable(imp[1:10,])
```

It can be seen that most algorithms have V1 as their most or 2nd most important attribute but the xgb model does not use it in its 1st 10. V3 and V5 are used in all models' top 10 features (except Adaboost.M1).

The most important variables for each model can also be plotted, to get an idea of their contribution to the model.

```
# plot 10 most important variables in the model.

plot(varImp(C5.0.mod), top=10, main="Variable importance - C5.0",
xlab="importance", ylab= "Variable"))
```

Note that `main` refers to the plot title, and `xlab` and `ylab` to the x and y axis labels.

Exercise

3. Check the most important attributes for all the models used above.

Bagging

We will build several models, one with CART (rpart), and the rest with the following bagged models: bagged Ada, bagged CART and random forest. For example

```
# CART

set.seed(123)

rpart.mod <- train(Class~., data=ionos, method="rpart", metric="Accuracy",
trControl=control)
```

Exercise

4. Build models for the rest of the algorithms. The methods are `"AdaBag"` (optional, see below), `"treebag"` and `"rf"`. Compare the results obtained. Which algorithm offers the best accuracy?
   Note: Bagged Ada (or AdaBag) takes a few minutes (3 minutes) to complete its execution so you may want to skip this algorithm.

Random forest options

Random forest has various options which can be tuned. One of them is to set the number for `mtry`, which contains the number of attributes to be considered at each split. In the lecture, we saw that it is recommended to consider a number which is the square root of the number of columns. You can do this as follows:

```
set.seed(123)
mtry <- sqrt(ncol(ionos))
tunegrid <- expand.grid(.mtry=mtry)
rf.mod2 <- train(Class~., data=ionos,
                 method="rf", metric="Accuracy", trControl=control,
                 tuneGrid=tunegrid, na.action=na.omit)
confusionMatrix(rf.mod2)
```

You could also have several different values in the `tuneGrid`. For example, to have the default value for `mtry` (i.e. 5) plus another 2 values, one which is the default plus 2 (i.e. 7) and one which is the default value plus 4 (i.e. 9), replace the 2nd and 3rd lines in the code above by the following:

```
mtry <- floor(sqrt(ncol(ionos)))
mtrylist <- c(mtry, mtry+2, mtry+4)
tunegrid <- expand.grid(.mtry=mtrylist)
```

Note that although random forest is quick, running it a few times in succession with tunegrid might make it observably slow.

Alternatively, a number of random tries could be set using `tuneLength`. This means that the algorithm will be run `tuneLength` times, each one with a different value of `mtry` chosen at random. For example, try 12 different values:

```
set.seed(123)
rf.mod3 <- train(Class~., data=ionos,
            method="rf", metric="Accuracy", trControl=control,
            tuneLength=12, na.action=na.omit)
```

The algorithm will report which mtry gave the best results from the random trial.

Stacking

The caretEnsemble package offers functions to combine the predictions of several models using the `caretStack()` function.

First we need to construct a set of base models to stack. We use the following base algorithms: Linear Discriminate Analysis (LDA), CART (rpart) and k-nearest Neighbours (kNN).

The caretEnsemble package offers function `caretList()` to create a list of models. These are the base models which will then be used by a meta-model.

Before we apply `caretList()` we want to ensure all algorithms use the same data partitions. As we are using 3 repeats of 10-fold cross validation, we can use function `createMultiFolds()` to create all the various data partitions needed. They can then be included in the `trainControl()` using the `index` option.

First, we create the data partitions.

```
# creating partitions

set.seed(123)
all3repeats <- createMultiFolds(ionos$Class, 3, 10)
#using the partitions by including them using the index parameter
control <- trainControl(method ="repeatedcv", number=10, repeats=3,
              savePredictions='final', index= all3repeats,
              classProbs=TRUE)
```

Then we define the list of base algorithms.

```
baseAlgo <- c('lda', 'rpart', 'knn', 'C5.0Tree', 'C5.0Rules')
```

Finally, we apply each of the algorithms to the dataset, using the partitions defined earlier in the train control.

```
baseModels <- caretList(Class~., data=ionos, trControl=control,
                  methodList=baseAlgo)
```

We can now check the results.

```
results <- resamples(baseModels)

summary(results)

dotplot(results)
```

C5.0Tree and C5.0Rules obtain the highest accuracy.

It is important that models are not very correlated, i.e. each model is an expert on something different (correlation < 0.75). Therefore, we check their correlation.

```
modelCor(results)

splom(results)
```

Correlations are not low for all models. C5.0Tree and C5.0Rules are highly correlated.

Exercise

5. Redefine baseModels so that C5.0Tree is not in the list.

Stacking the base models with Random Forest

To stack using Random Forest, as the meta-model:

```
stackControl <- trainControl(method="repeatedcv", number=10, repeats=3,
savePredictions='final', classProbs=TRUE)

set.seed(123)

ensembleStack <- caretStack(baseModels, method="rf", metric="Accuracy",
trControl=stackControl)

print(ensembleStack)
```

Has the accuracy improved?

Exercises

6. Change the meta-learner algorithm for the stacking exercise and compare the results you get with those obtained using GLM as the meta-learner. For example use C5.0.
7. Choose one of the datasets which you have seen in previous labs. Experiment with boosting, bagging and stacking.