Chapter 9

Random forests, boosting

9.1 Lab: Random forest, AdaBoost

Random forests are grown using library called `randomForest`. It is very simple to use, as you will see shortly.

First, load the `breastcancer.csv` data set. It is the full dataset used several times in the theory lectures. The task is to predict whether patient has benign or malignant type of growth. The tissue samples are taken for tests and various aspects of the cells are calculated: cell radius, texturedness and so on. Based of these measurements, a classifier has to decide to which class the patient belongs.

First, divide the dataset into train and test subsets:

```
idx.train = createDataPartition(y = data$diagnosis, p = 0.8,
list = FALSE)
train = data[idx.train, ]
test = data[-idx.train, ]
```

Training random forest is simple, you only need to select how many trees to grow. It is best to grow large forests, because increasing forest complexity does not result to the increase in test error:

```
library(randomForest)
rf=randomForest(diagnosis .,data=train,ntree=1000,importance=T)
```

A rule of thumb is to grow such a number of trees that the Out-Of-Bag (OOB) error no longer varies significantly. You can plot OOB error (overall and for each class separately) by simply using `plot` function:

```
plot(rf)
```

We can see that when the number of trees on the forest is not very large, the variance of OOB error is quite large. After reaching a particular number of trees it settles down and does not change a lot when adding more and more trees. It is important also to observe that further adding trees in the model does not increase the OOB error - increasing the complexity the random forest does not increase its generalization error.

Random forests produce a ranking of variables by their importance. Two options are possible: importance based on Gini impurity decrease and permutation importance, which is based on the decrease in accuracy if the values of the feature are shuffled randomly (thus breaking any ties to the class). Importances can be plotted as follows:

```
varImpPlot(rf)
```

Two plots of both importances are produced. Both importance measures produce approximately the same ranking of the variables: features based on worst and mean values of area, concavity, perimeter and radius are most important in classifying whether the patient has benign or malignant case.

Obtaining predictions is just as easy as in previous classifiers:

```
pred=predict(rf,test)
mean(pred==test$diagnosis)*100
confMat=table(true_lab=test$diagnosis,predicted=pred)
diag(confMat)/rowSums(confMat)
```

There are several options of library for AdaBoost algorithm: `adabag` and `gbm` for example. However, they have some major disadvantage, the first one is very slow, while the second one is for the binary classification problems. We will use `xgboost` library which provides a much more wider class of boosted algorithms, but the implementation of boosting is done in a different, more efficient manner that we spoke in the theory lecture. But the results are almost identical and therefore this library will be used further. It is a very complex library (similar to the `h2o` library for neural nets).

When using AdaBoost, or boosted trees, as these terms are almost synonymous, we have several choices: the number of boosting iterations (a.k.a. number of weak classifiers in the ensemble) and the type of weak classifiers. In the original AdaBoost method, the stumps (one split tree) were used as weak classifiers. We will consider at least several different sizes of decision trees as weak classifier candidates.

`xgboost` library, contrary to what you encountered in the previouse classifiers and corresponding libraries, it requires the class labels to be integer numbers, starting from 0. For example, if we have 3 class problem, then the class variable must be converted to values 0, 1 and 2. Such conversion can be carried out by the following lines of code:

```
train_label=as.numeric(train$diagnosis)-1
test_label=as.numeric(test$diagnosis)-1
```

Function `as.numeric` converts factors into numbers. Factors "B" and "M" are converted to 1 and 2. Thus we need to subtract 1 so that all class numbers starts from 0. In addition to this modification, `xgboost` requires that the data frame of features be converted to the matrix object. The boosting is performed by the following command:

```
library(xgboost)
bst = xgboost(data = as.matrix(train[,2:31]),
                    label = train_label,
                    num_class=2,
                    max_depth = 1,
                    nrounds = 100,
                    objective = "multi:softmax")
```

The option `max_depth` instructs to use the stump as a weak classifier. Increasing this values means that you are allowing to use decision trees that have more and more levels (splits). Value "multi:softmax" of `objective` option should be always used and not changed - it simply instructs to used multi-class classification strategy. Also, you always have to provide the number of classes with

parameter `num_class` - which is a bit strange, because inferring the number of classes from labels is trivial.

As before, we should always test our classifier:

```
pred=predict(bst,as.matrix(test[,2:31]))
mean(pred==test_label)*100
confMat=table(true_lab=test_label,predicted=pred)
diag(confMat)/rowSums(confMat)
```

You should check by yourself that in this case, increasing the size of ensemble does not cause overfitting. Also, increasing the size of a weak learner damages the accuracy.

9.2 Exercises with R: Smartphone-Based Recognition of Human Activities and Postural Transitions

The experiments were carried out with a group of 30 volunteers within an age bracket of 19-48 years. They performed a protocol of activities composed of six basic activities: three static postures (standing, sitting, lying) and three dynamic activities (walking, walking downstairs and walking upstairs). The experiment also included postural transitions that occurred between the static postures. These are: stand-to-sit, sit-to-stand, sit-to-lie, lie-to-sit, stand-to-lie, and lie-to-stand. All the participants were wearing a smartphone (Samsung Galaxy S II) on the waist during the experiment execution. We captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz using the embedded accelerometer and gyroscope of the device. The experiments were video-recorded to label the data manually. The obtained dataset was randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of 561 features was obtained by calculating variables from the time and frequency domain.

Note: Do not worry if you don't understand what Butterworth filter is. Just know that various signals are often prefiltered from noise using various mathematical models called filters. Also, frequency domain is another possible unheard off notion - to simply put it, it is an analysis of signals in terms of frequencies (for this often the Fourier transformation is performed on the signal).

More information about the features calculated for this dataset is provided in a separate file "features_info.txt".

In this problem, there are 12 different classes in total:

1. WALKING
2. WALKING_UPSTAIRS
3. WALKING_DOWNSTAIRS
4. SITTING
5. STANDING
6. LAYING
7. STAND_TO_SIT
8. SIT_TO_STAND
9. SIT_TO_LIE
10. LIE_TO_SIT
11. STAND_TO_LIE
12. LIE_TO_STAND

1. Explore the dataset: class distribution, at least 3 different plots with different pairs of variables coloured by the class. Provide a discussion regarding the amount of variables, class balance, class separability.

2. Apply random forest method: use at least 5 different `mtry` values as well as different numbers of trees in the forest. Report accuracies and discuss effect of `mtry` and `ntree` on the method performance. Accuracy must be investigated with respect overall accuracy as well as each class accuracies. Parameters `mtry` and `ntree` must be investigated in a wide range of values.

3. Investigate variable importance. Are there features which are unimportant (based on importance analysis)? Drop those features from the dataset and perform analysis which would answer the question whether variable importance analysis can be used as a kind of variable selection procedure (i.e. check whether dropping unimportant features does not deteriorate the overall performance).

4. Apply boosting method: explore the effect of number of weak learners on the method performance as well as the influence of the complexity of the weak learner (i.e. the effect of `max_depth`). An in-depth analysis with a boosting method must be performed as well. You must eventually answer questions like: does increasing complexity of a weak learner deteriorates the performance; is the boosting method robust against over-fitting. What about number of features reduction? Does it improve the performance of the algorithm?