Best orthogonalized subset selection (BOSS)

Sen Tian

2019-10-23

Installation

We maintain a Github page for the package and keep the most updated version there. To install, simply run the following commands in the console:

```
library(devtools)
install_github(repo="sentian/BOSSreg", subdir="r-package")
```

A stable version can be installed from CRAN using

```
install.packages(repo="BOSSreg", repos = "http://cran.us.r-project.org")
```

Introduction

BOSS is a least squares-based subset selection method. It is based on takes the following steps:

- order the predictors based on their partial correlations with the response;
- perform best subset regression upon the orthogonal basis of the ordered predictors;
- transform the coefficients back to the original space;
- choose the optimal solution using the selection rule AICc-hdf.

The hdf is a heuristic degrees of freedom for BOSS that can be plugged into a selection rule such as AICc-hdf, which can then be used as a selection rule for BOSS. AICc-hdf is defined as

$$\text{AICc-hdf} = n \log\left(\frac{\text{RSS}}{n}\right) + n\frac{n + \text{hdf}}{n - hdf - 2}.$$

More details can be referred to Tian et al. (2019).

This vignette is structured as follows. We start by simulating a dataset. We then introduce the components, functionalities and basic usage of the package. This is followed by a discussion contrasting BOSS and forward stepwise regression (FS). Finally, we study real data examples and compare BOSS with some popular regularization methods.

Simulated datasets

The model generating mechanism is $y = X\beta + \epsilon$. We consider a sparse model where only a few predictors matter, with a high signal-to-noise ratio. The detailed parameters are given as follows:

```
n = 200 # number of observations
p = 14 # number of predictors
p0 = 6 # number of active predictors (beta_j=0)
rho = 0.9 # correlation between predictors
nrep = 1000 # number of replications of y to be generated
SNR = 7 # signal-to-noise ratio
seed = 65 # the seed for reproducibility
```

We make the predictors with $\beta_j \neq 0$ pairwisely correlated with opposite effects. We generate 1000 replications of the response y are generated by fixing X. We assume the columns of X and y have 0 mean, so we can exclude the intercept term from model fitting.

```
library(MASS)
# function to generate the data
# columns of X have mean 0 and norm 1, y has mean 0
simu.data <- function(n, p, p0, rho, nrep, SNR, seed){
  # true beta
  beta = rep(0,p)
  beta = c(rep(c(1,-1),p0/2), rep(0,p-p0))
  names(beta) = paste0('X', seq(1,p))
  # covariance matrix
  covmatrix  = matrix(0,nrow=p,ncol=p)
  diag(covmatrix) = 1
  for(i in 1:(p0/2)){
    covmatrix[2*i-1,2*i] = covmatrix[2*i,2*i-1] = rho
  }
  # generate the predictors given the correlation structure
  set.seed(seed)
  x = mvrnorm(n,mu=rep(0,p),Sigma=covmatrix)
  x = scale(x,center=TRUE,scale=FALSE)
  colnorm = apply(x,2,function(m){sqrt(sum(m^2))})
  x = scale(x,center=FALSE,scale=colnorm) # standardization
  # sigma calculated based on SNR
  sd = sqrt(t(beta/colnorm)%*%covmatrix%*%(beta/colnorm) / SNR)
  mu = x%*%beta
  # generate replications of y by fixing X
  y = matrix(rep(mu,each=nrep),ncol=nrep,byrow=TRUE) +
    scale(matrix(rnorm(n*nrep,mean=0,sd=sd),nrow=n,ncol=nrep),center=TRUE,scale=FALSE)
  return(list(x=x, y=y, beta=beta, sigma=sd))
}
dataset = simu.data(n, p, p0, rho, nrep, SNR, seed)
x = dataset$x
y = dataset$y
beta = dataset$beta
mu = x%*%beta
sigma = dataset$sigma
```

The first $p_0 = 6$ predictors are active with $\beta_j \neq 0$.

```
print(beta)
#>  X1  X2  X3  X4  X5  X6  X7  X8  X9 X10 X11 X12 X13 X14
#>   1  -1   1  -1   1  -1   0   0   0   0   0   0   0   0
```

An illustration of the package

Fitting the model is simple.

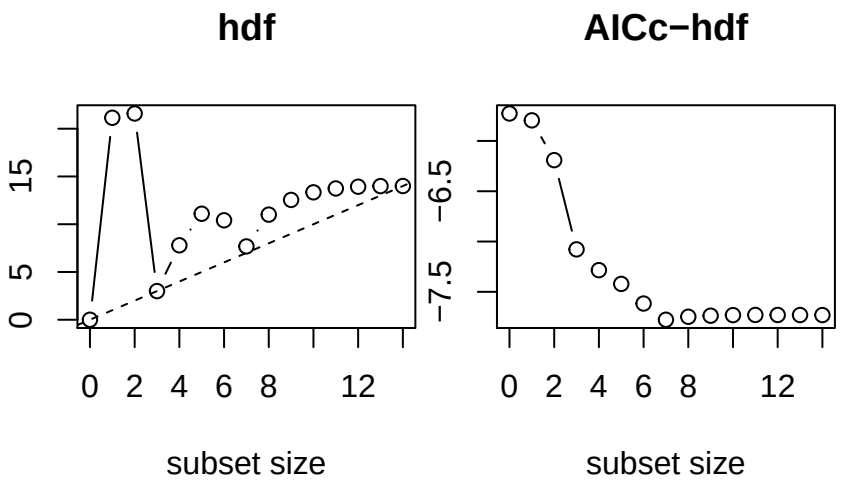
```
library(BOSSreg)
# choose a single replication as illustration
rep = seed
# fit the model
boss_model = boss(x, y[,rep], intercept = FALSE)
```

The 'boss' object contains estimated coefficient vectors for the entire solution paths of both BOSS and FS.

```
betahat_boss = boss_model$beta_boss
betahat_fs = boss_model$beta_fs
print(dim(betahat_boss))
#> [1] 14 15
```

By default, it also provides the hdf for BOSS and multiple information criteria.

```
# the heuristic degrees of freedom
plot(0:p, boss_model$hdf, main='hdf', ylab='', xlab='subset size', type='b')
abline(0, 1, lty=2)
# AICc-hdf
plot(0:p, boss_model$IC_boss$aicc, main='AICc-hdf', ylab='', xlab='subset size', type='b')
```



The optimal estimated coefficient vector and fitted mean vector can be obtained as follows.

```
# the default is chosen by AICc
betahat_aicc = coef(boss_model)
muhat_aicc = predict(boss_model, newx=x)
# use Cp rather than AICc
betahat_cp = coef(boss_model, ic='cp')
muhat_cp = predict(boss_model, newx=x)
```

In addition to information criteria, K-fold cross-validation (CV) with multiple replications can be used as a selection rule, with 10-fold CV with 1 replication the default choice.

```
# the default is 10-fold CV with 1 replication
set.seed(seed)
boss_cv_model = cv.boss(x, y[,rep], intercept=FALSE)
# coefficient vector selected by minimizing CV error
betahat_cv = coef(boss_cv_model)
# fitted values
muhat_cv = predict(boss_cv_model, newx=x)
```

Calling 'cv.boss' runs CV for FS as well.

```
# coefficient vector for FS selected by CV
betahat_fs_cv = coef(boss_cv_model, method='fs')
# fitted values
muhat_fs_cv = predict(boss_cv_model, newx=x, method='fs')
```

Here is a comparison of the coefficient vectors selected using different selection rules. The first three columns are for BOSS while the last column is for FS.

```
tmp = cbind(betahat_aicc, betahat_cp, betahat_cv, betahat_fs_cv)
dimnames(tmp) = list(dimnames(tmp)[[1]], c('boss_aicc', 'boss_cp', 'boss_cv', 'boss_fs_cv'))
print(tmp)
#> 14 x 4 sparse Matrix of class "dgCMatrix"
#>       boss_aicc     boss_cp     boss_cv  boss_fs_cv
#> X1   1.10223490  1.10223490  1.10223490  1.10284826
#> X2  -1.08347551 -1.08347551 -1.08347551 -1.08515601
#> X3   0.98945399  0.98945399  0.98945399  0.99385748
#> X4  -0.98187488 -0.98187488 -0.98187488 -0.98686091
#> X5   0.98030660  0.98030660  0.98030660  0.98393430
#> X6  -0.94454349 -0.94454349 -0.94454349 -0.94536470
#> X7   .           .           .           .
#> X8   .           .           .          -0.02397828
#> X9  -0.01841478 -0.01841478 -0.01841478 -0.01863818
#> X10  .           .           .           .
#> X11  .           .           .           .
#> X12  .           .           .           .
#> X13  .           .           .           .
#> X14  .           .           .           .
```

Comparing the solution paths of BOSS and FS

We see that FS gives a denser solution than BOSS in this case. Under the specific design of the true model, the true active predictors $(X_1, \cdots, X_6)$ are pairwisely correlated with opposite effects. Predictors (e.g. (X_1, X_2)) together lead to a high R^2 but each single one of them contributes little. As a result, FS can have trouble stepping in the true active predictors in an early stages. For example, as indicated below, the inactive predictor X_{12} joins in the first step. On the contrary, BOSS takes the same order of predictors as FS, and performs best subset regression on their orthogonal basis, providing the chance to re-evaluate (or re-order) each predictor.

```
# X9 joins first
print(boss_model$steps_x)
#>  X9  X4  X3  X5  X6  X1  X2  X8 X13 X12 X10 X14 X11  X7
#>   9   4   3   5   6   1   2   8  13  12  10  14  11   7
```

Let's set aside the selection rule for now, and compare the solution paths of the two methods. We calculate the average RMSE at each subset size based on 1000 replications. The RMSE is defined as

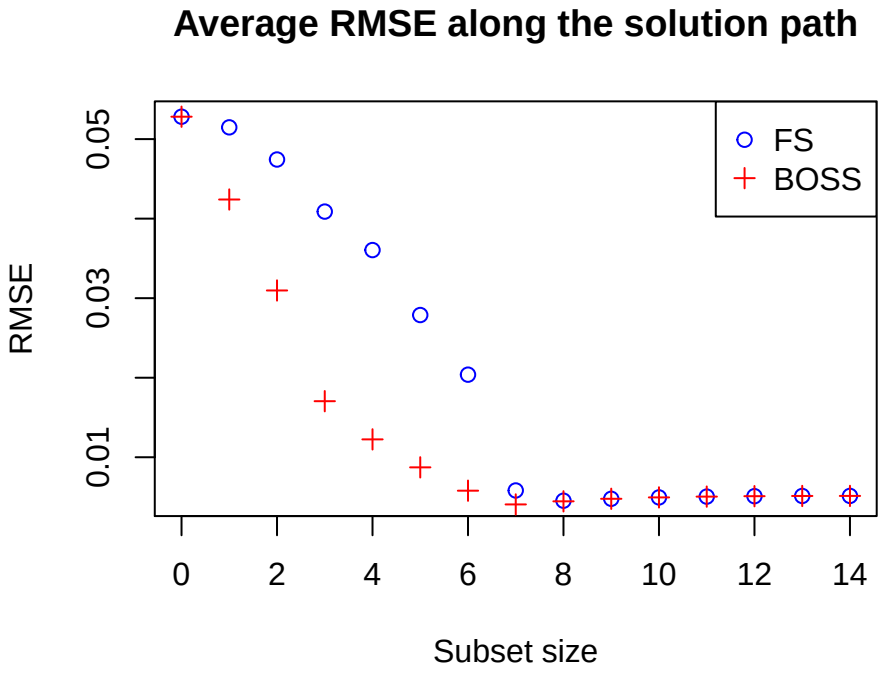
$$\text{RMSE} = \sqrt{\frac{1}{n}\|\hat{\mu} - X\beta\|_2^2}.$$

```
# function to calculate RMSE
calc.rmse <- function(muhat){
  sqrt( Matrix::colSums(sweep(muhat, 1, mu)^2) / n )
}
rmse_solutionpath = list(boss=list(), fs=list())
rmse = nvar = list(boss=c(), fs=c())
for(rep in 1:nrep){
  boss_cv_model = cv.boss(x, y[,rep], intercept=FALSE)
  # RMSE along the solution path
  rmse_solutionpath[['boss']][[rep]] = calc.rmse(x %*% boss_cv_model$boss$beta_boss)
```

```
  rmse_solutionpath[['fs']][[rep]] = calc.rmse(x %*% boss_cv_model$boss$beta_fs)
  # RMSE for the optimal subset selected via a selection rule
  rmse[['boss']][rep] = calc.rmse(predict(boss_cv_model$boss, newx = x)) # AICc
  rmse[['fs']][rep] = calc.rmse(predict(boss_cv_model, newx = x, method = 'fs')) # CV
  # number of variables
  nvar[['boss']][rep] = sum(coef(boss_cv_model$boss)!=0)
  nvar[['fs']][rep] = sum(coef(boss_cv_model, method='fs')!=0)
}
```

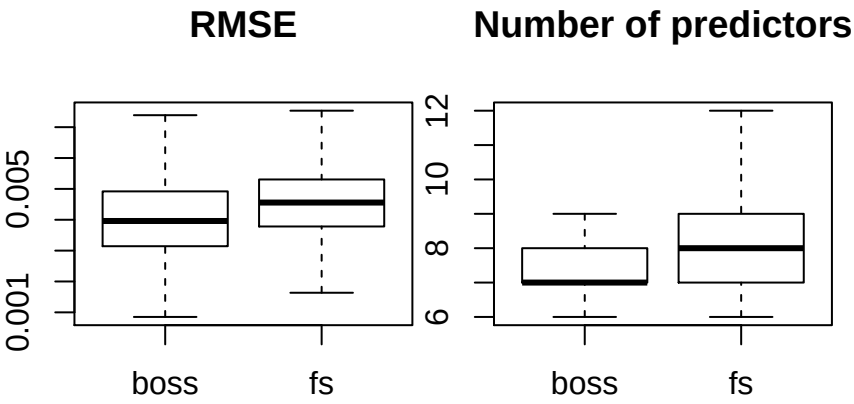
BOSS clearly provides a better solution path than FS in steps less than 8.

```
# average RMSE over replications
rmse_avg = lapply(rmse_solutionpath, function(xx){colMeans(do.call(rbind, xx))})
plot(0:p, rmse_avg$fs, col='blue', pch=1, main='Average RMSE along the solution path',
     ylab='RMSE', xlab='Subset size')
points(0:p, rmse_avg$boss, col='red', pch=3)
legend('topright', legend = c('FS', 'BOSS'), col=c('blue', 'red'), pch=c(1,3))
```



Next, we bring back the selection rule and compare their performances. The selection rule for BOSS is AICc-hdf and 10-fold CV for FS. BOSS shows a better predictive performance, and it provides sparser solutions than FS.

```
boxplot(rmse, outline=FALSE, main='RMSE')
boxplot(nvar, outline=FALSE, main='Number of predictors')
```



Real data examples

We compare the performance of BOSS with FS and some popular regularization methods on several real datasets. We consider four datasets from the StatLib library, 'boston housing', 'hitters', 'auto' and 'college'. An intercept term is included in all of the procedures. We present the results in this section and provide the code in the Appendix at the end of this document.

The selection rule is AICc for BOSS and LASSO, and 10-fold CV for FS and SparseNet. We use **R** packages *glmnet* and *sparsenet* to fit LASSO and SparseNet, respectively. We see that BOSS has the minimum RMSE for the 'hitters' and 'auto' datasets, while LASSO has the minimum RMSE for the 'boston housing' and 'college'. Due to an efficient implementation of the cyclic coordinate descent, the 'glmnet' algorithm provides an extremely fast LASSO solution. BOSS is also considerably computationally efficient, and is much faster than the remaining methods.

Dataset	n, p	Metrics	BOSS	FS	LASSO	SparseNet
boston	506, 13	RMSE	3.372	3.370	3.363	3.370
		# predictors	12.004	12.012	12.012	12.000
		running time (s)	0.028	0.248	0.005	0.267
hitters	263, 19	RMSE	233.853	237.335	234.064	235.242
		# predictors	11.152	10.586	14.205	12.783
		running time (s)	0.014	0.101	0.005	0.357
college	777, 17	RMSE	1565.476	1570.892	1564.807	1566.278
		# predictors	17.991	16.116	16.008	15.407
		running time (s)	0.095	0.835	0.007	0.482
auto	392, 6	RMSE	2.628	2.628	2.643	2.629
		# predictors	3.000	3.000	5.008	3.010
		running time (s)	0.009	0.079	0.004	0.156

Appendix: Code for the real data examples

The following code is used to pre-process the datasets. We remove all of the entries with 'NA' values. We recast binary categorical variables into $\{0, 1\}$ and remove categorical variables with more than two categories.

```
library(ISLR)
dataset = list()
# Boston Housing data
tmp = Boston
tmp = na.omit(tmp)
tmp$chas = as.factor(tmp$chas)
dataset$boston$x = data.matrix(tmp[,!names(tmp) %in% 'medv'])
```

```
dataset$boston$y = tmp$medv

# MLB hitters salary
tmp = Hitters
tmp = na.omit(tmp)
tmp[,c('League', 'Division', 'NewLeague')] =
  lapply(tmp[,c('League', 'Division', 'NewLeague')], as.factor)
dataset$hitters$x = data.matrix(tmp[,!(names(tmp) %in% c('Salary'))])
dataset$hitters$y = tmp$Salary

# # College data
tmp = College
tmp$Private = as.factor(tmp$Private)
dataset$college$x = data.matrix(tmp[,!(names(tmp) %in% c('Outstate'))])
dataset$college$y = tmp$Outstate

# Auto data
tmp = Auto
dataset$auto$x = data.matrix(tmp[,!(names(tmp) %in% c('mpg','name','origin'))])
dataset$auto$y = tmp$mpg
```

Code to calculate leave-one-out error, number of predictors and timing for each fitting procedure. Note that the following code thats around 20 minutes to run on a single core of a local machine with a 2.7 GHz i7 processer and 16 GB RAM.

```
library(glmnet)
library(sparsenet)
rmse <- function(y_hat, y){
  sqrt(sum( (y_hat - y)^2 / length(y)) )
}
rdresult <- function(x, y, nrep){
  p = dim(x)[2]

  allmethods = c('lasso','sparsenet','boss','fs')
  error = numvar = time = replicate(length(allmethods), rep(NA,nrep), simplify=F)
  names(error) = names(numvar) = names(time) = allmethods

  set.seed(seed)
  for(i in 1:nrep){
    index = 1:nrow(x)
    index = index[-i]

    x.train = x[index, , drop=FALSE]
    y.train = y[index]
    x.test = x[-index, , drop=FALSE]
    x.test.withint = cbind(rep(1,nrow(x.test)), x.test)
    y.test = y[-index]

    # BOSS
    ptm = proc.time()
    boss_model = boss(x.train, y.train, intercept = TRUE)
    time_tmp = proc.time() - ptm
    boss_pred = as.numeric( predict(boss_model, newx=x.test) )
    error$boss[i] = rmse(boss_pred, y.test)
```

```
    numvar$boss[i] = sum(coef(boss_model)!=0)
    time$boss[i] = time_tmp[3]

    # FS
    ptm = proc.time()
    boss_cv_model = cv.boss(x.train, y.train)
    time_tmp = proc.time() - ptm
    fs_pred = as.numeric( predict(boss_cv_model, newx=x.test, method='fs') )
    error$fs[i] = rmse(fs_pred, y.test)
    numvar$fs[i] = sum(coef(boss_cv_model, method='fs')!=0)
    time$fs[i] = time_tmp[3]

    # LASSO
    ptm = proc.time()
    lasso_model = glmnet(x.train, y.train, intercept=TRUE)
    lasso_aicc = as.numeric(calc.ic(predict(lasso_model, newx=x.train), y.train,
                                    ic='aicc', df=lasso_model$df+1))
    lasso_pred = predict(lasso_model, newx=x.test, s=lasso_model$lambda[which.min(lasso_aicc)])
    time_tmp = proc.time() - ptm
    error$lasso[i] = rmse(lasso_pred, y.test)
    numvar$lasso[i] = sum(coef(lasso_model, s=lasso_model$lambda[which.min(lasso_aicc)])!=0)
    time$lasso[i] = time_tmp[3]

    # Sparsenet
    ptm = proc.time()
    sparsenet_cv_model = cv.sparsenet(x.train, y.train)
    time_tmp = proc.time() - ptm
    sparsenet_pred = predict(sparsenet_cv_model, newx=x.test, which='parms.min')
    error$sparsenet[i] = rmse(sparsenet_pred, y.test)
    numvar$sparsenet[i] = sum(coef(sparsenet_cv_model, which='parms.min')!=0)
    time$sparsenet[i] = time_tmp[3]
  }
  return(list(error=error, numvar=numvar, time=time))
}
result = lapply(dataset, function(xx){rdresult(xx$x, xx$y, nrow(xx$x))})
```

This is the code to construct the table on page 6.

```
library(knitr)
library(kableExtra)
# function to extract the results
tmp_function <- function(method){
  unlist(lapply(result, function(xx){
    unlist(lapply(xx, function(yy){
      round(mean(yy[[method]]), 3)
      }))
    }))
}
tmp = data.frame(Dataset = rep(names(result), each=3),
  n_p = rep(unlist(lapply(dataset, function(xx){paste(dim(xx$x), collapse = ', ')})) , each=3),
  Metrics = rep(c('RMSE', '# predictors', 'running time (s)'), length(result)),
  BOSS = tmp_function('boss'),
  FS = tmp_function('fs'),
  LASSO = tmp_function('lasso'),
```

```
  SparseNet = tmp_function('sparsenet'))
rownames(tmp) = NULL
colnames(tmp)[2] = 'n, p'
kable(tmp, align = "c") %>%
  kable_styling(full_width = F) %>%
  column_spec(1, bold = T) %>%
  collapse_rows(columns = 1:2, valign = "middle")
```