Linear regression with correlated predictors

Jean Morrison

January 3, 2017

In this document I walk through the simulations in Section 3.1 and Appendix Section 3 of "Rank conditional coverage and confidence intervals in high dimensional problems" by Jean Morrison and Noah Simon. We make use of the `rcc` and `rccSims` packages which accompanies the paper and can be found at github.com/jean997/rcc and github.com/jean997/rccSims.

Problem Set-up

We imagine a case similar to a genome-wide association or an expression study. For each of 100 individuals, we have measured 1000 features. In a GWAS these would be genetic variants, while in an expression study these would be the abundances of different gene transcripts. In this simulation, the 1000 features are grouped into 100 blocks, each containing 10 features. Features within a block have a pairwise correlation of ρ – for this walk-through we use $\rho = 0.3$ but results are presented in the paper for $\rho = -0.1, 0, 0.3$ and 0.8. We will show code for generating the full set of results at the end. In each block of features, there is one feature that directly influences the outcome, y. The effect sizes for these features are drawn from a normal distribution.

To illustrate the simulations, we will first generate one data set and calculate several different sets of confidence intervals for the prameters.

```
library(MASS)
library(ggplot2)
library(rcc)
library(rccSims)
set.seed(1e7)
n.samp <- 100
n.block <- 100

#Covariance matrix for each block of features
s <- matrix(0.3, nrow=10, ncol=10)
diag(s) <- 1
#Effect sizes for each feature
beta <- list()
#for(i in 1:n.block) beta[[i]] <- rep(c(0, rnorm(n=100), 0), c(400, rep(1, 100), 500))
for(i in 1:n.block){
  beta[[i]] <- rep(0, 10)
  beta[[i]][5] <- rnorm(n=1)
}

#Generate correlated features
xs <- lapply(1:n.block, FUN=function(k){
      nk <- length(beta[[k]])
      mvrnorm(n=n.samp, mu = rep(0, nk), Sigma = s)
})
X <- do.call(cbind, xs)
#Generate outcome
y <- X%*% unlist(beta) + rnorm(n=n.samp, sd=1)
```

Parameter estimation

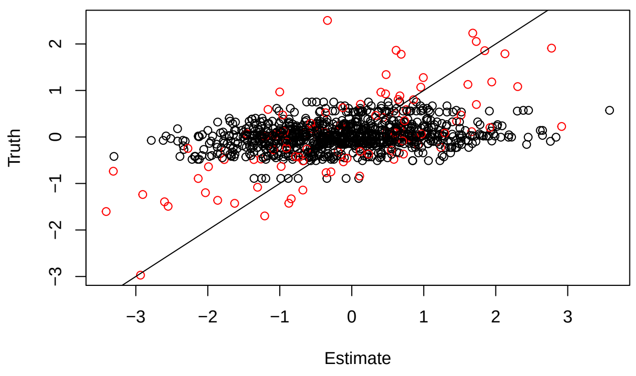
For each feature, we estimate the marginal association between the feature and the outcome using linear regression. For convenience (and some efficiency gain) we use the `many_lr` function in the `rccSims` package which is a convenience utility for running many single variable linear regressions.

```r
f_marg <- rccSims:::many_lr(y,X)
head(f_marg)
```

```
##     beta_hat    se_hat
## 1  0.8969749 0.9775843
## 2 -0.2651403 0.9404450
## 3  0.4854244 0.9898717
## 4  0.3684987 1.0274091
## 5  0.6158628 0.9571941
## 6  0.8207306 0.9952321
```

We can also calculate the true marginal association for each feature:

```r
truth <-  c()
for(k in 1:n.block){
    truth <- c(truth, s%*%beta[[k]])
}
f_marg$truth <- truth
plot(f_marg$beta, f_marg$truth, xlab="Estimate", ylab="Truth", col=rep(rep(c(1, 2, 1), c(4, 1, 5)), 10)
abline(0, 1)
```



Block-Based Ranking Scheme

We will use two different ranking schemes. In Section 3.1 we present results for a scheme where parameters are ranked simply on the absolute value of the test statistic. In Appendix Section 3 we discuss a ranking scheme where we first choose the most significant parameter in each block and then rank only these selected parameters.

To genmerate bootstrap confidence intervals, we will need a function that implements this block based ranking scheme:

```r
rank_block <- function(stats, use.abs, blocks){
  p <- length(stats)
  b <- unique(blocks)
  N <- length(b)
  if(use.abs) stats <- abs(stats)
  rank <- rep(NA, p)
  top_ix_block <- t(sapply(b, FUN=function(blk){
    ix <- which(blocks==blk)
    ixmax <- which.max(stats[ix])
    return(c(ix[ixmax], max(stats[ix])))
  }))
  o <- order(top_ix_block[,2], decreasing=TRUE)
  j <- top_ix_block[order(top_ix_block[,2], decreasing=TRUE),1]
  rank <- match(1:p, j)
  return(list("order"=j, "rank"=rank))
}
```

In general, `par_bs_ci` and `nonpar_bs_ci` in the `rcc` package can accept any ranking function that takes test statistics as the first arguement and `use.abs` as the second argument — `use.abs` indicates that ranking should be bsaed on the absolute value of the test statistic. The defualt ranking used by these functions is just the size of the (absolute) statistics. Ranking functions may accept additional arguments as well.

Here we get both the usual ranking and the block-based ranking for our parameter estimates:

```
blocks <- rep(1:n.block, each=10)
rnk_usual <- rcc:::basic_rank(f_marg$beta_hat/f_marg$se_hat, use.abs=TRUE)
rnk_block <- rank_block(f_marg$beta_hat/f_marg$se_hat, use.abs=TRUE, blocks=blocks)
f_marg$rank_block <- rnk_block$rank
block_ix <- which(!is.na(f_marg$rank_block))
f_marg$rank_usual <- rnk_usual$rank
```

Naive confidence intervals

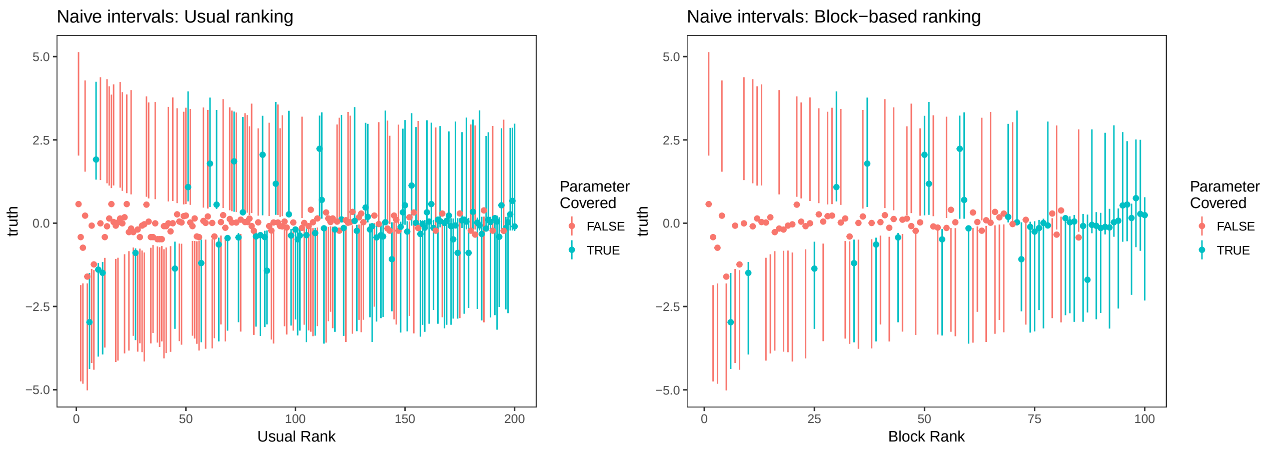
First we calculate the naive confidence intervals which don't depend on the ranking:

```
ci.naive <- cbind(f_marg$beta_hat - f_marg$se_hat*qnorm(0.95), f_marg$beta_hat + f_marg$se_hat*qnorm(0.9
sum(ci.naive[,1] <= f_marg$truth & ci.naive[,2] >= f_marg$truth, na.rm=TRUE)/1000
```

```
## [1] 0.876
```

Here we plot the naive intervals vs. both ranks

```
plot_cis(f_marg$rank_usual, ci.naive, f_marg$truth, plot.truth = TRUE, prop=0.2) + xlab("Usual Rank") +
plot_cis(f_marg$rank_block[block_ix], ci.naive[block_ix,], f_marg$truth[block_ix], plot.truth = TRUE) +
```



Parametric bootstrap

Next we calculate parametric bootstrap confidence intervals using the usual and block-based ranking schemes. For details on how the `par_bs_ci` function works, see Algorithm 2 and Appendix Algorithm 2 in the paper. You can also refer to the walk-through for section 1.5, here.

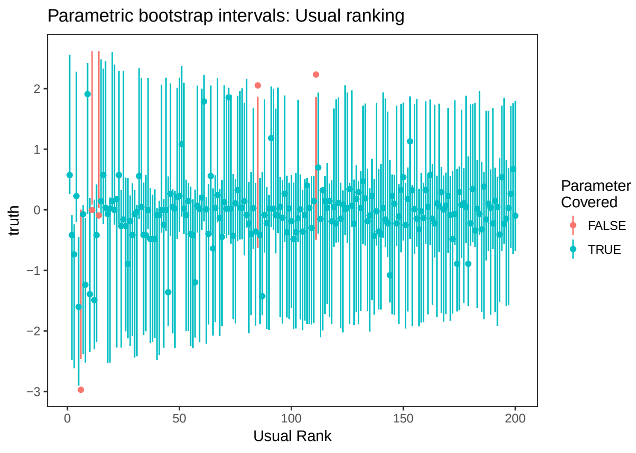
```r
#Block-based
ci.par.block <- par_bs_ci(beta=f_marg$beta_hat, se=f_marg$se_hat,
                          rank.func = rank_block, blocks=blocks)[, c("ci.lower", "ci.upper")]
mean(ci.par.block[,1] <= f_marg$truth & ci.par.block[,2] >= f_marg$truth, na.rm=TRUE)
```

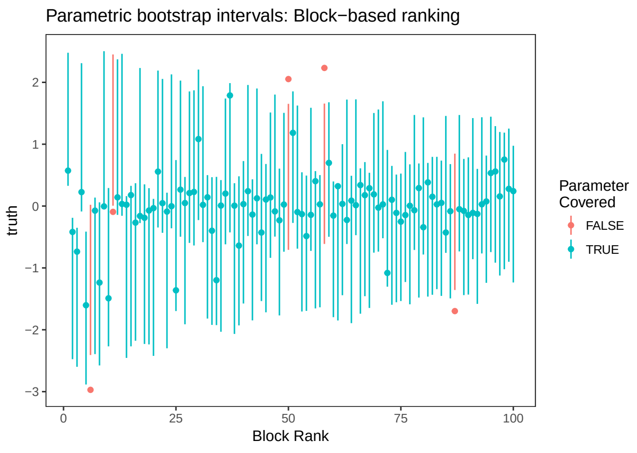
```
## [1] 0.95
```

```r
#Usual
ci.par.usual <- par_bs_ci(beta=f_marg$beta_hat, se=f_marg$se_hat)[, c("ci.lower", "ci.upper")]
mean(ci.par.usual[,1] <= f_marg$truth & ci.par.usual[,2] >= f_marg$truth, na.rm=TRUE)
```

```
## [1] 0.99
```

```r
plot_cis(f_marg$rank_usual, ci.par.usual, f_marg$truth, plot.truth = TRUE, prop=0.2) + xlab("Usual Rank
plot_cis(f_marg$rank_block[block_ix], ci.par.block[block_ix,], f_marg$truth[block_ix], plot.truth = TRU
```





Non-parametric bootstrap

The non-parametric boostrap is described in Algorithm 3 of the paper and implemented in the `nonpar_bs_ci` function of the `rcc` package. To compute the non-parametric bootstrap confidence intervals, we must supply a function that calculates parameter estimates from data. In `nonpar_bs_ci`, this is the `analysis.func` argument.

Here is the analysis function for our problem:

```r
lr_func <- function(data){
    y <- data[,1]
    X <- data[, -1]
    ests <- rccSims::many_lr(y, X, parallel=FALSE)
    df <- data.frame("estimate"=ests$beta_hat, "se"=ests$se_hat, "statistic"=ests$beta_hat/ests$se_hat)
    return(df)
}
```

For `nonpar_bs_ci` the analysis funciton may take only one argument so we supply a data frame or matrix that has y as the first column and the features as the subsequent columns.

Here we calculate the nonparametric boostrap confidence intervals using the usual and block-based ranking schemes. The `parallel` argument uses the `parallel` package to make use of multiple cores if available.

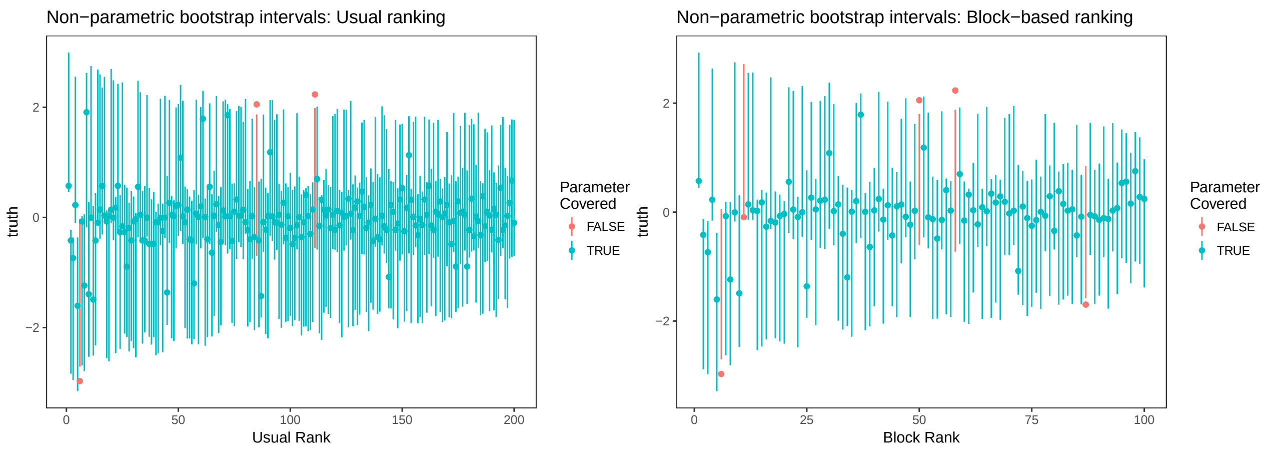
```
library(parallel)
data <- cbind(y, X)
#Usual
ci.nonpar.usual <- nonpar_bs_ci(data, analysis.func = lr_func,n.rep=1000,
                               level = 0.9, parallel=TRUE)[, c("ci.lower", "ci.upper")]
mean(ci.nonpar.usual[,1] <= f_marg$truth & ci.nonpar.usual[,2] >= f_marg$truth, na.rm=TRUE)
```

```
## [1] 0.991
```

```
#Block-based
ci.nonpar.block <- nonpar_bs_ci(data, analysis.func = lr_func,n.rep=1000, rank.func = rank_block,
                               level = 0.9, parallel=TRUE, blocks=blocks)[, c("ci.lower", "ci.upper")]
mean(ci.nonpar.block[,1] <= f_marg$truth & ci.nonpar.block[,2] >= f_marg$truth, na.rm=TRUE)
```

```
## [1] 0.95
```

```
plot_cis(f_marg$rank_usual, ci.nonpar.usual, f_marg$truth, plot.truth = TRUE, prop=0.2) + xlab("Usual Ra
plot_cis(f_marg$rank_block[block_ix], ci.nonpar.block[block_ix,], f_marg$truth[block_ix], plot.truth = T
```



Selection adjusted intervals of @Weinstein2013 and @Reid2014 are discussed in the walk-through for Section 1.5 and they look quite similar in this case so we will skip them here. They are included in the full simulation results below. We will show `ashr` credible intervals for this problem since, in this case the true parameter values are non-sparse (we are measuring the marginal effects) which leads to worse performance for `ashr`.

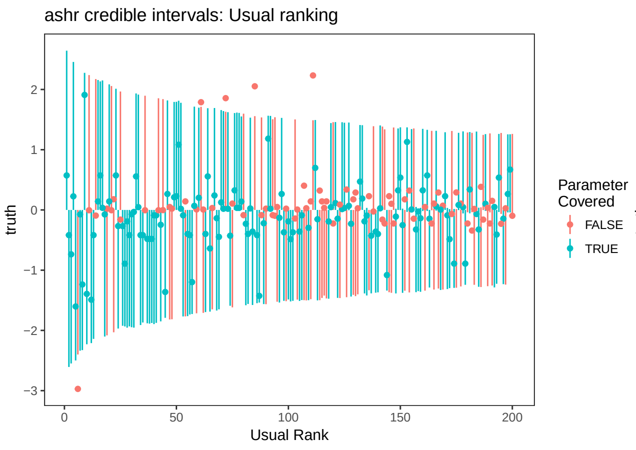
Empirical Bayes credible intervals (ashr; Stephens 2016)

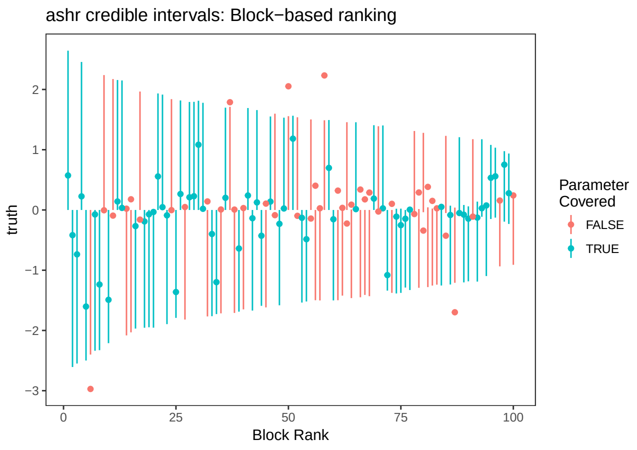
Here we generate the `ashr` credible intervals:

```
library(ashr)
ash.res <- ash(betahat = f_marg$beta_hat, sebetahat = f_marg$se_hat, mixcompdist = "normal")
ci.ash <- ashci(ash.res, level=0.9, betaindex = 1:1000, trace=FALSE)
mean(ci.ash[,1]<= f_marg$truth & ci.ash[,2] >= f_marg$truth)
```

```
## [1] 0.804
```

```
plot_cis(f_marg$rank_usual, ci.ash, f_marg$truth, plot.truth = TRUE, prop=0.2) + xlab("Usual Rank") + gg
plot_cis(f_marg$rank_block[block_ix], ci.ash[block_ix,], f_marg$truth[block_ix], plot.truth = TRUE) + xl
```





Simulations in Section 3.1 and Appendix section 3

Simulation results for usual ranking are shown in Section 3.1 and for block based ranking in Appendix Section 3. All of the steps in the previous section plus the intervals of @Weinstein2013 and @Reid2014 are implemented executed by the `cluster_sim` function in the `rccSims` package. We ran 400 simulations for each of four values of ρ. Since these take a little bit longer to run than the example in Section 1.5 (since we include the non-parametric bootstrap), we ran the simulations simulataneously as individual jobs submitted to a large cluster and each job had it's own seed. If you didn't have access to a cluster, the following loop would generate the same results (and uses the same seeds), but we recomend running these in parallel rather than in a loop.

```r
set.seed(5989615)
all.seeds <- floor(runif(n=400, min=1000, max=1e7))

nblock <- 100
beta <- list()
for(i in 1:nblock) beta[[i]] <- rep(c(0, rnorm(n=1), 0), c(4, 1, 5))

for(rho in c(-0.1, 0, 0.3, 0.8)){
  s <- matrix(rho, nrow=10, ncol=10)
  diag(s) <- 1
  Sigma <- list()
  for(i in 1:nblock) Sigma[[i]] <- s
  for(j in 1:400){
    results <- rccSims::cluster_sim(beta, Sigma, err.sd=1, n.samp=100,
                                    n.rep=1, seed=all.seeds[j], parallel=FALSE)
    save(results, file=paste0("cw_ranking", rho, "_n", j, ".RData"))
  }
}
```

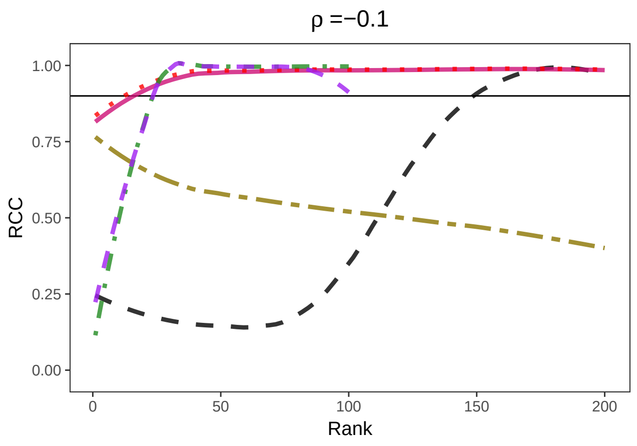
All of these results are included as built-in data sets to the `rccSims` package in the `lr_res` object, so you don't have to run them! `lr_res` is a list of length 4 with items corresponding to ($rho = -0.1, 0, 0.3$ and 0.8 respectively.

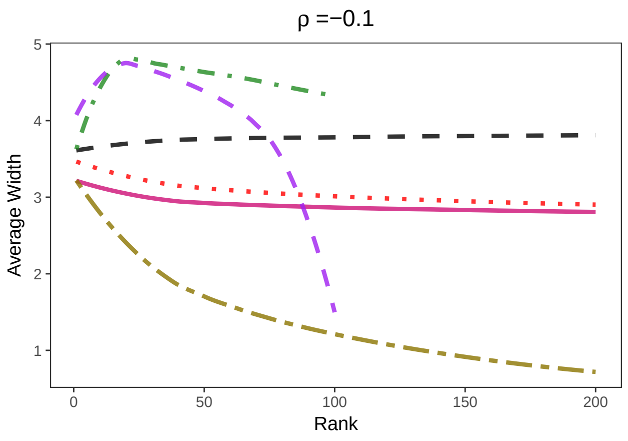
Here we plot the results for the usual ranking scheme (Section 3.1):

```r
library(tidyr)
data("lr_res", package="rccSims")
covplots <- list()
widthplots <- list()
```

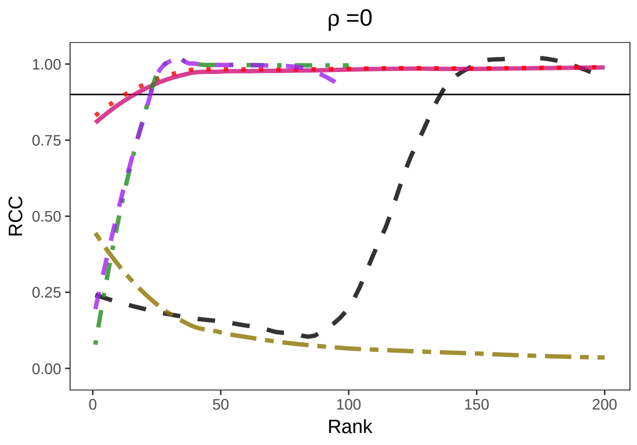
```r
tpart <- paste0("=", c(-0.1, 0, 0.3, 0.8)) #For titles
for(i in 1:4){
  lp <- "none"
  covplots[[i]] <- plot_coverage(lr_res[[i]], proportion=0.2,
      cols=c("black",  "deeppink3",  "red", "gold4", "forestgreen", "purple"),
      simnames=paste0(c("naive",  "par",    "nonpar", "ash", "wfb", "selInf1"), "_basic"),
      ltys= c(2, 1, 3, 6, 4, 2), span=0.5, y.range=c(-0.02, 1.02),
      legend.position = lp) + theme(plot.title=element_text(hjust=0.5)) + ggtitle(bquote(rho~.(tpart[i]
  widthplots[[i]] <- plot_width(lr_res[[i]], proportion=0.2,
      cols=c("black",  "deeppink3",  "red", "gold4", "forestgreen", "purple"),
      simnames=paste0(c("naive",  "par",    "nonpar", "ash", "wfb", "selInf1"), "_basic"),
      ltys= c(2, 1, 3, 6, 4, 2), span=0.5,
      legend.position = lp)+ theme(plot.title=element_text(hjust=0.5))+ ggtitle(bquote(rho~.(tpart[i]))

}
legend <- rccSims::make_sim_legend(legend.names = c("Marginal", "Parametric\nBootstrap",
                                       "Non-Parametric\nBootstrap", "ashr", "WFB", "RTT"),
            cols=c("black",  "deeppink3",  "red", "gold4", "forestgreen", "purple"),
            ltys= c(2, 1, 3, 6, 4, 2))
```

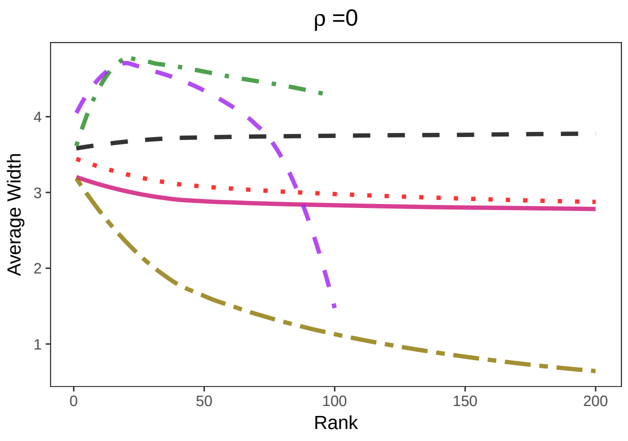
```r
covplots[[1]]
widthplots[[1]]
```



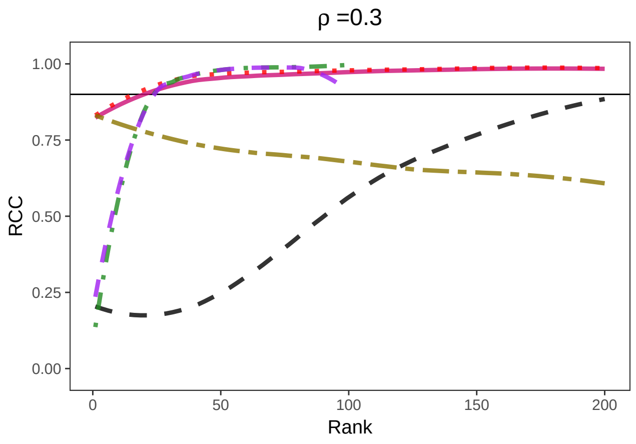


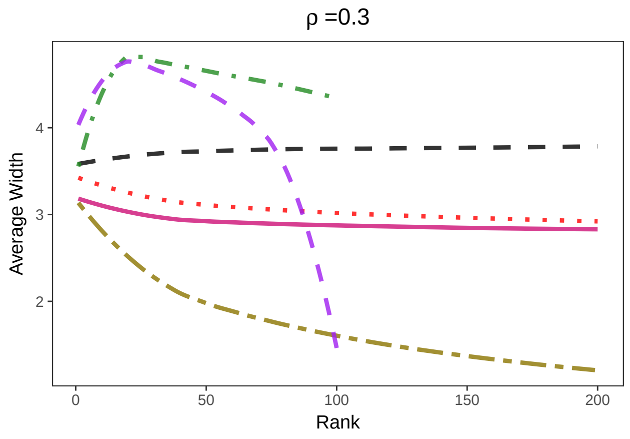
```r
covplots[[2]]
widthplots[[2]]
```



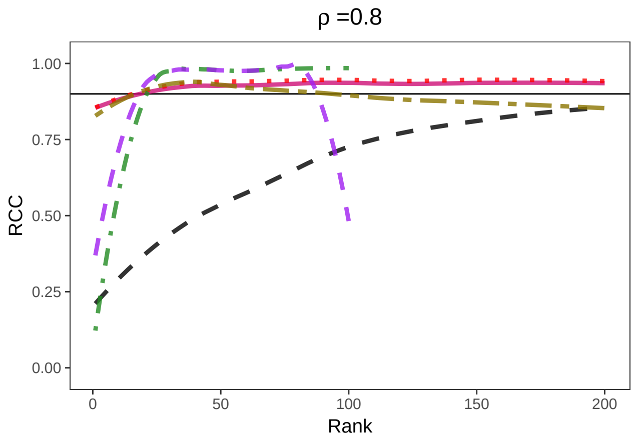


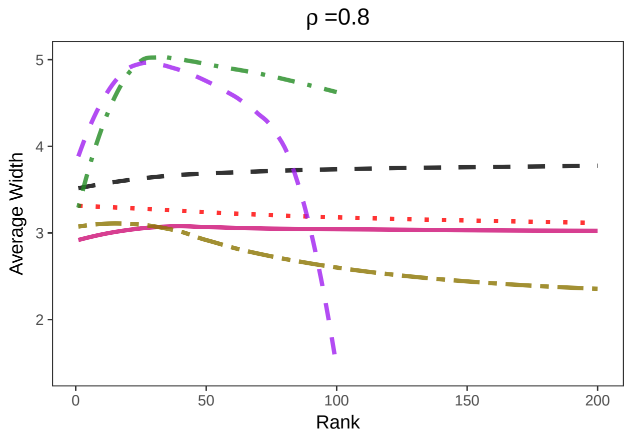
```r
covplots[[3]]
widthplots[[3]]
```





```r
covplots[[4]]
widthplots[[4]]
```

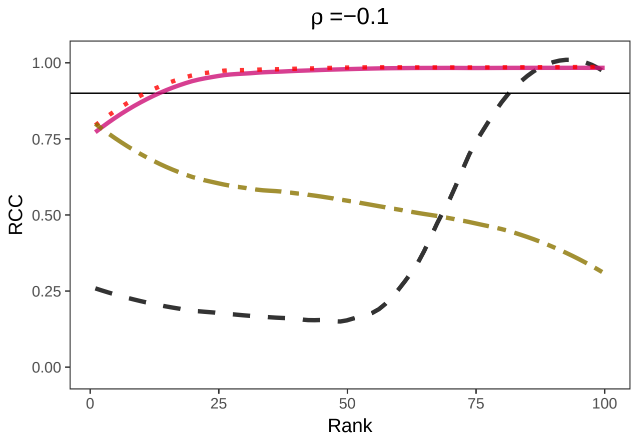


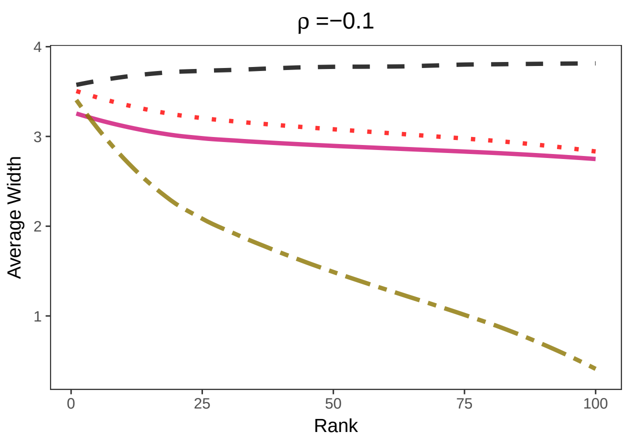


And here are plots using the block-based ranking scheme (Appendix Section 3). We don't include the selection adjusted intervals in these plots because they only consider selection schemes based on absolute value.

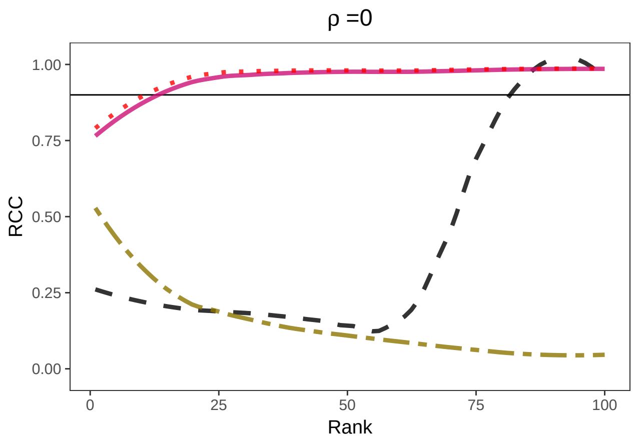
```r
covplots <- list()
widthplots <- list()
for(i in 1:4){
  lp <- "none"
  covplots[[i]] <- plot_coverage(lr_res[[i]], proportion=0.1,
      cols=c("black",  "deeppink3",  "red", "gold4"),
      simnames=paste0(c("naive",  "par",    "nonpar", "ash"), "_cw"),
      ltys= c(2, 1, 3, 6), span=0.5, y.range=c(-0.02, 1.02),
      legend.position = lp) + theme(plot.title=element_text(hjust=0.5)) + ggtitle(bquote(rho~.(tpart[i]
  widthplots[[i]] <- plot_width(lr_res[[i]], proportion=0.1,
      cols=c("black",  "deeppink3",  "red", "gold4"),
      simnames=paste0(c("naive",  "par",    "nonpar", "ash"), "_cw"),
      ltys= c(2, 1, 3, 6), span=0.5,
      legend.position = lp)+ theme(plot.title=element_text(hjust=0.5))+ ggtitle(bquote(rho~.(tpart[i]))

}
legend <- rccSims:::make_sim_legend(legend.names = c("Marginal", "Parametric\nBootstrap",
                                             "Non-Parametric\nBootstrap", "ashr"),
            cols=c("black",  "deeppink3",  "red", "gold4"),
            ltys= c(2, 1, 3, 6))
```

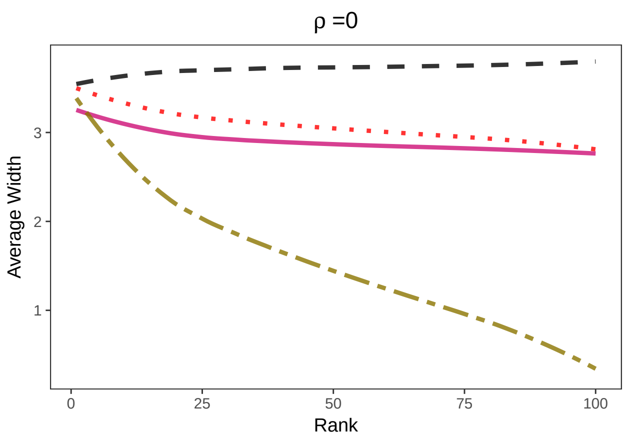
```
covplots[[1]]
widthplots[[1]]
```



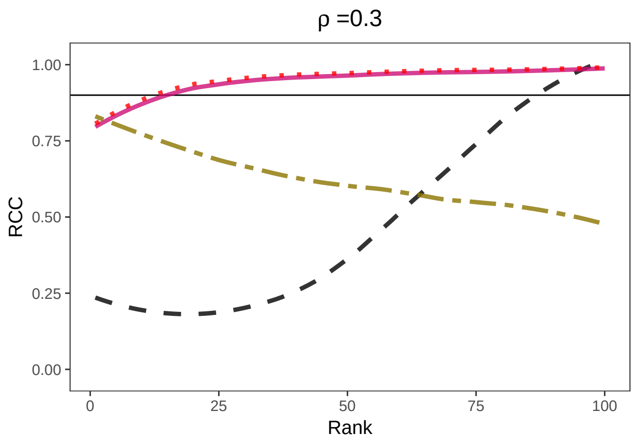


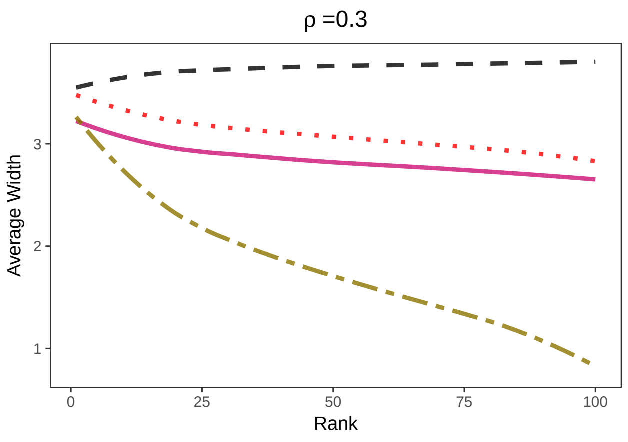
```
covplots[[2]]
widthplots[[2]]
```



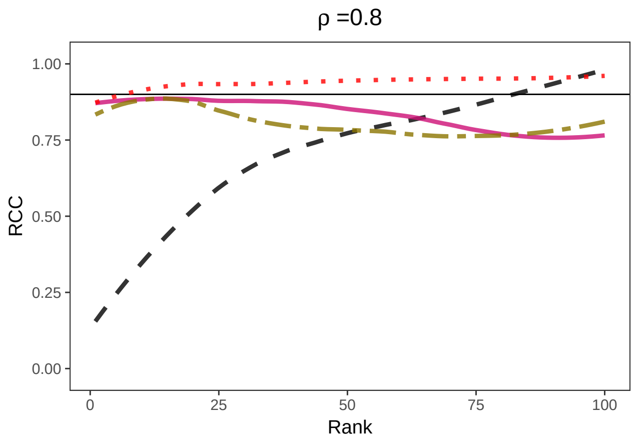


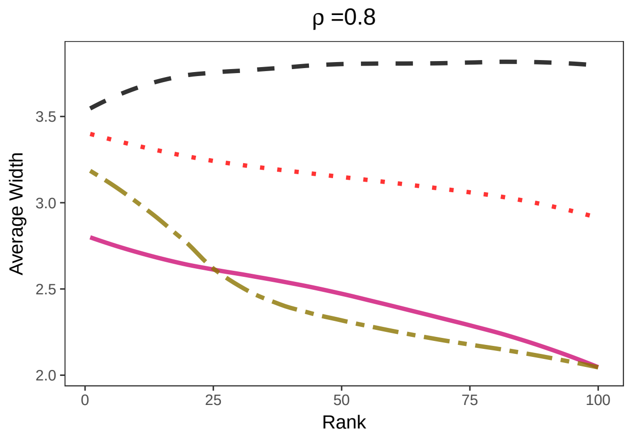
```
covplots[[3]]
widthplots[[3]]
```





```
covplots[[4]]
widthplots[[4]]
```


ρ =0.8
RCC
Rank


ρ =0.8
Average Width
Rank