TrainSel Usage

Deniz Akdemir, Julio Isidro Sanchez, Simon Rio and Javier Fernandez-Gonzalez

2023/09/20

Contents

Introduction

In this section, we will illustrate the use of the package 'TrainSel'. We will use the data sets provided within the package under the object named 'WheatData' throughout this presentation. The original data was obtained from the webpage https://triticeaetoolbox.org/. The data contains the genomewide marker data (at 4670 markers) and a simulated trait data (phenotypic measurements are simulated using an infinitismall model) for 200 wheat varieties.

We can load the library and 'WheatData' using the following code:

```
library(TrainSel)
```

```
## Loading required package: cluster

## Loading required package: parallel

## Loading required package: foreach

## Loading required package: doParallel

## Loading required package: iterators

##

## ************************************************************

##
```

```
##      This is 'TrainSel' package, v 3.0

## Copyright 2023 Deniz Akdemir, Julio Isidro Sanchez,

## Simon Rio, Javier Fernandez-Gonzalez. All rights reserved.

## Please read the LICENSE file for more information

## ************************************************************
```

```
data("WheatData")
```

Marker data is contained in the matrix object 'Wheat.M', a relationship matrix for the genotypes calculated from the marker matrix is in 'Wheat.K', and the plant height measurements are in 'Wheat.Y'. We can see the format of these data:

```
Wheat.M[1:5,1:5]
```

```
##       IWA1 IWA2 IWA3 IWA4 IWA5
## Line1    1    1   -1   -1   -1
## Line2    1    1    1   -1   -1
## Line3    1    1    1   -1   -1
## Line4    1    1    1    1   -1
## Line5    1    1    1   -1   -1
```

```
Wheat.K[1:5,1:5]
```

```
##            Line1      Line2      Line3      Line4      Line5
## Line1  1.9578361  0.8261588  0.7546713  0.5149389 -0.2697654
## Line2  0.8261588  2.0033376  0.5756170  0.5241991 -0.2315324
## Line3  0.7546713  0.5756170  1.9807687  0.7077443 -0.2888690
## Line4  0.5149389  0.5241991  0.7077443  2.2816612 -0.2645939
## Line5 -0.2697654 -0.2315324 -0.2888690 -0.2645939  1.8086996
```

```
Wheat.Y[1:5,]
```

```
##         id plant.height
## 1846 Line1     138.4471
## 250  Line2     122.5955
## 1541 Line3     134.7883
## 1516 Line4     121.4741
## 508  Line5     127.3920
```

Selection of a subset of genotypes for a phenotypic experiment in a single environment

Our aim is to build a predictive model for the plant height based on the genomewide marker data. The dataset contains the plant heights for all of these genotypes, however, for a moment, assume that we do not have the plant heights measurements for these 200 genotypes and currently only a subset of 160 candidate

genotypes are available to be used in the phenotypic experiment. Furthermore, since measuring plant height for 160 candidate genotypes via a phenotypic experiment can be costly, we assume that we can only perform the phenotypic experiment with say a maximum of 50 training genotypes selected from the 160. Following this phenotypic experiment, we can use the available marker data and the measured height of these 50 genotypes to train a genomic prediction model to make inferences about the heights of the remaining 110 genotypes or any other set of genotypes that we have the same genomewide marker data, for example, the 40 genotypes that were not available for the phenotypic experiment.

Selection of a subset of genotypes for a homogeneous design in a single environment

The simplest use case for 'TrainSel' is the situation where phenotypic experiment will only performed in one homogeneous environment. In this case, our purpose is to select a subset of size 50 from the 160 available genotypes so that the genomic prediction models that are trained on this 50 has a good generalization performance.

We distinguish between two cases of optimal training set selection based on whether we seek that generalize well for a specific target set of genotypes (Targeted optimization) or not (Un-targeted optimization). Not all optimization criteria are sensitive to this distinction, however when it is so this is reflected in how the optimization criteria is calculated.

Un-targeted optimization There are many different statistics that can be used for the untargeted optimization with a homogeneous design in a single environment. The package 'TrainSel' is designed to be flexible to be used with any design criteria, however, this means that the users need to program their own optimization functions. Below are some example functions that should get started for writing your own:

D-optimality criterion D-optimality criterion is a model based design criterion. The underlying model for the D-optimality criterion is a linear model. For the problem of selection of a subset of genotypes for a homogeneous design in a single environment a linear model relating the genotypic data to phenotypic measurements in the training data can be expressed as

$$y = 1\mu + f(M)\beta + \epsilon,$$

where y is the n vector of phenotypic measurements in the training data, μ is a scalar parameter for the mean of the phenotypic measurements, M is the $n \times m$ the marker matrix for the n training genotypes, $f(M)$ is a genomic features matrix with dimensions $n \times q$, β is the q vector of effects of genomic features, and ϵ is the residual error vector of length n. We further assume that the elements of ϵ are independent and identically distributed with a normal distribution with zero mean and variance σ_e^2. Under this model the variance of the estimators for β is known to be proportional to $[f(M)'f(M)]^{-1}$. D-optimal selection of n training genotypes from N individuals in the candidate set involves minimizing the determinant of this matrix (or equivalently maximizing the log-determinant of $f(M)'f(M)$). The feature matrix $f(M)$ is usually the first q principal components matrix for the marker matrix M and for this measure to be defined q should be less than n.

```
#We will use the first 30 principal components for this
Wheat.M_centered<-scale(Wheat.M, center=TRUE, scale=FALSE)
svdWheat.M_centered<-svd(Wheat.M_centered, nu=30, nv=30)

PC<-Wheat.M_centered%*%svdWheat.M_centered$v
dim(PC)
```

```
## [1] 200  30
```

```
dataDopt<-list(FeatureMat=PC)

DOPT<-function(soln, Data){
  Fmat<-Data[["FeatureMat"]]
  return(determinant(crossprod(Fmat[soln,]), logarithm=TRUE)$modulus)
}
```

In all of the following examples we will set the TrainSel algorithm parameters to the maximum allowed by the demo version. We will discuss in a later section how to use the `SetControlDefault` function to tune the TrainSel parameters when a license is available.

```
#Set parameters compatible with demo version
TSC<-SetControlDefault(size = "demo",
                       complexity = "low_complexity")

#You can also manually change the parameters, although generally we
#don't recommend it. Here we show how to decrease iterations from 100 to 90:
TSC$niterations
```

```
## [1] 100
```

```
TSC$niterations <- 90

#Be aware that, if you don't have a license key, altering the default parameters
#may exceed the demo limitations, precluding the algorithm from running!
```

```
TSOUTD<-TrainSel(Data=dataDopt,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "UOS",
           Stat = DOPT, control=TSC)
```

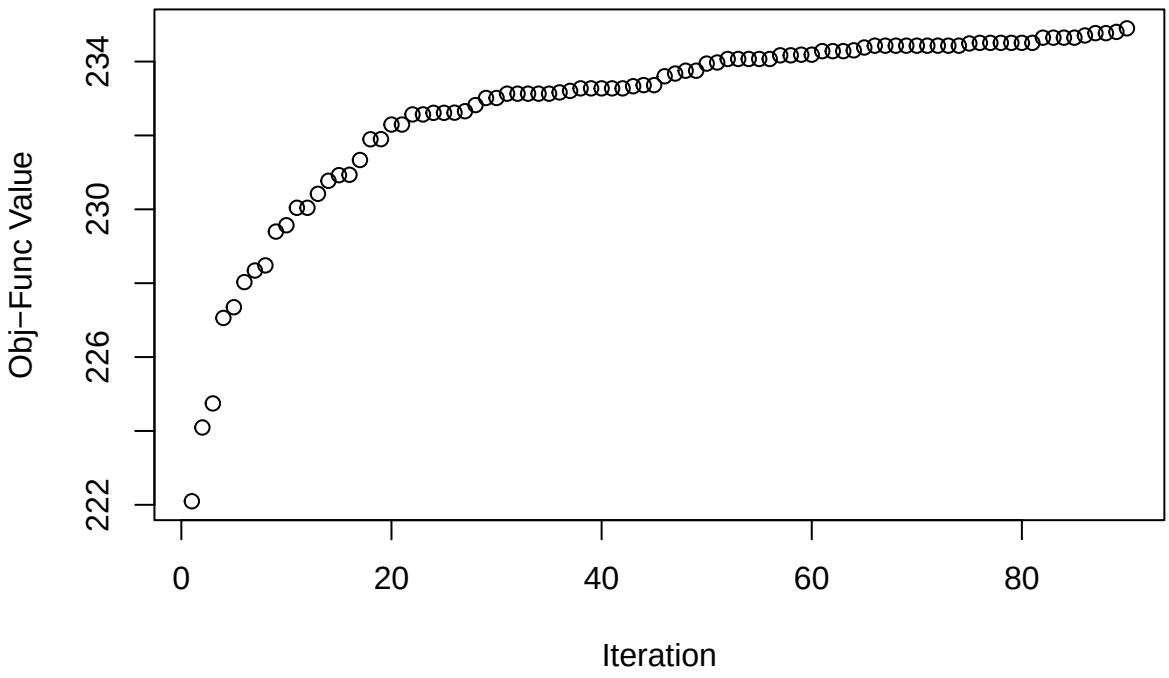
```
## [1] "Using demo version."
## Limited to selecting 100 integer and 3 double solutions with hyperparameters in SetControlDefault(si
## [1] "If you are interested in upgrading to full version, please contact us at j.isidro@upm.es"
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTD$BestSol_int])
```

```
## [1] "Line4"  "Line12" "Line13" "Line15" "Line16" "Line17"
```

Convergence should be checked after the algorithm finishes. We can do this by checking the path of the objective function through the iterations:

```
plot(TSOUTD$maxvec, xlab="Iteration",ylab="Obj-Func Value")
```



CDMEAN-optimality criterion CDMEAN-optimality criterion is also a model based criterion it is based on a G-BLUP mixed model. For the problem of selection of a subset of genotypes for a homogeneous design in a single environment, a G-BLUP model relating the genotypic data to phenotypic measurements in the training data can be expressed as

$$y = 1\mu + Zu + \epsilon$$

with μ a scalar parameter for the mean of the phenotypic measurements, Z the $n \times N$ design matrix for the N genotypes in the candidate set, $\epsilon \sim N_n(0, \sigma_e^2)$ independent of $u \sim N_q(0; \sigma_g^2 G)$.

For this model, the coefficient of determination matrix of $\widehat{u}$ for predicting u is given by

$$(GZ'PZG) \oslash G$$

where $P = V^{-1} - V^{-1}1(1'V^{-1}1)^{-1}1'V^{-1}$ is the projection matrix and $\oslash$ expresses the element-wise division.

The diagonals of this matrix are the coefficient of determination of the predictions for individual genotypes and the mean of these coefficient of determination values over the selected genotypes is called the CDMEAN-optimality criterion[1]. CDMEAN criterion takes values between 0 and 1 and the larger values are preferable. For the un-targeted optimization, the usual practice is to use CDMEAN that is calculated over the genotypes not included in the training set. We can program this objective function to be used in 'TrainSel' as follows:

```
# note that we are not using the target genotypes
dataCDMEANopt<-list(G=Wheat.K[1:160,1:160], lambda=1)

CDMEANOPT<-function(soln, Data){
  G<-Data[["G"]]
```

[1]A risk averse approach would entail maximizing the minimum of selected diagonals.

```
  lambda<-Data[["lambda"]]
  Vinv<-solve(G[soln,soln]+lambda*diag(length(soln)))
  outmat<-(G[,soln]%*%(Vinv-(Vinv%*%Vinv)/sum(Vinv))%*%G[soln,])/G
  return(mean(diag(outmat[-soln,-soln])))
}

TSOUTCD<-TrainSel(Data=dataCDMEANopt,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "UOS",
           Stat = CDMEANOPT,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTCD$BestSol_int])
```

```
## [1] "Line4"  "Line5"  "Line6"  "Line7"  "Line9"  "Line11"
```

Maximin distance criterion Maximin distance criterion is a non-parametric design criteria. An optimal training set of size n from the N candidates is selected by maximizing the minimum [2] genetic distance among the training genotypes, so this is a space filling design. Next, we show how to program this criterion in R:

```
dataMaximin<-list(DistMat=as.matrix(dist(Wheat.M_centered)))

MaximinOPT<-function(soln, Data){
  Dsoln<-Data[["DistMat"]][soln,soln]
  DsolnVec<-Dsoln[lower.tri(Dsoln,diag=FALSE)]
  return(min(DsolnVec))
}

TSOUTMaximin<-TrainSel(Data=dataMaximin,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "UOS",
           Stat = MaximinOPT,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTMaximin$BestSol_int])
```

```
## [1] "Line4"  "Line16" "Line17" "Line19" "Line20" "Line22"
```

Targeted optimization When the focus is on making inferences about the trait values for a known target set of genotypes is using genomic prediction, we can use what we call a targeted optimization criteria.

[2] We could also maximize the mean distance leading to Maximean criterion

Mean PEV criterion based on linear model Dopt criterion is not sensitive to information about the target set of genotypes. Nevertheless, a related linear model based criteria called the mean prediction error variance (Mean PEV) can be used when the genotypic data for the target set is available. This criteria relates to the average prediction error variance of the predictions for the target set of genotypes using the linear model in Equation 1.

```
##Target genotypes are in PC
dataPEVlm<-list(FeatureMat=PC, Target=161:200)


PEVlmOPT<-function(soln, Data){
  Fmat<-Data[["FeatureMat"]]
  targ<-Data[["Target"]]
  return(mean(diag(Fmat[targ,]%*%solve(crossprod(Fmat[soln,]))%*%t(Fmat[targ,]))))
}


TSOUTPEVlm<-TrainSel(Data=dataPEVlm,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "UOS",
           Stat = PEVlmOPT,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTPEVlm$BestSol_int])
```

```
## [1] "Line3"  "Line4"  "Line12" "Line15" "Line19" "Line23"
```

```


##This time the Target genotypes are in G
dataCDMEANTargetOpt<-list(G=Wheat.K, lambda=1, Target=161:200)

CDMEANOPTTarget<-function(soln, Data){
  G<-Data[["G"]]
  lambda<-Data[["lambda"]]
  targ<-Data[["Target"]]
  Vinv<-solve(G[soln,soln]+lambda*diag(length(soln)))
  outmat<-(G[,soln]%*%(Vinv-(Vinv%*%Vinv)/sum(Vinv))%*%G[soln,])/G
  return(mean(diag(outmat[targ,targ])))
}



TSOUTCDTarg<-TrainSel(Data=dataCDMEANTargetOpt,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "UOS",
           Stat = CDMEANOPTTarget,
```

```
            control=TSC,
            Verbose = FALSE)
```

Targeted CDMEAN criterion

```
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTCDTarg$BestSol_int])
```

```
## [1] "Line5"  "Line8"  "Line13" "Line17" "Line23" "Line27"
```

minimax criterion Minimize the maximum genomic distance of training genotypes to test genotypes.

```
dataMiniMax<-list(DistMat=as.matrix(dist(Wheat.M_centered)))

MiniMaxOPT<-function(soln, Data){
  Dsoln<-Data[["DistMat"]][soln,161:200]
  DsolnVec<-max(c(unlist(Dsoln)))
  return(-(DsolnVec))
}

TSOUTMinimax<-TrainSel(Data=dataMiniMax,
            Candidates = list(1:160),
            setsizes = c(50),
            settypes = "UOS",
            Stat = MiniMaxOPT,
            control=TSC,
            Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
head(rownames(Wheat.M)[TSOUTMinimax$BestSol_int])
```

```
## [1] "Line5"  "Line6"  "Line11" "Line14" "Line15" "Line20"
```

Multiple Design Criteria Maximize the mean genomic distance of training genotypes at the same time minimize the mean distance to the target genotypes.

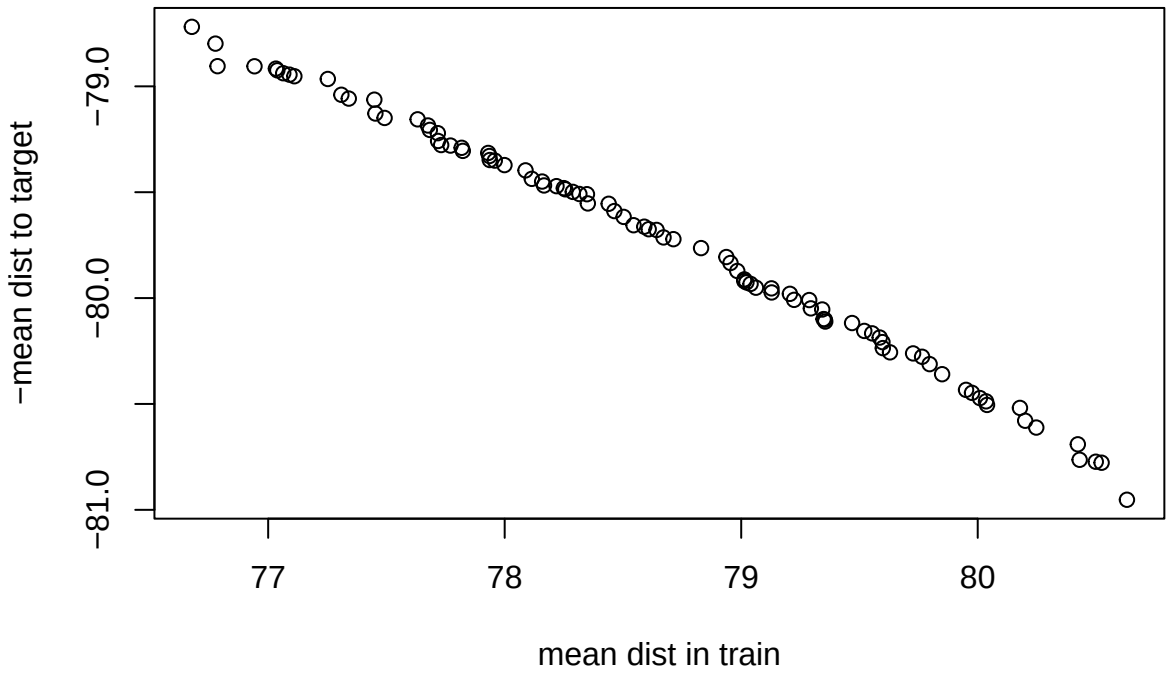
```
dataMultOpt<-list(DistMat=as.matrix(dist(Wheat.M_centered)))

MultOPT<-function(soln, Data){
  D<-Data[["DistMat"]]
  Dsoln<-D[soln,161:200]
  DsolnVec1<- -mean(c(unlist(Dsoln)))
  Dsoln2<-D[soln,soln]
  DsolnVec2<-mean(c(unlist(Dsoln2)))
  return(c(DsolnVec2,DsolnVec1))
}
TSOUTMultOPt<-TrainSel(Data=dataMultOpt,
            Candidates = list(1:160),
```

```
            setsizes = c(50),
            settypes = "UOS",
            Stat = MultOPT,
            nStat=2,
            control=TSC,
            Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

Plot the frontier solutions:

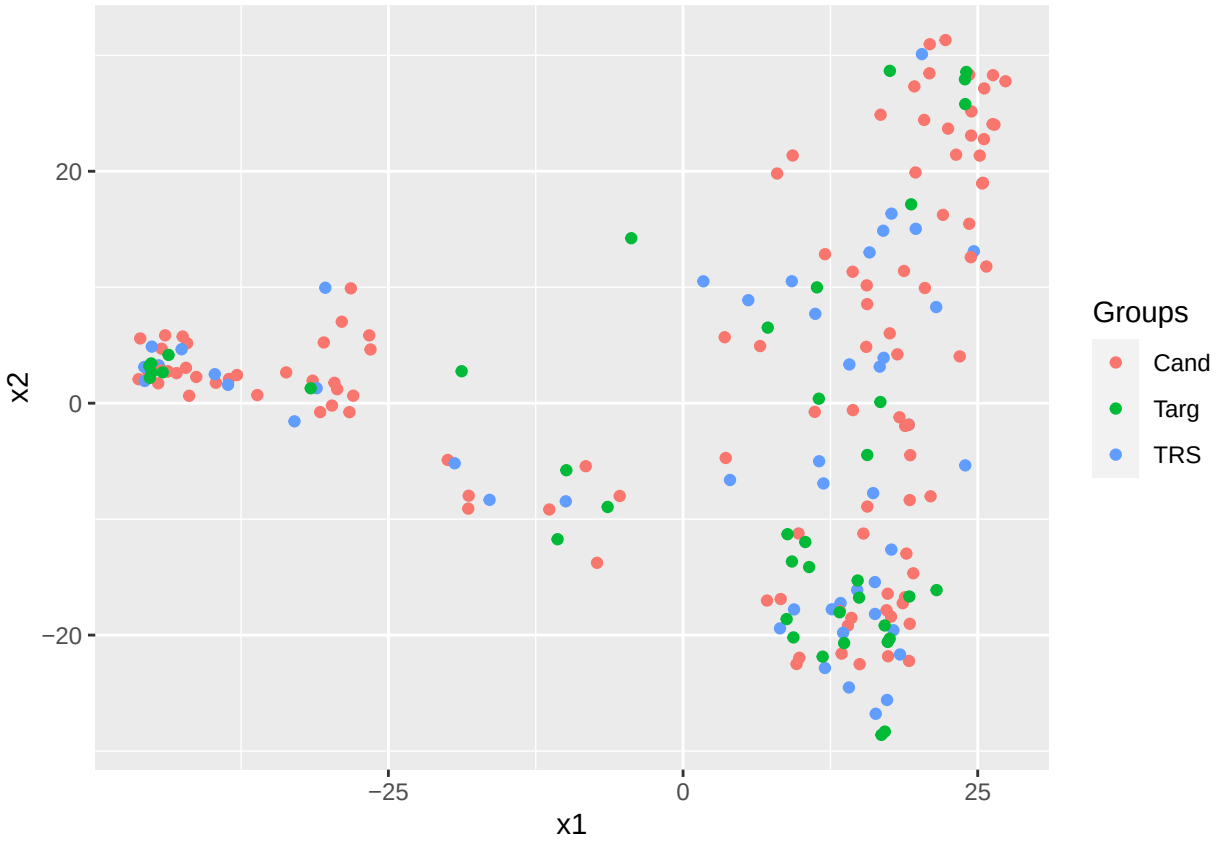
```
FrontierSols<-t(TSOUTMultOPt$BestVal)
plot(FrontierSols, xlab="mean dist in train",
     ylab="-mean dist to target")
```




```
Solns<-which((FrontierSols[,1]>=79) & (FrontierSols[,2]>= -80))# subset of solutions
Selected<-TSOUTMultOPt$BestSol_int[, Solns[1]]

Groups<-rep("Cand",200)
Groups[161:200]<-"Targ"
Groups[Selected]<-"TRS"
PlotData<-data.frame(x1=PC[,1], x2=PC[,2], Groups=factor(Groups))
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.2
```

```
p<-ggplot(PlotData, aes(x=x1,y=x2, color=Groups))+geom_point()
p
```



Selection of a subset of genotypes for an known design in a single environment

In certain cases, we are looking for conducting a phenotypic experiment with n training genotypes selected out of N candidate genotypes but in addition, we also have a particular blocking structure and environmental covariates involved in the design of the experiment. Suppose the matrix E is the $n \times p$ environmental covariates matrix. For instance, this matrix could be the design matrix for a row-column blocking within the environment. We assume still that we want make inferences about the genomic values after accounting for these covariates. In this case, the order in which the genotpes are positioned in the environment will be important. Perhaps, we would like to use similar genotypes in genotypes in dissimilar blocks and also we would like to observe as genetically distant genotypes within similar blocks.

Un-targeted optimization

D-optimality criterion with environmental covariates D-optimality criterion can be easily adopted for this purpose. FIrts we write the model as

$$y = E\beta_{env} + f(M)\beta_f + \epsilon$$

where y is the n vector of phenotypic measurements in the training data, E is the $n \times p$ design matrix for the environmental covariates, β_{env} is the p vector of the effects of the environmental covariates, M is the $n \times m$

the marker matrix for the n training genotypes, $f(M)$ is a genomic features matrix with dimensions $n \times q$, β_f is the q vector of effects of genomic features, and ϵ is the residual error vector of length n. We further assume that the elements of ϵ are independent and identically distributed with a normal distribution with zero mean and variance σ_e^2. Under this model the variance of the estimators for β is known to be proportional to $[f(M)'(I - E(E'E)^{-1}E')f(M)]^{-1}$. D-optimal selection of n training genotypes from N individuals in the candidate set involves minimizing the determinant of this matrix (or equivalently maximizing the log-determinant of $f(M)'(I - E(E'E)^{-1}E')f(M)$). The matrix $(I - E(E'E)^{-1}E')$ is the projection matrix to the orthogonal space of column space of E.

```
E<-data.frame(expand.grid(row=paste("row",1:5, sep="_"),
                          col=paste("col",1:10, sep="_")))
E$row<-as.factor(E$row)
E$col<-as.factor(E$col)


DesignE<-model.matrix(~row+col+row*col, data=E)

P<-diag(nrow(DesignE))-DesignE%*%solve(crossprod(DesignE))%*%t(DesignE)

dataDoptEnv<-list(FeatureMat=PC, Projection=P)

DOPTwithE<-function(soln, Data){
  Fmat<-Data[["FeatureMat"]]
  P<-Data[["Projection"]]
  return(determinant(crossprod(P%*%Fmat[soln,]), logarithm=TRUE)$modulus)
}

TSOUTDwithE<-TrainSel(Data=dataDoptEnv,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "OS",
           Stat = DOPTwithE,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
TSOUTDwithE$BestSol_int #order of this is important
```

```
##  [1]  21 103  70 139  65  94  79 105 109 112 135 104 142  47  81  18  15  58  53
## [20] 110  61  25  41 124 153  90  69 148 118  97 149  73  42  13  95  99 140  62
## [39]  83 154  78 128 108  31  71  49  45  17  85 125
```

```
head(rownames(Wheat.M)[TSOUTDwithE$BestSol_int])
```

```
## [1] "Line21"  "Line103" "Line70"  "Line139" "Line65"  "Line94"
```

```
E$GID<-rownames(Wheat.M)[TSOUTDwithE$BestSol_int]
###Here is the final design
head(E)
```

```
##     row   col     GID
## 1 row_1 col_1  Line21
## 2 row_2 col_1 Line103
## 3 row_3 col_1  Line70
## 4 row_4 col_1 Line139
## 5 row_5 col_1  Line65
## 6 row_1 col_2  Line94
```

CDMEAN-optimality criterion with environmental covariates We can also use environmental covariates with the CDMEAN-optimality criterion. In ordor to do this we first need to add the environmental covariates into the G-BLUP model. This model is written as

$$y = E\beta_{env} + +Zu + \epsilon$$

with E is the $n \times p$ design matrix for the environmental covariates, β_{env} is the p vector of the effects of the environmental covariates, Z is the $n \times N$ design matrix for the N genotypes in the candidate set, $\epsilon \sim N_n(0, \sigma_e^2)$ is independent of $u \sim N_q(0; \sigma_g^2 G)$.

For this model, the coefficient of determination matrix of $\widehat{u}$ for predicting u is given by

$$(GZ'PZG) \oslash G$$

where $P = V^{-1} - V^{-1}E(E'V^{-1}E)^{-1}E'V^{-1}$ is the projection matrix and $\oslash$ expresses the element-wise division.

```
dataCDMEANoptwithEnv<-list(G=Wheat.K[1:160,1:160],E=DesignE, lambda=1)

CDMEANOPTwithEnv<-function(soln, Data){
  G<-Data[["G"]]
  E<-Data[["E"]]

  lambda<-Data[["lambda"]]
  Vinv<-solve(G[soln,soln]+lambda*diag(length(soln)))
  outmat<-(G[,soln]%*%
             (Vinv-Vinv%*%E%*%solve(t(E)%*%Vinv%*%E)%*%t(E)
              %*%Vinv)%*%G[soln,])/G
  return(mean(diag(outmat[-soln,-soln])))
}


TSOUTCDwithENV<-TrainSel(Data=dataCDMEANoptwithEnv,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "OS",
           Stat = CDMEANOPTwithEnv,
           control=TSC,
           Verbose = FALSE)
```

```
## Convergence Achieved
##  (no improv in the last 'minitbefstop' iters).
```

```
E$GID<-rownames(Wheat.M)[TSOUTCDwithENV$BestSol_int]
###Here is the final design
head(E)
```

```
##     row   col     GID
## 1 row_1 col_1  Line82
## 2 row_2 col_1  Line37
## 3 row_3 col_1  Line22
## 4 row_4 col_1  Line31
## 5 row_5 col_1  Line32
## 6 row_1 col_2 Line149
```

Targeted optimization and allowing for replicates We can also do targeted optimization in this case with minimal change to the last code:

```
dataCDMEANoptwithEnvTarget<-list(G=Wheat.K,E=DesignE,Target=161:200, lambda=1)

CDMEANOPTwithEnvTarget<-function(soln, Data){
  G<-Data[["G"]]
  E<-Data[["E"]]
  targ<-Data[["Target"]]
  lambda<-Data[["lambda"]]
  Vinv<-solve(G[soln,soln]+lambda*diag(length(soln)))
  outmat<-(G[,soln]%*%
             (Vinv-Vinv%*%E%*%solve(t(E)%*%Vinv%*%E)%*%t(E)%*%Vinv)
           %*%G[soln,])/G
  return(mean(diag(outmat[targ,targ])))
}



TSOUTCDwithENVTarg<-TrainSel(Data=dataCDMEANoptwithEnvTarget,
           Candidates = list(1:160),
           setsizes = c(50),
           settypes = "OMS",
           Stat = CDMEANOPTwithEnvTarget,
           control=TSC,
           Verbose = FALSE)
```

```
## Convergence Achieved
##  (no improv in the last 'minitbefstop' iters).
```

```
E$GID<-rownames(Wheat.M)[TSOUTCDwithENVTarg$BestSol_int]
###Here is the final design
head(E)
```

```
##     row   col     GID
## 1 row_1 col_1  Line28
## 2 row_2 col_1  Line95
## 3 row_3 col_1 Line143
## 4 row_4 col_1 Line108
## 5 row_5 col_1  Line23
## 6 row_1 col_2 Line111
```

Design for a phenotypic experiment in multiple environments

In practice, most genomic selection experiments are performed over multiple environments. Designing genomic selection over multiple environments means we would like to choose training genotypes to use in each of these environments and for genomic selection the distribution of the alleles within and between the different environments can be arranged optimally for obtaining better generalization performance.

Using CDMEAN-optimality criterion for genomic selection experiment design in multiple environments As before, we first need to state the underlying model. For multi-environmental trials, a commonly used genomic prediction model is the multi-environmental G-BLUP model. Suppose we have 3 environments, in Environment 1 we can accommodate 30 genotypes in a 6-rows 5-columns design, in Environment 2 we can accommodate 20 genotypes in a 4-rows 5-columns design, and in Environment 3 we can accommodate 50 genotypes in an unknown homogeneous design. We assume that the genomic covariance of the environments is (proportionally) equal to the matrix

$$V_g = \begin{pmatrix} 1.0 & .7 & .5 \\ .7 & 1.2 & .8 \\ .5 & .8 & 1.5 \end{pmatrix}$$

We assume that the residual errors in these environments are correlated with the following covariance matrix

$$V_e = \begin{pmatrix} 1.0 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 1.0 \end{pmatrix}.$$

We can express the model for the training data as follows:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = E\beta_{env} + \begin{pmatrix} Z_1 u_1 \\ Z_2 u_2 \\ Z_3 u_3 \end{pmatrix} + \begin{pmatrix} Z_1 \epsilon_1 \\ Z_2 \epsilon_2 \\ Z_3 \epsilon_3 \end{pmatrix}$$

where E is the $n \times p$ design matrix for the environmental covariates, β_{env} is the p vector of the effects of the environmental covariates, y_i is an n_i vector and Z_i is the $n_i \times N$ design matrix for the N genotypes in the candidate set in environment i for $i = 1, 2, 3$. In addition, we assume $(\epsilon_1, \epsilon_2, \epsilon_3) \sim N_{N\times 3}(0, I_N, V_e)$ is independent of $(u_1, u_2, u_3) \sim N_{N\times 3}(0; G, V_g)$. This means that the vectorized form of these matrices $\epsilon = vec(\epsilon_1, \epsilon_2, \epsilon_3)$ and $u = vec(u_1, u_2, u_3)$ are independently distributed as $N_{3N}(0, V_e \otimes I_N)$ and $N_{3N}(0, V_g \otimes G)$.

In our case $n_1 = 30, n_2 = 20, n_3 = 50$, and $n = n_1 + n_2 + n_3 = 100$. Here is how you can approach this problem using 'TrainSel':

```
Vg=matrix(c(1.0 , .7 , .5 , .7 , 1.2 ,.8 , .5 , .8 , 1.5), 3,3)
Ve=matrix(c(1.0, 0,0  , 0,1.5,0 , 0,0,1.0), 3,3)
rownames(Vg)<-colnames(Vg)<-rownames(Ve)<-colnames(Ve)<-paste("E",1:3, sep="")
G<-kronecker(Vg, Wheat.K, make.dimnames = TRUE)
R<-kronecker(Ve, diag(nrow(Wheat.K)), make.dimnames = TRUE)
#### Note the shape of G (same as R)
head(rownames(G))
```

```
## [1] "E1:Line1" "E1:Line2" "E1:Line3" "E1:Line4" "E1:Line5" "E1:Line6"
```

```
tail(rownames(G))
```

```
## [1] "E3:Line195" "E3:Line196" "E3:Line197" "E3:Line198" "E3:Line199"
## [6] "E3:Line200"
```

```
dim(G)
```

```
## [1] 600 600
```

By examining the shape of the G matrix above we see that the candidate set are on the 1st through 160th, 200th through 360th and,400th through 560th rows and columns of this matrix. We want to use 30 from 1 through 160, 20 from 200 through 360 and 50 from 400 through 560. The first two sets are ordered and last one is unordered. We are also going to assume duplicates within an environment are not allowed.

Design Matrix for the environments is obtained below. I am assuming no interaction effects between rows and columns.

```
E1<-(expand.grid(row=paste("row",1:6, sep="_"),
                 col=paste("col",1:5, sep="_")))


E2<-(expand.grid(row=paste("row",1:4, sep="_"),
                 col=paste("col",1:5, sep="_")))


E3<-data.frame(row=paste("row",rep(1,50),sep="_"),
               col=paste("col",rep(1,50), sep="_"))



EnvData<-data.frame(Env=c(rep("E1", 30), rep("E2", 20),rep("E3", 50)),
                 rbind(E1,E2,E3))
DesignE<-model.matrix(~Env+Env/(row+col), data=EnvData)
DesignE<-DesignE[,colSums(DesignE)>0]
## This is the names of the fixed effects design matrix
colnames(DesignE)
```

```
##  [1] "(Intercept)"    "EnvE2"          "EnvE3"          "EnvE1:rowrow_2"
##  [5] "EnvE2:rowrow_2" "EnvE1:rowrow_3" "EnvE2:rowrow_3" "EnvE1:rowrow_4"
##  [9] "EnvE2:rowrow_4" "EnvE1:rowrow_5" "EnvE1:rowrow_6" "EnvE1:colcol_2"
## [13] "EnvE2:colcol_2" "EnvE1:colcol_3" "EnvE2:colcol_3" "EnvE1:colcol_4"
## [17] "EnvE2:colcol_4" "EnvE1:colcol_5" "EnvE2:colcol_5"
```

```
dataCDMEANoptwithEnvME<-list(G=G,R=R, E=DesignE)
Target<-c(161:200, 361:400,561:600)
# we will exclude these since we assume nontargeted
CDMEANOPTwithEnvME<-function(soln, Data){
  G<-Data[["G"]]
  R<-Data[["R"]]
  E<-Data[["E"]]
  Vinv<-solve(G[soln,soln]+R[soln,soln])
  outmat<-(G[,soln]
           %*%(Vinv-Vinv%*%E%*%solve(t(E)%*%Vinv%*%E)%*%t(E)%*%Vinv)
           %*%G[soln,])/G
  return(mean(diag(outmat[-c(soln, Target),-c(soln, Target)])))
  #returning the mean CD on the remaining GIDs
  #(1:200 after deleting Target and the training.)
```

```
}


TSOUTCDwithENVME<-TrainSel(Data=dataCDMEANoptwithEnvME,
           Candidates = list(1:160, 201:360,401:560),
           setsizes = c(30,20,50),
           settypes = c("OS","OS","UOS"),
           Stat = CDMEANOPTwithEnvME,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
##Putting GIDs in the design #first 30 are Env1
E1$GID<-rownames(G)[TSOUTCDwithENVME$BestSol_int[1:30]]
###Here is the final design for env 1
head(E1)
```

```
##     row   col       GID
## 1 row_1 col_1 E1:Line69
## 2 row_2 col_1 E1:Line57
## 3 row_3 col_1 E1:Line45
## 4 row_4 col_1 E1:Line64
## 5 row_5 col_1 E1:Line10
## 6 row_6 col_1 E1:Line75
```

```
##second 20 are Env2

E2$GID<-rownames(G)[TSOUTCDwithENVME$BestSol_int[31:50]]
###Here is the final design for env 2
head(E2)
```

```
##     row   col        GID
## 1 row_1 col_1 E2:Line158
## 2 row_2 col_1  E2:Line50
## 3 row_3 col_1 E2:Line125
## 4 row_4 col_1 E2:Line156
## 5 row_1 col_2  E2:Line54
## 6 row_2 col_2 E2:Line135
```

```
##Last 50 are Env3
E3$GID<-rownames(G)[TSOUTCDwithENVME$BestSol_int[51:100]]
###Here is the final design for env 3
head(E3)
```

```
##     row   col       GID
## 1 row_1 col_1  E3:Line3
## 2 row_1 col_1  E3:Line5
## 3 row_1 col_1  E3:Line9
## 4 row_1 col_1 E3:Line11
## 5 row_1 col_1 E3:Line16
## 6 row_1 col_1 E3:Line21
```

Implementing a penalty function for the total number of genotypes in the experiment

Suppose we want to restrict the total number of unique genotypes used in the above multi-environmental design to between 80 and 90. We can use a penalty function approach to impose this constraint: A penalty function forces the optimization algorithm to find desired solutions by assigning the points that don't satisfy the constraints values that is small with respect to the values of the optimization criteria (the size of this value might depend on how far the solutions are from satisfying the constrains), if the constraints are satisfied then the then the penalty functions returns zero.

```
PenaltyFunction<-function(soln){
  soln1<-soln[1:30]
  soln2<-soln[31:50]-200
  soln3<-soln[51:100]-400
  numuniquegeno<-length(unique(c(soln1,soln2,soln3)))
  if( (numuniquegeno<80) |  (90<numuniquegeno)){
    return(min(numuniquegeno-80, 90-numuniquegeno))
     } else {
    return(0)
    }
}
```

```
CDMEANOPTwithEnvMEwithPenalty<-function(soln, Data){
  penalty<-PenaltyFunction(soln)
  if (penalty==0){
  G<-Data[["G"]]
  R<-Data[["R"]]
  E<-Data[["E"]]
  Vinv<-solve(G[soln,soln]+R[soln,soln])
  outmat<-(G[,soln]
           %*%(Vinv-Vinv%*%E%*%solve(t(E)%*%Vinv%*%E)%*%t(E)%*%Vinv)
           %*%G[soln,])/G
  return(mean(diag(outmat[-c(soln, Target),-c(soln, Target)])))
  } else {return(penalty)}
}
```

```
TSOUTCDwithENVMEwithPenalty<-TrainSel(Data=dataCDMEANoptwithEnvME,
           Candidates = list(1:160, 201:360,401:560),
           setsizes = c(30,20,50),
           settypes = c("OS","OS","UOS"),
           Stat = CDMEANOPTwithEnvMEwithPenalty,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
E1$GID<-rownames(G)[TSOUTCDwithENVMEwithPenalty$BestSol_int[1:30]]
###Here is the final design for env 1
head(E1)
```

```
##     row   col        GID
## 1 row_1 col_1 E1:Line146
```

```
## 2 row_2 col_1  E1:Line29
## 3 row_3 col_1  E1:Line51
## 4 row_4 col_1 E1:Line117
## 5 row_5 col_1  E1:Line46
## 6 row_6 col_1 E1:Line152
```

```
E2$GID<-rownames(G)[TSOUTCDwithENVMEwithPenalty$BestSol_int[31:50]]
###Here is the final design for env 2
head(E2)
```

```
##     row   col        GID
## 1 row_1 col_1  E2:Line47
## 2 row_2 col_1  E2:Line83
## 3 row_3 col_1 E2:Line105
## 4 row_4 col_1  E2:Line99
## 5 row_1 col_2  E2:Line31
## 6 row_2 col_2 E2:Line115
```

```
E3$GID<-rownames(G)[TSOUTCDwithENVMEwithPenalty$BestSol_int[51:100]]
###Here is the final design for env 3
head(E3)
```

```
##     row   col       GID
## 1 row_1 col_1  E3:Line3
## 2 row_1 col_1  E3:Line4
## 3 row_1 col_1  E3:Line9
## 4 row_1 col_1 E3:Line10
## 5 row_1 col_1 E3:Line11
## 6 row_1 col_1 E3:Line12
```

Other usage examples in plant breeding

'TrainSel' can be adopted to be used in optimization of problems other than the design of training populations for genomic prediction. The following examples demonstrate use of the package in some mixed integer optimization problems related to plant breeding.

Unsupervised Variable Selection

In some cases, we are interested in a subset of markers that represent the information contained in the all of the available markers. We can use the distance correlation measure between the marker data with all of the markers and the marker data with a selected set of markers to aid us in this process. We want to maximize the distance correlation measure for a given set of markers of size say 100. You can do this with the following code:

```
library(energy)
```

```
## Warning: package 'energy' was built under R version 4.2.3
```

```
dataVarSel<-list(dist(Wheat.M_centered), Wheat.M_centered)
funVarSel<-function(soln,Data){
out<-bcdcor(Data[[1]],dist(Data[[2]][,soln]))
return(out)
}

TSOUTVarSel<-TrainSel(Data=dataVarSel,
           Candidates = list(1:ncol(Wheat.M_centered)),
           setsizes = c(100),
           settypes = c("UOS"),
           Stat = funVarSel,
           control=TSC,
           Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
#to check convergence
#plot(TSOUTVarSel$maxvec, "Distance Correlation")
AnchorMarkers<-TSOUTVarSel$BestSol_int
```

Optimal parental proportions and the Pareto front

In this example we want to select 3 parents out of the 40 genotypes in the target set and provide parental proportions for those 3 parents so that the mean GEBVs (estimated using a model based on the anchor markers) for the progeny of these parents mixed in these proportions would be maximized and inbreeding of the progeny is minimized.

We first need to estimate the marker effects. In this example we will only use the anchor markers, and the model is trained on all 160 candidate genotypes.

```
library(EMMREML)
```

```
## Warning: package 'EMMREML' was built under R version 4.2.3
```

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

```
mixedmodelout<-emmreml(y=Wheat.Y[1:160,2],
                       X=matrix(1, nrow=160, ncol=1),
                       Z=Wheat.M_centered[1:160,AnchorMarkers],
                       K=diag(length(AnchorMarkers)))
markereffects<-mixedmodelout$uhat
```

Using the estimated marker effects we can calculate GEBVs for the 40 prospective parents in the Target set. We also subset the genomic relationship matrix for these 40 genotypes from the whole genomic relationship matrix.

```
GEBVs40<-Wheat.M_centered[161:200,AnchorMarkers]%*%markereffects
K40<-Wheat.K[161:200,161:200]
```

Since we know the phenotypic values for the Target in our case, we can povide an estimate of the accuracy for the GEBVs in the Target set:

```r
cor(GEBVs40,Wheat.Y[161:200,2])
```

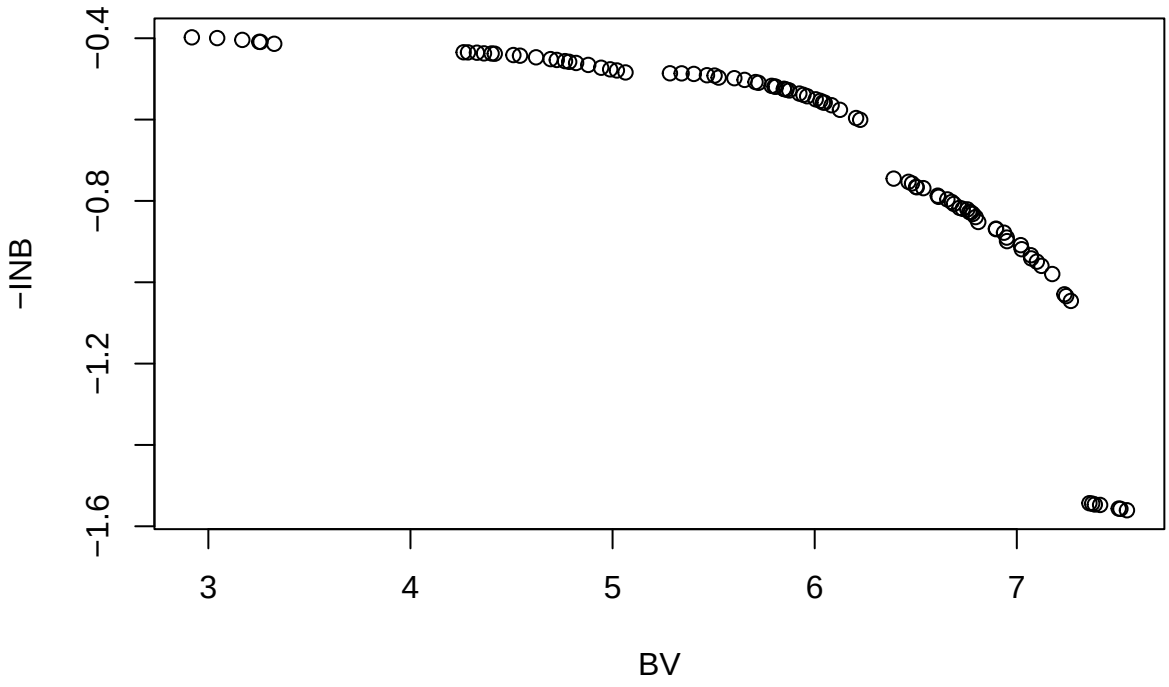
```
##           [,1]
## [1,] 0.7620872
```

Now, we are ready to setup the problem: We want to select 3 genotypes as parents and also obtain the parental proportions for these parents so that the mean GEBVs for the progeny is maximized while the inbreeding in the progeny is minimized.

```r
dataOptProp<-list(GEBVs40,K40)
funOptProp<-function(soln_int,soln_dbl,Data){
  props<-soln_dbl/sum(soln_dbl) #to rescale the solution to sum up to 1.
  BV<-crossprod(Data[[1]][soln_int],props)
  Inb<-t(props)%*%Data[[2]][soln_int,soln_int]%*%props
  return(c(BV,-c(Inb)))
}



TSOUTOptProp<-TrainSel(Data=dataOptProp,
           Candidates = list(1:40, c(.001,.999)),
           setsizes = c(3, 3),
           settypes = c("OS","DBL"),
           Stat = funOptProp,
           nStat = 2,
           Verbose = FALSE,
           control = TSC)
```

```
## Maximum number of iterations reached.
```

```r
plot(t(TSOUTOptProp$BestVal), xlab="BV", ylab="-INB")
```



Optimal genomic mating

2 objectives, minimize inbreeding maximize gain We want to select 100 mates so that the mean GEBVs for the progeny is maximized while the inbreeding in the progeny is minimized.

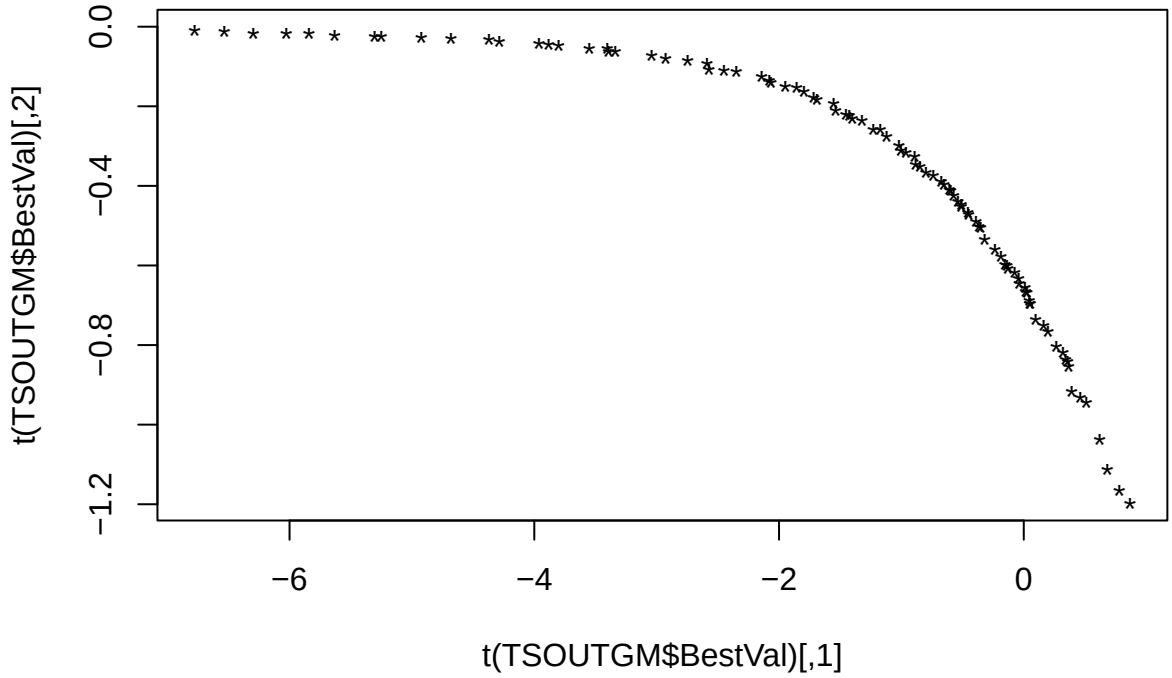
```
library(rrBLUP)
##All possible mates
#males 161:200
#females 161:200
#this gives 1600 possible mates.
DFMates<-expand.grid(rownames(Wheat.M[161:180,]),rownames(Wheat.M[181:200,]))
M1<-Wheat.M[161:180,AnchorMarkers]
M2<-Wheat.M[181:200,AnchorMarkers]
M<-rbind(M1, M2)
K<-A.mat(M)
BVParents<-M%*%markereffects
```

```
###selecting 50 mates,  setsizes=50
###Note Stat=BV_INB, nStat = 2,
BV_INB<-function(soln){
  DFMatessoln<-DFMates[soln,]
  P1<-factor(DFMatessoln[,1], levels=rownames(M1))
  P2<-factor(DFMatessoln[,2], levels=rownames(M2))
  P1<-model.matrix(~P1-1)
  P2<-model.matrix(~P2-1)
```

```
  P<-cbind(P1/2,P2/2)
  PKP<-mean(tcrossprod(P%*%K,P)) #inbreeding
  BVmatessoln<-mean(P%*%BVParents)# gain
  out<-c(BVmatessoln,-as.numeric(PKP))
  names(out)<-c("Gain","-Inb")
  return(out)
}



TSOUTGM<-TrainSel(Candidates = list(1:nrow(DFMates)),
                  setsizes=50,
                  settypes = c("UOMS"),
                  Stat=BV_INB,
                  nStat = 2,
                  Verbose = FALSE,
                  control = TSC)
```

```
## Maximum number of iterations reached.
```

```
plot(t(TSOUTGM$BestVal), pch="*")
```




```
##dimension of the problem * number of solutions
dim(TSOUTGM$BestSol_int)
```

```
## [1] 50 94
```

```
########solution 1
head(DFMates[TSOUTGM$BestSol_int[,1],])
```

```
##          Var1    Var2
## 41    Line161 Line183
## 46    Line166 Line183
## 112   Line172 Line186
## 112.1 Line172 Line186
## 112.2 Line172 Line186
## 112.3 Line172 Line186
```

```
TSOUTGM$BestVal[,1]
```

```
## [1]  0.04881438 -0.68972672
```

```
########solution 2
head(DFMates[TSOUTGM$BestSol_int[,2],])
```

```
##         Var1    Var2
## 32   Line172 Line182
## 32.1 Line172 Line182
## 39   Line179 Line182
## 41   Line161 Line183
## 46   Line166 Line183
## 46.1 Line166 Line183
```

```
TSOUTGM$BestVal[,2] ##, etc,...
```

```
## [1] -3.96142540 -0.03749246
```

3 objectives, minimize inbreeding maximize gain and crossvariance When concentrating on mates instead of the parental proportions we can introduce another statistic nto the optimization problem. We want to have high within family variance for the selected mates. We have included two functions to calculate the mean variance of a mating plan, namely, 'calculatecrossvalueM1' and 'calculatecrossvalueM2'.

```
##############################
 BV_VAR_INB<-function(soln){
   DFMatessoln<-DFMates[soln,]
   P1<-factor(DFMatessoln[,1], levels=rownames(M1))
   P2<-factor(DFMatessoln[,2], levels=rownames(M2))
   P1<-model.matrix(~P1-1)
   P2<-model.matrix(~P2-1)
   P<-cbind(P1/2,P2/2)
  BVmatessoln<-mean(P%*%BVParents)# gain


   varI<-mean(sapply(1:nrow(DFMatessoln), function(i){
     p1<-DFMatessoln[i,1]
```

```
    p2<-DFMatessoln[i,2]
    m1<-M1[p1,]
    m2<-M2[p2,]
   calculatecrossvalueM1(round(m1),round(m2),markereffects)
   ## calculatecrossvalueM2(round(m1),round(m2),markereffects, markermap)
  }))

 PKP<-mean(tcrossprod(P%*%K,P)) #inbreeding

  out<-c(BVmatessoln, varI,-as.numeric(PKP))
  names(out)<-c("Gain", "Var","-Inb")
  return(out)

}
```

Note that we we would need to increase the 'npop' and 'niterations' parameters to get a good coverage of the frontier surface. This can be done automatically using the SetControlDefault function (explained later). However, it would exceed demo limitations.

```
####Three objectives, minimize inbreeding maximize gain and cross-variance
###selecting 50 mates, setsizes=50
TSOUT<-TrainSel(Candidates = list(1:nrow(DFMates)),
                setsizes=5,
                settypes = c("UOMS"),
                Stat=BV_VAR_INB,
                nStat = 3,
                control = TSC,
                Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
TSOUT$BestVal[,1:5] # values for first 5 results
```

```
##              [,1]         [,2]        [,3]       [,4]        [,5]
## [1,] -5.956143368 -5.769073223 -7.87578220 -5.8086769 -4.46556834
## [2,]  0.001362463  0.002198758  0.12951878  0.2618012  0.03849388
## [3,] -0.068184756 -0.103724948 -0.07373532 -0.1194161 -0.11096694
```

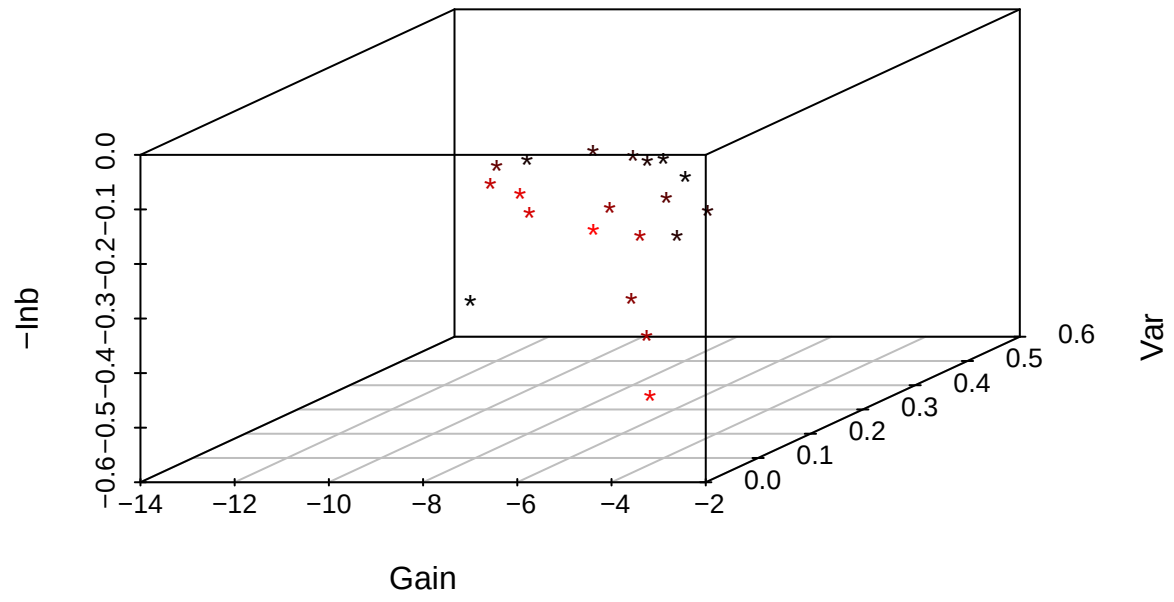
```
ncol(TSOUT$BestVal) # Number of solutions found
```

```
## [1] 20
```

```
library(scatterplot3d)
```

```
## Warning: package 'scatterplot3d' was built under R version 4.2.3
```

```
sd3<-scatterplot3d(t(TSOUT$BestVal), pch="*",highlight.3d=TRUE, xlab="Gain",
                   ylab="Var", zlab="-Inb")
```




```
########solution 1
 head(DFMates[TSOUT$BestSol_int[,1],])
```

```
##         Var1    Var2
## 97   Line177 Line185
## 102  Line162 Line186
## 208  Line168 Line191
## 340  Line180 Line197
## 394  Line174 Line200
```

```
 TSOUT$BestVal[,1]
```

```
## [1] -5.956143368  0.001362463 -0.068184756
```

```
# ########solution 20
 head(DFMates[TSOUT$BestSol_int[,10],])
```

```
##         Var1    Var2
## 55   Line175 Line183
## 113  Line173 Line186
## 174  Line174 Line189
## 282  Line162 Line195
## 393  Line173 Line200
```

```
 TSOUT$BestVal[,10]# ###,etc,...
```

```
## [1] -3.1927290143  0.0009388288 -0.4374289508
```

```
fobj <- function(x)  (x^2+x)*cos(x)
lbound <- -10; ubound <- 10
x<-seq(lbound, ubound, length=100)
```

```
TSout<-TrainSel(Candidates=list(c(lbound,ubound)),
                setsizes = 1,
                settypes = c("DBL"),
                Stat=fobj,
                Verbose = FALSE,
                control = TSC)
```

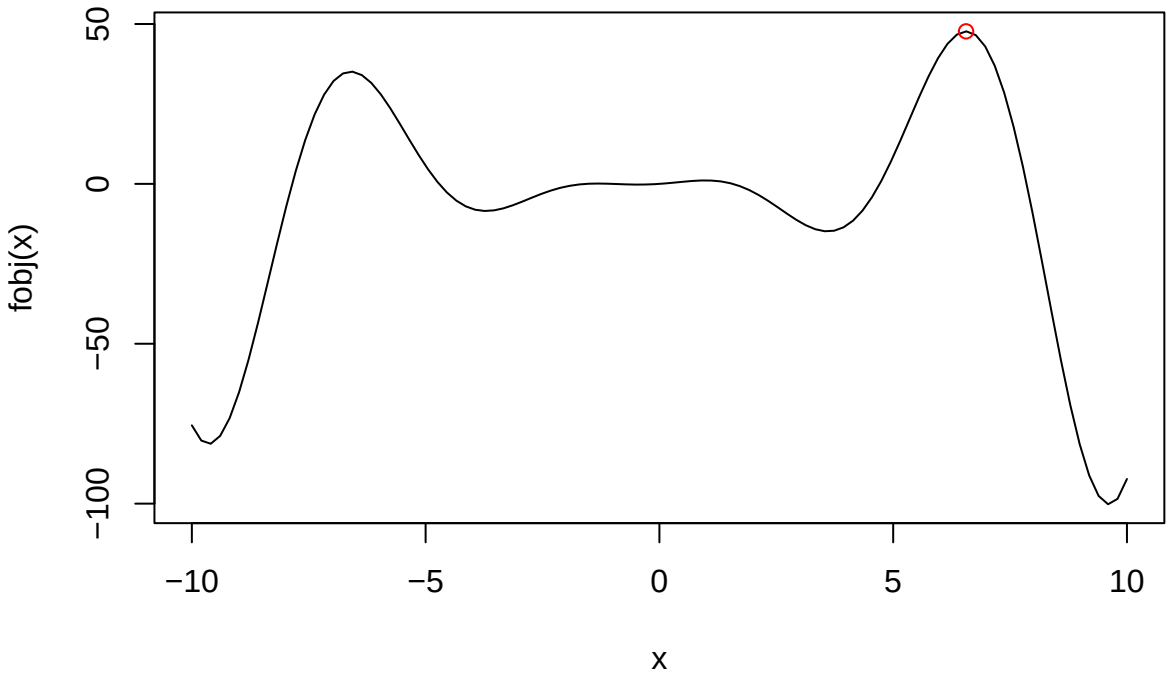
Finding maximum of a function with a single continuous input

```
## Convergence Achieved
##  (no improv in the last 'minitbefstop' iters).
```

```
TSout$BestSol_DBL
```

```
## [1] 6.55883
```

```
plot(x, fobj(x), type="l")
points(TSout$BestSol_DBL,TSout$BestVal, col="red")
```




```
Rastrigin <- function( x)
{
  x1=x[1]
  x2=x[2]
 -(20 + x1^2 + x2^2 - 10*(cos(2*pi*x1) + cos(2*pi*x2)))
}
```

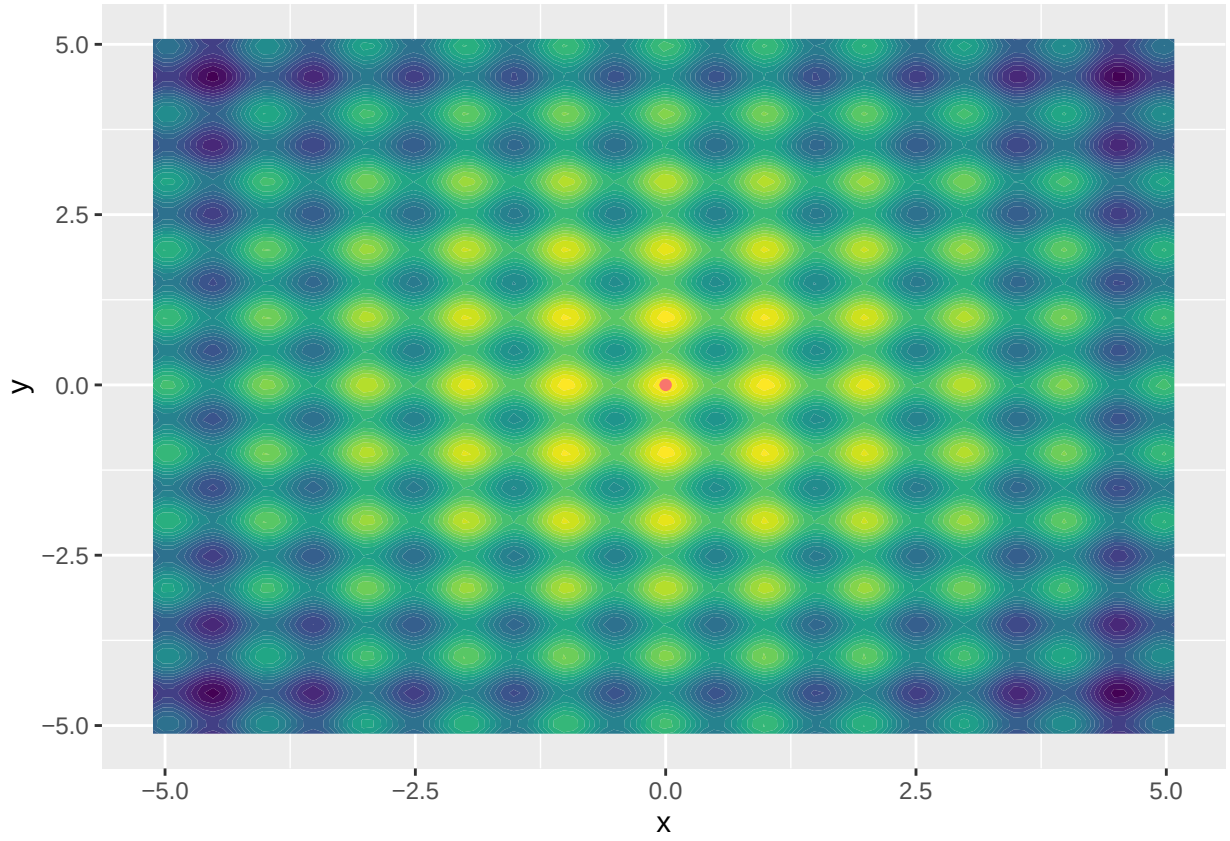
```
TSout2<-TrainSel(Candidates=list(c(-10, 5.12), c(-5.12, 10)),
                 setsizes = 2,
                settypes = c("DBL", "DBL"),
                Stat=Rastrigin,
                Verbose = FALSE,
                control = TSC)
```

Finding maximum of a function with two continuous inputs

```
## Convergence Achieved
##  (no improv in the last 'minitbefstop' iters).
```

```
x1best<-TSout2$BestSol_DBL[1]
x2best<-TSout2$BestSol_DBL[2]
xbest<-data.frame(x=x1best, y=x2best)
x1 <- x2 <- seq(-5.12, 5.12, by = 0.1)
x12<-expand.grid(x1, x2)
f <- apply(x12, 1,function(x){Rastrigin(x)})
plotdata<-cbind(x12,f)
colnames(plotdata)<-c("x","y","z")
```

```
library(ggplot2)
p<-ggplot(plotdata, aes(x, y)) + stat_contour_filled(aes(z = z), bins=30)
p<-p+ geom_point(data = xbest, aes(color="blue"))+theme(legend.position = "none")
p
```




```
library(devtools)
```

```
## Loading required package: usethis
```

```
library(BBmisc)
```

```
## Warning: package 'BBmisc' was built under R version 4.3.2
```

```
##
## Attaching package: 'BBmisc'
```

```
## The following object is masked from 'package:base':
##
##     isFALSE
```

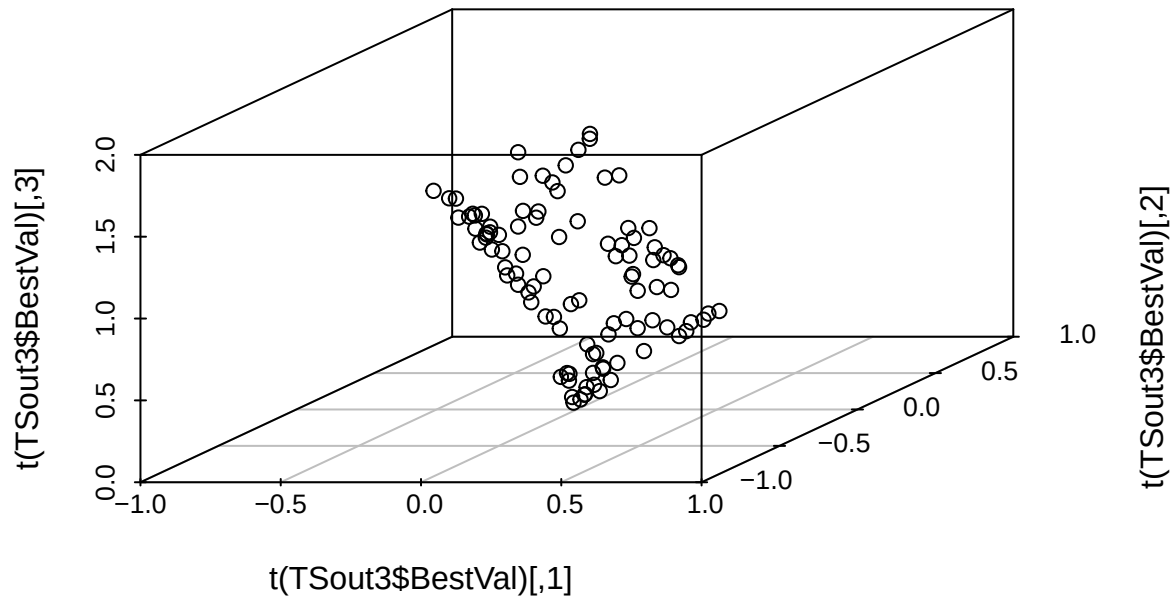
```
fn = function(x){
  return(c(x[1]^2-x[2]^2,sin(x[2]^2-x[1]^2),x[1]^2+x[2]^2))
  }
lower = c(0,0,0)
upper = c(1,1,1)
fn(c(0,1))
```

```
## [1] -1.000000  0.841471  1.000000
```

```
tsControl=TSC
tsControl$niterations=10 #reduce the iterations to accelerate the analysis
#more iterations would be needed.
TSout3<-TrainSel(Candidates=list(c(lower[1], upper[1]),
                                 c(lower[2], upper[2]),
                                 c(lower[3], upper[3])),
                 setsizes = 2,
                 settypes = c("DBL", "DBL"),
                 Stat=fn,
                 nStat=3,
                 control=tsControl,
                 Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
scatterplot3d::scatterplot3d(t(TSout3$BestVal))
```



Hyperparameters

SetControlDefault and complexity setting. One important aspect for maximizing TrainSel performance is finding the optimal parameters to include in the "TrainSel_Control" object. To that end, SetControlDefault function is available. It contains 6 settings: 4 **size** levels and 2 **complexity** levels. It automatically generates a TrainSel_Control object with the optimal parameters for the desired size and complexity. A larger size results in more iterations performed, which increases the likelihood of reaching convergence. "low_complexity" has a higher convergence rate than "high_complexity" but it is more susceptible to the random start and has a higher chance of getting stuck in a local maximum.

In this example we will showcase the influence of the **complexity** setting using a toy evaluation metric with a known true best and a known local maximum. We have 200 values, 100 positive and 100 negative, and TrainSel has to select 50 of them that maximize the absolute value of their sum. That way, the best solutions are either selecting only positive or only negative values. In this case, selecting the 50 largest positive values is the true optimum while selecting only negative values is slightly worse and it can be considered a local optimum in which the optimization may get stuck.

```
candidates <- 1:200
size <- 100
values <- c(-((100:1)/100)^2, #first 100 values are negative
            ((3:102)/100)^2) #last 100 values are positive
DataComp <- list()
DataComp[["values"]] <- values

head(values)
```

```
## [1] -1.0000 -0.9801 -0.9604 -0.9409 -0.9216 -0.9025
```

```
tail(values)
```

```
## [1] 0.9409 0.9604 0.9801 1.0000 1.0201 1.0404
```

```
TestEval <- function(soln, DataComp) {
  values <- DataComp[["values"]]
  return(abs(sum(values[soln])))
}
```

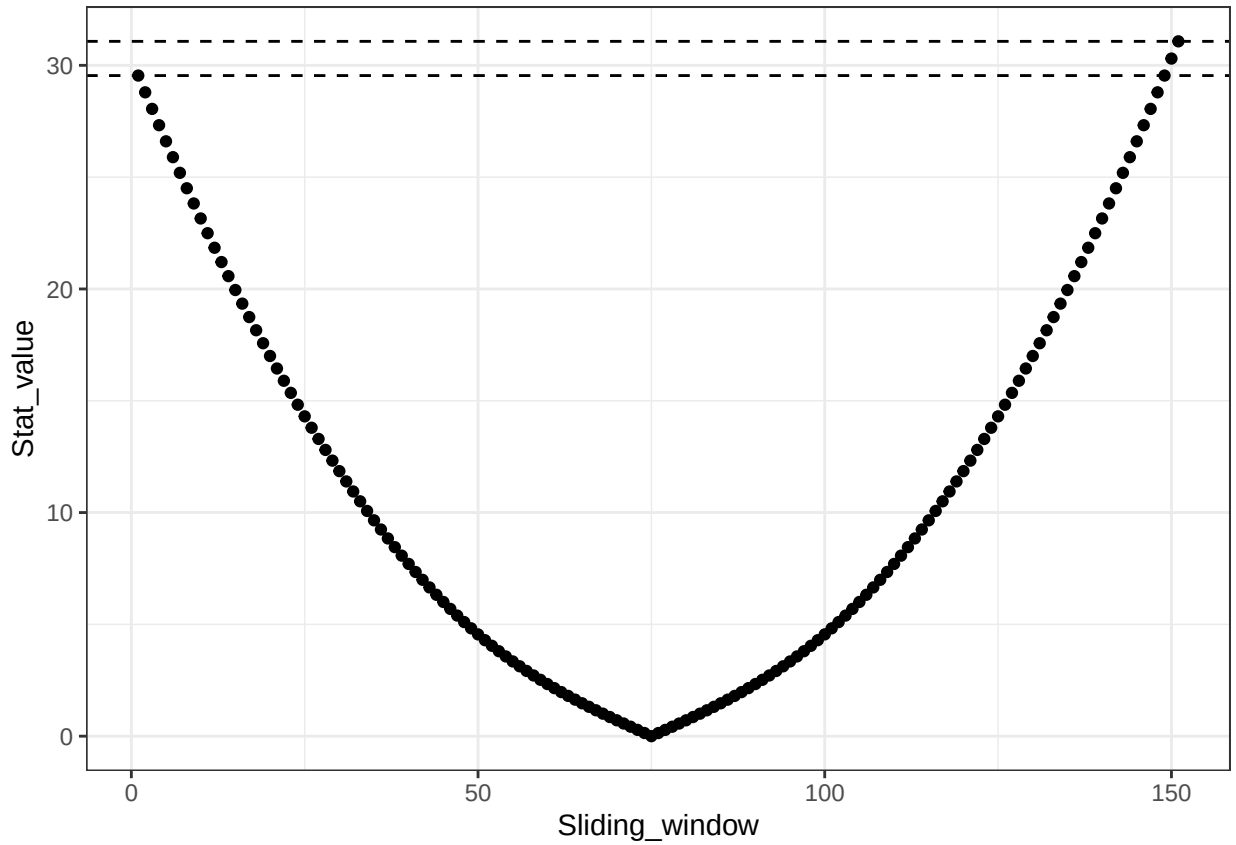
```
knownTrueBest <- abs(sum(values[151:200]))
LocalMaximum <- abs(sum(values[1:50]))
knownTrueBest
```

```
## [1] 31.0725
```

```
LocalMaximum
```

```
## [1] 29.5425
```

```
#Evolution of the evaluation metric against a sliding window that
#initially selects the most negative values and gradually moves
#towards the positive values. The two dashed lines correspond to the
#the global optimum and the local maximum
library(ggplot2)
selected <- list()
value <- c()
for (i in 1:151) {
  selected[[i]] <- 1:50+i-1
  value <- c(value, TestEval(selected[[i]], DataComp))
}
plotdf <- data.frame(Stat_value=value,
                     Sliding_window=1:length(value))

ggplot(plotdf) +
  theme_bw() +
  geom_point(aes(x=Sliding_window, y=Stat_value)) +
  geom_hline(yintercept = knownTrueBest, linetype = "dashed")+
  geom_hline(yintercept = LocalMaximum, linetype = "dashed")
```



In the figure above, it can be seen that the global optimum (stat value of 31.0725) is reached when the largest positive values are selected, while a local maximum (stat value of 29.5425) is obtained when the smallest negative values are selected. This local maximum is only slightly worse than the true optimum and it can act as a "trap" for an optimization algorithm, as it can be easily misidentified with the optimum. Here we will show how the "high_complexity" setting massively reduces the likelihood of TrainSel getting "stuck" in the local maximum:

First, we will perform optimization 10 times using the **"low_complexity"** setting. When working with "low_complexity", only the genetic algorithm is used.

```
#The code below is not run in this example as the hyperparameters used
#exceed demo limitations. Feel free to run it if you have acquired a license key.

#Generate TrainSel_Control object with the desired parameters
tsLowComp <- SetControlDefault(size = "small",
                               complexity = "low_complexity",
                               verbose = FALSE)
tsLowComp$progress <- FALSE #hide progress bar

results_low_comp <- c()
for (i in 1:10) {
  set.seed(i)
  TSOUTC<-TrainSel(Data=DataComp,
                   Candidates = list(1:200),
                   setsizes = c(50),
                   settypes = "UOS",
                   Stat = TestEval,
```

```
                  control=tsLowComp,
                  Verbose = FALSE,
                  Username = NULL, #you will need a valid license key to run this!
                  Password = NULL) #you will need a valid license key to run this!
  results_low_comp <- c(results_low_comp, TSOUTC$BestVal)
}
```

Next, we will repeat optimization with the **"high_complexity"** setting. In this case, optimization has 2 distinct phases:

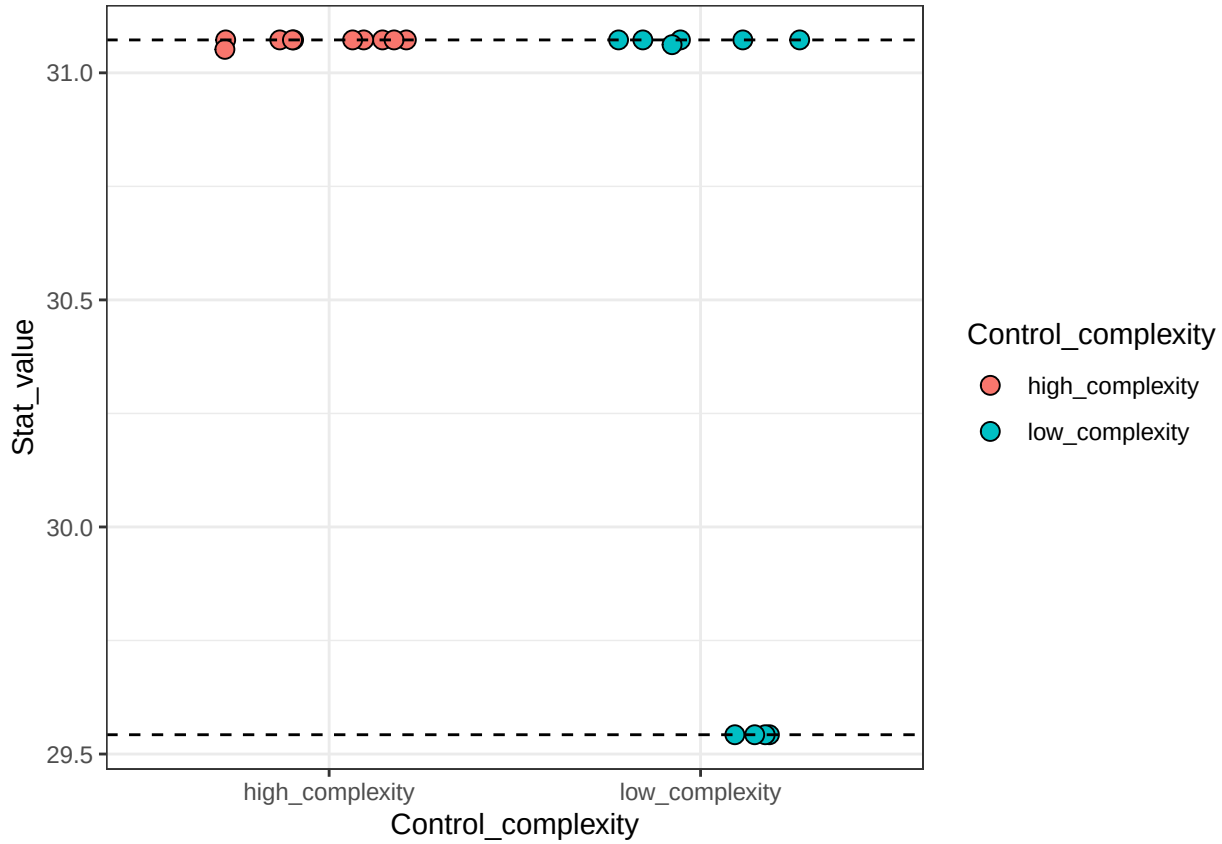
1) Islands optimization. Several small optimization runs with different random starts using only genetic algorithm.
2) Main step. Its initial solutions are the best ones found by the islands in the previous step. Here, genetic algorithm and simulated annealing are combined, with the latter increasing diversity of the selected solutions and improving exploration of the search space, increasing likelihood of finding global optimum.

```
#The code below is not run in this example as the "high_complexity" setting
#exceeds demo limitations. Feel free to run it if you have acquired a license key.

#Generate TrainSel_Control object with the desired parameters
tsHighComp <- SetControlDefault(size = "small",
                                complexity = "high_complexity",
                                verbose = FALSE)
tsHighComp$progress <- FALSE #hide progress bar

results_high_comp <- c()
for (i in 1:10) {
  set.seed(i)
  TSOUTC<-TrainSel(Data=DataComp,
                   Candidates = list(1:200),
                   setsizes = c(50),
                   settypes = "UOS",
                   Stat = TestEval,
                   control=tsHighComp,
                   Verbose = FALSE,
                   Username = NULL, #you will need a valid license key to run this!
                   Password = NULL) #you will need a valid license key to run this!
  results_high_comp <- c(results_high_comp, TSOUTC$BestVal)
}
```

Finally, we can compare the results of high and low complexity:

```
load("non_demo_results.RData") #as the previous analyses could not be run within
#the demo version, we load here the results (objects "results_low_comp" and
#"results_high_comp") we obtained using a valid license key

plotdf <- data.frame(Stat_value=c(results_low_comp,results_high_comp),
                     Control_complexity=c(rep("low_complexity", 10),
                                          rep("high_complexity", 10)))

ggplot(plotdf) +
  theme_bw() +
```

```
  geom_point(aes(x=Control_complexity, y=Stat_value, fill=Control_complexity),
             position = position_jitter(w = 0.3, h = 0),
             size = 3,
             shape = 21) +
  geom_hline(yintercept = knownTrueBest, linetype = "dashed")+
  geom_hline(yintercept = LocalMaximum, linetype = "dashed")
```



In the plot above, the two dashed lines correspond to the true best solution and the local optimum. The "high_complexity" setting always found the best solution, while "low_complexity" got stuck in the local optimum 40% of the optimization runs.

Estimation of computational time. The optimization process usually involves calculating the evaluation metric tens or hundreds of thousands of times. As a result, if the evaluation metric is relatively slow, time can become the limiting factor. Here, we provide an easy way to estimate the time needed for optimization, which can be an invaluable decision support tool:

```
#Toy example with a evaluation metric of known running time.
TimeEval <- function(soln, DataTime) {
  sleep <- DataTime[["sleep"]]
  Sys.sleep(sleep) #we use this to control the computational time of the function
  return(1)
}


TimeControl <- SetControlDefault(size = "demo",
```

```
                                  complexity = "low_complexity",
                                  verbose = FALSE)

#Reduce iterations and population size to accelerate computations
TimeControl$npop = 4
TimeControl$nelite = 2
TimeControl$niterations = 8

CandidatesTime = list(1:200)
setsizesTime = c(50)
settypesTime = "UOS"
```

Slow evaluation metric. In this situation, we have a relatively slow evaluation metric. This is the kind of scenario in which the time can become a limiting factor and in which it is important to have an accurate estimation of the computational time.

```
DataTime <- list(sleep = 1) #1s computational time




Time_resuls <- TimeEstimation(Stat = TimeEval,
               Data = DataTime,
               Candidates = CandidatesTime,
               setsizes = setsizesTime,
               settypes = settypesTime,
               control = TimeControl,
               n_average = 5,
               verbose = FALSE)

#To estimate time needed to run the evaluation metric, it was run "n_average" times.
#The results (in seconds) are stored here:
Time_resuls$time_Stat_vector
```

```
## [1] 1.025504 1.023344 1.024504 1.024273 1.028002
```

```
#Number of times the evaluation metric has to be run during optimization
Time_resuls$eval_total
```

```
## [1] 20
```

```
#Estimated total time (in seconds) needed to perform optimization
Time_resuls$time_total_seconds
```

```
## [1] 20.50251
```

```
#Let's check if the estimation and the reality match:
startTime <- Sys.time()
TSOUTT<-TrainSel(Data=DataTime,
                 Candidates = CandidatesTime,
```

```
                  setsizes = setsizesTime,
                  settypes = settypesTime,
                  Stat = TimeEval,
                  control=TimeControl,
                  Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
endTime <- Sys.time()
#Very close to the estimation
endTime - startTime
```

```
## Time difference of 21.03196 secs
```

The time estimation was very accurate and can be used to decide the maximum number of iterations that can be performed with the computational resources available.

Fast evaluation metric. In this situation, computational time should not be an issue, making time estimation less relevant.

```
DataTime <- list(sleep = 0)

Time_resuls <- TimeEstimation(Stat = TimeEval,
                              Data = DataTime,
                              Candidates = CandidatesTime,
                              setsizes = setsizesTime,
                              settypes = settypesTime,
                              control = TimeControl,
                              n_average = 10,
                              verbose = FALSE)
Time_resuls$time_total_seconds
```

```
## [1] 4.768372e-05
```

```
startTime <- Sys.time()
TSOUTT<-TrainSel(Data=DataTime,
                  Candidates = CandidatesTime,
                  setsizes = setsizesTime,
                  settypes = settypesTime,
                  Stat = TimeEval,
                  control=TimeControl,
                  Verbose = FALSE)
```

```
## Maximum number of iterations reached.
```

```
endTime <- Sys.time()
#Not close to the time estimation
endTime - startTime
```

```
## Time difference of 0.519423 secs
```

In this case the estimation was several orders of magnitudes off. The reason for it is that, when the evaluation metric is relatively slow as in the first example, the total time needed for optimization is dominated by the time needed to run the evaluation metric, while internal TrainSel processes can be disregarded. On the contrary, when the evaluation metric is close to instant, computational time is dominated by internal TrainSel processes. `TimeEstimation` function only considers the time needed to run the evaluation metric, which makes it inaccurate for extremely fast evaluation metrics. However, in this case optimization would be fast and computational time would not be a limiting factor, reducing the relevance of the inaccuracy of time estimation. The computational bottleneck in optimization is the time required by the evaluation metric, which is very well estimated by the `TimeEstimation` function

Hyperparameter Tuning Pipeline.

When optimizing the hyperparameters, it is important to consider the existing trade-off between **size** and **complexity** in SetControlDefault. "low_complexity" results in a high convergence rate (i.e. faster optimization), but it increases the likelihood of finding a local maximum instead of the global one and makes optimization more susceptible to the random start (i.e. worse results). "high_complexity" solves these problems at the expense of reducing convergence rate, which is problematic when computational time is a limiting factor and convergence is not reached. As a general rule, the priority should be reaching convergence (or getting very close). As a result, increasing complexity will only result in gains if convergence can be reliably reached.

If previous experience with a given optimization problem is available, it is possible to know which parameters should be used. However, this is often not the case. In this situation, we suggest using a combination of `TimeEstimation` and `SetControlDefault` functions to find the optimal parameters:

1. Use `TimeEstimation` to find if computational time is the limiting factor for a given set of parameters:

- If computational time is not a problem, set **size** to "large" and **complexity** to "high_complexity" in SetControlDefault.
- If computational time is the limiting factor, set **complexity** to "low_complexity" and **size** as large as possible.

2. The results from the optimization with the parameters described above can inform which parameters to use if a similar optimization problem has to be solved in the future:

- If convergence was not reached and computational time allows for it, **size** should be increased. If **size** was already set to "large", we advise manually increasing "niterations" in the "TrainSel_Control" object generated by `SetControlDefault` function.
- If convergence was reached late (after 70% of "niterations"), keep current parameters. You can check the iteration in which optimization finished by looking at the length of the "maxvec" vector in the TrainSel output.
- If convergence was reached at an intermediate point (around 30-70% of "niterations" needed), keep current **size** and increase **complexity** to "high_complexity".
- If convergence was reached early (before 30% of "niterations"), decrease **size** and increase **complexity** to "high_complexity".

A decision tree summarizing this steps is available in Figure 1 and we include a small example on how to do this process:

```
#Tune parameters example

DataTime <- list(sleep = 1) #1s computational time for evaluation metric
```

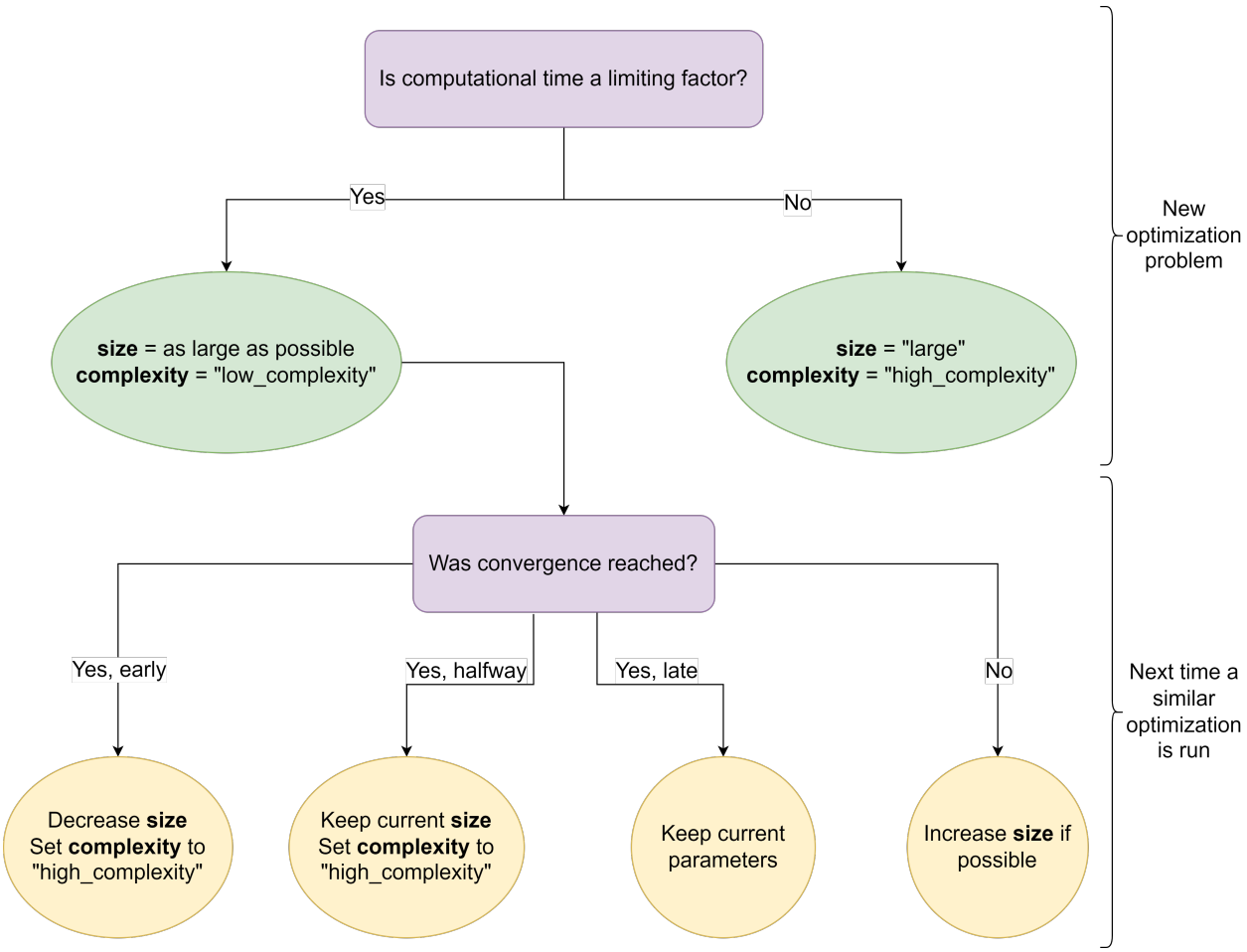


Figure 1: Decision tree for hyperparameter tuning

```r
#1.1) check if optimization is doable with desired parameters or if it takes too long
exampleCNTRL <- SetControlDefault(size = "large",
                                  complexity = "high_complexity",
                                  verbose = FALSE)




Time_resuls <- TimeEstimation(Stat = TimeEval,
               Data = DataTime,
               Candidates = CandidatesTime,
               setsizes = setsizesTime,
               settypes = settypesTime,
               control = exampleCNTRL,
               n_average = 5,
               verbose = FALSE)

#Estimated total time (in hours) needed to perform optimization
Time_resuls$time_total_seconds/3600
```

```
## [1] 189.4357
```

```r
#This is too slow (around 7 days!). You can accelerate computations by
#tuning the hyperparameters


#1.2) Reduce parameters to match desired time limit
exampleCNTRL <- SetControlDefault(size = "demo",
                                  complexity = "low_complexity",
                                  verbose = FALSE)

Time_resuls <- TimeEstimation(Stat = TimeEval,
               Data = DataTime,
               Candidates = CandidatesTime,
               setsizes = setsizesTime,
               settypes = settypesTime,
               control = exampleCNTRL,
               n_average = 5,
               verbose = FALSE)

#Reducing the parameters allows to reduce time to around 2-3 hours, which is doable
Time_resuls$time_total_seconds/3600
```

```
## [1] 2.729503
```

```r
#Several hours is still to slow for this example, so we will use a toy example
#with very low number of iterations
#Reduce iterations and population size to accelerate computations
exampleCNTRL$npop = 4
exampleCNTRL$nelite = 2
exampleCNTRL$niterations = 8
```

```r
exampleCNTRL$minitbefstop = 1 #stop after 1 iteration without improving fitness
#We don't recommend modifying "minitbefstop" in normal TrainSel operations, we are
#doing it here only for the sake of the example


Time_resuls <- TimeEstimation(Stat = TimeEval,
               Data = DataTime,
               Candidates = CandidatesTime,
               setsizes = setsizesTime,
               settypes = settypesTime,
               control = exampleCNTRL,
               n_average = 5,
               verbose = FALSE)

#In this toy example, only around 20 seconds are needed.
#Let's try it
Time_resuls$time_total_seconds
```

```
## [1] 20.41564
```

```r
#2) Run optimization with the selected parameters

#Let's check if the estimation and the reality match:
startTime <- Sys.time()
TSOUTT<-TrainSel(Data=DataTime,
                 Candidates = CandidatesTime,
                 setsizes = setsizesTime,
                 settypes = settypesTime,
                 Stat = TimeEval,
                 control=exampleCNTRL,
                 Verbose = FALSE)
```

```
## Convergence Achieved
##  (no improv in the last 'minitbefstop' iters).
```

```r
endTime <- Sys.time()
#Lower time than estimation because convergence was reached early
endTime - startTime
```

```
## Time difference of 10.80631 secs
```

```r
#2.1) Check if convergence was reached

#Convergence was achieved in iteration number 3 out of the 8 iterations we
#set as limit. Therefore, we can conclude that the parameters used were enough
length(TSOUTT$maxvec) #number of iterations run before convergence
```

```
## [1] 3
```

```
exampleCNTRL$niterations #iteration number limit
```

```
## [1] 8
```

```
#2.2) If the optimization had been far from convergence, higher iterations would
#have been required
```

License key

This document only has examples that can be run with the demo version of TrainSel, but if the desired training set size is larger than 100 integer or 3 double solutions, upgrading to the full version is required:

```
#If this is run, it will generate an error as the Demo limitations are exceeded
#(setsizes > 100)
TSOUTCD<-TrainSel(Data=dataCDMEANopt,
                  Candidates = list(1:160),
                  setsizes = c(120),
                  settypes = "UOS",
                  Stat = CDMEANOPT,
                  control=TSC,
                  Verbose = FALSE)

#To obtain a license key, you can contact us at j.isidro@upm.es
#We currently distribute free license keys for public bodies, such us,
#Universities, and non-profit organizations.
#Payment is required for private purposes or profit organizations


#If you have received a license key, you can activate the full version by including
#your username and password in the TrainSel function:


#If this is run, it will generate an error as the username and password in
#this example are not valid
username <- 0
password <- 0
TSOUTCD<-TrainSel(Data=dataCDMEANopt,
                  Candidates = list(1:160),
                  setsizes = c(120),
                  settypes = "UOS",
                  Stat = CDMEANOPT,
                  control=TSC,
                  Username = username, #Include username here
                  Password = password, #Include password here
                  Verbose = FALSE)
```