Simulation Examples for Linear Models

Package Setup

```
if (!require("devtools", quietly = TRUE))
  install.packages("devtools")
```

```
Warning: package 'usethis' was built under R version 4.4.1
```

```
devtools::install_github("anonstats123/Nullstrap")
```

```
Skipping install of 'Nullstrap' from a github remote, the SHA1 (4519a73e) has not changed
since last install.
  Use `force = TRUE` to force installation
```

```
library(PRROC)
library(Nullstrap)
library(MASS)
```

```
Warning: package 'MASS' was built under R version 4.4.1
```

```
library(knockoff)
```

Simulation Data Generation

```
n <- 500
p <- 200
nonzero_coefs <- 30
Amp <- 0.25
rho <- 0.6
Theta.8 <- toeplitz(rho^(0:(p - 1)))
X <- mvrnorm(n, rep(0, p), Sigma = Theta.8)
X <- scale(X)
beta <- rep(0, p)
beta[1:nonzero_coefs] <- sample(c(-Amp, Amp), nonzero_coefs, replace = TRUE)
Signal_index <- 1:nonzero_coefs
true_labels <- beta != 0
y <- X %*% beta + rnorm(n)
y <- y - mean(y)
```

Statistical Metrics Function

```
fdp = function(selected) sum(beta[selected] == 0) / max(1, length(selected))
power = function(selected) sum(beta[selected] != 0) / sum(beta != 0)
aupr = function(statistic) pr.curve(scores.class0 = statistic,
```

```
                                    weights.class0 = true_labels,
                                    curve = FALSE)$auc.integral
```

Nullstrap

```
result_Nullstrap <- nullstrap_filter(
  X, y, fdr_value = 0.1, best_lambda = NULL, B_reps = NULL, dist_type = "normal",
  model_type = "linear"
)
```

```
Loading required package: Matrix

Warning: package 'Matrix' was built under R version 4.4.1

Loaded glmnet 4.1-8

Warning: package 'survival' was built under R version 4.4.1

Warning: package 'eha' was built under R version 4.4.1
```

```
Nullstrap_FDR <- length(which(result_Nullstrap$statistic[setdiff(1:p, Signal_index)] >=
                              result_Nullstrap$threshold)) / max(length(result_Nullstrap$s
Nullstrap_Power <- length(which(result_Nullstrap$statistic[Signal_index] >=
                                result_Nullstrap$threshold)) / length(Signal_index)
Nullstrap_AUPR <- aupr(result_Nullstrap$statistic)
cat("Nullstrap FDR:", Nullstrap_FDR, "\n")
```

```
Nullstrap FDR: 0
```

```
cat("Nullstrap Power:", Nullstrap_Power, "\n")
```

```
Nullstrap Power: 0.8333333
```

```
cat("Nullstrap AUPR:", Nullstrap_AUPR, "\n")
```

```
Nullstrap AUPR: 0.9698301
```

Model-X

```
result_mx <- knockoff.filter(X, y, fdr = 0.1)
mx_FDR = fdp(result_mx$selected)
mx_Power = power(result_mx$selected)
mx_AUPR = aupr(abs(result_mx$statistic))
cat("Model-X FDR:", mx_FDR, "\n")
```

```
Model-X FDR: 0
```

```r
cat("Model-X Power:", mx_Power, "\n")
```

```
Model-X Power: 0
```

```r
cat("Model-X AUPR:", mx_AUPR, "\n")
```

```
Model-X AUPR: 0.8788262
```

Fixed-X

```r
result_fx <- knockoff.filter(X = X, y = y, knockoffs = create.fixed,
                             statistic = stat.glmnet_lambdasmax, fdr = 0.1)
fx_FDR = fdp(result_fx$selected)
fx_Power = power(result_fx$selected)
fx_AUPR = aupr(abs(result_fx$statistic))
cat("Fixed-X FDR:", fx_FDR, "\n")
```

```
Fixed-X FDR: 0
```

```r
cat("Fixed-X Power:", fx_Power, "\n")
```

```
Fixed-X Power: 0
```

```r
cat("Fixed-X AUPR:", fx_AUPR, "\n")
```

```
Fixed-X AUPR: 0.6866516
```

Data Splitting

```r
source("./DS.R")
source("./analys.R")
result_ds <- DS(X, y, num_split = 1, q = 0.1)
ds_FDR <- fdp(result_ds$DS_feature)
ds_Power <- power(result_ds$DS_feature)
ds_AUPR = aupr(abs(result_ds$DS_statistic))
cat("Data Splitting FDR:", ds_FDR, "\n")
```

```
Data Splitting FDR: 0.1333333
```

```r
cat("Data Splitting Power:", ds_Power, "\n")
```

```
Data Splitting Power: 0.8666667
```

```r
cat("Data Splitting AUPR:", ds_AUPR, "\n")
```

```
Data Splitting AUPR: 0.8965079
```

Multiple Data Splitting

```r
source("./DS.R")
source("./analys.R")
result_mds <- DS(X, y, num_split = 50, q = 0.1)
mds_FDR <- fdp(result_mds$MDS_feature)
mds_Power <- power(result_mds$MDS_feature)
mds_AUPR = aupr(abs(result_mds$MDS_statistic))
cat("Multiple Data Splitting FDR:", mds_FDR, "\n")
```

```
Multiple Data Splitting FDR: 0.04166667
```

```r
cat("Multiple Data Splitting Power:", mds_Power, "\n")
```

```
Multiple Data Splitting Power: 0.7666667
```

```r
cat("Multiple Data Splitting AUPR:", mds_AUPR, "\n")
```

```
Multiple Data Splitting AUPR: 0.9436678
```

BH

```r
SoftThreshold <- function(x, lambda ) {
  #
  # Standard soft thresholding
  #
  if (x>lambda){
    return (x-lambda);}
  else {
    if (x< (-lambda)){
      return (x+lambda);}
    else {
      return (0);}
  }
}

InverseLinftyOneRow <- function ( sigma, i, mu, maxiter=50, threshold=1e-2 ) {
  p <- nrow(sigma);
  rho <- max(abs(sigma[i,-i])) / sigma[i,i];
  mu0 <- rho/(1+rho);
  beta <- rep(0,p);

  if (mu >= mu0){
    beta[i] <- (1-mu0)/sigma[i,i];
    returnlist <- list("optsol" = beta, "iter" = 0);
    return(returnlist);
```

```
  }

  diff.norm2 <- 1;
  last.norm2 <- 1;
  iter <- 1;
  iter.old <- 1;
  beta[i] <- (1-mu0)/sigma[i,i];
  beta.old <- beta;
  sigma.tilde <- sigma;
  diag(sigma.tilde) <- 0;
  vs <- -sigma.tilde%*%beta;

  while ((iter <= maxiter) && (diff.norm2 >= threshold*last.norm2)){

    for (j in 1:p){
      oldval <- beta[j];
      v <- vs[j];
      if (j==i)
        v <- v+1;
      beta[j] <- SoftThreshold(v,mu)/sigma[j,j];
      if (oldval != beta[j]){
        vs <- vs + (oldval-beta[j])*sigma.tilde[,j];
      }
    }

    iter <- iter + 1;
    if (iter==2*iter.old){
      d <- beta - beta.old;
      diff.norm2 <- sqrt(sum(d*d));
      last.norm2 <-sqrt(sum(beta*beta));
      iter.old <- iter;
      beta.old <- beta;
      if (iter>10)
        vs <- -sigma.tilde%*%beta;
    }
  }

  returnlist <- list("optsol" = beta, "iter" = iter)
  return(returnlist)
}

InverseLinfty <- function(sigma, n, resol=1.5, mu=NULL, maxiter=50, threshold=1e-2, verbo
  isgiven <- 1;
  if (is.null(mu)){
    isgiven <- 0;
  }

  p <- nrow(sigma);
  M <- matrix(0, p, p);
  xperc = 0;
  xp = round(p/10);
```

```
  for (i in 1:p) {
    if ((i %% xp)==0){
      xperc = xperc+10;
      if (verbose) {
        print(paste(xperc,"% done",sep="")); }
    }
    if (isgiven==0){
      mu <- (1/sqrt(n)) * qnorm(1-(0.1/(p^2)));
    }
    mu.stop <- 0;
    try.no <- 1;
    incr <- 0;
    while ((mu.stop != 1)&&(try.no<10)){
      last.beta <- beta
      output <- InverseLinftyOneRow(sigma, i, mu, maxiter=maxiter, threshold=threshold)
      beta <- output$optsol
      iter <- output$iter
      if (isgiven==1){
        mu.stop <- 1
      }
      else{
        if (try.no==1){
          if (iter == (maxiter+1)){
            incr <- 1;
            mu <- mu*resol;
          } else {
            incr <- 0;
            mu <- mu/resol;
          }
        }
        if (try.no > 1){
          if ((incr == 1)&&(iter == (maxiter+1))){
            mu <- mu*resol;
          }
          if ((incr == 1)&&(iter < (maxiter+1))){
            mu.stop <- 1;
          }
          if ((incr == 0)&&(iter < (maxiter+1))){
            mu <- mu/resol;
          }
          if ((incr == 0)&&(iter == (maxiter+1))){
            mu <- mu*resol;
            beta <- last.beta;
            mu.stop <- 1;
          }
        }
      }
      try.no <- try.no+1
    }
    M[i,] <- beta;
  }
```

```
  return(M)
}

NoiseSd <- function( yh, A, n ){
  ynorm <- sqrt(n)*(yh/sqrt(diag(A)));
  sd.hat0 <- mad(ynorm);

  zeros <- (abs(ynorm)<3*sd.hat0);
  y2norm <- sum(yh[zeros]^2);
  Atrace <- sum(diag(A)[zeros]);
  sd.hat1 <- sqrt(n*y2norm/Atrace);

  ratio <- sd.hat0/sd.hat1;
  if (max(ratio,1/ratio)>2)
    print("Warning: Noise estimate problematic");

  s0 <- sum(zeros==FALSE);
  return (list( "sd" = sd.hat1, "nz" = s0));
}

Lasso <- function( X, y, lambda = NULL, intercept = TRUE){
  #
  # Compute the Lasso estimator:
  # - If lambda is given, use glmnet and standard Lasso
  # - If lambda is not given, use square root Lasso
  #
  p <- ncol(X);
  n <- nrow(X);

  if  (is.null(lambda)){
    lambda <- sqrt(qnorm(1-(0.1/p))/n);
    outLas <- slim(X,y,lambda=c(lambda),method="lq",q=2,verbose=FALSE);
    # Objective : sqrt(RSS/n) +lambda *penalty
    if (intercept==TRUE) {
      return (c(as.vector(outLas$intercept),as.vector(outLas$beta)))
    }  else {
      return (as.vector(outLas$beta));
    }
  } else {
    outLas <- glmnet(X, y, family = c("gaussian"), alpha =1, intercept = intercept );
    # Objective :1/2 RSS/n +lambda *penalty
    if (intercept==TRUE){
      return (as.vector(coef(outLas,s=lambda)));
    } else {
      return (as.vector(coef(outLas,s=lambda))[2:(p+1)]);
    }
  }
}

SSLasso <- function (X, y, alpha=0.05, lambda = NULL, mu = NULL, intercept = TRUE,
                     resol=1.3, maxiter=50, threshold=1e-2, verbose = TRUE) {
```

```r
#
# Compute confidence intervals and p-values.
#
# Args:
#   X     :  design matrix
#   y     :  response
#   alpha :  significance level
#   lambda:  Lasso regularization parameter (if null, fixed by sqrt lasso)
#   mu    :  Linfty constraint on M (if null, searches)
#   resol :  step parameter for the function that computes M
#   maxiter: iteration parameter for computing M
#   threshold : tolerance criterion for computing M
#   verbose : verbose?
#
# Returns:
#   noise.sd: Estimate of the noise standard deviation
#   norm0   : Estimate of the number of 'significant' coefficients
#   coef    : Lasso estimated coefficients
#   unb.coef: Unbiased coefficient estimates
#   low.lim : Lower limits of confidence intervals
#   up.lim  : upper limit of confidence intervals
#   pvals   : p-values for the coefficients
#
p <- ncol(X);
n <- nrow(X);
pp <- p;
col.norm <- 1/sqrt((1/n)*diag(t(X)%*%X));
X <- X %*% diag(col.norm);

htheta <- Lasso (X,y,lambda=lambda,intercept=intercept);

if (intercept==TRUE){
  Xb <- cbind(rep(1,n),X);
  col.norm <- c(1,col.norm);
  pp <- (p+1);
} else {
  Xb <- X;
}
sigma.hat <- (1/n)*(t(Xb)%*%Xb);

if ((n>=2*p)){
  tmp <- eigen(sigma.hat)
  tmp <- min(tmp$values)/max(tmp$values)
}else{
  tmp <- 0
}

if ((n>=2*p)&&(tmp>=1e-4)){
 M <- solve(sigma.hat)
}else{
 M <- InverseLinfty(sigma.hat, n, resol=resol, mu=mu, maxiter=maxiter, threshold=thres
```

```r
  }

  unbiased.Lasso <- as.numeric(htheta + (M%*%t(Xb)%*%(y - Xb %*% htheta))/n);

  A <- M %*% sigma.hat %*% t(M);
  noise <- NoiseSd( unbiased.Lasso, A, n );
  s.hat <- noise$sd;

  interval.sizes <- qnorm(1-(alpha/2))*s.hat*sqrt(diag(A))/(sqrt(n));

  if  (is.null(lambda)){
    lambda <- s.hat*sqrt(qnorm(1-(0.1/p))/n);
  }

  addlength <- rep(0,pp);
  MM <- M%*%sigma.hat - diag(pp);
  for (i in 1:pp){
    effectivemuvec <- sort(abs(MM[i,]),decreasing=TRUE);
    effectivemuvec <- effectivemuvec[0:(noise$nz-1)];
    addlength[i] <- sqrt(sum(effectivemuvec*effectivemuvec))*lambda;
  }

  htheta <- htheta*col.norm;
  unbiased.Lasso <- unbiased.Lasso*col.norm;
  interval.sizes <- interval.sizes*col.norm;
  addlength <- addlength*col.norm;

  if (intercept==TRUE){
    htheta <- htheta[2:pp];
    unbiased.Lasso <- unbiased.Lasso[2:pp];
    interval.sizes <- interval.sizes[2:pp];
    addlength <- addlength[2:pp];
  }
  p.vals <- 2*(1-pnorm(sqrt(n)*abs(unbiased.Lasso)/(s.hat*col.norm[(pp-p+1):pp]*sqrt(diag

  returnList <- list("noise.sd" = s.hat,
                     "norm0" = noise$nz,
                     "coef" = htheta,
                     "unb.coef" = unbiased.Lasso,
                     "low.lim" = unbiased.Lasso - interval.sizes - addlength,
                     "up.lim" = unbiased.Lasso + interval.sizes + addlength,
                     "pvals" = p.vals
  )
  return(returnList)
}

fit = cv.glmnet(X, y, intercept = F, alpha = 1)
best_lambda <- 0.5*fit$lambda.min
if(n > p){
  fit_lm = lm(y ~ X - 1)
  fit_lm_pvalue = summary(fit_lm)$coef[, 4]
```

```
  p_adjusted_BH = p.adjust(fit_lm_pvalue, method = "BH")
  bh_AUPR = aupr(-fit_lm_pvalue)
  } else {
  bh_result <- SSLasso(X, y, lambda = best_lambda, intercept = FALSE)
  pvals = bh_result$pvals
  p_adjusted_BH = p.adjust(pvals, method = "BH")
  bh_AUPR <- aupr(-pvals)
  }
bh_Power = length(which(p_adjusted_BH[Signal_index] < 0.1)) /length(Signal_index)
bh_FDR = length(which(p_adjusted_BH[setdiff(1:p, Signal_index)] < 0.1)) /
  max(length(which(p_adjusted_BH < 0.1)), 1)
cat("BH FDR:", bh_FDR, "\n")
```

```
BH FDR: 0
```

```
cat("BH Power:", bh_Power, "\n")
```

```
BH Power: 0.7333333
```

```
cat("BH AUPR:", bh_AUPR, "\n")
```

```
BH AUPR: 0.9373924
```

BHq

```
SoftThreshold <- function(x, lambda ) {
  #
  # Standard soft thresholding
  #
  if (x>lambda){
    return (x-lambda);}
  else {
    if (x< (-lambda)){
      return (x+lambda);}
    else {
      return (0);}
  }
}

InverseLinftyOneRow <- function ( sigma, i, mu, maxiter=50, threshold=1e-2 ) {
  p <- nrow(sigma);
  rho <- max(abs(sigma[i,-i])) / sigma[i,i];
  mu0 <- rho/(1+rho);
  beta <- rep(0,p);

  if (mu >= mu0){
    beta[i] <- (1-mu0)/sigma[i,i];
    returnlist <- list("optsol" = beta, "iter" = 0);
```

```
    return(returnlist);
  }

  diff.norm2 <- 1;
  last.norm2 <- 1;
  iter <- 1;
  iter.old <- 1;
  beta[i] <- (1-mu0)/sigma[i,i];
  beta.old <- beta;
  sigma.tilde <- sigma;
  diag(sigma.tilde) <- 0;
  vs <- -sigma.tilde%*%beta;

  while ((iter <= maxiter) && (diff.norm2 >= threshold*last.norm2)){

    for (j in 1:p){
      oldval <- beta[j];
      v <- vs[j];
      if (j==i)
        v <- v+1;
      beta[j] <- SoftThreshold(v,mu)/sigma[j,j];
      if (oldval != beta[j]){
        vs <- vs + (oldval-beta[j])*sigma.tilde[,j];
      }
    }

    iter <- iter + 1;
    if (iter==2*iter.old){
      d <- beta - beta.old;
      diff.norm2 <- sqrt(sum(d*d));
      last.norm2 <-sqrt(sum(beta*beta));
      iter.old <- iter;
      beta.old <- beta;
      if (iter>10)
        vs <- -sigma.tilde%*%beta;
    }
  }

  returnlist <- list("optsol" = beta, "iter" = iter)
  return(returnlist)
}

InverseLinfty <- function(sigma, n, resol=1.5, mu=NULL, maxiter=50, threshold=1e-2, verbo
  isgiven <- 1;
  if (is.null(mu)){
    isgiven <- 0;
  }

  p <- nrow(sigma);
  M <- matrix(0, p, p);
  xperc = 0;
```

```
  xp = round(p/10);
  for (i in 1:p) {
    if ((i %% xp)==0){
      xperc = xperc+10;
      if (verbose) {
        print(paste(xperc,"% done",sep="")); }
    }
    if (isgiven==0){
      mu <- (1/sqrt(n)) * qnorm(1-(0.1/(p^2)));
    }
    mu.stop <- 0;
    try.no <- 1;
    incr <- 0;
    while ((mu.stop != 1)&&(try.no<10)){
      last.beta <- beta
      output <- InverseLinftyOneRow(sigma, i, mu, maxiter=maxiter, threshold=threshold)
      beta <- output$optsol
      iter <- output$iter
      if (isgiven==1){
        mu.stop <- 1
      }
      else{
        if (try.no==1){
          if (iter == (maxiter+1)){
            incr <- 1;
            mu <- mu*resol;
          } else {
            incr <- 0;
            mu <- mu/resol;
          }
        }
        if (try.no > 1){
          if ((incr == 1)&&(iter == (maxiter+1))){
            mu <- mu*resol;
          }
          if ((incr == 1)&&(iter < (maxiter+1))){
            mu.stop <- 1;
          }
          if ((incr == 0)&&(iter < (maxiter+1))){
            mu <- mu/resol;
          }
          if ((incr == 0)&&(iter == (maxiter+1))){
            mu <- mu*resol;
            beta <- last.beta;
            mu.stop <- 1;
          }
        }
      }
      try.no <- try.no+1
    }
    M[i,] <- beta;
```

```
  }
  return(M)
}

NoiseSd <- function( yh, A, n ){
  ynorm <- sqrt(n)*(yh/sqrt(diag(A)));
  sd.hat0 <- mad(ynorm);

  zeros <- (abs(ynorm)<3*sd.hat0);
  y2norm <- sum(yh[zeros]^2);
  Atrace <- sum(diag(A)[zeros]);
  sd.hat1 <- sqrt(n*y2norm/Atrace);

  ratio <- sd.hat0/sd.hat1;
  if (max(ratio,1/ratio)>2)
    print("Warning: Noise estimate problematic");

  s0 <- sum(zeros==FALSE);
  return (list( "sd" = sd.hat1, "nz" = s0));
}

Lasso <- function( X, y, lambda = NULL, intercept = TRUE){
  #
  # Compute the Lasso estimator:
  # - If lambda is given, use glmnet and standard Lasso
  # - If lambda is not given, use square root Lasso
  #
  p <- ncol(X);
  n <- nrow(X);

  if  (is.null(lambda)){
    lambda <- sqrt(qnorm(1-(0.1/p))/n);
    outLas <- slim(X,y,lambda=c(lambda),method="lq",q=2,verbose=FALSE);
    # Objective : sqrt(RSS/n) +lambda *penalty
    if (intercept==TRUE) {
      return (c(as.vector(outLas$intercept),as.vector(outLas$beta)))
    }  else {
      return (as.vector(outLas$beta));
    }
  } else {
    outLas <- glmnet(X, y, family = c("gaussian"), alpha =1, intercept = intercept );
    # Objective :1/2 RSS/n +lambda *penalty
    if (intercept==TRUE){
      return (as.vector(coef(outLas,s=lambda)));
    } else {
      return (as.vector(coef(outLas,s=lambda))[2:(p+1)]);
    }
  }
}

SSLasso <- function (X, y, alpha=0.05, lambda = NULL, mu = NULL, intercept = TRUE,
```

```
                    resol=1.3, maxiter=50, threshold=1e-2, verbose = TRUE) {
  #
  # Compute confidence intervals and p-values.
  #
  # Args:
  #   X     :  design matrix
  #   y     :  response
  #   alpha :  significance level
  #   lambda:  Lasso regularization parameter (if null, fixed by sqrt lasso)
  #   mu    :  Linfty constraint on M (if null, searches)
  #   resol :  step parameter for the function that computes M
  #   maxiter: iteration parameter for computing M
  #   threshold : tolerance criterion for computing M
  #   verbose : verbose?
  #
  # Returns:
  #   noise.sd: Estimate of the noise standard deviation
  #   norm0   : Estimate of the number of 'significant' coefficients
  #   coef    : Lasso estimated coefficients
  #   unb.coef: Unbiased coefficient estimates
  #   low.lim : Lower limits of confidence intervals
  #   up.lim  : upper limit of confidence intervals
  #   pvals   : p-values for the coefficients
  #
  p <- ncol(X);
  n <- nrow(X);
  pp <- p;
  col.norm <- 1/sqrt((1/n)*diag(t(X)%*%X));
  X <- X %*% diag(col.norm);

  htheta <- Lasso (X,y,lambda=lambda,intercept=intercept);

  if (intercept==TRUE){
    Xb <- cbind(rep(1,n),X);
    col.norm <- c(1,col.norm);
    pp <- (p+1);
  } else {
    Xb <- X;
  }
  sigma.hat <- (1/n)*(t(Xb)%*%Xb);

  if ((n>=2*p)){
    tmp <- eigen(sigma.hat)
    tmp <- min(tmp$values)/max(tmp$values)
  }else{
    tmp <- 0
  }

  if ((n>=2*p)&&(tmp>=1e-4)){
   M <- solve(sigma.hat)
  }else{
```

```
    M <- InverseLinfty(sigma.hat, n, resol=resol, mu=mu, maxiter=maxiter, threshold=thres
  }

  unbiased.Lasso <- as.numeric(htheta + (M%*%t(Xb)%*%(y - Xb %*% htheta))/n);

  A <- M %*% sigma.hat %*% t(M);
  noise <- NoiseSd( unbiased.Lasso, A, n );
  s.hat <- noise$sd;

  interval.sizes <- qnorm(1-(alpha/2))*s.hat*sqrt(diag(A))/(sqrt(n));

  if  (is.null(lambda)){
    lambda <- s.hat*sqrt(qnorm(1-(0.1/p))/n);
  }

  addlength <- rep(0,pp);
  MM <- M%*%sigma.hat - diag(pp);
  for (i in 1:pp){
    effectivemuvec <- sort(abs(MM[i,]),decreasing=TRUE);
    effectivemuvec <- effectivemuvec[0:(noise$nz-1)];
    addlength[i] <- sqrt(sum(effectivemuvec*effectivemuvec))*lambda;
  }

  htheta <- htheta*col.norm;
  unbiased.Lasso <- unbiased.Lasso*col.norm;
  interval.sizes <- interval.sizes*col.norm;
  addlength <- addlength*col.norm;

  if (intercept==TRUE){
    htheta <- htheta[2:pp];
    unbiased.Lasso <- unbiased.Lasso[2:pp];
    interval.sizes <- interval.sizes[2:pp];
    addlength <- addlength[2:pp];
  }
  p.vals <- 2*(1-pnorm(sqrt(n)*abs(unbiased.Lasso)/(s.hat*col.norm[(pp-p+1):pp]*sqrt(diag

  returnList <- list("noise.sd" = s.hat,
                     "norm0" = noise$nz,
                     "coef" = htheta,
                     "unb.coef" = unbiased.Lasso,
                     "low.lim" = unbiased.Lasso - interval.sizes - addlength,
                     "up.lim" = unbiased.Lasso + interval.sizes + addlength,
                     "pvals" = p.vals
  )
  return(returnList)
}

fit = cv.glmnet(X, y, intercept = F, alpha = 1)
best_lambda <- 0.5*fit$lambda.min
if(n > p){
  fit_lm = lm(y ~ X - 1)
```

```r
  fit_lm_pvalue = summary(fit_lm)$coef[, 4]
  p_adjusted_BHq = p.adjust(fit_lm_pvalue, method = "BY")
  bhq_AUPR = aupr(-fit_lm_pvalue)
  } else {
  bhq_result <- SSLasso(X, y, lambda = best_lambda, intercept = FALSE)
  pvals = bhq_result$pvals
  p_adjusted_BHq = p.adjust(pvals, method = "BY")
  bhq_AUPR = aupr(-pvals)
  }
bhq_Power = length(which(p_adjusted_BHq[Signal_index] < 0.1)) /length(Signal_index)
bhq_FDR = length(which(p_adjusted_BHq[setdiff(1:p, Signal_index)] < 0.1)) /
  max(length(which(p_adjusted_BHq < 0.1)), 1)
cat("BHq FDR:", bhq_FDR, "\n")
```

```
BHq FDR: 0
```

```r
cat("BHq Power:", bhq_Power, "\n")
```

```
BHq Power: 0.6
```

```r
cat("BHq AUPR:", bhq_AUPR, "\n")
```

```
BHq AUPR: 0.9373924
```