Parallelisation of R with h2o

Lab III: Regularisation example via movie_reviews

Dr. Simon Caton

Introduction

h2o supports a large number of Generalised Linear Models. In fact, too many to cover here. See here for an extensive overview with examples https://h2o-release.s3.amazonaws.com/h2o/rel-slater/9/docs-website/h2o-docs/booklets/GLM_Vignette.pdf.

If you need a refresh of (or introduction to) regularisation, I would suggest viewing the following YouTube video lectures from the ISLR book: https://www.youtube.com/playlist?list=PL5-da3qGB5IB-Xdpj_uXJpLGiRfv9UVXI

Basic Built-in Example

The link above provides a much more expansive explanation of the following (simple) example (pages 11-12), but the basic call for a logistic regression in h2o is as follows:

```
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()
path = system.file("extdata", "prostate.csv", package = "h2o")
h2o_df = h2o.importFile(path)
h2o_df$CAPSULE = as.factor(h2o_df$CAPSULE)
```

```
binomial.fit = h2o.glm(y = "CAPSULE",
          x = c("AGE", "RACE", "PSA", "GLEASON"),
          training_frame = h2o_df, family = "binomial")
```

```
h2o.performance(binomial.fit)
```

```
## H2OBinomialMetrics: glm
## ** Reported on training data. **
##
## MSE:  0.1786802
## RMSE:  0.4227058
## LogLoss:  0.5289
## Mean Per-Class Error:  0.2598975
## AUC:  0.7927644
## Gini:  0.5855288
## R^2:  0.2571069
## Residual Deviance:  401.964
## AIC:  411.964
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##          0   1    Error      Rate
## 0      155  72 0.317181   =72/227
## 1       31 122 0.202614   =31/153
## Totals 186 194 0.271053  =103/380
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold    value idx
## 1                       max f1  0.298968 0.703170 191
## 2                       max f2  0.205231 0.803080 294
## 3                 max f0point5  0.531385 0.687606 108
## 4                 max accuracy  0.531385 0.736842 108
## 5                max precision  0.997166 1.000000   0
## 6                   max recall  0.087936 1.000000 359
## 7              max specificity  0.997166 1.000000   0
## 8             max absolute_mcc  0.298968 0.471116 191
## 9   max min_per_class_accuracy  0.408870 0.713656 172
## 10 max mean_per_class_accuracy  0.298968 0.740103 191
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

To undertake classification (it's a categorical dependent variable), we would follow the same approach as we did for the MNIST dataset:

- create a training, and testing sample
- build the model as above
- use *h2o.performance()* and/or *h2o.predict()*

However, in the presence of a large number of predictors, or when the data set is wide, we want to avoid over-fitting, reduce variance of the prediction error and handle correlated predictors. Similarly,it is also advantageous to reduce the number of components our model has to aid its generalisability and interpretability. Regularisation through the introduction of penalties assists in both these respects. The two most common penalized models are ridge regression and the lasso, with the elastic net combining both penalties. For large (wide) datasets, finding a good selection of variables for a number of tuning parameters can be quite computationally expensive, although both the ridge regression and lasso do a fair bit to reduce computational time vs. for example a best subset selection.

Typically in R, we would use the glmnet function for this, which is reasonably fast. Yet, h2o also provides the ability as we saw to optimise over the tuning parameters too.

To regularise the sample model above via an elastic net, we would do the following:

```
#random sample via h2o's sample method
split <- h2o.splitFrame(h2o_df, ratios = .75)
training <- split[[1]]
testing <- split[[2]]

binomial.fit.regularised <- h2o.glm(y = "CAPSULE",
                    x = c("AGE", "RACE", "PSA", "GLEASON"),
                    family = "binomial",
                    training_frame = training,
                    alpha = 0.5,
                    lambda_search = TRUE,
                    nfolds = 5)
```

```
h2o.performance(binomial.fit.regularised, newdata=testing)
```

```
## H2OBinomialMetrics: glm
##
## MSE:  0.1821809
## RMSE:  0.4268265
## LogLoss:  0.5376573
## Mean Per-Class Error:  0.2410628
## AUC:  0.815942
## Gini:  0.6318841
## R^2:  0.2711884
## Residual Deviance:  97.85362
## AIC:  107.8536
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##         0  1    Error    Rate
## 0      32 14 0.304348  =14/46
## 1       8 37 0.177778   =8/45
## Totals 40 51 0.241758  =22/91
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold    value idx
## 1                       max f1  0.291601 0.770833  50
## 2                       max f2  0.220185 0.869565  72
## 3                 max f0point5  0.493786 0.751445  31
## 4                 max accuracy  0.291601 0.758242  50
## 5                max precision  0.962430 1.000000   0
## 6                   max recall  0.119707 1.000000  84
## 7              max specificity  0.962430 1.000000   0
## 8             max absolute_mcc  0.291601 0.521668  50
## 9   max min_per_class_accuracy  0.408881 0.739130  45
## 10 max mean_per_class_accuracy  0.291601 0.758937  50
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

Note the additional parameters:

- alpha: controls the elastic net penalty distribution between the l1 and l2 norms. It can have any value in the [0, 1] range or a vector of values (which triggers grid search). If $\alpha = 0$, h2o solves the GLM

with ridge regression, while if $\alpha = 1$ it does so with the lasso penalty. Defaults at 0.5 (hence the same outcome as above).
- lambda_search: h2o will seek to optimise λ (the regularisation strength). The λ range is any positive value or a vector of values (which triggers grid search). Note: Lambda values are capped at λ_{max}, which is the smallest λ for which the solution is all zeros (except for the intercept term). Defaults to FALSE.
- nfolds: number of folds for K-fold cross-validation. Defaults to 0.

We can also leverage a grid search for parameters like α similar to how we did for the random forest.

Undertaking the same with glmnet would be as follows:

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-13
```

```
library(caret)
```

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

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

```
df <- read.csv(file = system.file("extdata", "prostate.csv", package = "h2o"))
df$CAPSULE <- as.factor(df$CAPSULE)
index <- sample(1:nrow(df), nrow(df)*.75, replace = F)
training <- df[index, ]
testing <- df[-index, ]

t1 <- Sys.time()
fit <- cv.glmnet(x = as.matrix(training[,c("AGE", "RACE", "PSA", "GLEASON")]),
                 y = training$CAPSULE,
                 family = 'binomial',
                 alpha = 0.5,
                 nfolds=5)
print( difftime( Sys.time(), t1, units = 'sec'))
```

```
## Time difference of 0.2039092 secs
```

```
pred <- predict(fit, newx=as.matrix(testing[,c("AGE", "RACE", "PSA", "GLEASON")]),
  type = 'response')[, 1]

#pick a confidence threshold, e.g. 0.5 (we can optimise this easily enough though)
pred <- ifelse(pred>0.5, 1, 0)

confusionMatrix(pred, testing$CAPSULE)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 50 27
##          1  4 14
##
##                Accuracy : 0.6737
##                  95% CI : (0.5698, 0.7664)
##     No Information Rate : 0.5684
```

```
##     P-Value [Acc > NIR] : 0.02345
##
##                   Kappa : 0.2868
##  Mcnemar's Test P-Value : 7.772e-05
##
##             Sensitivity : 0.9259
##             Specificity : 0.3415
##          Pos Pred Value : 0.6494
##          Neg Pred Value : 0.7778
##              Prevalence : 0.5684
##          Detection Rate : 0.5263
##    Detection Prevalence : 0.8105
##       Balanced Accuracy : 0.6337
##
##        'Positive' Class : 0
##
```

At this point, we would still need to determine at least the value of α, and our confidence threshold (the latter is probably slightly more challenging methodologically, but from a programming perspective can essentially be undertaken with a for loop). Before we concern ourselbves with this, we could also use built-in parallelism for this method as follows:

```
library(doMC)
registerDoMC(detectCores())
t2 <- Sys.time()
fit <- cv.glmnet(x = as.matrix(training[,c("AGE", "RACE", "PSA", "GLEASON")]),
                 y = training$CAPSULE,
                 family = 'binomial',
                 alpha = 0.5,
                 nfolds = 5,
                 parallel = TRUE)
print( difftime( Sys.time(), t2, units = 'sec'))
```

```
## Time difference of 0.2675769 secs
```

However, for a dataset of this size, this is somewhat overengineering the problem! The overhead of using multiple cores and synchronising across them is larger (evidentally) than not parallelising in the first place! However, as we will see in the example below, if runtime is important, the above code will outperform h2o.

Analysing Movie Reviews

So let's articulate the type of classification problem we have here: a dichotomous dependent variable "good" and "not good" movie review, our independent variable (as it currently stands) in this case is our review text. Obviously, we cannot use this as is, some preprocessing is required.

Let's start just by sampling the data:

```
library(text2vec)
data("movie_review")

set.seed(2018)
trainIndex <- createDataPartition(movie_review$sentiment, p = 0.8,
                                  list = FALSE,
                                  times = 1)
```

```
movie_train <- movie_review[trainIndex, ]
movie_test <- movie_review[-trainIndex, ]
```

Next, we need to tokenize the text; essentially find all of the individual words in the text. Doing this means we have a list of words. We can then process each word in our list:

- Convert them to lower case, thus Cat and cat become the same word.
- We can (but aren't here) stem the words, i.e. cut the end of the word off. For example, competing (and compete), would become compet this removes common suffixes reducing variation in the list of words.
- We can also (but aren't here) remove stop words like the, he, she, it. These words are extremely common and thus it is not likely they really add to our model.
- We can also (but aren't here) remove punctuation and special characters (e.g. emoji). Otherwise "win," and "win" would be considered as different words.

The last 3 forms of preprocessing are excluded here, just to simplify the code.

```
prep_fun = tolower
tok_fun = word_tokenizer

it_train = itoken(movie_train[['review']],
             preprocessor = prep_fun,
             tokenizer = tok_fun,
             ids = movie_train[['ids']],
             progressbar = FALSE)

it_test = itoken(movie_test[['review']],
             preprocessor = prep_fun,
             tokenizer = tok_fun,
             ids = movie_test[['ids']],
             progressbar = FALSE)
```

Next, we build a vocabulary (a list of all unique words across all reviews):

```
vocab <- create_vocabulary(it_train)
```

Each word will later become one possible component in our model – it gets big very quickly!

Finally, tokenize the corpus of reviews and build a sparse matrix (called a document term matrix) in each column is the frequency of word occurences, for each review (one per row). From here, we'll be able to model reviews according to word frequencies.

```
vectorizer <- vocab_vectorizer(vocab)

dtm_train <- create_dtm(it_train, vectorizer)
dtm_test <- create_dtm(it_test, vectorizer)
```

These matrices are quite big, and very sparse! We have predictors! Many of which will not be useful at all.

```
dim(dtm_train)
```

```
## [1]  4000 38320
```

Using the glmnet package, we can build a logistic regression, and regularise (in parallel) as follows. Here we are also optimising based on the area under the ROC curve:

```
t1 <- Sys.time()
fit <- cv.glmnet(x = dtm_train, y = movie_train[['sentiment']],
                 family = 'binomial',
                 # lasso + ridge
```

```
                  alpha = 0.5,
                  # interested area unded ROC curve
                  type.measure = "auc",
                  # 5-fold cross-validation
                  nfolds = 5,
                  parallel = TRUE)

print( difftime( Sys.time(), t1, units = 'sec'))
```

```
## Time difference of 3.946934 secs
```

```
pred <- predict(fit, dtm_test, type = 'response')[, 1]

glmnet:::auc(movie_test[['sentiment']], pred)
```

```
## [1] 0.9292578
```

Whilst this performance is ok, we can do a lot better. Note also, that by using the glmnet implementation of auc we can remove the need to determine a confidence threshold for predicting our dependent variable, which for the purposes of this session is convenient.

To do the same with h2o, we have to handle a little challenge: h2o is not very good at receiving sparse matrices. For a dataset like this, it won't be too bad, but if we increase the number of reviews significantly, *as.h2o()* will be really slow! Even with packages like data.table and slam installed, which help it deal with sparse matrices.

For this, we can use the SVMLight representation of a sparse matrix:

```
library(sparsio)
write_svmlight(dtm_train, y = movie_train[['sentiment']], file="train.svmlight")
write_svmlight(dtm_test, y = movie_test[['sentiment']], file="test.svmlight")
```

This creates a better representation of the sparse matrix represented as follows:

1 4662:1 8096:1 9698:1 10827:1 ...

Where 1 reflects the value of the dependent variable (positive), then only non-zero columns are represented. We can import this kind of data much faster into h2o, than we could using *as.h2o()*

```
h2o_training <- h2o.importFile(path=paste0(getwd(), "/train.svmlight"))
h2o_testing <- h2o.importFile(path=paste0(getwd(), "/test.svmlight"))
```

Note that the names of the columns have changed. They have been renamed C1, ... CN this information was lost in the conversion to the SVMLight format. However, C1 is our dependent variable. We could also fix this easily enough if it bothered us significantly via the names function.

```
y <- "C1"

t2 <- Sys.time()
glm_fit1 <- h2o.glm(y = y,
                    family = "binomial",
                    training_frame = h2o_training,
                    alpha = 0.5,
                    lambda_search = TRUE,
                    nfolds = 5)
```

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

```
## Time difference of 512.3626 secs
```

```
h2o.performance(glm_fit1, xval = TRUE, newdata = h2o_testing)
```

```
## H2OBinomialMetrics: glm
##
## MSE:  0.1073666
## RMSE:  0.3276685
## LogLoss:  0.3484456
## Mean Per-Class Error:  0.1377558
## AUC:  0.9259813
## Gini:  0.8519627
## R^2:  0.5704716
## Residual Deviance:  696.8913
## AIC:  4816.891
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
##          0   1    Error       Rate
## 0      426  80 0.158103    =80/506
## 1       58 436 0.117409    =58/494
## Totals 484 516 0.138000  =138/1000
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold    value idx
## 1                       max f1  0.527801 0.863366 203
## 2                       max f2  0.198314 0.904636 298
## 3                 max f0point5  0.638770 0.868626 169
## 4                 max accuracy  0.560958 0.862000 194
## 5                max precision  0.999936 1.000000   0
## 6                   max recall  0.006483 1.000000 391
## 7              max specificity  0.999936 1.000000   0
## 8             max absolute_mcc  0.527801 0.724807 203
## 9   max min_per_class_accuracy  0.564265 0.859684 193
## 10 max mean_per_class_accuracy  0.527801 0.862244 203
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

In this case, although we can marginally improve an AU, the compute time here probably wasn't worth it. We'd actually be better off this time (sorry!!) working a little harder by hand, and using the parallelisation backend of cv.glmnet to tune the regularisation parameters. However, if you are/were happy to wait.

The silver lining here I suppose is that we could use a grid search for the α parameter. We assumed that an elastic net would be the best option here, which it may not be.

For cv.glmnet this is quite straight forward, especially once we have established a local cluster for the implementation:

```
alpha <- seq(0,10)/10
auc <- c()
models <- c()

t3 <- Sys.time()
for (i in 1:length(alpha)) {
  fit <- cv.glmnet(x = dtm_train, y = movie_train[['sentiment']],
                   family = 'binomial',
                   # lasso + ridge
                   alpha = alpha[i],
                   # interested area unded ROC curve
```

```
                  type.measure = "auc",
                  # 5-fold cross-validation
                  nfolds = 5,
                  parallel = TRUE)


  pred <- predict(fit, dtm_test, type = 'response')[, 1]
  auc[i] <- glmnet:::auc(movie_test[['sentiment']], pred)
  # this is optional (we're not going to use the model again)
  models[i] <- fit
}
print( difftime( Sys.time(), t3, units = 'sec'))
```

```
## Time difference of 40.23531 secs
```

Which in comparison to the approx. 11 mins it took to do just one optimisation for α with h2o is a quite a difference. For reference, our best model now has an AUC of:

```
max(auc)
```

```
## [1] 0.9379471
```

using α:

```
alpha[which(auc == max(auc))]
```

```
## [1] 0.1
```

Summary

So to summarise, as this may seem like it was a waste of time exploring h2o in this context. Yet, there are a couple of important points to consider:

Performance: h2o can out perform cv.glmnet in the case where α was set to 0.5 (change the seed and redo the sample, and you may find one of these example, but inevitably many of these examples will be crowded out by computational time needed) – it may eventually outperform glmnet if we instrumented the grid search to optimise α. However, the improvment may be marginal.

Computational time: the idea of computational time is different for different users. A pragmoatic user, may only consider wall clock time (i.e. the time from start to finish as experienced by the user), by building a sufficiently large cluster (i.e. combining multiple cloud VMs into 1 h2o cluster), the wall clock time (as measured through this lab) can potentially be reduced to be more competitive with cv.glmnet (via the doMC library). The flip side here is CPU time (i.e., the sum of all time spent across all cores computing the solution), which obviously in the case of h2o will be far higher than for cv.glmnet. However, when we consider other scenarios, such as the random forest or PCA in the previous lab, here we see something more benficial.

Trade off: it will be up to you whenever you use parallel implementations like h2o to decide for yourself where its benefits are. It's important that sessions like this one do not gloryify such libraries, as they may not always be "better" depending on your notion(s) of quality. For example, based on the observations of this session, you'd probably tend to do PCA with h2o, and an elastic net with cv.glmnet (+ doMC).

Be pragmatic: you do also need to factor in development time (i.e. coding time) into you notions of performance. This, often overlooked, component of "performance" can sway your perspectives a little, but it's highly subjective and individualised.

Exercise

To close, and for adding a little light on the closing discussion obove. Take the opening example of the prostate dataset, and compare and constrast h2o vs. a parallelised (or not) cv.glmnet with a (pseudo) grid search for α trying to create for yourself your own answer(s) to the four points above.