Random Forest

Random forests use bootstrap aggregating to reduce the variance of the outcomes. In the first step, bootstrapping (sampling with replacement) is used to create `B` training sets from the population with the same size as the original training set. Hereafter, a separate tree for each of these training sets is built. Trees are grown using recursive binary splitting on the training data until a node reaches some minimum number of observations. The idea is that the tree should go from impure (equal mixing of classes) to pure (each leaf corresponds to one class exactly). The splits are determined such that they decrease variance, error and impurity. Random forests decorrelate the trees by considering only `m` of all `p` predictors as split candidates, whereby often `m = sqrt(p)`.

Classification trees predict that each observation belongs to the most commonly occurring class (i.e. majority vote) of training observations in the region to which it belongs. The classification error rate is the fraction of the number of misclassified observations and the total number of classified observations. The Gini index and cross-entropy measures determine the level of impurity in order to decide on the best split at each node. In the final step, the average of the classification prediction results of all `B` trees is computed from the majority vote. The accuracy is computed as the out-of-bag (OOB) error and/or the test set error.

As each bootstrap samples from the training set with replacement, about $^2/_3$ of the observations are not sampled and some are sampled multiple times. In the case of `B` trees in the forest, each observation is left out of approximately `B`/ 3 trees. The non-sampled observations are used as test set and the `B`/ 3 trees are used for out-of-sample predictions. In random forests, pruning is not needed as potential over-fitting is (partially) mitigated by the usage of bootstrapped samples and multiple decorrelated random trees.

We start by tuning the number of variables that are randomly sampled as candidates at each split,`mtry`. We make use of the `caret` framework, which makes it easy to train and evaluate a large number of different types of models. For random forests, we have the `repeatedcv` method perform five-fold cross-validation with five repetitions. For now, we build a random forest containing 200 trees because previous analyses with these data showed that the error does not decrease substantially when the number of trees is larger than 200, while a larger number of trees does require more computational power. We will see later on that 200 trees is indeed sufficient for this analysis. We let the algorithm determine what the best model is based on the accuracy metric, and we ask the algorithm to run the model for `pca.dims` (= 17) different values of `mtry`. We first specify the controls in `rf_rand_control`: we perform 5-fold cross-validation with 5 repeats (`method = "cv"`, `number = 5` and `repeats = 5`), allow parallel computation (`allowParallel = TRUE`) and save the predicted values (`savePredictions = TRUE`).

```
library(caret)
rf_rand_control = trainControl(method = "repeatedcv",
                               search = "random",
                               number = 5,
                               repeats = 5,
                               allowParallel = TRUE,
                               savePredictions = TRUE)
set.seed(1234)
rf_rand = train(x = train.images.pca,
                y = train.data.pca$label,
                method = "rf",
                ntree = 200,
                metric = "Accuracy",
                trControl = rf_rand_control,
                tuneLength = pca.dims)

print(rf_rand)
```

```
Random Forest

60000 samples
   17 predictor
   10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 5 times)
Summary of sample sizes: 48000, 48000, 48000, 48000, 48000, 48000, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
   1    0.8474167  0.8304630
   4    0.8535733  0.8373037
   5    0.8535600  0.8372889
   9    0.8528367  0.8364852
  10    0.8529333  0.8365926
  11    0.8526467  0.8362741
  12    0.8523000  0.8358889
  15    0.8495867  0.8328741
  16    0.8481267  0.8312519

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 4.
```

We can check the model performance on both the training and test sets by means of different metrics using a custom function, `model_performance`, which can be found on my Github.

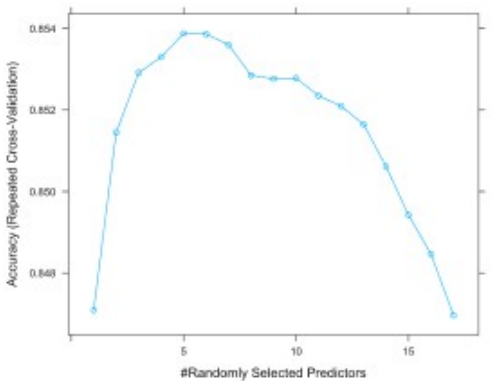
```
mp.rf.rand = model_performance(rf_rand, train.images.pca, test.images.pca,
                               train.data.pca$label, test.data.pca$label,
"random_forest_random")
```

```
accuracy_train precision_train recall_train  F1_train accuracy_test precision_test recall_test  F1_test            model
     0.8535755       0.8811304       0.841 0.8285812        0.8474       0.8809662       0.829 0.814742 random_forest_random
```

We can also use the `caret` framework to perform a grid search with pre-specified values for `mtry` rather than a random search as above.

```
rf_grid_control = trainControl(method = "repeatedcv",
                               search = "grid",
                               number = 5,
                               repeats = 5,
                               allowParallel = TRUE,
                               savePredictions = TRUE)
set.seed(1234)
rf_grid = train(x = train.images.pca,
                y = train.data.pca$label,
                method = "rf",
                ntree = 200,
                metric = "Accuracy",
                trControl = rf_grid_control,
                tuneGrid = expand.grid(.mtry = c(1:pca.dims)))

plot(rf_grid)
```




```
mp.rf.grid = model_performance(rf_grid, train.images.pca, test.images.pca,
                               train.data.pca$label, test.data.pca$label,
"random_forest_grid")
```

```
accuracy_train precision_train recall_train  F1_train accuracy_test precision_test recall_test  F1_test           model
     0.8538767       0.88164    0.8407333 0.8207214        0.8484        0.88077       0.832 0.8160863 random_forest_grid
```

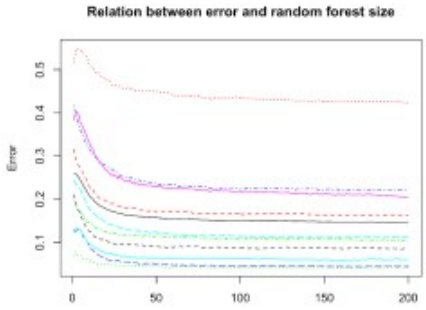
As shown by the results, the random search selects `mtry=4` as the optimal parameter, resulting in 85% training and test set accuracies. The grid search selects `mtry=5` and achieves similar accuracies for both values of 4 and 5 for `mtry`. We can see from the results that according to `rf_rand`, `mtry` values of 4 and 5 lead to very similar results,

which also goes for `mtry` values of 5 and 6 for `rf_grid`. Although the results of `rf_rand` and `rf_grid` are very similar, we choose the best model on the basis of accuracy and save this in `rf_best`. For this model, we'll look at the relationship between the error and random forest size as well as the receiver operating characteristic (ROC) curves for every class. Let's start by subtracting the best performing model from `rf_rand` and `rf_grid`.

```
rf_models = list(rf_rand$finalModel, rf_grid$finalModel)
rf_accs = unlist(lapply(rf_models, function(x){ sum(diag(x$confusion)) /
sum(x$confusion) }))
rf_best = rf_models[[which.max(rf_accs)]]
```

Next, we plot the relationship between the size of the random forest and the error using the `plot()` function from the `randomForest` package.

```
library(randomForest)
plot(rf_best, main = "Relation between error and random forest size")
```



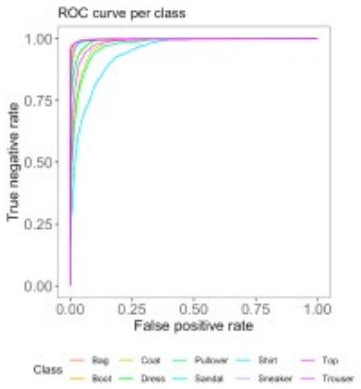
We observe from this plot that the error does not decrease anymore for any of the classes after about 100 trees, and so we can conclude that our forest size of 200 is sufficient. We can also use the `varImpPlot()` function from the `randomForest` package to plot the importance for each variable. I will not show that here because it's not as meaningful given that our variables are principal components of the actual pixels, but it's good to keep in mind when extending these analyses to other data.

Finally, we plot the ROC curves for every class. On the x-axis of an ROC plot, we usually have the false positive rate (false positive / (true negative + false positive)) and on the y-axis the true positive rate (true positive / (true positive + false negative)). Essentially, the ROC plot helps us to compare the performance of our model with respect to predicting different classes. The area underneath each curve is the proportion of correct classifications for that particular class. Therefore, the further the curve is "drawn" towards the top left from the 45 degrees line, the better the classification for that class. We first need to obtain the data for the ROC curve for every class (or clothing category) in our data, which we bind together by rows, including a label for the classes.

```
library(ROCR)
library(plyr)
pred_roc = predict(rf_best, test.images.pca, type = "prob")
classes = unique(test.data.pca$label)
classes = classes[order(classes)]
plot_list = list()
for (i in 1:length(classes)) {
  actual = ifelse(test.data.pca$label == classes[i], 1, 0)
  pred = prediction(pred_roc[, i], actual)
  perf = performance(pred, "tpr", "fpr")
  plot_list[[i]] = data.frame(matrix(NA, nrow = length(perf@x.values[[1]]), ncol = 2))
  plot_list[[i]]['x'] = perf@x.values[[1]]
  plot_list[[i]]['y'] = perf@y.values[[1]]
}
plotdf = rbind.fill(plot_list)
plotdf["Class"] = rep(cloth_cats, unlist(lapply(plot_list, nrow)))
```

Next, we plot the ROC curves for every class. Note that we use the custom plotting theme `my_theme()` as defined in the the second blog post of this series.

```
ggplot() +
  geom_line(data = plotdf, aes(x = x, y = y, color = Class)) +
  labs(x = "False positive rate", y = "True negative rate", color = "Class") +
  ggtitle("ROC curve per class") +
  theme(legend.position = c(0.85, 0.35)) +
  coord_fixed() +
  my_theme()
```



We observe from the ROC curves that shirts and pullovers are most often misclassified, whereas trousers, bags, boots and sneakers are most often correctly classified. A possible explanation for this could be that shirts and pullovers can be very similar in shape to other categories, such as tops, coats and dresses; whereas bags, trousers, boots and sneakers are more dissimilar to other categories in the data.

Gradient-Boosted Trees

While in random forests each tree is fully grown and trained independently with a random sample of data, in boosting every newly built tree incorporates the error from the previously built tree. That is, the trees are grown sequentially on an adapted version of the initial data, which does not require bootstrap sampling. Because of this, boosted trees are usually smaller and more shallow than the trees in random forests, improving the tree where it does not work well enough yet. Boosting is often said to outperform random forests, which is mainly because the approach learns slowly. This learning rate can be controlled by the shrinkage parameter, which we'll tune later.

In boosting, it's important to tune the parameters well and play around with different values of the parameters, which can easily be done using the `caret` framework. These parameters include the learning rate, `eta`, the minimal required loss reduction to further partition on a leaf node of the tree, `gamma`, the maximal depth of a tree `max_depth`, the number of trees in the forest, `nrounds`, the minimum number of observations in the trees' nodes, `min_child_weight`, the fraction of the training set observations randomly selected to grow trees, `subsample`, and the proportion of independent variables to use for each tree, `colsample_bytree`. An overview of all parameters can be found here. Again, we use the `caret` framework to tune our boosting model.

```
xgb_control = trainControl(
  method = "cv",
  number = 5,
  classProbs = TRUE,
  allowParallel = TRUE,
  savePredictions = TRUE
)
```

Next, we define the possible combinations of the tuning parameters in the form of a grid, named `xgb_grid`.

```
xgb_grid = expand.grid(
  nrounds = c(50, 100),
  max_depth = seq(5, 15, 5),
  eta = c(0.002, 0.02, 0.2),
  gamma = c(0.1, 0.5, 1.0),
  colsample_bytree = 1,
  min_child_weight = c(1, 2, 3),
```

```
  subsample = c(0.5, 0.75, 1)
)
```

We set the seed and then train the model onto the transformed principal components of the training data using `xgb_control` and `xgb_grid` as specified earlier. Note that because of the relatively large number of tuning parameters, and thus the larger number of possible combinations of these parameters (`nrow(xgb_grid) = 486`), this may take quite a long time to run.

```
set.seed(1234)
xgb_tune = train(x = train.images.pca,
                 y = train.classes,
                 method = "xgbTree",
                 trControl = xgb_control,
                 tuneGrid = xgb_grid
)
xgb_tune
```

(Note that the output of xgb_tune has been truncated for this post.)

```
eXtreme Gradient Boosting

60000 samples
   17 predictor
   10 classes: 'Bag', 'Boot', 'Coat', 'Dress', 'Pullover', 'Sandal', 'Shirt', 'Sneaker', 'Top', 'Trouser'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 48000, 48000, 48000, 48000, 48000
Resampling results across tuning parameters:

  eta    max_depth  gamma  min_child_weight  subsample  nrounds  Accuracy   Kappa
  0.002  5          0.1    1                 0.50        50      0.7004000  0.7437778
  0.002  5          0.1    1                 0.50       100      0.7722667  0.7469630
  0.002  5          0.1    1                 0.75        50      0.7656833  0.7396481
  0.002  5          0.1    1                 0.75       100      0.7686833  0.7429815
  0.002  5          0.1    1                 1.00        50      0.7581000  0.7312222
  0.002  5          0.1    1                 1.00       100      0.7641500  0.7379444
  0.002  5          0.1    2                 0.50        50      0.7698833  0.7443148
  0.002  5          0.1    2                 0.50       100      0.7724500  0.7471667
  0.002  5          0.1    2                 0.75        50      0.7651167  0.7390185
  0.002  5          0.1    2                 0.75       100      0.7684667  0.7427407
  0.002  5          0.1    2                 1.00        50      0.7580500  0.7311667
  0.002  5          0.1    2                 1.00       100      0.7641000  0.7378889
  0.002  5          0.1    3                 0.50        50      0.7694667  0.7438519
  0.002  5          0.1    3                 0.50       100      0.7719500  0.7466111
  0.002  5          0.1    3                 0.75        50      0.7659833  0.7399815
  0.002  5          0.1    3                 0.75       100      0.7682833  0.7425370
  0.002  5          0.1    3                 1.00        50      0.7578333  0.7309259
  0.002  5          0.1    3                 1.00       100      0.7639333  0.7377037
  0.002  5          0.5    1                 0.50        50      0.7693833  0.7437593
  0.002  5          0.5    1                 0.50       100      0.7717000  0.7461333
  0.002  5          0.5    1                 0.75        50      0.7654667  0.7394074
  0.002  5          0.5    1                 0.75       100      0.7683000  0.7425556
  0.002  5          0.5    1                 1.00        50      0.7581167  0.7312407
  0.002  5          0.5    1                 1.00       100      0.7640833  0.7378704
  0.002  5          0.5    2                 0.50        50      0.7695667  0.7439630
  0.002  5          0.5    2                 0.50       100      0.7717000  0.7461333
  0.002  5          0.5    2                 0.75        50      0.7654333  0.7393704
  0.002  5          0.5    2                 0.75       100      0.7682500  0.7425000
  0.002  5          0.5    2                 1.00        50      0.7581000  0.7312222
  0.002  5          0.5    2                 1.00       100      0.7641500  0.7379444
  0.002  5          0.5    3                 0.50        50      0.7696167  0.7440185
  0.002  5          0.5    3                 0.50       100      0.7723667  0.7470741
  0.002  5          0.5    3                 0.75        50      0.7653667  0.7392963
  0.002  5          0.5    3                 0.75       100      0.7680167  0.7422407
  0.002  5          0.5    3                 1.00        50      0.7578500  0.7309444

The final values used for the model were nrounds = 100, max_depth = 15, eta = 0.2, gamma = 0.1, colsample_bytree = 1,
 min_child_weight = 3 and subsample = 0.75.
```

Let's have a look at the tuning parameters resulting in the highest accuracy, and the model performance overall.

```
xgb_tune$results[which.max(xgb_tune$results$Accuracy), ]
```

```
    eta max_depth gamma colsample_bytree min_child_weight subsample nrounds Accuracy     Kappa  AccuracySD     KappaSD
448 0.2        15   0.1              1                3      0.75     100  0.86205 0.8467222 0.003380554 0.003676171
```

```
mp.xgb = model_performance(xgb_tune, train.images.pca, test.images.pca,
                           train.classes, test.classes, "xgboost")
```

```
accuracy_train precision_train recall_train  F1_train accuracy_test precision_test recall_test   F1_test   model
       0.86205       0.9526255       0.9585 0.9555537        0.8548       0.9495549        0.96 0.9547489 xgboost
```

The optimal combination of tuning parameter values resulted in 86.2% training and 85.5% testing accuracies. Although there may be some slight overfitting going on, the model performs a bit better than the random forest, as was expected. Let's have a look at the confusion matrix for the test set predictions to observe what clothing categories are mostly correctly or wrongly classified.

```
table(pred = predict(xgb_tune, test.images.pca),
      true = test.classes)
```

	true									
pred	Bag	Boot	Coat	Dress	Pullover	Sandal	Shirt	Sneaker	Top	Trouser
Bag	960	2	5	5	6	4	18	0	10	1
Boot	2	941	0	0	0	27	0	46	0	0
Coat	2	0	782	28	111	0	104	0	6	6
Dress	8	0	30	884	11	2	31	0	33	21
Pullover	8	0	110	9	772	0	118	0	12	1
Sandal	5	15	0	0	0	922	1	32	4	0
Shirt	8	0	72	34	81	0	572	0	101	3
Sneaker	4	42	0	0	0	45	0	922	0	0
Top	3	0	1	30	17	0	155	0	832	7
Trouser	0	0	0	18	2	0	1	0	2	961

As we saw with the random forests, pullovers, shirts and coats are most often mixed up, while trousers, boots, bags and sneakers are most often correctly classified.