Parallelisation of R with h2o

Lab II: Classification example via MNIST

Dr. Simon Caton

Building Classification Models 101

If you've never built a classification model before, here is a quick overview. If you have skip this section.

We have a single categorical dependent variable Y. E.g. for MNIST, this is the feature called label, i.e. a value between 0 and 9 representing the hand-drawn number in the image. Our aim is to use a vector of p predictor variables X (also called inputs, regressions, covariates, features, independent variables, ...) to predict (or classify) unseen instances of X. To do this, we have some amount of training data (i.e. sample of the entire dataset used for building our classification model) $(x_1, y_1)...(x_N, y_N)$, corresponding to our observations (instances, examples) of these measurements.

Using the training data, we aspire to:

1. Predict unseen test cases (using labeled test data for performance evaluation),
2. Typically understand which features influence the outcome (and how),
3. Assess the quality of our predictions and correspondingly the model we have built.

We need reasonable assertions of "quality" and also a clear understanding of what determines "high" quality. To capture performance, we can us a series of measures, many of which will be familiar, such as RMSE, MSE, etc. Ones that may be less familiar, but which are used in this as well as the next session are derivable from a coincidence matrix that captures how often the model correctly predicts a class value. For a binary classification problem this would correspond to the following:

Predicted / Actual	Yes	No
Pred Yes	True Positive (TP)	False Positive (FP)
Pred No	False Negative (FN)	True Negative (TN)

From this, we can compute:

- True positive rate (also called hit rate, or recall): $\frac{TP}{(TP+FN)}$
- False positive rate (also called false alarm rate): $\frac{TN}{(TN+FP)}$
- Accuracy: $\frac{(TP+TN)}{(TP+TN+FP+FN)}$
- Error Rate: 1 – accuracy or $\frac{(FP+FN)}{(TP+TN+FP+FN)}$
- Sensitivity (the proportion of positive examples correctly classified): $\frac{TP}{(TP+FN)}$
- Specificity (the proportion of negative examples correctly classified): $\frac{TN}{(TN+FP)}$

In a multi-class problem, Sensitivity and Specificity become "is" vs. "is not" of a given class label. Accuracy and error rate, become class accuracy similar to Sensitivity and Specificity, or emcompass all classes.

Random Forest on MNIST

This time, we'll use a number of principal components for reasons of wait time within the lab setting, and also because PCA performed quite well for the dataset. We'll also only take a smaller (stratified) sample of the data for model building.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
library(h2o)
```

```
##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit http://docs.h2o.ai
##
## ----------------------------------------------------------------------
```

```
##
## Attaching package: 'h2o'
```

```
## The following objects are masked from 'package:stats':
##
##     cor, sd, var
```

```
## The following objects are masked from 'package:base':
##
##     &&, %*%, %in%, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc
```

```
h2o.init()

mnist <- read.csv("MNISTTrain.csv", header=T)
mnist$label <- factor(mnist$label)

h2o.mnist <- h2o.importFile(path=paste0(getwd(), "/MNISTTrain.csv"), header=T)
```

```
columsKeep <- names(which(colSums(mnist[, -1]) > 0))
h2o.mnist <- h2o.mnist[c("label", columsKeep)]
mnist <- mnist[c("label", columsKeep)]
h2o.mnist[,1] <- as.factor(h2o.mnist[,1])

set.seed(2018)
#stratified sampling
index <- createDataPartition(mnist$label, p = .25, list = FALSE)

h2o.training <- h2o.mnist[index, -1]
h2o.testing <- h2o.mnist[-index, -1]
```

Now please compute the Principle Components using the training data, and apply the first 35 to both the training and testing samples of the dataset.

So, you need to:

1. Run PCA (*h2o.prcomp*) on the training frame,
2. Apply the loadings to both the training (name this h2o.training.pca) and testing (name this h2o.testing.pca) samples, and finally
3. Cast the output back into an H2OFrame using *as.h2o()*

Note that h2o will by default remove (i.e., ignore) features that are constant. This unless handled appropriately will cause problems when we apply the loadings. Depending on your sample, you may encouter constant columns, use the parameter *ignore_const_cols = FALSE* to switch this feature off.

You should have two h2o frames that look something like this (if you need a little help, check the previous lab session):

```
dim(h2o.training.pca)
```

```
## [1] 10503    35
```

```
dim(h2o.testing.pca)
```

```
## [1] 31497    35
```

Whilst h2o can do a lot for us, it cannot yet optimise the number of PCs that we use. However, as this is a fairly commonly used dataset, usually n between 35 and 45 is fine.

Now, to build our classification model. Standard R using the same default settings as h2o would be as follows (assuming we'd had also undertaken the above steps for on the original mnist data):

```
library(randomForest)
pca.train$label <- mnist[index, 1]
```

We added the depending variable back in again for training, and now we could build our model. However, I wouldn't suggest to run the code segment below, it takes a while to run.

```
t1 <- Sys.time()
rf <- randomForest(label ~., data=pca.train, ntree=50)
print( difftime( Sys.time(), t1, units = 'sec'))

pred <- predict(rf, newdata=pca.test)

confusionMatrix(pred)
```

The same with h2o (again using default parameter settings). First add the depending variables back in, and build a variable that contains the name of the dependent (required by h2o):

```
#add the depending variable back in again
h2o.training.pca <- h2o.cbind(h2o.training.pca, as.h2o(factor(mnist[index,1])))
names(h2o.training.pca)

#rename the dependent (it's current called x)
names(h2o.training.pca)[36] <- "label"

#do same for the testing sample
h2o.testing.pca <- h2o.cbind(h2o.testing.pca, as.h2o(factor(mnist[-index,1])))
names(h2o.testing.pca)[36] <- "label"

y <- "label"
```

Now, build, and evaluate.

```
t2 <- Sys.time()
h2o.rf <- h2o.randomForest(y=y, training_frame = h2o.training.pca)
```

```r
print( difftime( Sys.time(), t2, units = 'sec'))
```

```
## Time difference of 6.392019 secs
```

```r
h2o.performance(h2o.rf, newdata=h2o.testing.pca, xval=TRUE)
```

```
## H2OMultinomialMetrics: drf
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.1461568
## RMSE: (Extract with `h2o.rmse`) 0.3823047
## Logloss: (Extract with `h2o.logloss`) 0.4869588
## Mean Per-Class Error: 0.06975772
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4    5    6    7    8    9  Error
## 0      3028    2    4    2    1   19   28    5    9    1 0.0229
## 1         1 3453   16    9    1   13    2    7    8    3 0.0171
## 2        44   17 2889   33   21    8   22   40   49    9 0.0776
## 3        14    9   58 2958    6   92   10   26   66   24 0.0935
## 4        14   14   16    2 2836    1   25   12   20  114 0.0714
## 5        26    9   13   69   12 2622   35   12   23   25 0.0787
## 6        37    7   25    2    6   36 2982    1    6    0 0.0387
## 7        23   38   42    3   26    5    2 3068   10   83 0.0703
## 8        37   36   34   87   27   55   16   18 2709   28 0.1109
## 9        40   10   13   52  118   25    3   82   23 2775 0.1165
## Totals 3264 3595 3110 3217 3054 2876 3125 3271 2923 3062 0.0691
##                   Rate
## 0      =     71 / 3,099
## 1      =     60 / 3,513
## 2      =    243 / 3,132
## 3      =    305 / 3,263
## 4      =    218 / 3,054
## 5      =    224 / 2,846
## 6      =    120 / 3,102
## 7      =    232 / 3,300
## 8      =    338 / 3,047
## 9      =    366 / 3,141
## Totals = 2,177 / 31,497
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## ===============================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.930882
## 2   2  0.970886
## 3   3  0.982951
## 4   4  0.987459
## 5   5  0.989840
## 6   6  0.991269
## 7   7  0.991872
```

```
## 8   8  0.992253
## 9   9  0.992476
## 10 10  1.000000
```

We can add in a k-fold cross validation into the previous call by using the nfolds parameter.

Note that we haven't actually asked h2o to optimise the forest yet, we just used the default settings. To do this, we just need to instruct h2o which parameters it should tune. Typically, for a random forest, we would want to tune at least:

- ntrees: the number of trees, and potentially also
- mtries (mtry in R's randomForest library): number of variables randomly sampled as candidates at each split – note that making this variable too high will result in increasing the model's bias.

For this, we can instruct h2o to do this for us, but we need to provide it some idea of what we want these values to be. For this we use the h2o Grid Search (note the first value in the list is the default setting).

Also to help, we'll give h2o a validation sample – this will not be used for training, but later for ranking our models. It should also help to mitigate overfitting. For this, we'll use 1/3 of our remaining testing data (25% overall). Note that this can take a while to run (but you can reduce the runtime significantly by reducing the number of values per tuning parameter).

```
rf_params <- list(ntrees = c(50, 100, 200, 500),
                  mtries = c(round(sqrt(35), 0), 3, 8, 11, 14))

#stratified sampling
validation <- createDataPartition(mnist[-index,1], p = 1/3, list = FALSE)
h2o.validation <- h2o.testing.pca[validation, ]

t3 <- Sys.time()
rf_grid <- h2o.grid("randomForest", y = y,
                    grid_id = "rf_grid",
                    training_frame = h2o.training.pca,
                    validation_frame = h2o.validation,
                    hyper_params = rf_params)
```

```
print( difftime( Sys.time(), t3, units = 'sec'))
```

```
## Time difference of 1083.064 secs
```

```
#now retrieve the models and sort according our prefered performance metric
rf_grid <- h2o.getGrid(grid_id = "rf_grid",
                           sort_by = "accuracy",
                           decreasing = TRUE)

# Grab the top RF model, chosen by accuracy
best_rf <- h2o.getModel(rf_grid@model_ids[[1]])

print(best_rf)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: drf
## Model ID:  rf_grid_model_36
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1             500                     5000            26702190        12
```

```
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        20   19.06480        145        616   420.48340
##
##
## H2OMultinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("RTMP_sid_89eb_12")`
## MSE: (Extract with `h2o.mse`) 0.1716639
## RMSE: (Extract with `h2o.rmse`) 0.4143234
## Logloss: (Extract with `h2o.logloss`) 0.517119
## Mean Per-Class Error: 0.0628635
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4   5    6    7   8    9  Error
## 0      1008    0    2    2    2   5   10    0   3    1 0.0242
## 1         0 1149    5    3    3   2    4    1   2    2 0.0188
## 2         9    3  979   10    5   0    2   17  18    2 0.0632
## 3         1    3   24  999    1  18    6    7  14   15 0.0818
## 4         2    7    3    0  953   1    8    4   5   35 0.0639
## 5         9    3    3   26    7 873   11    3   7    7 0.0801
## 6        10    4    2    0    5  14  994    0   6    0 0.0396
## 7         1    9   16    0   11   1    0 1039   2   22 0.0563
## 8         5   10    9   31    5  22    7    7 909   11 0.1053
## 9        10    5    5   12   24   5    1   28  10  947 0.0955
## Totals 1055 1193 1048 1083 1016 941 1043 1106 976 1042 0.0622
##                  Rate
## 0      =   25 / 1,033
## 1      =   22 / 1,171
## 2      =   66 / 1,045
## 3      =   89 / 1,088
## 4      =   65 / 1,018
## 5      =     76 / 949
## 6      =   41 / 1,035
## 7      =   62 / 1,101
## 8      =  107 / 1,016
## 9      =  100 / 1,047
## Totals = 653 / 10,503
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.937827
## 2   2  0.976293
## 3   3  0.987718
## 4   4  0.993716
## 5   5  0.996382
## 6   6  0.997620
```

```
## 7   7  0.998381
## 8   8  0.998857
## 9   9  0.999048
## 10 10  1.000000
##
##
## H2OMultinomialMetrics: drf
## ** Reported on validation data. **
##
## Validation Set Metrics:
## =====================
##
## Extract validation frame with `h2o.getFrame("RTMP_sid_89eb_14")`
## MSE: (Extract with `h2o.mse`) 0.1726149
## RMSE: (Extract with `h2o.rmse`) 0.4154695
## Logloss: (Extract with `h2o.logloss`) 0.5209388
## Mean Per-Class Error: 0.0591096
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4   5    6    7   8    9  Error
## 0      1017    0    2    2    0   4    5    1   2    0 0.0155
## 1         0 1149   10    2    1   3    2    2   2    0 0.0188
## 2        13    2  975   13    9   4    8    6  14    0 0.0661
## 3         4    3   21 1006    1  19    4   11  14    5 0.0754
## 4         1    6    4    0  958   0    7    1   4   37 0.0589
## 5         7    2    4   16    2 898    6    5   3    6 0.0537
## 6        11    2    4    1    3  11 1002    0   0    0 0.0309
## 7         2   11   14    1    4   2    1 1037   3   25 0.0573
## 8         5   16   11   36    8  26    4    2 900    8 0.1142
## 9         2    4    5   15   40   6    1   22  10  942 0.1003
## Totals 1062 1195 1050 1092 1026 973 1040 1087 952 1023 0.0587
##                  Rate
## 0      =   16 / 1,033
## 1      =   22 / 1,171
## 2      =   69 / 1,044
## 3      =   82 / 1,088
## 4      =   60 / 1,018
## 5      =     51 / 949
## 6      =   32 / 1,034
## 7      =   63 / 1,100
## 8      =  116 / 1,016
## 9      =  105 / 1,047
## Totals = 616 / 10,500
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.941333
## 2   2  0.978381
## 3   3  0.988381
## 4   4  0.992857
## 5   5  0.995619
```

```
## 6   6  0.996762
## 7   7  0.998286
## 8   8  0.999143
## 9   9  0.999810
## 10 10  1.000000
```

```
h2o.performance(best_rf, newdata=h2o.testing.pca[-validation, ], xval=TRUE)
```

```
## H2OMultinomialMetrics: drf
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.17023
## RMSE: (Extract with `h2o.rmse`) 0.4125894
## Logloss: (Extract with `h2o.logloss`) 0.5149765
## Mean Per-Class Error: 0.05890397
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## ========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           0    1    2    3    4    5    6    7    8    9  Error
## 0      2018    0    2    2    4    9   21    2    7    1 0.0232
## 1         0 2307    9    9    0    5    1    5    3    3 0.0149
## 2        18   11 1953   21   11    4    8   27   31    4 0.0647
## 3         3    7   38 2002    4   45    3   12   43   18 0.0795
## 4         2   10    8    1 1924    2   11    3    6   69 0.0550
## 5        15    3    7   46   11 1771   18    6    7   13 0.0664
## 6        17    4   14    0    3   21 2002    1    6    0 0.0319
## 7         8   26   24    2   16    2    1 2067    5   49 0.0605
## 8        14   23   16   52   12   28   10   13 1842   21 0.0931
## 9        17    5    6   42   75   10    0   45    9 1885 0.0998
## Totals 2112 2396 2077 2177 2060 1897 2075 2181 1959 2063 0.0584
##                   Rate
## 0      =     48 / 2,066
## 1      =     35 / 2,342
## 2      =    135 / 2,088
## 3      =    173 / 2,175
## 4      =    112 / 2,036
## 5      =    126 / 1,897
## 6      =     66 / 2,068
## 7      =    133 / 2,200
## 8      =    189 / 2,031
## 9      =    209 / 2,094
## Totals = 1,226 / 20,997
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## =======================================================================
## Top-10 Hit Ratios:
##     k hit_ratio
## 1   1  0.941611
## 2   2  0.977568
## 3   3  0.987951
## 4   4  0.993523
## 5   5  0.996380
## 6   6  0.997523
```

```
## 7   7  0.998381
## 8   8  0.999286
## 9   9  0.999667
## 10 10  1.000000
```

So this is a noticeable improvement, our mean per-class error is now reasonably better. Note that there are better candidate models for this dataset, such as a support vector machine, and XGBoost (the latter has been implemented by h2o, and the former sadly largely replaced by deeplearning models now). The process to train and optimise XGBoost or a deep learner would be largely the same the same as above.

Note that we have been a little over excited with the choice of parameters for optimisation (specifically w.r.t. mtries), but for the purposes of demonstrating a grid search this is fine.

Exercise

If you find that you have extra time, build and optimise a classification model of you choice (or aligned with your own interest) either on MNIST, or a similar dataset. Alternatively, you can also experiment with alternative tuning parameters. E.g.:

- max_depth: Maximum tree depth (size of each tree in the forest), and
- min_rows: Fewest allowed (weighted) observations in a leaf.