

# Package ‘fastfurious’

November 9, 2015

**Type** Package

**Title** fastfurious

**Version** 0.1-1

**Date** 2015-08-03

**Author** Gino Tesei <gtesei@yahoo.com>

**Maintainer** Gino Tesei <gtesei@yahoo.com>

**Description** fast-furiuos gathers code, models and meta-models I needed in my Machine Learning Lab but I didn't found on the shelf.

**License** MIT + file LICENSE

**URL** <https://github.com/gteseifast-furious>

**BugReports** <https://github.com/gteseifast-furious/issues>

**VignetteBuilder** knitr

**Suggests** knitr, lattice (>= 0.20), ggplot2 (>= 1.0.0), testthat, Cubist, arm, MASS, kknn, kernlab, ipred, randomForest, pROC

**Depends** R (>= 2.10), caret

**Imports** parallel, subselect, plyr, xgboost, magrittr, stringr, e1071, glmnet, verification

## R topics documented:

ff.bindPath . . . . .	2
ff.blend . . . . .	2
ff.corrFilter . . . . .	4
ff.createEnsemble . . . . .	5
ff.encodeCategoricalFeature . . . . .	7
ff.extractDateFeature . . . . .	8
ff.featureFilter . . . . .	9
ff.getBestBlenderPerformance . . . . .	10
ff.getBestBlenderTune . . . . .	10
ff.getMaxCuncurrentThreads . . . . .	11
ff.getPath . . . . .	11
ff.getPathBindings . . . . .	12
ff.kmeans . . . . .	12
ff.makeFeatureSet . . . . .	13
ff.pca . . . . .	14

ff.plotPerformance.reg . . . . .	16
ff.poly . . . . .	16
ff.setBasePath . . . . .	17
ff.setMaxCuncurrentThreads . . . . .	17
ff.summaryBlender . . . . .	18
ff.trainAndPredict.class . . . . .	18
ff.trainAndPredict.reg . . . . .	20
ff.verifyBlender . . . . .	22
RMSE.xgb . . . . .	23
RMSLE.xgb . . . . .	23

<b>Index</b>	<b>24</b>
--------------	-----------

---

ff.bindPath	<i>Bind an absolute path for a kind of resources.</i>
-------------	---

---

## Description

Bind an absolute path for a kind of resources.

## Usage

```
ff.bindPath(type, sub_path, createDir = FALSE)
```

## Arguments

type	the type of resource.
sub_path	the suffix to concatenate to the absolute path to get the absolute path of the kind of resource.
createDir	set to TRUE to create the directory if it does not exist

## Examples

```
ff.setBasePath(getwd())
if(! dir.exists("mydata") ) dir.create(mydata)
ff.bindPath(type = "data",sub_path = "mydata")
```

---

ff.blend	<i>Given a tuned regression model, finds more performant tuning configurations using Nelder/Mead, quasi-Newton and conjugate-gradient algorithms.</i>
----------	---

---

## Description

Given a tuned regression model, finds more performant tuning configurations using Nelder/Mead, quasi-Newton and conjugate-gradient algorithms.

## Usage

```
ff.blend(bestTune, caretModelName, Xtrain, y, controlObject, max_secs = 10 *
  60, seed = NULL, method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B",
    "SANN"), useInteger = TRUE, parallelize = TRUE, verbose = TRUE)
```

## Arguments

<code>bestTune</code>	a data.frame with best tuned parameters of specified model.
<code>caretModelName</code>	a string specifying which model to use. Possible values are <code>lm</code> , <code>bayesglm</code> , <code>glm</code> , <code>glmStepAIC</code> , <code>rlm</code> , <code>knn</code> , <code>pls</code> , <code>ridge</code> , <code>enet</code> , <code>svmRadial</code> , <code>treebag</code> , <code>gbm</code> , <code>rf</code> , <code>cubist</code> , <code>avNNet</code> , <code>xgbTreeGTJ</code> , <code>xgbTree</code>
<code>Xtrain</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>y</code>	the output variable as numeric vector
<code>controlObject</code>	a list of values that define how this function acts. Must be a caret <code>trainControl</code> object
<code>max_secs</code>	the max number of seconds as time constraint
<code>seed</code>	a user specified seed. Useful for replicable execution (e.g. passing the same seed to the <a href="#">ff.verifyBlender</a> function) if the control object involves random steps for creating resamples.
<code>method</code>	the method to use. Possible values are <code>c(Nelder-Mead, BFGS, CG, L-BFGS-B, SANN)</code> .
<code>useInteger</code>	TRUE if the tuning grid is composed of integers and not of continuous numbers.
<code>parallelize</code>	TRUE to enable parallelization (require <code>parallel</code> ).
<code>verbose</code>	TRUE to enable verbose mode.

## Value

a list of lists (one for each specified optimization method) with components `par` (best set of parameters found), `value` (the value of `fn` corresponding to `par`), `counts` (a two-element integer vector giving the number of calls to `fn` and `gr` respectively; this excludes those calls needed to compute the Hessian, if requested, and any calls to `fn` to compute a finite-difference approximation to the gradient), `convergence` (an integer code. 0 indicates successful completion which is always the case for SANN and Brent), `message` (a character string giving any additional information returned by the optimizer, or NULL), `seed` (the used seed). For further details see [optim](#).

## References

<https://stat.ethz.ch/pipermail/r-devel/2010-August/058081.html>

## Examples

```
## suppress warnings raised because there few obs
warn_def = getOption(warn)
options(warn=-1)

## data
Xtrain <- data.frame( a = rep(1:5 , each = 2), b = 10:1,
  c = rep(as.Date(c("2007-06-22", "2004-02-13")),5) )
Xtest <- data.frame( a = rep(2:6 , each = 2), b = 1:10,
  c = rep(as.Date(c("2007-03-01", "2004-05-23")),5) )
Ytrain = 1:10 + runif(nrow(Xtrain))
```

```

## encode datasets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D"))
Xtrain = l$traindata
Xtest = l$testdata

## make a caret control object
controlObject <- trainControl(method = "repeatedcv",
                               repeats = 1, number = 2)

## train and predict
tp = ff.trainAndPredict.reg(Ytrain=Ytrain ,
                           Xtrain=Xtrain ,
                           Xtest=Xtest ,
                           model.label = "cubist" ,
                           controlObject=controlObject)

pred_test = tp$pred
model = tp$model
secs = tp$secs

## blender
gBlender = ff.blend(bestTune = tp$model$bestTune,
                    caretModelName = "cubist" ,
                    Xtrain = Xtrain ,
                    y = Ytrain, controlObject = tp$model$control,
                    max_secs = 3,
                    seed = 123,
                    method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
                    useInteger = TRUE,
                    parallelize = TRUE,
                    verbose = FALSE)

ff.summaryBlender(gBlender)
ff.getBestBlenderPerformance(gBlender)
bestTune = ff.getBestBlenderTune(gBlender)
ff.verifyBlender (gBlender,Xtrain=Xtrain,y=Ytrain,seed=123,
                  controlObject=tp$model$control,caretModelName = "cubist")

## restore warnings
options(warn=warn_def)

```

---

ff.corrFilter

---

*Filter a data.frame of numeric according to a given threshold of correlation*


---

## Description

Filter a data.frame of numeric according to a given threshold of correlation

## Usage

```
ff.corrFilter(Xtrain, Xtest, y, abs_th = NULL, rel_th = 1,
              method = "pearson")
```

**Arguments**

Xtrain	a train set data.frame of numeric
Xtest	a test set data.frame of numeric
y	the output variable (as numeric vector)
abs_th	an absolute threshold (= number of data frame columns)
rel_th	a relative threshold (= percentage of data frame columns)
method	a character string indicating which correlation method is to be used for the test. One of "pearson", "kendall", or "spearman".

**Value**

a list of filtered train set and test set with correlation test results

**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = c(4:1,6,6), c = rep(1,6))
Xtest <- Xtrain + runif(nrow(Xtrain))
y = 1:6
l = ff.corrFilter(Xtrain=Xtrain,Xtest=Xtest,y=y,rel_th=0.5)
Xtrain.filtered = l$Xtrain
Xtest.filtered = l$Xtest
```

---

ff.createEnsemble	<i>Create an ensemble of a tuned model</i>
-------------------	--

---

**Description**

Create an ensemble of a tuned model

**Usage**

```
ff.createEnsemble(Xtrain, Xtest, y, caretModelName, bestTune, predTest = NULL,
  removePredictorsMakingIllConditionedSquareMatrix_forLinearModels = TRUE,
  controlObject, parallelize = TRUE, verbose = TRUE, regression = TRUE,
  ...)
```

**Arguments**

Xtrain	the encoded data.frame of train data. Must be a data.frame of numeric
Xtest	the encoded data.frame of test data. Must be a data.frame of numeric
y	the output variable as numeric vector
caretModelName	a string specifying which model to use. Possible values for regression are lm, bayesglm, glm, glmStepAIC, rlm, knn, pls, ridge, enet, svmRadial, treebag, gbm, rf, cubist, avNNet, xgbTreeGTJ, xgbTree.
bestTune	a data.frame with best tuned parameters of specified model.
predTest	test set prediction (numeric vector). If available, passing it through this paramter the function doesn't compute it again for creating the esemble.

<code>removePredictorsMakingIllConditionedSquareMatrix_forLinearModels</code>	TRUE for removing predictors making ill-conditioned square matrices in case of fragile linear models, i.e. <code>c(rlm,pls,ridge,enet)</code> for regression.
<code>controlObject</code>	a list of values that define how this function acts. Must be a caret <code>trainControl</code> object
<code>parallelize</code>	TRUE to enable parallelization (require <code>parallel</code> ).
<code>verbose</code>	TRUE to enable verbose mode.
<code>regression</code>	TRUE to create an ensemble of a tuned regression model and FALSE to create an ensemble of a tuned classification model.
<code>...</code>	arguments passed to the regression routine.

### Value

a list of train and test predictions.

### Examples

```
## suppress warnings raised because there few obs
warn_def = getOption(warn)
options(warn=-1)

## data
Xtrain <- data.frame( a = rep(1:10 , each = 2), b = 20:1,
  c = rep(as.Date(c("2007-06-22", "2004-02-13")),10) )
Xtest <- data.frame( a = rep(2:11 , each = 2), b = 1:20,
  c = rep(as.Date(c("2007-03-01", "2004-05-23")),10) )
Ytrain = 1:20 + runif(nrow(Xtrain))

## encode datasets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D"))
Xtrain = l$traindata
Xtest = l$testdata

## make a caret control object
controlObject <- trainControl(method = "repeatedcv", repeats = 1, number = 2)

tp = ff.trainAndPredict.reg(Ytrain=Ytrain ,
  Xtrain=Xtrain ,
  Xtest=Xtest ,
  model.label = "cubist" ,
  controlObject=controlObject)

pred_test = tp$pred
model = tp$model
secs = tp$secs

## create ensemble
en = ff.createEnsemble(Xtrain = Xtrain,
  Xtest = Xtest,
  y = Ytrain,
  bestTune = tp$model$bestTune ,
  caretModelName = "cubist" ,
  parallelize = TRUE,
  removePredictorsMakingIllConditionedSquareMatrix_forLinearModels = TRUE,
  controlObject = tp$model$control)
```

```

predTrain = en$predTrain
predTest = en$predTest

## restore warnings
options(warn=warn_def)

```

---

```
ff.encodeCategoricalFeature
```

*Encode a generic predictor as a categorical features using both observations of train set and test for levels. It's anyway possible to adopt more levels by using the parameter levels. Notice that modeling a generic vector, e.g. c(1,2,3,4,5,2,3) as a categorical predictor xor a numeric predictor is a modeling choice (eventually to be assessed by cross-validation).*

---

## Description

Encode a generic predictor as a categorical features using both observations of train set and test for levels. It's anyway possible to adopt more levels by using the parameter levels. Notice that modeling a generic vector, e.g. c(1,2,3,4,5,2,3) as a categorical predictor xor a numeric predictor is a modeling choice (eventually to be assessed by cross-validation).

## Usage

```

ff.encodeCategoricalFeature(data.train, data.test, colname.prefix,
  asNumericSequence = F, replaceWhiteSpaceInLevelsWith = NULL,
  levels = NULL, remove1DummyVar = FALSE)

```

## Arguments

<code>data.train</code>	the observations of the predictor in train set.
<code>data.test</code>	the observations of the predictor in test set.
<code>colname.prefix</code>	the prefix of output data frame.
<code>asNumericSequence</code>	set T if the predictor is a numeric sequence filling any possible hole between min and max in observations that could occur both in train set and test set.
<code>replaceWhiteSpaceInLevelsWith</code>	replace possible spaces in the train/test name of feature.
<code>levels</code>	the levels of the categorical feature. Must be NULL if <code>asNumericSequence</code> is T.
<code>remove1DummyVar</code>	T to remove one dummy variable. Why? First, if you know the values of the first C - 1 dummy variables, you know the last one too and it is more economical to use C - 1. Secondly, if the model has slopes and intercepts (e.g. linear regression), the sum of all of the dummy variables will add up to the intercept (usually encoded as a "1") and that is bad for the math involved. On the other hand, there are models like penalized methods (such as ridge regression) that seldom penalize the intercept, so a C-1 encoded variable could cause the other category effects to be penalized towards the reference category effect.

**Value**

the list of trainset and testset after applying the specified filters

**References**

<http://appliedpredictivemodeling.com/blog/2013/10/23/the-basics-of-encoding-categorical-data-fo>

**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = 6:1, c = letters[1:6])
Xtest <- data.frame( a = rep(2:4 , each = 2), b = 1:6, c = letters[6:1])
print(Xtrain)
#   a b c
# 1 1 6 a
# 2 1 5 b
# 3 2 4 c
# 4 2 3 d
# 5 3 2 e
# 6 3 1 f

l = ff.encodeCategoricalFeature (Xtrain$c , Xtest$c , "c")
l$traindata
#      c_1 c_2 c_3 c_4 c_5 c_6
# 7      1  0  0  0  0  0
# 8      0  1  0  0  0  0
# 9      0  0  1  0  0  0
# 10     0  0  0  1  0  0
# 11     0  0  0  0  1  0
# 12     0  0  0  0  0  1

Xtrain[,c] = NULL
Xtest[,c] = NULL
Xtrain = cbind(Xtrain,l$traindata)
Xtest = cbind(Xtest,l$testdata)
```

---

**ff.extractDateFeature** *Extracts a numerical feature from a date predictor. The feature is built as the difference in days from the oldest date in both the train set and test set and any given observation.*

---

**Description**

Extracts a numerical feature from a date predictor. The feature is built as the difference in days from the oldest date in both the train set and test set and any given observation.

**Usage**

```
ff.extractDateFeature(data.train, data.test)
```

**Arguments**

**data.train**        the observations of the predictor in train set.  
**data.test**        the observations of the predictor in test set.



**Value**

the list of trainset and testset after applying the specified encoding and the related date range

**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = 6:1,
  c = rep(as.Date(c("2007-06-22", "2004-02-13")),3) )
Xtest <- data.frame( a = rep(2:4 , each = 2), b = 1:6,
  c = rep(as.Date(c("2007-03-01", "2004-05-23")),3) )
l = ff.extractDateFeature(Xtrain$c,Xtest$c)
Xtrain[,c] = NULL
Xtest[,c] = NULL
Xtrain = cbind(Xtrain,c=l$traindata)
Xtest = cbind(Xtest,c=l$testdata)
```

---

ff.featureFilter	<i>Filter predictors according to specified criteria.</i>
------------------	---

---

**Description**

Filter predictors according to specified criteria.

**Usage**

```
ff.featureFilter(traindata, testdata, y = NULL,
  removeOnlyZeroVariancePredictors = FALSE,
  performVarianceAnalysisOnTrainSetOnly = TRUE, correlationThreshold = NULL,
  removePredictorsMakingIllConditionedSquareMatrix = TRUE,
  removeIdenticalPredictors = TRUE, removeHighCorrelatedPredictors = TRUE,
  featureScaling = TRUE, verbose = TRUE)
```

**Arguments**

traindata	the train set
testdata	the test set
y	the response variable. Must be not NULL if correlationThreshold is not NULL.
removeOnlyZeroVariancePredictors	TRUE to remove only zero variance predictors
performVarianceAnalysisOnTrainSetOnly	TRUE to perform the variance analysis on the train set only
correlationThreshold	a correlation threshold above which keeping predictors (considered only if removeOnlyZeroVariancePredictors is FALSE).
removePredictorsMakingIllConditionedSquareMatrix	TRUE to predictors making ill conditioned square matrices
removeIdenticalPredictors	TRUE to remove identical predictors (using base::identical function)
removeHighCorrelatedPredictors	TRUE to remove high correlated predictors
featureScaling	TRUE to perform feature scaling
verbose	TRUE to set verbose mode

**Value**

the list of trainset and testset after applying the specified filters

**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = c(4:1,6,6), c = rep(1,6))
Xtest <- Xtrain + runif(nrow(Xtrain))
l = ff.featureFilter (traindata = Xtrain,
                     testdata = Xtest,
                     removeOnlyZeroVariancePredictors=TRUE)

Xtrain = l$traindata
Xtest = l$testdata
```

---

ff.getBestBlenderPerformance

*Helper function that given a blender object returns the best optimization method.*

---

**Description**

Helper function that given a blender object returns the best optimization method.

**Usage**

```
ff.getBestBlenderPerformance(blender)
```

**Arguments**

blender            a blender object

**Value**

a numeric of best score and as object name the best performant method name.

**See Also**

[ff.blend](#) for examples.

---

ff.getBestBlenderTune    *Helper function that given a blender object returns the best tuning parameters found by the blender.*

---

**Description**

Helper function that given a blender object returns the best tuning parameters found by the blender.

**Usage**

```
ff.getBestBlenderTune(blender, truncate = TRUE)
```

**Arguments**

blender	a blender object
truncate	TRUE to cut at the first tuning best configuration in case there are more than one optimal tuning configurations.

**Value**

a data.frame of the best tuning parameters.

**See Also**

[ff.blend](#) for examples.

---

```
ff.getMaxCuncurrentThreads
```

*Get the max number of cuncurrent threads.*

---

**Description**

Get the max number of cuncurrent threads.

**Usage**

```
ff.getMaxCuncurrentThreads()
```

**Examples**

```
ff.getMaxCuncurrentThreads()
```

---

```
ff.getPath
```

*Get the absolute path for a kind of resources.*

---

**Description**

Get the absolute path for a kind of resources.

**Usage**

```
ff.getPath(type = "base")
```

**Arguments**

type	the type of resource.
------	-----------------------

**Value**

the absolute path for a kind of resources (as character)

**Examples**

```
ff.setBasePath(./)
ff.getPath() ## equivalent to ff.getPath(type="base")
```

---

<code>ff.getPathBindings</code>	<i>Get the list of bindings, i.e. (type resource,absolute path) pairs as a list</i>
---------------------------------	---

---

### Description

Get the list of bindings, i.e. (type resource,absolute path) pairs as a list

### Usage

```
ff.getPathBindings()
```

### Value

the list of bindings

### Examples

```
ff.setBasePath(getwd())
if(! dir.exists("mydata") ) dir.create(mydata)
ff.bindPath(type = "data",sub_path = "mydata")
ff.getPathBindings()
```

---

<code>ff.kmeans</code>	<i>An useful wrapper of <a href="#">kmeans</a> performing k-means clustering on the given trainset / testset (Xtrain / Xtest) and assuming a number of cluster from 1:max_centers. The best number of cluster is computed so that the variation in the within group sum of squares between two subsequent number of clusters is maximized in absolute value.</i>
------------------------	--

---

### Description

An useful wrapper of [kmeans](#) performing k-means clustering on the given trainset / testset (Xtrain / Xtest) and assuming a number of cluster from 1:max\_centers. The best number of cluster is computed so that the variation in the within group sum of squares between two subsequent number of clusters is maximized in absolute value.

### Usage

```
ff.kmeans(Xtrain, Xtest, max_centers = 10, nstart = 5, iter.max = 10,
  doPlot = FALSE, verbose = FALSE)
```

### Arguments

<code>Xtrain</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>Xtest</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>max_centers</code>	the max number of clusters to be evaluated
<code>nstart</code>	how many random sets should be chosen? Such a parameter is passed to <a href="#">kmeans</a> .

iter.max	the maximum number of iterations allowed. Such a parameter is passed to <a href="#">kmeans</a> .
doPlot	a logical value indicating whether plotting.
verbose	a logical value indicating whether verbose mode should be enabled.
varThreshold	a threshold indicating the proportion of variance that should be explained. Must be a numeric between 0 and 1.

### Value

a list whose components are the max number of clusters evaluated (max\_centers), the best number of clusters (best\_n\_clusters), the sequence of within groups sum of squares across number of clusters (wss), assuming best\_n\_clusters as number of clusters a vector of integers (from 1:best\_n\_clusters) indicating the cluster to which each point is allocated in the train set (K.train) and test set (K.test).

### Examples

```
## data
Xtrain <- data.frame(a = rep(1:10 , each = 2), b = 20:1,
                    c = rep(as.Date(c("2007-06-22", "2004-02-13")),10) , d = 20:1)
Xtest <- data.frame(a = rep(2:11 , each = 2), b = 1:20,
                   c = rep(as.Date(c("2007-03-01", "2004-05-23")),10) , d = 1:20)

## encode data sets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D","N"))
Xtrain = l$traindata
Xtest = l$testdata

ffKmeans = ff.kmeans(Xtrain = Xtrain , Xtest = Xtest ,
                     max_centers = 10 , verbose = TRUE)

best_n_clusters = ffKmeans$best_n_clusters

K.train = ffKmeans$K.train
K.test = ffKmeans$K.test
```

---

ff.makeFeatureSet	<i>Encode the feature set according to meta data passed as input.</i>
-------------------	---

---

### Description

Encode the feature set according to meta data passed as input.

### Usage

```
ff.makeFeatureSet(data.train, data.test, meta, scaleNumericFeatures = FALSE,
                  parallelize = FALSE, remove1DummyVarInCatPreds = FALSE)
```

**Arguments**

<code>data.train</code>	the observations of the predictor in train set.
<code>data.test</code>	the observations of the predictor in test set.
<code>meta</code>	the meta data. It should be a vector of the character C, N, D, e.g. <code>c(N,C,D)</code> of the same length of the train set / test set columns
<code>scaleNumericFeatures</code>	set to TRUE to center and scale numeric features
<code>parallelize</code>	set to TRUE to enable parallelization (require <code>parallel</code> package)
<code>remove1DummyVarInCatPreds</code>	T to remove one dummy variable in encoding categorical predictors. For further details see <a href="#">ff.encodeCategoricalFeature</a> .

**Value**

the list of trainset and testset after applying the specified encodings

**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = 6:1,
  c = rep(as.Date(c("2007-06-22", "2004-02-13")),3) )
Xtest <- data.frame( a = rep(2:4 , each = 2), b = 1:6,
  c = rep(as.Date(c("2007-03-01", "2004-05-23")),3) )
l = ff.makeFeatureSet(Xtrain,Xtest,c(C,N,D))
Xtrain = l$traindata
Xtest = l$testdata
```

ff.pca

*An useful wrapper of [prcomp](#) performing a principal components analysis on the given trainset / testset (Xtrain / Xtest).*

**Description**

An useful wrapper of [prcomp](#) performing a principal components analysis on the given trainset / testset (Xtrain / Xtest).

**Usage**

```
ff.pca(Xtrain, Xtest, center = TRUE, scale. = FALSE,
  removeZeroVarPredictors = TRUE, varThreshold = 0.95, doPlot = TRUE,
  verbose = FALSE)
```

**Arguments**

<code>Xtrain</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>Xtest</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>center</code>	a logical value indicating whether the variables should be shifted to be zero centered. Alternately, a vector of length equal the number of columns of data can be supplied. The value is passed to <code>scale</code> .

scale.	a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place. The default is FALSE for consistency with S, but in general scaling is advisable. Alternatively, a vector of length equal the number of columns of data can be supplied. The value is passed to scale.
removeZeroVarPredictors	a logical value indicating whether removing zero variance predictors before calling <code>prcomp</code> preventing errors due to the fact that the latter cannot rescale a constant/zero column to unit variance.
varThreshold	a threshold indicating the proportion of variance that should be explained. Must be a numeric between 0 and 1.
doPlot	a logical value indicating whether plotting the proportion of variance explained vs. principal components.
verbose	a logical value indicating whether verbose mode should be enabled.

## Value

a list whose components are the number of principal components (`numComp`), the number of principal components to hold so that the proportion of variance explained by each subsequent principal component drops off as an elbow in the screen plot (`numComp.elbow`), the number of principal components explaining a given (specified by the `varThreshold` input parameter) proportion of variance (`numComp.threshold`), the threshold indicating the proportion of variance that should be explained (`varThreshold`), the cumulative sum of proportion of variance explained by each principal component (`cumVar`), the proportion of variance explained by each principal component (`var`), the principal components for train and test set (`PC.train` and `PC.test`)

## Examples

```
## data
Xtrain <- data.frame(a = rep(1:10 , each = 2), b = 20:1,
                    c = rep(as.Date(c("2007-06-22", "2004-02-13")),10) , d = 20:1)
Xtest  <- data.frame(a = rep(2:11 , each = 2), b = 1:20,
                    c = rep(as.Date(c("2007-03-01", "2004-05-23")),10) , d = 1:20)

## encode data sets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D","N"))
Xtrain = l$traindata
Xtest  = l$testdata

ffPCA = ff.pca(Xtrain = Xtrain , Xtest = Xtest , center = TRUE , scale. = TRUE ,
               removeZeroVarPredictors = TRUE ,
               varThreshold = 0.95 , doPlot = FALSE , verbose = TRUE)

numComp <- ffPCA$numComp
numComp.elbow <- ffPCA$numComp.elbow
numComp.threshold <- ffPCA$numComp.threshold

PC_Xtrain_95Var = ffPCA$PC.train[1:numComp.threshold,,drop=FALSE]
PC_Xtest_95Var  = ffPCA$PC.test[1:numComp.threshold,,drop=FALSE]
```

---

```
ff.plotPerformance.reg
```

*Plot predicted values vs. observed / residual values.*

---

### Description

Plot predicted values vs. observed / residual values.

### Usage

```
ff.plotPerformance.reg(observed, predicted, main = NULL)
```

### Arguments

observed	the observed output variables (numeric vector).
predicted	the predicted values (numeric vector).
main	a string as a title for the plot

### Examples

```
obs = 1:10
preds = obs + runif(length(obs))
ff.plotPerformance.reg(observed = obs , predicted = preds, main="Predicted vs. observed/residual")
```

---

```
ff.poly
```

*Make polynomial terms of a data.frame*

---

### Description

Make polynomial terms of a data.frame

### Usage

```
ff.poly(x, n, direction = 0)
```

### Arguments

x	a data.frame of numeric
n	the polynomial degree
direction	if set to 0 returns the terms $x^{(1/n)}, x^{(1/(n-1))}, \dots, x, x^2, \dots, x^n$ . If set to -1 returns the terms $x^{(1/n)}, x^{(1/(n-1))}, \dots, x$ . If set to 1 returns the terms $x, x^2, \dots, x^n$ .

### Value

the data.frame with the specified polynomial terms



**Examples**

```
Xtrain <- data.frame( a = rep(1:3 , each = 2), b = c(4:1,6,6), c = rep(1,6))
Xtest <- Xtrain + runif(nrow(Xtrain))
data = rbind(Xtrain,Xtest)
data.poly = ff.poly(x=data,n=3)
Xtrain.poly = data.poly[1:nrow(Xtrain),]
Xtest.poly = data.poly[(nrow(Xtrain)+1):nrow(data),]
```

---

ff.setBasePath	<i>Set base path</i>
----------------	----------------------

---

**Description**

Set base path

**Usage**

```
ff.setBasePath(path)
```

**Arguments**

path	the absolute path.
------	--------------------

**Examples**

```
ff.setBasePath("./")
```

---

ff.setMaxCuncurrentThreads	<i>Set the max number of cuncurrent threads.</i>
----------------------------	--

---

**Description**

Set the max number of cuncurrent threads.

**Usage**

```
ff.setMaxCuncurrentThreads(nThreads = 2)
```

**Arguments**

nThreads	max number of cuncurrent threads.
----------	-----------------------------------

**Examples**

```
ff.setMaxCuncurrentThreads(4)
```

---

<code>ff.summaryBlender</code>	<i>Helper function that given a blender object returns a numeric vector of performances (one for each optimization method).</i>
--------------------------------	---

---

### Description

Helper function that given a blender object returns a numeric vector of performances (one for each optimization method).

### Usage

```
ff.summaryBlender(blender)
```

### Arguments

<code>blender</code>	a blender object
----------------------	------------------

### Value

a numeric vector of performances (one for each optimization method)

### See Also

[ff.blend](#) for examples.

---

<code>ff.trainAndPredict.class</code>	<i>Trains a specified classification model on the given train set and predicts on the given test set.</i>
---------------------------------------	---

---

### Description

Trains a specified classification model on the given train set and predicts on the given test set.

### Usage

```
ff.trainAndPredict.class(Ytrain, Xtrain, Xtest, model.label, controlObject,
  best.tuning = FALSE, verbose = FALSE,
  removePredictorsMakingIllConditionedSquareMatrix_forLinearModels = TRUE,
  metric.label = "auc", xgb.metric.fun = NULL, xgb.maximize = FALSE,
  xgb.foldList = NULL, xgb.eta = NULL, xgb.max_depth = NULL,
  xgb.cv.default = TRUE, xgb.param = NULL, ...)
```

**Arguments**

<code>Ytrain</code>	the output variable as numeric vector
<code>Xtrain</code>	the encoded data.frame of train data. Must be a data.frame of numeric
<code>Xtest</code>	the encoded data.frame of test data. Must be a data.frame of numeric
<code>model.label</code>	a string specifying which model to use.
<code>controlObject</code>	a list of values that define how this function acts. Must be a caret trainControl object for all models except that for xgbTreeGTJ.
<code>best.tuning</code>	TRUE to use more dense tuning grid or custom routine/tuning grid if available
<code>verbose</code>	TRUE to enable verbose mode.
<code>removePredictors</code>	MakingIllConditionedSquareMatrix_forLinearModels TRUE for removing predictors making ill-conditioned square matrices in case of fragile linear models.
<code>metric.label</code>	the label of function to optimize/minimize.
<code>xgb.metric.fun</code>	custom function to optimize/minimize for xgbTreeGTJ.
<code>xgb.maximize</code>	TRUE to maximize the specified xgb.metric.fun.
<code>xgb.foldList</code>	custom resampling folds list for xgbTreeGTJ.
<code>xgb.eta</code>	custom eta parameter for xgbTreeGTJ.
<code>xgb.max_depth</code>	custom max_depth parameter for xgbTreeGTJ.
<code>xgb.cv.default</code>	TRUE for using xgboost::xgb.cv function (mandatory in case of fix nrounds), FALSE for using the internal ff.xgb.cv function. The main advantage of the latter is that it doesn't need to restart nrounds in case for the specified nrounds cross validation error is still decreasing.
<code>xgb.param</code>	custom parameters for XGBoost.
<code>...</code>	arguments passed to the regression routine.

**Value**

a list of test predictions, model and number of excecuting seconds.

**Examples**

```
## suppress warnings raised because of few obs
warn_def = getOption(warn)
options(warn=-1)

## data
Xtrain <- data.frame( a = rep(1:10 , each = 2), b = 20:1,
                     c = rep(as.Date(c("2007-06-22", "2004-02-13")),10) , d = 20:1)
Xtest  <- data.frame( a = rep(2:11 , each = 2), b = 1:20,
                     c = rep(as.Date(c("2007-03-01", "2004-05-23")),10) , d = 1:20)
Ytrain = c(rep(1,10),rep(0,10))

## encode datasets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D","N"))
Xtrain = l$traindata
Xtest  = l$testdata

## make a caret control object
controlObject <- trainControl(method = "repeatedcv", repeats = 2, number = 3 ,
```

```

summaryFunction = twoClassSummary , classProbs = TRUE)
tp = ff.trainAndPredict.class(Ytrain=Ytrain ,
                              Xtrain=Xtrain ,
                              Xtest=Xtest,
                              model.label = "svmRadial" ,
                              controlObject=controlObject,
                              verbose=TRUE ,
                              best.tuning=TRUE)

pred_test = tp$pred
model = tp$model
elapsed.secs = tp$secs

bestTune = l$model$bestTune
best_ROC = max(tp$model$results$ROC)

## restore warnings
options(warn=warn_def)

```

---

```
ff.trainAndPredict.reg
```

*Trains a specified model on the given train set and predicts on the given test set.*

---

## Description

Trains a specified model on the given train set and predicts on the given test set.

## Usage

```

ff.trainAndPredict.reg(Ytrain, Xtrain, Xtest, model.label, controlObject,
  best.tuning = FALSE, verbose = FALSE,
  removePredictorsMakingIllConditionedSquareMatrix_forLinearModels = TRUE,
  xgb.metric.fun = RMSLE.xgb, xgb.maximize = FALSE,
  xgb.metric.label = "RMSLE", xgb.foldList = NULL, xgb.eta = NULL,
  xgb.max_depth = NULL, xgb.cv.default = TRUE, xgb.param = NULL, ...)

```

## Arguments

Ytrain	the output variable as numeric vector
Xtrain	the encoded data.frame of train data. Must be a data.frame of numeric
Xtest	the encoded data.frame of test data. Must be a data.frame of numeric
model.label	a string specifying which model to use. Possible values are lm, bayesglm, glm, glmStepAIC, rlm, knn, pls, ridge, enet, svmRadial, treebag, gbm, rf, cubist, avNNet, xgbTreeGTJ, xgbTree
controlObject	a list of values that define how this function acts. Must be a caret trainControl object for all models except that for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
best.tuning	TRUE to use more dense tuning grid or custom routine if available
verbose	TRUE to enable verbose mode.

removePredictorsMakingIllConditionedSquareMatrix_forLinearModels	TRUE for removing predictors making ill-conditioned square matrices in case of fragile linear models, i.e. c(rlm,pls,ridge,enet).
xgb.metric.fun	custom function to optimize/minimize for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.maximize	TRUE to maximize the specified xgb.metric.fun. Only for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.metric.label	custom label of function to optimize/minimize for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.foldList	custom resampling folds list for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.eta	custom eta parameter for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.max_depth	custom max_depth parameter for xgbTreeGTJ and xgbTree. In the latter case only if best.tuning is TRUE.
xgb.cv.default	TRUE for using xgboost::xgb.cv function (mandatory in case of fix nrounds), FALSE for using the internal ff.xgb.cv function. The main advantage of the latter is that it doesn't need to restart nrounds in case for the specified nrounds cross validation error is still decreasing.
xgb.param	custom parameters for XGBoost.
...	arguments passed to the regression routine.

## Value

a list of test predictions, model and number of excecuting seconds.

## Examples

```
## suppress warnings raised because of few obs
warn_def = getOption(warn)
options(warn=-1)

## data
Xtrain <- data.frame( a = rep(1:10 , each = 2), b = 20:1,
  c = rep(as.Date(c("2007-06-22", "2004-02-13")),10) )
Xtest <- data.frame( a = rep(2:11 , each = 2), b = 1:20,
  c = rep(as.Date(c("2007-03-01", "2004-05-23")),10) )
Ytrain = 1:20 + runif(nrow(Xtrain))

## encode datasets
l = ff.makeFeatureSet(Xtrain,Xtest,c("C","N","D"))
Xtrain = l$traindata
Xtest = l$testdata

## make a caret control object
controlObject <- trainControl(method = "repeatedcv", repeats = 1, number = 2)

tp = ff.trainAndPredict.reg(Ytrain=Ytrain ,
  Xtrain=Xtrain ,
  Xtest=Xtest ,
  model.label = "cubist" ,
```

```

                                controlObject=controlObject)

    pred_test = tp$pred
    model = tp$model
    elapsed.secs = tp$secs

    ## restore warnings
    options(warn=warn_def)

```

---

ff.verifyBlender	<i>Helper function that given a blender object replicates the execution in order to verify performances.</i>
------------------	--

---

## Description

Helper function that given a blender object replicates the execution in order to verify performances.

## Usage

```
ff.verifyBlender(blender, Xtrain, y, seed = NULL, controlObject,
                 caretModelname)
```

## Arguments

blender	a blender object
Xtrain	the train set
y	the output variable as numeric vector
seed	the seed used by the blender, if applicable. If the blender used one, it is necessary for replicating blender performances.
controlObject	a list of values that define how this function acts. Must be a caret trainControl object. It must be the same used by the blender.
caretModelname	a string specifying which model to use. Possible values are lm, bayesglm, glm, glmStepAIC, rlm, knn, pls, ridge, enet, svmRadial, treebag, gbm, rf, cubist, avNNet, xgbTreeGTJ, xgbTree. It must be the same model name used by the blender.

## Value

a numeric as difference in performance between blender and replicated execution.

## See Also

[ff.blend](#) for examples.

---

RMSE.xgb	<i>Root mean square error</i>
----------	-------------------------------

---

**Description**

Root mean square error

**Usage**

```
RMSE.xgb(preds, dtrain)
```

**Arguments**

preds	the predicted values (numeric vector).
dtrain	the xgboost train set object.

**Value**

a list of metric label / values

---

RMSLE.xgb	<i>Root mean square logistic error</i>
-----------	--

---

**Description**

Root mean square logistic error

**Usage**

```
RMSLE.xgb(preds, dtrain, th_err = 1.5)
```

**Arguments**

preds	the predicted values (numeric vector).
dtrain	the xgboost train set object.
th_err	a threshold in case predictions are negative.

**Value**

a list of metric label / values

# Index

`ff.bindPath`, [2](#)  
`ff.blend`, [2](#), [10](#), [11](#), [18](#), [22](#)  
`ff.corrFilter`, [4](#)  
`ff.createEnsemble`, [5](#)  
`ff.encodeCategoricalFeature`, [7](#), [14](#)  
`ff.extractDateFeature`, [8](#)  
`ff.featureFilter`, [9](#)  
`ff.getBestBlenderPerformance`, [10](#)  
`ff.getBestBlenderTune`, [10](#)  
`ff.getMaxCuncurrentThreads`, [11](#)  
`ff.getPath`, [11](#)  
`ff.getPathBindings`, [12](#)  
`ff.kmeans`, [12](#)  
`ff.makeFeatureSet`, [13](#)  
`ff.pca`, [14](#)  
`ff.plotPerformance.reg`, [16](#)  
`ff.poly`, [16](#)  
`ff.setBasePath`, [17](#)  
`ff.setMaxCuncurrentThreads`, [17](#)  
`ff.summaryBlender`, [18](#)  
`ff.trainAndPredict.class`, [18](#)  
`ff.trainAndPredict.reg`, [20](#)  
`ff.verifyBlender`, [3](#), [22](#)  
  
`kmeans`, [12](#), [13](#)  
  
`optim`, [3](#)  
  
`prcomp`, [14](#), [15](#)  
  
`RMSE.xgb`, [23](#)  
`RMSLE.xgb`, [23](#)