Appendix S9: Code for case study 1: Model averaging of GLMS, with a comparison of approaches for computing model weights

Dormann et al.

06 December, 2017

Contents

1 Introduction

In the following sections we shall briefly present different approaches for averaging predictions from likelihood-based models (here simple linear models). We demonstrate their application using simulated data that aims to mimic real-life situations, where predictors exhibit collinearity, and the response is not a simple and clean function of predictors (details below). This is meant as an illustration of how the different model-weight-deriving approaches can be implemented, but not a how-to guide advising what to do. Typically some data dredging may lead to the set of models to be averaged, which may be much larger than our set of 16 in this example.

2 Data simulation

We start by simulating a data set:

1. We generate two orthogonal (uncorrelated) and uniformly distributed predictors; we call these two predictors "SET1".
2. We generate a second pair of predictors (SET2), so that they have strong (but not perfect) collinearity with SET1. To do so, we first rotate the predictors in SET1 to obtain a new pair of orthogonal predictors, and then add noise.
3. We generate the response y; to do so, we first create a response 'y1' based on SET1, and a response 'y2' by using SET2 and the original regression coefficients rotated (this way, 'y1' and 'y2' would have been identical if we had not added noise to SET2). We assign 'y1' to half of the sites and 'y2' to the other half. The idea behind this is that the response in some sites will be better predicted by SET1, and in other sites by SET2. By putting all predictors together in the model,we can expect a number of competing models to have similar explanatory power.

The final response includes some additional noise for more realism.

```
N <- 70  #total number of data points (70 each for training and testing)

simdata <- function(N){
  # PREDICTORS
  P <- 2    # number of predictors, if want more will need a larger rotation matrix
  preds1 <- matrix(runif(2*N*P),nrow=2*N,ncol=P)   # three predictors (SET1)
  #rot1   <- c( 0.36, 0.48, -0.8, -0.80, 0.60, 0.0, 0.48, 0.64, 0.6)
  rot1   <- c( 0.96, -0.26, 0.26, 0.96)
  rot1m  <- matrix(rot1, nrow=P, ncol=P)           # rotation matrix; t(rot1m) = solve(rot1m)
  preds2 <- preds1 %*% rot1m                      # rotated predictors (SET2)
  preds2 <- preds2 + matrix(runif(2*N*P, min=0, max=0.2), nrow=2*N, ncol=P) # with a bit of noise
  preds  <- cbind(preds1, preds2)

  # RESPONSE
  b0 <- 1; b1 <- 4; b2 <- 2;   #regression coefficients
  y1 <- b0 + b1 * preds[,1] + b2 * preds[,2]   # response as determined by preds1
  bs <- c(b1, b2) %*% rot1m; b3 <- bs[1]; b4 <- bs[2]; #rotated coefficients
  y2 <- b0 + b3*preds[,3] + b4*preds[,4]   # response as determined by preds2
  tmp <- sample(1:(2*N), size=N)   #some sites get y1 and some y2
  y <- y1; y[tmp] <- y2[tmp]
  y <- y + rnorm(2*N, sd=1)  # some noise, for more realism

  # COMPILE ALL DATA
  mydf<-data.frame(y=y, p=preds)    #dataframe with response and all predictors
}
```

```
set.seed(5)
dats <- simdata(N)
train <- dats[1:N,]
test <- dats[(N+1) : (2*N),]
```

This data set will be fitted using 16 different linear models, differing in the number of predictors (1 to 5, including the intercept, in all possible combinations). The row numbers (1 - 16) refer to the models in the main text (there called "m1" - "m16"; m1 is the intercept-only model).

```
m.all <- lm(y~., data=train) # fit linear terms only; use only first half of the data
library(MuMIn)
options(na.action = "na.fail")
mytab <- dredge(m.all, rank=AIC)
mytab
```

```
Global model call: lm(formula = y ~ ., data = train)
---
Model selection table
   (Intrc)   p.1     p.2     p.3    p.4 df   logLik   AIC delta weight
10  0.8004 3.723                  2.135  4 -103.288 214.6  0.00  0.284
4   1.0170 4.285  2.0420                4 -103.393 214.8  0.21  0.256
12  0.8724 3.941  0.8243          1.300  5 -103.208 216.4  1.84  0.113
14  0.8534 4.186         -0.4514  2.015  5 -103.266 216.5  1.96  0.107
8   1.0660 4.804  1.9040 -0.5436         5 -103.360 216.7  2.14  0.097
13  0.4328                3.4710  3.120  4 -104.987 218.0  3.40  0.052
16  0.9108 4.296  0.7844 -0.3570  1.246  6 -103.194 218.4  3.81  0.042
7   0.7227        3.1470  4.2840         4 -105.954 219.9  5.33  0.020
15  0.4488        0.2721  3.5400  2.863  5 -104.979 220.0  5.38  0.019
6   1.8560 8.964         -4.8380         4 -106.759 221.5  6.94  0.009
2   2.0220 4.435                         3 -111.874 229.7 15.17  0.000
11  0.4682       -8.0800         10.940  4 -116.230 240.5 25.88  0.000
5   2.5950                3.8470         3 -122.303 250.6 36.03  0.000
9   1.6760                        3.596  3 -127.476 261.0 46.37  0.000
3   3.0760        2.4330                 3 -135.504 277.0 62.43  0.000
1   4.3680                               2 -140.595 285.2 70.61  0.000
Models ranked by AIC(x)
```

```
model.list <- get.models(mytab, subset=NA)
# sort the model list from 1 : M (to make all outputs follow the same sequence,
# from simplest to full model):
model.list <- model.list[order(as.numeric(rownames(mytab)))]
M <- length(model.list)
```

```
truth <- test[,1] # we use test as stand-in
preds <- sapply(model.list, predict, newdata=test)
```

For each of the 16 models, we now compute their RMSE:

```
singleRMSEs <- apply(preds, 2, function(x) (sqrt(mean((x - truth)^2))))
```

Additionally, mimicking optimal model selection, we store the best model for this data set. This model, the "best possible model of a run", is included as a reference only, to see whether model averaging can exceed model selection. Together with the full model (m16), run's best is serving as a yardstick for model averaging.

```
RMSEsinglebest <- min(singleRMSEs, na.rm=T)
singleRMSEs
```

```
       1        2        3        4        5        6        7        8        9       10
1.736192 1.302883 1.575529 1.104876 1.497730 1.195540 1.115050 1.107267 1.475688 1.120529
      11       12       13       14       15       16
1.339535 1.112717 1.135443 1.121223 1.132715 1.113752
```

3 The general form of model weights

Model weights for all performance-related indices κ have the same form: $w_m = \frac{e^{\kappa_m - \kappa_{\min}}}{\sum_{k \in \mathcal{M}} e^{\kappa_k - \kappa_{\min}}}$. For a given method, we need to replace κ with the appropriate measure of model performance. For likelihood-based approaches, this could be, for example, $\kappa = \ell_m$.

4 Equal weights: 1/M

The simplest of model averages gives each model the same weight. $\kappa_1 = \ldots = \kappa_m = 1/M = 1/16$ in this case.

```
weighted1overM <- preds %*% rep(1/M, M)
(RMSE1overM <- sqrt(mean((weighted1overM - truth)^2)))
```

```
[1] 1.126825
```

This may serve as a reference for all other methods.

5 Median of predictions

Similarly simple is to compute the median of predictions at each point. One could keep a tally of how often a model was used to compute the median, but we did not do that here.

```
medians <- apply(preds, 1, median)
(RMSEmedian <- sqrt(mean((medians - truth)^2)))
```

```
[1] 1.10716
```

6 Reversible-jump MCMC

The rjMCMC has to be tailored to the problem at hand. Specifically, we have to define two steps as functions: updating the parameters (updateparam) and updating the model (updatemodel). Both functions are a bit too long to be presented here and are given in a separate file, which we load before further analysis.

```
source("RJMCMCfunctions.R")
X <- as.matrix(cbind(1, train[,-1])) # include 1 for the intercept
y <- train$y
```

We initialise the rjMCMC with a randomly choosen model and random values for the model parameters (e.g. from the prior):

```
# starting values for coefficients
beta_vec <- runif(ncol(X), -1, 1)
# starting value for residual noise estimate sigma
sigma <- rgamma(1, 1, 1)
```

The likelihood-value is needed as input when we start the actual Markov chain:

```
llikhood <- sum(dnorm(y, X%*%beta_vec, sigma, log = T))

niter <- 1e5
burnin <- niter/2
thin <- 20

(niter-burnin)/thin
```

```
[1] 2500
```

```
# store values
beta_mat <- matrix(0, nrow = (niter-burnin)/thin, ncol = length(beta_vec))
sigma_vec <- c()

#choose prior parameters for coefficients
prior_sigma_beta <- 3
prior_pars <- c(0, prior_sigma_beta)

#choose parameters of proposal distributions for updates
prop_sigma_beta <- 3
prop_pars <- c(0, prop_sigma_beta)
# now run rjMCMC:
system.time({
for (iter in 1:niter){
  temp1 <- updateparam(beta_vec, sigma, llikhood, prior_beta = "norm", prior_beta_par =
                         prior_pars, prop_beta = "norm", prop_beta_par = prop_pars[2])
  beta_vec <- temp1$beta_vec
  sigma <- temp1$sigma
  llikhood <- temp1$llikhood
  # between models update
  temp2 <- updatemodel(beta_vec, sigma, llikhood,  prior_beta = "norm", prior_beta_par =
                         prior_pars, prop_beta = "norm", prop_beta_par = prop_pars)
  # read current values for beta, current model and log-likelihood value
  beta_vec <- temp2$beta_vec
  llikhood <- temp2$llikhood
  # Store output
  if (iter>burnin & (iter-burnin) %% thin == 0){
    ind_st  <-  ceiling((iter-burnin)/thin)
    beta_mat[ind_st, ] <- beta_vec
    sigma_vec[ind_st] <- sigma
  }
  #there are 16 possible models, count the number of times each model appears
  model_mat <- matrix(0, nrow = nrow(beta_mat), ncol = 1)
  for(sim in 1:nrow(beta_mat)){
    model_mat[sim,] <- paste(c((1:length(beta_vec))[beta_mat[sim,1:length(beta_vec)]!=0]),
                             sep="", collapse = "+")
  }
}
})
```

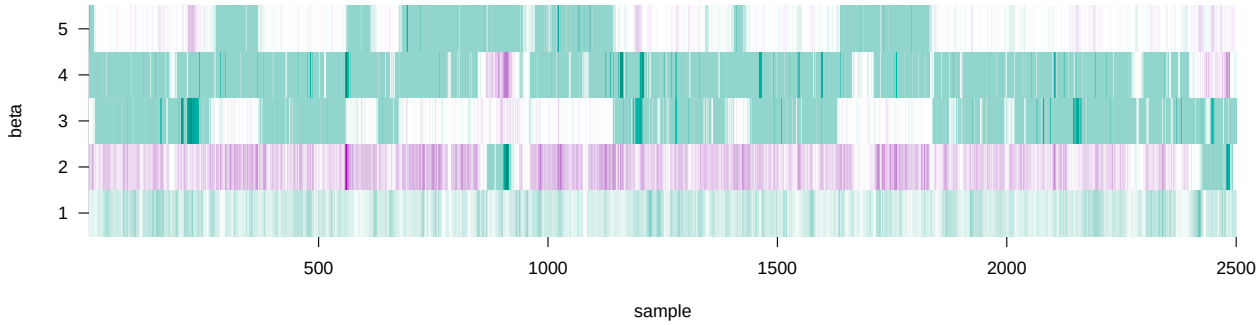
```
    user   system  elapsed
1458.404    6.715 1482.714
```

That took a while (approximately 30 minutes)! At this point it seems imperial to check the run diagnostics, which is far beyond the scope of our excercise. Instead, we carry on regardless, emphasising that this is not

what we would normally do before having checked the rjMCMC diagnostics! (Note that the final results suggest that the rjMCMC did perform very well indeed, suggesting that we did choose sufficient chain lengths.)

The next lines of code summarise the results, compute model weights and sort the labels so that the models are in the same sequence as for all other runs.

```
model_weights <- sort(round(table(model_mat)/nrow(model_mat), 5), TRUE)
allmodelnames <- c("1", paste0("1+", unlist(sapply(1:4, function(n)
    apply(combn(2:5, n), 2, paste0, collapse="+")))))
weightsrjMCMC <- rep(0, M)
names(weightsrjMCMC) <- allmodelnames
# put values into vector of M models:
for (i in 1:length(model_weights)){# loop through weights of rjMCMC
  ind <- which(allmodelnames == names(model_weights[i]) )
  weightsrjMCMC[ind] <- model_weights[i]
}
library(colorspace)
image(y=1:5, x=1:nrow(beta_mat), z=beta_mat, ylab="beta", xlab="sample",
      col=diverge_hcl(50, h=c(180, 300), c=100, l=c(40,100)), las=1)
```




```
# MANUALLY CHECK sequence of models to be the same as after dredge!!
rightSequence <- c(1,2,3,6,4,7,9,12,5,8,10,13,11,14,15,16)
# this is the order in which rjMCMC-models should occur:
#allmodelnames[rightSequence] # check this against varIndMatrix below
(weightsrjMCMC <- weightsrjMCMC[rightSequence])
```

```
      1     1+2     1+3   1+2+3     1+4   1+2+4   1+3+4 1+2+3+4     1+5
 0.0000  0.0000  0.0000  0.1808  0.0000  0.0000  0.0108  0.0844  0.0000
   1+2+5   1+3+5 1+2+3+5   1+4+5 1+2+4+5 1+3+4+5 1+2+3+4+5
 0.3336  0.0000  0.1840  0.0104  0.1180  0.0112  0.0668
```

The figure indicates estimates (positive in green, negative in red) of model parameters (betas). Parameters are intercept and number 1 to 4 (from bottom to top). Which of the 16 models is fitted is hard to see, as they differ in the combination of parameters, but the intercept is always fitted (top row). Note the trade-off between row "5" and "3": either one is fitted, or the other, but rarely both. Similarly, there seems to be a positive correlation between row "4" and "2", with sign switches.

First we use rjMCMC only to derive model weights and compute the RMSE based on the fitted linear models.

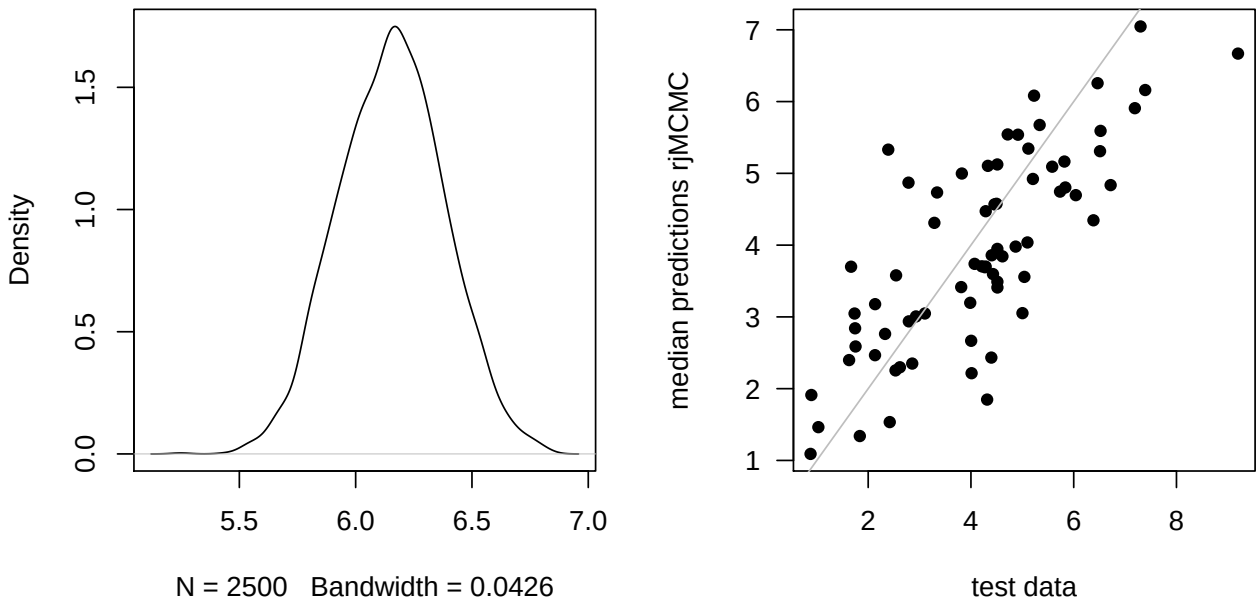
```
weightedPredsrjMCMC <- preds %*% weightsrjMCMC
(RMSErjMCMC <- sqrt(mean((weightedPredsrjMCMC - truth)^2)))
```

```
[1] 1.112931
```

More consistent with the Bayesian setup, and for completeness, here is how one could use the rjMCMC directly. Since the samples are stored, we can use them to make the prediction, yielding 2500 (or so) predictions, from

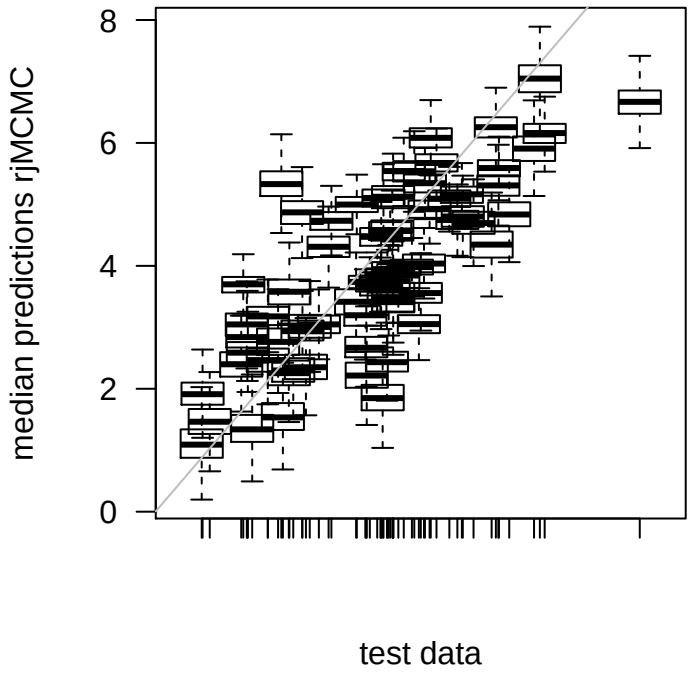
which we shall use the median (assuming that this would be a good summary of the predictions). Note that we do not use the information on the weights, because they are implicit in the beta_mat-values.

```
testpredsrjMCMC <- as.matrix(cbind(1, test[,-1])) %*% t(beta_mat)
par(mfrow=c(1,2), mar=c(5,5,1,1))
# example for prediction to first data point 1 of test data:
plot(density(testpredsrjMCMC[1,]), main="")
plot(truth, apply(testpredsrjMCMC, 1, median), ylab="median predictions rjMCMC",
     xlab="test data", las=1, pch=16)
abline(0,1, col="grey")
```



The left figure show the posterior distribution of a prediction to a single new data point. The right figure compares the test data and the rjMCMC-predictions.

```
boxplot(t(testpredsrjMCMC), at=truth, ylab="median predictions rjMCMC", xlab="test data",
las=1, pch=16, outline=F, names=rep("", length(truth)))
abline(0,1, col="grey")
```



The ticks on the x-axis indicate the position of values, while scaling is the same on both (line indicates 1:1).

```
cor(apply(testpredsrjMCMC, 1, median), truth)
```

```
[1] 0.7738438
```

```
(RMSErjMCMCmedian <- sqrt(mean((apply(testpredsrjMCMC, 1, median) - truth)^2)))
```

```
[1] 1.115324
```

7 Bayes Factor-based weights

```
# first set up an indicator matrix for all models:
varIndMatrix <- ifelse(is.na(mytab[, 1:5]), 0, 1)
varIndMatrix <- varIndMatrix[order(as.numeric(rownames(varIndMatrix))),]
# sort sequence, same as for model.list!

X <- cbind(1, train[1:N, -1]) # only the predictors, plus intercept as first column
Y <- train[1:N, 1]
library(BayesianTools)
likelihood <- function(x, option = 1){
  # a function to switch between the four models and compute the respective likelihood
  # x is a vector with betas, plus a parameter for sd, i.e. NCOL(X)+1
  res = as.matrix(X) %*% (x[-(NCOL(X)+1)] * varIndMatrix[option, ])
  # sets all parameters to 0 that are not in the model
  ll = sum(dnorm(res - Y), sd = x[NCOL(X)+1], log = T)
  return(ll)
}

prior <- function(x){
  # double-exponential/Laplace prior, with lambda set to 10
  # this is a lasso-shrinkage prior
```

```
  ll = sum(dexp(abs(10*x)), log = T)
  return(ll)
}

# For illustration only: setting up two models to be fitted (for
# details see package BayesianTools):
#setup1 = createBayesianSetup(likelihood = function(x) likelihood(x, option  = 1),
#  prior=createPrior(density = prior, lower = c(rep(-5, NCOL(X)), 0.0001), upper =
#  c(rep(5, NCOL(X)),5)))
#setup2 = createBayesianSetup(likelihood = function(x) likelihood(x, option  = 2),
#  prior=createPrior(density = prior, lower = c(rep(-5, NCOL(X)), 0.0001), upper =
#  c(rep(5, NCOL(X)),5)))
# For illustration only: fitting the two models:
#res1 <- runMCMC(setup1, sampler = "Metropolis", settings = list(iterations = 30000,
#                                        startValue = c(rep(0, NCOL(X)), 0.01)))
#res2 <- runMCMC(setup2, sampler = "Metropolis", settings = list(iterations = 30000,
#                                        startValue = c(rep(0, NCOL(X)),0.01)))

# Instead, we loop the setup, and lapply the run:
setups <- list()
for (m in 1:M){
  setups[[m]] <- createBayesianSetup(likelihood = function(x) likelihood(x, option  = m),
                           prior=createPrior(density = prior, lower =
                           c(rep(-5, NCOL(X)), 0.0001), upper = c(rep(5, NCOL(X)),5)))
}
system.time({
resBayesFits <- lapply(setups, runMCMC, sampler = "Metropolis", settings = list(iterations =
                                    40000, startValue = c(rep(0, NCOL(X)), 0.01)))
})
# extract the marginal likelihoods:
ML <- unlist(sapply(resBayesFits, function(x) marginalLikelihood(x)[1]))
names(ML) <- paste0("m", 1:16)
# compute Bayes Factor weights:
```

```
(weightsBF <- exp(ML) / sum(exp(ML)))
```

```
      ln.m       ln.m       ln.m       ln.m       ln.m       ln.m       ln.m       ln.m
0.05665825 0.05189973 0.06428040 0.05348265 0.06070942 0.06195644 0.06002052 0.05209902
      ln.m       ln.m       ln.m       ln.m       ln.m       ln.m       ln.m       ln.m
0.06515604 0.06299156 0.06399811 0.06663114 0.07230987 0.07167333 0.07641655 0.05971697
```

Although we may in a typical setup use the MCMC-samples to derive an averaged model prediction, we here use Bayes factors solely to derive Bayesian model weights, and compute predictions from them for comparison with the other methods. The code above is a good starting place for computing any other posterior parameter of interest.

```
#weightsBF
weightedPredsBF <- preds %*% weightsBF
(RMSEBF <- sqrt(mean((weightedPredsBF - truth)^2)))
```

```
[1] 1.129579
```

Note that rmMCMC and Bayes factor approximate the thing: the probability of model i. That these two approaches do not converge here to the same values is a consequence of different priors and different ways to compute model weights. It is beyond the scope of this study to provide a detailed discussion of how to implement different Bayesian fitting procedures and prior-choices.

8 AIC-based model weights

When we can compute an AIC for each model, $\kappa = -0.5AIC_m$, yielding what has been informally called "Akaike weights". Because our data set is rather small, we use the sample-size corrected AIC (although it is not clear that this is always a good idea: Richards 2007).

```
library(MuMIn)
(AICs <- unlist(sapply(model.list, AICc)))
```

```
       1        2        3        4        5        6        7        8        9       10
285.3700 230.1108 277.3720 215.4013 250.9702 222.1333 220.5231 217.6579 261.3150 215.1921
      11       12       13       14       15       16
241.0757 217.3530 218.5901 217.4694 220.8954 219.7211
```

```
#weights:
(weightsAIC <- exp(-0.5*(AICs-min(AICs)))/sum(exp(-0.5*(AICs-min(AICs)))))
```

```
           1            2            3            4            5            6
1.749322e-16 1.746835e-04 9.541528e-15 2.731395e-01 5.160458e-09 9.430672e-03
           7            8            9           10           11           12
2.109563e-02 8.838091e-02 2.926458e-11 3.032517e-01 7.265200e-07 1.029373e-01
          13           14           15           16
5.545535e-02 9.711856e-02 1.751236e-02 3.150260e-02
```

```
weightedPredsAIC <- preds %*% weightsAIC
(RMSEAIC <- sqrt(mean((weightedPredsAIC - truth)^2)))
```

```
[1] 1.111273
```

9 BIC-based model weights

BIC-weights are fully analogous to AIC-weights, replacing the AIC with the BIC: $\kappa = -0.5BIC_m$. This measure has been proposed as approximating the Bayes factor for large data sets (albeit with an error). It aims at identifying the "truest" model in the set, rather than providing best prediction.

```
(BICs <- unlist(sapply(model.list, BIC)))
```

```
       1        2        3        4        5        6        7        8        9       10
289.6879 236.4926 283.7538 223.7799 257.3520 230.5119 228.9017 227.9629 267.6968 223.5707
      11       12       13       14       15       16
249.4543 227.6580 226.9687 227.7743 231.2004 231.8787
```

```
#weights:
(weightsBIC <- exp(-0.5*(BICs-min(BICs)))/sum(exp(-0.5*(BICs-min(BICs)))))
```

```
           1            2            3            4            5            6
1.698690e-15 6.043835e-04 3.301251e-14 3.482227e-01 1.785455e-08 1.202306e-02
           7            8            9           10           11           12
2.689459e-02 4.300539e-02 1.012519e-10 3.866124e-01 9.262326e-07 5.008840e-02
          13           14           15           16
7.069944e-02 4.725706e-02 8.521363e-03 6.070303e-03
```

Note that the sequence of weights has reversed, simply because the BIC gives a higher penalty to each parameter (log(50)) than the AIC (2).

```
weightedPredsBIC <- preds %*% weightsBIC
(RMSEBIC <- sqrt(mean((weightedPredsBIC - truth)^2)))
```

```
[1] 1.110979
```

10 Mallows' Cp weights

Model avearging based on Mallows' Cp has been shown to be "optimal" for linear models with known weights (see main text). We compute it as follows:

```
(Cps <- sapply(model.list, Cp))
```

```
       1        2        3        4        5        6        7        8        9
234.20670 106.07691 208.37181  85.66710 142.90156  94.31505  92.17030  88.07399 165.66024
      10       11       12       13       14       15       16
 85.41151 123.62426  87.69116  89.65987  87.83707  92.24308  90.20776
```

```
#weights:
(weightsCp <- exp(-0.5*(Cps-min(Cps)))/sum(exp(-0.5*(Cps-min(Cps)))))
```

```
          1           2           3           4           5           6
1.603877e-33 1.067091e-05 6.533489e-28 2.884923e-01 1.076046e-13 3.821701e-03
          7           8           9          10          11          12
1.116820e-02 8.659354e-02 1.229834e-18 3.278200e-01 1.651361e-09 1.048613e-01
         13          14          15          16
3.918461e-02 9.748361e-02 1.076911e-02 2.979500e-02
```

Apparently Mallow's Cp is very similar to AIC.

```
weightedPredsCp <- preds %*% weightsCp
(RMSECp <- sqrt(mean((weightedPredsCp - truth)^2)))
```

```
[1] 1.11157
```

11 Widely applicable information criterion (WAIC)

```
library(blmeco)
waics <- sapply(model.list, function(x) WAIC(glm(x))$WAIC2)
# requires reformulation as glm due to a small programming bug in WAIC
waics
```

```
      1       2       3       4       5       6       7       8       9      10
283.3087 227.5151 275.3416 211.3727 249.0777 219.8672 217.6136 214.3379 258.8516 212.6249
     11      12      13      14      15      16
238.2994 214.1848 216.3114 215.0513 218.6907 217.3689
```

```
(weightsWAIC <- exp(-0.5*(waics-min(waics)))/sum(exp(-0.5*(waics-min(waics)))))
```

```
          1           2           3           4           5           6
1.004280e-16 1.310029e-04 5.393698e-15 4.193255e-01 2.722796e-09 5.997926e-03
          7           8           9          10          11          12
1.850809e-02 9.520609e-02 2.054141e-11 2.241980e-01 5.963469e-07 1.027821e-01
         13          14          15          16
3.549130e-02 6.664236e-02 1.080079e-02 2.091624e-02
```

```
weightedPredsWAIC <- preds %*% weightsWAIC
(RMSEWAIC <- sqrt(mean((weightedPredsWAIC - truth)^2)))
```

```
[1] 1.109224
```

(Note that there are two slightly different ways WAIC computes the number of parameters in the model, pwaic1 and pwaic2, and neither yields the actual rank of the model. We chose the one closer to the model's actual number of parameters, its rank (pwaic2, and accordingly WAIC2). Until WAIC is implemented independently, these values should be taken with a note of caution, as also stated by the authors! Also note that Watanabe's widely applicable *Bayesian* information criterion is identical to BIC for regular statistical model: Watanabe 2013, page 869.)

12 Leave-one-out cross-validation (LOOCV)

AIC supposedly approximates Kullback-Leibler-divergence, as does LOO. The latter has the charm of being applicable also for likelihood-free methods and thus we shall briefly introduce here how to implement it.

```
looAll <- matrix(NA, nrow=N, ncol=M)
for (i in 1:N){
  fm.loo <- lapply(model.list, function(x) update(x, .~., data=train[-i, ]))
  looAll[i,] <- suppressWarnings(sapply(fm.loo, function(x) predict(x,
                                    newdata=train[i, ,drop=F])))
}
head(looAll)
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]     [,9]
[1,] 4.369896 2.867223 4.093642 2.680111 3.040154 3.001784 2.516781 2.692114 3.944494
[2,] 4.352126 5.053333 3.626831 4.424828 5.042496 4.898578 4.198724 4.441183 3.417073
[3,] 4.357351 6.126385 4.057798 5.814249 5.982125 5.892730 5.774946 5.809187 4.471281
[4,] 4.393922 3.300018 5.377329 4.154461 3.001437 3.985675 4.071504 4.179632 5.294302
[5,] 4.387133 2.461055 4.649265 2.749684 3.105246 2.106607 3.297954 2.673890 4.493909
[6,] 4.357326 5.131482 4.831542 5.509323 4.595625 5.633266 5.240173 5.544744 5.005612
        [,10]    [,11]    [,12]    [,13]    [,14]    [,15]    [,16]
[1,] 2.852231 3.993450 2.745515 2.793554 2.860879 2.725044 2.750470
[2,] 4.375784 3.866720 4.385477 4.161481 4.387795 4.160988 4.392709
[3,] 5.910643 5.709681 5.875621 5.922954 5.902205 5.916586 5.870561
[4,] 4.032534 4.095168 4.108838 3.934500 4.066638 3.964398 4.133193
[5,] 2.840006 3.859905 2.806714 3.327252 2.770312 3.325680 2.749380
[6,] 5.394579 4.747374 5.448082 5.135737 5.432106 5.147216 5.476698
```

```
# now choose a criterion for evaluation, e.g. RMSE, compared to omitted data point(s):
RMSE <- apply(looAll, 2, function(x) sqrt(mean((x-Y)^2)))
# or R2:
R2 <- apply(looAll, 2, function(x) cor(x, Y)^2)
# turn into weights:
(weightsRMSE <- (exp(-1*(RMSE-min(RMSE))))/sum(exp(-1*(RMSE-min(RMSE)))))
```

```
 [1] 0.03501430 0.06380730 0.03885629 0.07224235 0.05225455 0.06832852 0.06931172
 [8] 0.07072270 0.04707533 0.07249342 0.05803859 0.07135032 0.07050630 0.07094007
[15] 0.06930340 0.06975484
```

Note that for R^2 a higher value is better, while for RMSE it is worse. Hence, the "-1" in the previous formula turns into a "$+1$" here:

```
(weightsR2 <- (exp((R2-min(R2))))/sum(exp((R2-min(R2)))))
```

```
 [1] 0.09621947 0.06047523 0.03870896 0.06612104 0.05139685 0.06361219 0.06425790
 [8] 0.06519629 0.04673279 0.06627436 0.05617601 0.06558445 0.06502667 0.06533298
[15] 0.06427552 0.06460931
```

```r
# Now compute weighted prediction RMSE-loo:
weightedPredslooRMSE <- preds %*% weightsRMSE
(RMSEloormse <- sqrt(mean((weightedPredslooRMSE - truth)^2)))
```

```
[1] 1.117005
```

```r
# Now compute weighted prediction R2-loo:
weightedPredslooR2 <- preds %*% weightsR2
(RMSElooR2 <- sqrt(mean((weightedPredslooR2 - truth)^2)))
```

```
[1] 1.127831
```

Using LOO is apparently not difficult, and RMSE and R^2 in this simple and non-representative case were good options.

13 Bayesian Model Averaging using Expectation Maximisation (BMA-EM)

BMA-EM is implemented as "ensemble Bayesian model averaging" (EBMA) in the package EBMAforecast. It requires setting up a test-train data set first to avoid giving all weight to the most overfitting approach.

```r
library(EBMAforecast)
set.seed(1)
trainsplit <- sample(rep(c(T, F), N/2))
trainsub1 <- train[trainsplit, ]
trainsub2 <- train[!trainsplit, ]
trainsub1fits <- lapply(model.list, update, .~. , data=trainsub1) # re-fit models on trainsub1
bmafits <- sapply(trainsub1fits, predict, newdata=trainsub2) # predict them to trainsub2
bmaY <- trainsub2$y # to train the EMA-algorithm
EBMAdata <- makeForecastData(.predCalibration=bmafits, .outcomeCalibration=bmaY,
                             .modelNames=paste0("m", 1:M)) # make dataset
EBMAfit <- calibrateEnsemble(EBMAdata, model="normal") # compute weights
EBMAfit@modelWeights
```

```
        m1         m2         m3         m4         m5         m6         m7         m8
0.00000000 0.52851014 0.00000000 0.02880101 0.00000000 0.08378223 0.35890662 0.00000000
        m9        m10        m11        m12        m13        m14        m15        m16
0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
```

```r
weightedPredsEBMA <- preds %*% EBMAfit@modelWeights
(RMSEebma <- sqrt(mean((weightedPredsEBMA - truth)^2)))
```

```
[1] 1.168728
```

14 Naive bootstrap

Here we re-run the models on a bootstrap of the data and count how often a model comes out best. "Best" is quantified by AIC (according to Buckland et al.'s original suggestion).

```r
Nboots <- 1000
bestCounter <- rep(0, M)
for (i in 1:Nboots){
  bsfits <- lapply(model.list, update, .~. , data=train[sample(nrow(train), nrow(train),
```

```
    replace=T),]) # re-fit models on train1
  bsAICs <- sapply(bsfits, AIC)
  bestCounter[which.min(bsAICs)] <- bestCounter[which.min(bsAICs)] + 1
}
(weightsboot <- bestCounter/sum(bestCounter))
```

```
 [1] 0.000 0.001 0.000 0.310 0.000 0.074 0.057 0.022 0.000 0.334 0.000 0.033 0.098 0.047
[15] 0.006 0.018
```

```
weightedPredsBoot <- preds %*% weightsboot
(RMSEboot <- sqrt(mean((weightedPredsBoot - truth)^2)))
```

```
[1] 1.111003
```

Not very convincing result, as already pointed at by Wagenmaker et al. (2004).

15 Stacking

In stacking, we create a train/test data set. The models fitted to the train data then predict to the test, and their predictions are combined using **optimal** model weights (minimising RMSE; this optimisation is not always successful and then a new partitioning is used; the error messages can hence be ignored). The weights thus derived are stored, and the whole procedure is repeated many times. The average weights of many replicates are the final stacking model weights.

```
stacking <- function(test.preds, test.obs){
    # this function computes the optimal weight for a single train/test split;
    # from the models fitted to the training data it uses the predictions to the test;
    # then it optimises the weight vector across the models for combining these
    # predictions to the observed data in the test;
    # trick 1: each weight is between 0 and 1: w <- exp(-w)
    # trick 2: weights sum to 1: w <- w/sum(w)
    #
    # weights are weights for each model, between -infty and +infty!
    # preds are predictions from each of the models

    if (NCOL(test.preds) >= length(test.obs)) stop("Increase the test set! More models
       than test points.")

    # now do an internal splitting into "folds" data sets:
    weightsopt <- function(ww){
        # function to compute RMSE on test data
        w <- c(1, exp(ww)); w <- w/sum(w) ## w all in (0,1) SIMON;
        # set weight1 always to 1, other weights are scaled accordingly
        # (this leads to a tiny dependence of optimal weights on whether model1 is any
        # good or utter rubbish;
        # see by moving the 1 to the end instead -> 3rd digit changes)
        pred <- as.vector(test.preds %*% w)
        return(sqrt(mean((pred - test.obs)^2)))
    }

    ops <- optim(par=runif(NCOL(test.preds)-1), weightsopt, method="BFGS")
    if (ops$convergence != 0) stop("Optimisation not converged!")
    round(c(1, exp(ops$par))/sum(c(1, exp(ops$par))), 4)
}
```

```
Nstack <- 1000
weightsStack <- matrix(NA, ncol=M, nrow=Nstack)
colnames(weightsStack) <- paste0("m", 1:M)
i = 0
while (i < Nstack){
  trainsplit <- sample(rep(c(T, F), floor(N/2)))
  trainsub1 <- train[trainsplit, ]
  trainsub2 <- train[!trainsplit, ]
  trainsub1fits <- lapply(model.list, update, .~. , data=trainsub1) # re-fit models on train1
  stackpreds <- sapply(trainsub1fits, predict, newdata=trainsub2) # predict them to train2
  optres <- try(stacking(test.preds=stackpreds, test.obs=trainsub2$y))
  if (inherits(optres, "try-error"))  next;
  i = i + 1
  weightsStack[i,] <- optres
  rm(trainsplit, trainsub1, trainsub2, trainsub1fits, stackpreds)
  #print(i)
}
# visualise, if you want:
#plot(density(weightsStack[,1], from=0, to=1), las=1, lwd=2, xlim=c(0,1), col="grey20")
#for (j in 2:M) lines(density(weightsStack[,j], from=0, to=1), lwd=2, col=paste0("grey", j*20))
#legend("topright", col=paste0("grey", (1:M)*5), lwd=3, legend=paste0("m", 1:M), cex=1.5, bty="n")
```

```
(weightsStacking <- colSums(weightsStack)/sum(weightsStack))
```

```
         m1          m2          m3          m4          m5          m6          m7
0.001543727 0.089186343 0.011127893 0.191998122 0.011049691 0.108782582 0.068479085
         m8          m9         m10         m11         m12         m13         m14
0.068571486 0.053979234 0.141930255 0.037551350 0.061922971 0.066628453 0.053988834
        m15         m16
0.019538738 0.013721237
```

```
weightedPredsStack <- preds %*% weightsStacking
(RMSEstack <- sqrt(mean((weightedPredsStack - truth)^2)))
```

```
[1] 1.110741
```

That works rather well!

16 Jackknife

The jackknife model averaging optimises the fit of the prediction onto an omitted data point. It is in a way similar to stacking, but requires only N steps (N = number of data points).

```
# 1. fit the candidate models, omitting one data point at a time:
J <- matrix(NA, N, M) # matrix with jackknifed predictions
for (i in 1:N){
  # re-fit models on train with one less data point:
  jfits <- lapply(model.list, update, .~. , data=train[-i,])
  # predict them to omitted data point:
  J[i,] <- sapply(jfits, predict, newdata=train[i, , drop=F])
    rm(jfits)
}
# 2. compute RMSE for a value of w, given J:
weightsopt <- function(ww, J){
```

```
    # function to compute RMSE on test data
    # at some point to also use likelihood instead of RMSE, but primarily for 0/1 data
    w <- c(1, exp(ww)); w <- w/sum(w)
    Jpred <- J %*% w
    return(sqrt(mean((Jpred - train$y)^2)))
}
ops <- optim(par=runif(NCOL(J)-1), weightsopt, method="BFGS", control=list(maxit=5000), J=J)
if (ops$convergence != 0) stop("Not converged!")
round(weightsJMA <- c(1, exp(ops$par))/sum(c(1, exp(ops$par))),3)
```

```
 [1] 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
```

```
weightedPredsJMA <- preds %*% weightsJMA
(RMSEjma <- sqrt(mean((weightedPredsJMA - truth)^2)))
```

```
[1] 1.120528
```

Not exactly glorious, but better than some.

17 Bates-Granger

This proposes to use the prediction error (and the covariance in prediction errors) of the models to estimate weights. The algebraic procedure is fast, but whether the estimates for the covariance is stable has been called into question. Again we have to employ a train/test split in order to estimate prediction errors without giving all weight to overfitted models.

```
set.seed(1)
trainsplit <- sample(rep(c(T, F), floor(N/2)))
trainsub1 <- train[trainsplit, ]
trainsub2 <- train[!trainsplit, ]
trainsub1fits <- lapply(model.list, update, .~. , data=trainsub1) # re-fit models on train1
trainsub2preds <- sapply(trainsub1fits, predict, newdata=trainsub2) # predict them to train2
trainsub2resid <- trainsub2$y - trainsub2preds

Sigma <- cov(trainsub2resid[,-c(4,6,7,8,10:16)])
# PROBLEM: models are nested; removed models manually.
ones <- rep(1, ncol(Sigma))
(weightsBGsome <- solve(t(ones) %*% solve(Sigma) %*% ones)%*%ones%*%solve(Sigma) )
```

```
             1         2        3         5           9
[1,] -0.5439211 0.8658415 0.489955 0.2614798 -0.07335519
```

```
weightsBG <- rep(0, M)
weightsBG[as.numeric(colnames(weightsBGsome))] <- weightsBGsome

weightedPredsBG <- preds %*% weightsBG
(RMSEbg <- sqrt(mean((weightedPredsBG - truth)^2)))
```

```
[1] 1.209345
```

Not bad!

18 Cos-squared weights

Works with correlation in predictions, thus no splitting of the train data required.

```
csweights <- function(R, eps=1E-6, maxit=50, verbose=FALSE){
  # implements Garthwaite & Mubwandarikwa's cos-square scheme (their appendix)
  # eps and maxit are chosen without much testing; not converging within 20 iterations
  # doesn't actually mean that something is wrong; mostly it works much faster,
  # though, i.e. within only a few iterations.
  require(expm)
  D1 <- diag(rep(2, NCOL(R)))
  D2 <- diag(NCOL(R))
  counter = 0
  while (any( abs(diag(D1) - diag(D2)) > eps)){
      ED <- eigen(D1 %*% R %*% D1)
      Q <- ED$vectors
      Lambda <- diag(ED$values)
      ## test:
      #Q %*% Lambda %*% solve(Q) # fine
      Lambda12 <- sqrtm(Lambda)
      E <- solve(D1) %*% Q %*% Lambda12 %*% solve(Q)
      D2 <- D1
      D1 <- diag( diag(Re(E)))
      counter <- counter + 1
      if (verbose) cat(counter, " ")
      if (counter >= maxit){
        warning("Maximum number of iterations reached without convergence!")
        break
      }
  }
  w <- diag(D2)^2 / sum(diag(D2)^2)
  return(w)
}
R <- cor(preds)
R[1:5, 1:8]
```

```
   1           2           3         4          5         6         7         8
1  1          NA          NA        NA         NA        NA        NA        NA
2 NA  1.00000000 -0.01325623 0.8828473  0.9463922 0.9438419 0.8431865 0.8843464
3 NA -0.01325623  1.00000000 0.4579157 -0.2703518 0.2511825 0.4969853 0.4545609
4 NA  0.88284728  0.45791569 1.0000000  0.7144278 0.9571259 0.9830897 0.9997571
5 NA  0.94639216 -0.27035183 0.7144278  1.0000000 0.7865196 0.7010841 0.7124771
```

Here we see that the intercept-only model has a constant prediction and hence no correlation can be computed (yielding NA). We thus drop the intercept-only model from the set of models, i.e. assign a weight of 0.

```
R <- cor(preds[,-1])
(weightsCS <- c(0, csweights(R, verbose=F, maxit=500)))
```

```
 [1] 0.000000e+00 4.065223e-15 2.885132e-01 1.707060e-08 2.409820e-01 2.155152e-01
 [7] 1.254118e-02 1.229909e-15 1.437033e-14 5.706505e-16 2.424484e-01 1.920697e-15
[13] 1.446313e-15 5.283942e-16 6.913051e-15 1.271160e-15
```

```
weightedPredsCS <- preds %*% weightsCS
(RMSEcs <- sqrt(mean((weightedPredsCS - truth)^2)))
```

```
[1] 1.210089
```

Clearly, cos-squared is not ideally suited for nested models. The optimisation takes very long to converge, because the linear combinations of predictions (from nested models) cause problems.

19 Model-based model combinations

In MBMC we use a 'supra-model' to combine the predictions of different models. To avoid overfitting, i.e. the supra-model relying most on the best-fitting, but not necessarily best-predicting model, we again split the data into train1 and train2 to derive the model combination. This model is then applied to the full data, as above.

```
set.seed(1)
trainsplit <- sample(rep(c(T, F), floor(N/2)))
trainsub1 <- train[trainsplit, ]
trainsub2 <- train[!trainsplit, ]
trainsub1fits <- lapply(model.list, update, .~. , data=trainsub1) # re-fit models on trainsub1
mbmcfits <- sapply(trainsub1fits, predict, newdata=trainsub2) # predict them to train2
colnames(mbmcfits) <- paste0("mbmcfits", 1:M)
summary(mbmc1 <- lm(trainsub2$y ~ ., data=as.data.frame(mbmcfits))) # this is a simple linear model
```

```

Call:
lm(formula = trainsub2$y ~ ., data = as.data.frame(mbmcfits))

Residuals:
    Min      1Q  Median      3Q     Max
-2.4928 -0.5698 -0.1120  0.7167  1.8985

Coefficients: (12 not defined because of singularities)
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.35737    1.71827  -1.372    0.180
mbmcfits1         NA         NA      NA       NA
mbmcfits2    0.86584    0.77583   1.116    0.273
mbmcfits3    0.48995    0.67522   0.726    0.474
mbmcfits4         NA         NA      NA       NA
mbmcfits5    0.26148    0.80335   0.325    0.747
mbmcfits6         NA         NA      NA       NA
mbmcfits7         NA         NA      NA       NA
mbmcfits8         NA         NA      NA       NA
mbmcfits9   -0.07336    0.64278  -0.114    0.910
mbmcfits10        NA         NA      NA       NA
mbmcfits11        NA         NA      NA       NA
mbmcfits12        NA         NA      NA       NA
mbmcfits13        NA         NA      NA       NA
mbmcfits14        NA         NA      NA       NA
mbmcfits15        NA         NA      NA       NA
mbmcfits16        NA         NA      NA       NA

Residual standard error: 1.103 on 30 degrees of freedom
Multiple R-squared:  0.6569,    Adjusted R-squared:  0.6112
F-statistic: 14.36 on 4 and 30 DF,  p-value: 1.165e-06
```

```
# alternatively, we can use a machine-learning algorithm, e.g. ANN or randomForest:
library(randomForest)
(mbmc2 <- randomForest(x=mbmcfits, y=trainsub2$y))
```

```
Call:
 randomForest(x = mbmcfits, y = trainsub2$y)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 5

          Mean of squared residuals: 1.172048
                    % Var explained: 61.42
```

There simply isn't much to improve using MBMC in this specific case study: R^2s are relatively low.

```
mbmcpreds <- sapply(trainsub1fits, predict, newdata=test)
colnames(mbmcpreds) <- paste0("mbmcfits", 1:M)
# MBMC prediction with the linear model:
weightedPredsMBMC1 <- predict(mbmc1, newdata=as.data.frame(mbmcpreds))
(RMSEmbmc1 <- sqrt(mean((weightedPredsMBMC1 - truth)^2)))
```

```
[1] 1.128412
```

```
# MBMC prediction with the GAM:
#weightedPredsMBMC1b <- predict(mbmc1b, newdata=as.data.frame(mbmcpreds))
#(RMSEmbmc1b <- sqrt(mean((weightedPredsMBMC1b - truth)^2)))
# MBMC prediction with randomForest:
weightedPredsMBMC2 <- predict(mbmc2, newdata=as.data.frame(mbmcpreds))
(RMSEmbmc2 <- sqrt(mean((weightedPredsMBMC2 - truth)^2)))
```

```
[1] 1.254349
```

Investing into more complicated ways to combine predictions, as with the randomForest, does *not* seem to add benefit over the simple lm. The GAM couldn't be fitted due to nestedness of parameters across the models (for the same reason lm has all the NAs, de facto combining only four models).

20 Repeated evaluation and summary

The RMSE on this specific test data set look like this:

```
RMSEapproaches <- c(RMSEsinglebest, singleRMSEs[16], RMSE1overM, RMSEmedian, RMSErjMCMC,
                    RMSErjMCMCmedian, RMSEBF, RMSEAIC, RMSEBIC, RMSECp, RMSEWAIC,
                    RMSEloormse, RMSElooR2,RMSEebma,RMSEboot,RMSEstack, RMSEjma, RMSEbg,
                    RMSEcs, RMSEmbmc1, RMSEmbmc2)
results <- sapply(RMSEapproaches, round, 3)
names(results) <- c("run's best", "full", "1/M", "median", "rjMCMC", "rjMCMCmedian",
                    "BayesFactor", "AIC", "BIC", "Cp", "WAIC", "LOO-CV rmse", "LOO-CV R2",
                    "BMA-EM", "boot", "stacking", "JMA", "Bates-Granger", "cos-squared",
                    "MBMClm", "MBMCrf")
sort(results, decreasing=F)
```

```
   run's best        median          WAIC           AIC           BIC          boot
        1.105         1.107         1.109         1.111         1.111         1.111
     stacking            Cp       rjMCMC          full  rjMCMCmedian   LOO-CV rmse
```

```
        1.111         1.112         1.113         1.114         1.115         1.117
          JMA           1/M     LOO-CV R2        MBMClm   BayesFactor        BMA-EM
        1.121         1.127         1.128         1.128         1.130         1.169
Bates-Granger   cos-squared        MBMCrf
        1.209         1.210         1.254
```

Now we can repeat all these analyses a few times to see what the overall ranking in a situation like this would be. This takes quite a few hours, mind you!

In addition to the 16 models and the 19 weighting approaches, we add the following comparisons:

- run's best: the best-performing model for each specific run (which will differ between replicates);
- all single models (m1 – m16);
- the full model (identical to m16, hence we omit m16 from the analysis);
- the direct rjMCMC computation of the averaged prediction (which is the median of the posterior across models), in addition to using rjMCMC-derived model weights.

```
R <- 100 # put this to at least 100
M <- 16 # number of models
weights.arr <- array(NA, dim=c(R, 21+M, M)) # repeats, approaches, models
results.mat <- matrix(NA, nrow=R, ncol=21+M)
colnames(results.mat) <- c("run's best", "full", "1/M", "median", "rjMCMC", "rjMCMC median",
                           "Bayes Factor", "AIC", "BIC", "Cp", "WAIC", "LOO-CV rmse", "LOO-CV R2",
                           "BMA-EM", "boot", "stacking", "JMA", "Bates-Granger", "cos-squared",
                           "MBMC lm", "MBMC rf", paste0("m", 1:M))
# "run's best" is the best model for each replicate, so a different model each time.
for (r in 1:R){
  set.seed(r)
  N <- 70 # number of data points
  ...
  results <- sapply(RMSEapproaches, round, 3)
  results.mat[r,] <- results
}
```

We omit the R-code from the html document here, which is exactly as above.

Let's have a look how much the weights actually vary, on average, between methods:

```
weights.mat <- apply(weights.arr, c(2,3), mean, na.rm=T)
weights.mat[22:37, ] <- diag(1, 16) # the 16 single models
rownames(weights.mat) <- colnames(results.mat)
colnames(weights.mat) <- paste0("m", 1:16)
round(weights.mat, 3)
```

Next, we summarise the results and sort by lowest RMSE.

```
sort(apply(results.mat, 2, median), decreasing=F)
```

```
   run's best rjMCMC median           BIC        median           m10        rjMCMC
     1.063065      1.069285      1.073677      1.075439      1.075833      1.076162
         boot          WAIC           AIC            Cp      stacking           m14
     1.076196      1.076510      1.076636      1.077738      1.078791      1.079108
          JMA           m13            m4            m8            m7           m15
     1.079277      1.079653      1.080117      1.080845      1.082479      1.084068
         full           m16           m12        BMA-EM  Bayes Factor           1/M
     1.085747      1.085747      1.086834      1.103869      1.105911      1.109704
    LOO-CV R2   LOO-CV rmse       MBMC lm            m6       MBMC rf            m2
     1.109730      1.122975      1.134644      1.152553      1.181280      1.206159
```

```
Bates-Granger   cos-squared          m11           m5           m9           m3
     1.206202      1.209085     1.263535     1.360272     1.470197     1.587185
           m1
     1.677216
```

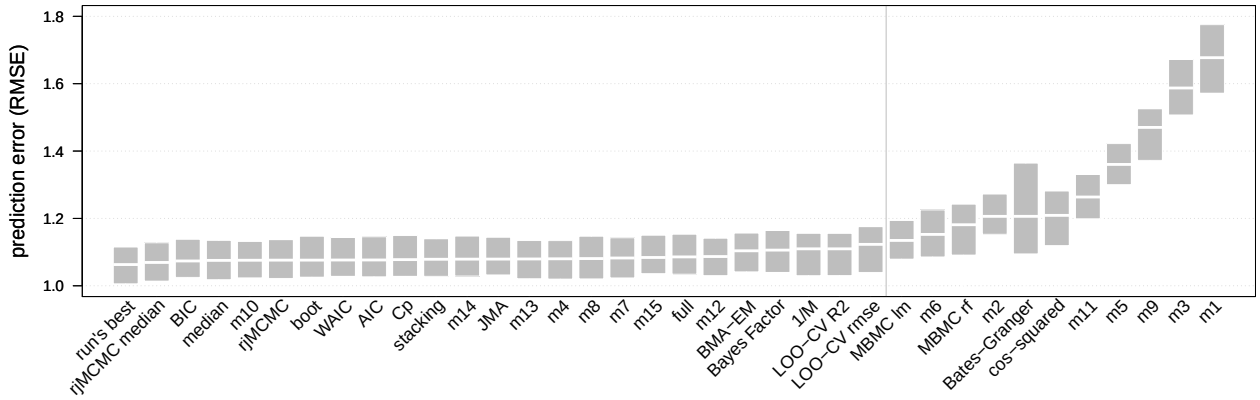
```
sort(apply(results.mat, 2, sd), decreasing=F)
```

```
   run's best           m4          m12       median         boot          m10
   0.08881114   0.08885856   0.08913990   0.09030180   0.09087613   0.09093199
         full          m16           m8     stacking         WAIC           Cp
   0.09094871   0.09094871   0.09107277   0.09114004   0.09133095   0.09150290
          AIC          JMA       rjMCMC          BIC rjMCMC median          m14
   0.09152372   0.09168529   0.09191871   0.09197703   0.09216574   0.09299313
           m7          m15       BMA-EM           m6          m13  Bayes Factor
   0.09308613   0.09312510   0.09385217   0.09412491   0.09633670   0.09722074
          1/M     LOO-CV R2   LOO-CV rmse      MBMC lm           m2           m9
   0.09730378   0.09735920   0.09960649   0.10134090   0.10315366   0.10909616
          m11           m5           m3  cos-squared      MBMC rf           m1
   0.10924339   0.11190077   0.11377994   0.11429703   0.11788296   0.12726591
Bates-Granger
   1.36874598
```

Note that m16 is the same as the full model, so we remove m16 from further analysis.

```
library(xtable)
out <- round(apply(weights.arr, c(2,3), mean, na.rm=T), 3)
out[22:37, ] <- diag(1, 16) # the single models
oo <- order(apply(results.mat, 2, median))[-c(20)] # rm m16 = full model
out.ordered <- cbind(out, apply(results.mat, 2, median))[oo, ]
print(xtable(out.ordered, digits=c(1,rep(2, 16),3)), file="case1weights.txt")
```

```
oo <- order(apply(results.mat, 2, median))[-c(20)]
par(mar=c(8,4.5,1,1))
boxplot(results.mat[, oo], las=2, col="grey", border="white", ylab="prediction error (RMSE)",
        cex.lab=1.3, ylim=c(1, 1.8), show.names=F, xlim=c(1, 36))
abline(v=25.5, col="grey")
abline(h=c(1,1.2,1.4,1.6,1.8), col="lightgrey", lty=3)
boxplot(results.mat[, oo], las=2, col="grey", border="white", ylab="prediction error (RMSE)",
        cex.lab=1.3, ylim=c(1, 1.8), add=T, show.names=F) # to plot on top of grid
text((1:36)+0.2, .95, labels=colnames(results.mat[,oo]), srt=45, xpd=T, adj=1, cex=1.2)
```



Methods to the right of the vertical line are very similar in predictive performance. Note that "run's best" refers to a different model at each run, while m10 is the best model across all runs. M10 is the model with

predictors p1 and p4.

In summary, the above code shows how a variety of model averaging approaches can be implemented for likelihood-based models (and many of them also for models without a likelihood). It was not the aim of this appendix to provide an extensive evaluation of the approaches. For example, rjMCMC and Bayes Factor approximate the same goal and should yield near-identical results. This is not a problem of the method as such, only of our automatised handling of it.

Similar, methods that require a train-test split (stacking, BMA-EM, Bates-Granger, MBMCs) may perform better with different splitting regimes than our half-half. Such optimisation has to be explored by simulations beyond the scope of our review.