Package 'GROAN'

July 12, 2017

Type Package

Title Genomic Regression Workbench

Version 1.0.0

Date 2017-07-12

Author Nelson Nazzicari & Filippo Biscarini

Maintainer Nelson Nazzicari <nelson.nazzicari@crea.gov.it>

Description Workbench for testing genomic regression accuracy on (optionally noisy) phenotypes.

License GPL-3 | file LICENSE

LazyData TRUE

RoxygenNote 6.0.1

Depends R (>= 2.10)

Imports plyr, rmarkdown, rrBLUP

Suggests BGLR, e1071, ggplot2, knitr, randomForest

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2017-07-12 19:30:16 UTC

R topics documented:

`addRegressor` *Add an extra regressor to a Workbench*

Description

This function adds a regressor to an existing GROAN.Workbench object.

Usage

```
addRegressor(wb, regressor, regressor.name = regressor, ...)
```

Arguments

`wb`	the GROAN.Workbench instance to be updated
`regressor`	regressor function
`regressor.name`	string that will be used in reports. Keep in mind that when deciding names.
`...`	extra parameters are passed to the regressor function

Value

an updated instance of the original GROAN.Workbench

See Also

createWorkbench GROAN.run

Examples

```
#creating a Workbench with all default arguments
wb = createWorkbench()
#adding a second regressor
wb = addRegressor(wb, regressor = phenoRegressor.dummy, regressor.name = 'dummy')

## Not run:
#trying to add again a regressor with the same name would result in a naming conflict error
wb = addRegressor(wb, regressor = phenoRegressor.dummy, regressor.name = 'dummy')
## End(Not run)
```

`createNoisyDataset` *Noisy Data Set Constructor*

Description

This function creates a GROAN.NoisyDataset object (or fails trying). The class will contain all noisy data set components: genotypes and/or covariance matrix, phenotypes, strata (optional), a noise injector function and its parameters.
You can have a general description of the created object using the overridden print.GROAN.NoisyDataset function.

Usage

```
createNoisyDataset(name, genotypes = NULL, covariance = NULL, phenotypes,
  strata = NULL, extraCovariates = NULL, ploidy = 2,
  noiseInjector = noiseInjector.dummy, ...)
```

Arguments

`name`	A string defining the dataset name, used later do identify this particular instance in reports and result files. It is advisable for it to be it somewhat meaningful (to you, GROAN simply reports it as it is)
`genotypes`	Matrix or dataframe containing SNP genotypes, one row per sample (N), one column per marker (M), 0/1/2 format (for diploids) or 0/1/2.../ploidy in case of polyploids
`covariance`	square (NxN) matrix of covariances between samples
`phenotypes`	numeric array, M slots
`strata`	array of M slots, describing the strata each data point belongs to. This is used for stratified crossvalidation (see `createWorkbench`)
`extraCovariates`	dataframe of optional extra covariates (N lines, one column per extra covariate). Numeric ones will be normalized, string and categorical ones will be transformed in stub TRUE/FALSE variables (one per possible value, see model.matrix).
`ploidy`	number of haploid sets in the cell. Defaults to 2 (diploid).
`noiseInjector`	name of a noise injector function, defaults to noiseInjector.dummy
`...`	further arguments are passed along to noiseInjector

Value

a GROAN.NoisyDataset object.

See Also

GROAN.run createNoisyDataset

Examples

```
#For more complete examples see the package vignette
#creating a noisy dataset with normal noise
nds = createNoisyDataset(
  name = 'PEA, normal noise',
  genotypes = GROAN.pea.SNPs,
  phenotypes = GROAN.pea.yield,
  noiseInjector = noiseInjector.norm,
  mean = 0,
  sd = sd(GROAN.pea.yield) * 0.5
)
```

---

createRunId *Generate a random run id*

Description

This function returns a partially random alphanumeric string that can be used to identify a single run.

Usage

```
createRunId()
```

Value

a partially random alphanumeric string

---

createWorkbench *Workbench constructor*

Description

This function creates a GROAN.Workbench instance (or fails trying). The created object contains:
a) one regressor with its own specific configuration
b) the crossvalidation parameters, describing the experiment.
You can have a general description of the created object using the overridden print.GROAN.Workbench function.
It is possible to add other regressors to the created `GROAN.Workbench` object using addRegressor.
Once the `GROAN.Workbench` is created it must be passed to GROAN.run to start the experiment.

Usage

```
createWorkbench(folds = 10, reps = 5, stratified = FALSE,
  outfolder = NULL, saveHyperParms = FALSE, saveExtraData = FALSE,
  regressor = phenoRegressor.rrBLUP, regressor.name = "default regressor",
  ...)
```

Arguments

folds	defaults to 10, as in 10-folds crossvalidation
reps	number of times the whole test must be repeated, defaults to 5
stratified	boolean indicating whether the crossvalidation should be standard (the default) or stratified (keeping the same proportion of data strata in each fold). If no strata are present in the GROAN.NoisyDataSet object and stratified==TRUE all samples will be considered belonging to the same strata ("dummyStrata")
outfolder	folder where to save the data. If NULL (the default) nothing will be saved. Filenames are standardized. If existing, accuracy and hyperparameter files will be updated, otherwise are created. ExtraData cannot be updated, so unique filenames will be generated using runId (see GROAN.run)
saveHyperParms	boolean indicating if the hyperparameters from regressor training should be saved in outfolder. Defaults to FALSE.
saveExtraData	boolean indicating if extradata from regressor training should be saved in outfolder as R objects (using the save function). Defaults to FALSE.
regressor	regressor function. Defaults to phenoRegressor.rrBLUP
regressor.name	string that will be used in reports. Keep that in mind when deciding names. Defaults to "default regressor"
...	extra parameter are passed to regressor function

Value

An instance of GROAN.Workbench

See Also

addRegressor GROAN.run createNoisyDataset

Examples

```
#creating a Workbench with all default arguments
wb1 = createWorkbench()
#another Workbench, with different crossvalidation
wb2 = createWorkbench(folds=5, reps=20)
#a third one, with a different regressor and extra parameters passed to regressor function
wb3 = createWorkbench(regressor=phenoRegressor.BGLR, regressor.name='Bayesian Lasso', type='BL')
```

getNoisyPhenotype *Generate an instance of noisy phenotypes*

Description

Given a `Noisy Dataset` object, this function applies the noise injector to the data and returns a noisy version of it. It is useful for inspecting the noisy injector effects.

Usage

```
getNoisyPhenotype(nds)
```

Arguments

`nds`	a `Noisy Dataset` object

Value

the phenotypes contained in `nds` with added noise.

GROAN.pea.kinship *Data - kinship among pea lines*

Description

A square dataframe containing the realized kinships between pairs of pea lines. Values were computed following the Astle & Balding metric. Higher values represent a higher degree of genetic similarity between lines. This metric mainly accounts for additive genetic contributions (as an alternative to dominant contributions).

Usage

```
GROAN.pea.kinship
```

Format

A data frame with 103 rows and 103 variables. Row and column names are pea lines.

Source

Annicchiarico et al., *GBS-Based Genomic Selection for Pea Grain Yield under Severe Terminal Drought*, The Plant Genome, Volume 10. doi: 10.3835/plantgenome2016.07.0072

GROAN.pea.SNPs *Data - pea SNP genotypes*

Description

SNP genotypes for 103 pea lines. Each line of this data frame represent a single pea line. Each column a SNP marker. Values can either be 0, 1, or 2, representing the three possible genotypes (AA, Aa, and aa, respectively).

Usage

```
GROAN.pea.SNPs
```

Format

A data frame with 103 rows and 647 variables. Each row represent a pea line, each column a SNP marker

Source

Annicchiarico et al., *GBS-Based Genomic Selection for Pea Grain Yield under Severe Terminal Drought*, The Plant Genome, Volume 10. doi: 10.3835/plantgenome2016.07.0072

GROAN.pea.yield *Data - pea phenotypes (yield)*

Description

Phenotype dataset, containing data on pea grain yield [t/ha] for each pea line.

Usage

```
GROAN.pea.yield
```

Format

A named array with 103 slots.

Source

Annicchiarico et al., *GBS-Based Genomic Selection for Pea Grain Yield under Severe Terminal Drought*, The Plant Genome, Volume 10. doi: 10.3835/plantgenome2016.07.0072

GROAN.run *Compare Genomic Regressors on a Noisy Dataset*

Description

This function runs the experiment described in a GROAN.Workbench object on the data contained in a GROAN.NoisyDataSet object. It returns a GROAN.Result object, which have a summary function for quick inspection and can be fed to plotResult for visual comparisons. The experiment statistics are computed via measurePredictionPerformance.
Each time this function is invoked it will refer to a runId - an alphanumeric string identifying each specific run. The runId is usually generated internally, but it is possible to pass it if the intention is to join results from different runs for analysis purposes.

Usage

```
GROAN.run(nds, wb, run.id = createRunId())
```

Arguments

nds	a GROAN.NoisyDataSet object, containing the data (genotypes, phenotypes and so forth) plus a noiseInjector function
wb	a GROAN.Workbench object, containing the regressors to be tested together with the description of the experiment
run.id	an alphanumeric string identifying this specific run. If not passed it is generated using createRunId

Value

a GROAN.Result object

See Also

measurePredictionPerformance

Examples

```
## Not run:
#Complete examples are found in the vignette
vignette('GROAN.vignette', package='GROAN')

#Minimal example
#1) creating a noisy dataset with normal noise
nds = createNoisyDataset(
  name = 'PEA, normal noise',
  genotypes = GROAN.pea.SNPs,
  phenotypes = GROAN.pea.yield,
  noiseInjector = noiseInjector.norm,
  mean = 0,
```

```
  sd = sd(GROAN.pea.yield) * 0.5
)

#2) creating a GROAN.WorkBench using default regressor and crossvalidation preset
wb = createWorkbench()

#3) running the experiment
res = GROAN.run(nds, wb)

#4) examining results
summary(res)
plotResult(res)

## End(Not run)
```

`measurePredictionPerformance`
Measure Performance of a Prediction

Description

This method returns several performance metrics for the passed predictions.

Usage

```
measurePredictionPerformance(truevals, predvals)
```

Arguments

`truevals`	true values
`predvals`	predicted values

Value

A named array with the following fields:

pearson Pearson's correlation

spearman Spearmans' correlation (order based)

rmse Root Mean Square Error

mae Mean Absolute Error

coeff_det Coefficient of determination

noiseInjector.dummy *Noise Injector dummy function*

Description

This noise injector does not add any noise. Passed `phenotypes` are simply returned. This function is useful when comparing different regressors on the same dataset without the effect of extra injected noise.

Usage

```
noiseInjector.dummy(phenotypes)
```

Arguments

`phenotypes`	input phenotypes. This object will be returned without checks.

Value

the same passed `phenotypes`

See Also

Other noiseInjectors: `noiseInjector.norm`, `noiseInjector.swapper`, `noiseInjector.unif`

Examples

```
phenos = rnorm(10)
all(phenos == noiseInjector.dummy(phenos)) #TRUE
```

noiseInjector.norm *Inject Normal Noise*

Description

This function adds to the passed `phenotypes` array noise sampled from a normal distribution with the specified mean and standard deviation.
The function can interest the totality of the passed phenotype array or a random subset of it (commanded by `subset` parameter).

Usage

```
noiseInjector.norm(phenotypes, mean = 0, sd = 1, subset = 1)
```

Arguments

phenotypes	an array of numbers.
mean	mean of the normal distribution.
sd	standard deviation of the normal distribution.
subset	integer in [0,1], the proportion of original dataset to be injected

Value

An array, of the same size as phenotypes, where normal noise has been added to the original phenotype values.

See Also

Other noiseInjectors: noiseInjector.dummy, noiseInjector.swapper, noiseInjector.unif

Examples

```
#a sinusoid signal
phenos = sin(seq(0,5, 0.1))
plot(phenos, type='p', pch=16, main='Original (black) vs. Injected (red), 100% affected')

#adding normal noise to all samples
phenos.noise = noiseInjector.norm(phenos, sd = 0.2)
points(phenos.noise, type='p', col='red')

#adding noise only to 30% of the samples
plot(phenos, type='p', pch=16, main='Original (black) vs. Injected (red), 30% affected')
phenos.noise.subset = noiseInjector.norm(phenos, sd = 0.2, subset = 0.3)
points(phenos.noise.subset, type='p', col='red')
```

---

noiseInjector.swapper *Swap phenotypes between samples*

Description

This function introduces swap noise, i.e. a number of couples of samples will have their phenotypes swapped.
The number of couples is computed so that the total fraction of interested phenotypes approximates subset.

Usage

```
noiseInjector.swapper(phenotypes, subset = 0.1)
```

Arguments

phenotypes	an array of numbers
subset	fraction of phenotypes to be interested by noise.

Value

the same passed phenotypes, but with some elements swapped

See Also

Other noiseInjectors: noiseInjector.dummy, noiseInjector.norm, noiseInjector.unif

Examples

```
#a set of phenotypes
phenos = 1:10
#swapping two elements
phenos.sw2 = noiseInjector.swapper(phenos, 0.2)
#swapping four elements
phenos.sw4 = noiseInjector.swapper(phenos, 0.4)
#swapping four elements again, since 30% of 10 elements
#is rounded to 4 (two couples)
phenos.sw4.again = noiseInjector.swapper(phenos, 0.3)
```

---

noiseInjector.unif *Inject Uniform Noise*

Description

This function adds to the passed phenotypes array noise sampled from a uniform distribution with the specified range.
The function can interest the totality of the passed phenotype array or a random subset of it (commanded by subset parameter).

Usage

```
noiseInjector.unif(phenotypes, min = 0, max = 1, subset = 1)
```

Arguments

phenotypes	an array of numbers.
min, max	lower and upper limits of the distribution. Must be finite.
subset	integer in [0,1], the proportion of original dataset to be injected

Value

An array, of the same size as phenotypes, where uniform noise has been added to the original phenotype values.

See Also

Other noiseInjectors: noiseInjector.dummy, noiseInjector.norm, noiseInjector.swapper

Examples

```
#a sinusoid signal
phenos = sin(seq(0,5, 0.1))
plot(phenos, type='p', pch = 16, main='Original (black) vs. Injected (red), 100% affected')

#adding normal noise to all samples
phenos.noise = noiseInjector.unif(phenos, min=0.1, max=0.3)
points(phenos.noise, type='p', col='red')

#adding noise only to 30% of the samples
plot(phenos, type='p', pch = 16, main='Original (black) vs. Injected (red), 30% affected')
phenos.noise.subset = noiseInjector.unif(phenos, min=0.1, max=0.3, subset = 0.3)
points(phenos.noise.subset, type='p', col='red')
```

---

phenoRegressor.BGLR *Regression using BGLR package*

Description

This is a wrapper around BGLR. As such, it won't work if BGLR package is not installed.
Genotypes are modeled using the specified `type`. If `type` is 'RKHS' (and only in this case) the covariance/kinship matrix `covariances` is required, and it will be modeled as matrix K in BGLR terms. In all other cases genotypes and covariances are put in the model as X matrices.
Extra covariates, if present, are modeled as FIXED effects.

Usage

```
phenoRegressor.BGLR(phenotypes, genotypes, covariances, extraCovariates,
  type = c("FIXED", "BRR", "BL", "BayesA", "BayesB", "BayesC", "RKHS"), ...)
```

Arguments

`phenotypes`	phenotypes, a numeric array (n x 1), missing values are predicted
`genotypes`	SNP genotypes, one row per phenotype (n), one column per marker (m), values in 0/1/2 for diploids or 0/1/2/...ploidy for polyploids. Can be NULL if `covariances` is present.
`covariances`	square matrix (n x n) of covariances. Can be NULL if `genotypes` is present.
`extraCovariates`	extra covariates set, one row per phenotype (n), one column per covariate (w). If NULL no extra covariates are considered.
`type`	character literal, one of the following: FIXED (Flat prior), BRR (Gaussian prior), BL (Double-Exponential prior), BayesA (scaled-t prior), BayesB (two component mixture prior with a point of mass at zero and a scaled-t slab), BayesC (two component mixture prior with a point of mass at zero and a Gaussian slab)
`...`	extra parameters are passed to BGLR

Value

The function returns a list with the following fields:

- `predictions` : an array of (n) predicted phenotypes, with NAs filled and all other positions repredicted (useful for calculating residuals)
- `hyperparams` : empty, returned for compatibility
- `extradata` : list with information on trained model, coming from BGLR

See Also

BGLR

Other phenoRegressors: `phenoRegressor.RFR`, `phenoRegressor.SVR`, `phenoRegressor.dummy`, `phenoRegressor.rrBLUP`

Examples

```
## Not run:
#using the GROAN.pea dataset, we regress on the dataset and predict the first ten phenotypes
phenos = GROAN.pea.yield
phenos[1:10]  = NA

#calling the regressor with Bayesian Lasso
results = phenoRegressor.BGLR(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  type = 'BL', nIter = 2000 #BGLR-specific parameters
)

#examining the predictions
plot(GROAN.pea.yield, results$predictions,
     main = 'Train set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results$predictions[1:10], pch=16, col='red')

#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

## End(Not run)
```

phenoRegressor.dummy *Regression dummy function*

Description

This function is for development purposes. It returns, as "predictions", an array of random numbers. It accept the standard inputs and produces a formally correct output. It is, obviously, quite fast.

Usage

```
phenoRegressor.dummy(phenotypes, genotypes, covariances, extraCovariates)
```

Arguments

phenotypes	phenotypes, numeric array (n x 1), missing values are predicted
genotypes	SNP genotypes, one row per phenotype (n), one column per marker (m), values in 0/1/2 for diploids or 0/1/2/...ploidy for polyploids. Can be NULL if covariances is present.
covariances	square matrix (n x n) of covariances. Can be NULL if genotypes is present.
extraCovariates	extra covariates set, one row per phenotype (n), one column per covariate (w). If NULL no extra covariates are considered.

Value

The function should return a list with the following fields:

- predictions : an array of (k) predicted phenotypes
- hyperparams : named array of hyperparameters selected during training
- extradata : any extra information

See Also

Other phenoRegressors: phenoRegressor.BGLR, phenoRegressor.RFR, phenoRegressor.SVR, phenoRegressor.rrBLUP

Examples

```
#genotypes are not really investigated. Only
#number of test phenotypes is used.
phenoRegressor.dummy(
  phenotypes = c(1:10, NA, NA, NA),
  genotypes = matrix(nrow = 13, ncol=30)
)
```

phenoRegressor.RFR *Random Forest Regression using package randomForest*

Description

This is a wrapper around randomForest and related functions. As such, this function will not work if randomForest package is not installed. There is no distinction between regular covariates (genotypes) and extra covariates (fixed effects) in random forest. If extra covariates are passed, they are put together with genotypes, side by side. Same thing happens with covariances matrix. This can bring to the scientifically questionable but technically correct situation of regressing on a big matrix made of SNP genotypes, covariances and other covariates, all collated side by side. The function makes no distinction, and it's up to the user understand what is correct in each specific experiment.

WARNING: this function can be *very* slow, especially when called on thousands of SNPs.

Usage

```
phenoRegressor.RFR(phenotypes, genotypes, covariances, extraCovariates,
  ntree = ceiling(length(phenotypes)/5), ...)
```

Arguments

`phenotypes`	phenotypes, a numeric array (n x 1), missing values are predicted
`genotypes`	SNP genotypes, one row per phenotype (n), one column per marker (m), values in 0/1/2 for diploids or 0/1/2/...ploidy for polyploids. Can be NULL if `covariances` is present.
`covariances`	square matrix (n x n) of covariances. Can be NULL if `genotypes` is present.
`extraCovariates`	extra covariates set, one row per phenotype (n), one column per covariate (w). If NULL no extra covariates are considered.
`ntree`	number of trees to grow, defaults to a fifth of the number of samples (rounded up). As per `randomForest` documentation, it should not be set to too small a number, to ensure that every input row gets predicted at least a few times
`...`	any extra parameter is passed to `randomForest::randomForest()`

Value

The function returns a list with the following fields:

- `predictions` : an array of (k) predicted phenotypes
- `hyperparams` : named vector with the following keys: ntree (number of grown trees) and mtry (number of variables randomly sampled as candidates at each split)
- `extradata` : the object returned by `randomForest::randomForest()`, containing the full trained forest and the used parameters

See Also

randomForest

Other phenoRegressors: phenoRegressor.BGLR, phenoRegressor.SVR, phenoRegressor.dummy, phenoRegressor.rrBLUP

Examples

```
## Not run:
#using the GROAN.pea dataset, we regress on the dataset and predict the first ten phenotypes
phenos = GROAN.pea.yield
phenos[1:10]  = NA

#calling the regressor with random forest
results = phenoRegressor.RFR(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  ntree = 20,
  mtry = 200 #randomForest-specific parameters
)

#examining the predictions
plot(GROAN.pea.yield, results$predictions,
     main = 'Train set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results$predictions[1:10], pch=16, col='red')

#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

## End(Not run)
```

---

phenoRegressor.rrBLUP *SNP-blup using rrBLUP package*

Description

This is a wrapper around rrBLUP function mixed.solve. Genotypes are modeled as random effects (matrix Z) and extra covariates, if present, as fixed effects (matrix X). Argument covariances is ignored.
Please note that this function won't work if SNPs are not passed and if rrBLUP package is not installed.

Usage

```
phenoRegressor.rrBLUP(phenotypes, genotypes, covariances = NULL,
  extraCovariates = NULL, ...)
```

Arguments

phenotypes	phenotypes, a numeric array (n x 1), missing values are predicted
genotypes	SNP genotypes, one row per phenotype (n), one column per marker (m), values in 0/1/2 for diploids or 0/1/2/...ploidy for polyploids. Can be NULL if covariances is present.
covariances	square matrix (n x n) of covariances, ignored (included for coherence with other regressors).
extraCovariates	optional extra covariates set, one row per phenotype (n), one column per covariate (w). If NULL no extra covariates are considered.
...	extra parameters are passed to rrBLUP::mixed.solve

Value

The function returns a list with the following fields:

- predictions : an array of (k) predicted phenotypes
- hyperparams : named vector with the following keys: Vu, Ve, beta, LL
- extradata : list with information on trained model, coming from mixed.solve

See Also

mixed.solve

Other phenoRegressors: phenoRegressor.BGLR, phenoRegressor.RFR, phenoRegressor.SVR, phenoRegressor.dummy

Examples

```
## Not run:
#using the GROAN.pea dataset, we regress on the dataset and predict the first ten phenotypes
phenos = GROAN.pea.yield
phenos[1:10]  = NA

#calling the regressor with ridge regression BLUP on SNPs
results = phenoRegressor.rrBLUP(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  SE = TRUE, return.Hinv = TRUE #rrBLUP-specific parameters
)

#examining the predictions
```

```
plot(GROAN.pea.yield, results$predictions,
     main = 'Train set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results$predictions[1:10], pch=16, col='red')

#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

## End(Not run)
```

`phenoRegressor.SVR` *Suppor Vector Regression using package e1071*

Description

This is a wrapper around several functions from `e1071` package (as such, it won't work if e1071 package is not installed). This function implements Support Vector Regressions, meaning that the data points are projected in a transformed higher dimensional space where linear regression is possible.

`phenoRegressor.SVR` can operate in three modes: run, train and tune.
In **run** mode you need to pass the function an already tuned/trained SVR model, typically obtained either directly from e1071 functions (e.g. from svm, best.svm and so forth) or from a previous run of `phenoRegressor.SVR` in a different mode. The passed model is applied to the passed dataset and predictions are returned.
In **train** mode a SVR model will be trained on the passed dataset using the passed hyper parameters. The trained model will then be used for predictions.
In **tune** mode you need to pass one or more sets of hyperparameters. The best combination of hyperparameters will be selected through crossvalidation. The best performing SVR model will be used for final predictions. This mode can be very slow.

There is no distinction between regular covariates (genotypes) and extra covariates (fixed effects) in Support Vector Regression. If extra covariates are passed, they are put together with genotypes, side by side. Same thing happens with covariances matrix. This can bring to the scientifically questionable but technically correct situation of regressing on a big matrix made of SNP genotypes, covariances and other covariates, all collated side by side. The function makes no distinction, and it's up to the user understand what is correct in each specific experiment.

Usage

```
phenoRegressor.SVR(phenotypes, genotypes, covariances, extraCovariates,
  mode = c("tune", "train", "run"), tuned.model = NULL,
  scale.pheno = TRUE, scale.geno = FALSE, ...)
```

Arguments

phenotypes	phenotypes, a numeric array (n x 1), missing values are predicted
genotypes	SNP genotypes, one row per phenotype (n), one column per marker (m), values in 0/1/2 for diploids or 0/1/2/...ploidy for polyploids. Can be NULL if covariances is present.
covariances	square matrix (n x n) of covariances. Can be NULL if genotypes is present.
extraCovariates	extra covariates set, one row per phenotype (n), one column per covariate (w). If NULL no extra covariates are considered.
mode	this parameter decides what will happen with the passed dataset • mode = "tune" : hyperparameters will be tuned on a grid (you may want to specify its values using extra params) with a call to e1071::tune.svm. Use this option if you have no idea about the optimal choice of hyperparameters. This mode can be very slow. • mode = "train" : an SVR will be trained on the train dataset using the passed hyperparameters (if you know them). This more invokes e1071::train • mode = "run" : you already have a tuned and trained SVR (put it into tuned.model) and want to use it. The fastest mode.
tuned.model	a tuned and trained SVR to be used for prediction. This object is only used if mode is equal to "run".
scale.pheno	if TRUE (default) the phenotypes will be scaled and centered (before tuning or before applying the passed tuned model).
scale.geno	if TRUE the genotypes will be scaled and centered (before tuning or before applying the passed tuned model. It is usually not a good idea, since it leads to worse results. Defaults to FALSE.
...	all extra parameters are passed to e1071::svm or e1071::tune.svm

Value

The function returns a list with the following fields:

- predictions : an array of (n) predicted phenotypes
- hyperparams : named vector with the following keys: gamma, cost, coef0, nu, epsilon. Some of the values may not make sense given the selected model, and will contain default values from e1071 library.
- extradata : depending on mode parameter, extradata will contain one of the following: 1) a SVM object returned by e1071::tune.svm, containing both the best performing model and the description of the training process 2) a newly trained SVR model 3) the same object passed as tuned.model

See Also

svm, tune.svm, best.svm from e1071 package

Other phenoRegressors: phenoRegressor.BGLR, phenoRegressor.RFR, phenoRegressor.dummy, phenoRegressor.rrBLUP

Examples

```
## Not run:
### WARNING ###
#The 'tuning' part of the example can take quite some time to run,
#depending on the computational power.

#using the GROAN.pea dataset, we regress on the dataset and predict the first ten phenotypes
phenos = GROAN.pea.yield
phenos[1:10] = NA

#--------- TUNE ---------
#tuning the SVR on a grid of hyperparameters
results.tune = phenoRegressor.SVR(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  mode = 'tune',
  kernel = 'linear', cost = 10^(-3:+3) #SVR-specific parameters
)

#examining the predictions
plot(GROAN.pea.yield, results.tune$predictions,
     main = 'Mode = TUNING\nTrain set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results.tune$predictions[1:10], pch=16, col='red')

#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results.tune$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results.tune$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

#--------- TRAIN ---------
#training the SVR, hyperparameters are given
results.train = phenoRegressor.SVR(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  mode = 'train',
  kernel = 'linear', cost = 0.01 #SVR-specific parameters
)

#examining the predictions
plot(GROAN.pea.yield, results.train$predictions,
     main = 'Mode = TRAIN\nTrain set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results.train$predictions[1:10], pch=16, col='red')
```

```
#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results.train$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results.train$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

#--------- RUN ---------
#we recover the trained model from previous run, predictions will be exactly the same
results.run = phenoRegressor.SVR(
  phenotypes = phenos,
  genotypes = GROAN.pea.SNPs,
  covariances = NULL,
  extraCovariates = NULL,
  mode = 'run',
  tuned.model = results.train$extradata
)

#examining the predictions
plot(GROAN.pea.yield, results.run$predictions,
     main = 'Mode = RUN\nTrain set (black) and test set (red) regressions',
     xlab = 'Original phenotypes', ylab = 'Predicted phenotypes')
points(GROAN.pea.yield[1:10], results.run$predictions[1:10], pch=16, col='red')

#printing correlations
test.set.correlation  = cor(GROAN.pea.yield[1:10], results.run$predictions[1:10])
train.set.correlation = cor(GROAN.pea.yield[-(1:10)], results.run$predictions[-(1:10)])
writeLines(paste(
  'test-set correlation :', test.set.correlation,
  '\ntrain-set correlation:', train.set.correlation
))

## End(Not run)
```

`plotResult` *Plot results of a run*

Description

This function uses ggplot2 package (which must be installed) to graphically render the result of a run. The function receive as input the output of GROAN.run and returns a ggplot2 object (that can be further customized). Currently implemented types of plot are:

- `box` : boxplot, showing the distribution of repetitions. See geom_boxplot
- `bar` : barplot, showing the average over repetitions. See stat_summary
- `bar_conf95` : same as 'bar', but with 95% confidence intervals

Usage

```
plotResult(res, variable = c("pearson", "spearman", "rmse", "time_per_fold",
  "coeff_det", "mae"), plot.type = c("box", "bar", "bar_conf95"),
  strata = c("no_strata", "avg_strata", "single"))
```

Arguments

`res`	a result data frame containing the output of GROAN.run
`variable`	name of the variable to be plotted
`plot.type`	a string indicating the type of plot to be obtained
`strata`	string determining behaviour toward strata. If `'no_strata'` will plot accuracies not considering strata. If `'avg_strata'` will average single strata accuracies. If `'single'` each strata will be represented separately.

Value

a ggplot2 object

`print.GROAN.NoisyDataset` *Print a GROAN Noisy Dataset object*

Description

Short description for class GROAN.NoisyDataset, created with createNoisyDataset.

Usage

```
## S3 method for class 'GROAN.NoisyDataset'
print(x, ...)
```

Arguments

`x`	object of class GROAN.NoisyDataset.
`...`	ignored, put here to match S3 function signature

Value

This function returns the original `GROAN.NoisyDataset` object invisibly (via invisible(x))

print.GROAN.Workbench *Print a GROAN Workbench object*

Description

Short description for class GROAN.Workbench, created with createWorkbench.

Usage

```
## S3 method for class 'GROAN.Workbench'
print(x, ...)
```

Arguments

x	object of class GROAN.Workbench.
...	ignored, put here to match S3 function signature

Value

This function returns the original GROAN.Workbench object invisibly (via invisible(x))

summary.GROAN.Result *Summary of GROAN.Result*

Description

Performance metrics are averaged over repetitions, so that a data.frame is produced with one row per dataset/regressor/extra_covariates/strata/samples/markers/folds combination.

Usage

```
## S3 method for class 'GROAN.Result'
summary(object, ...)
```

Arguments

object	an object returned from GROAN.run
...	additional arguments ignored, added for compatibility to generic summary function

Value

a data.frame with averaged statistics

Index