

Package ‘ModelMap’

January 2, 2012

Type Package

Title Creates Random Forest and Stochastic Gradient Boosting Models, and applies them to GIS .img files to build detailed prediction maps.

Version 2.1.1

Date 2011-04-21

Depends R (>= 2.8.1), randomForest, gbm, PresenceAbsence, rgdal, fields

Author Elizabeth Freeman, Tracey Frescino

Maintainer Elizabeth Freeman <eafreeman@fs.fed.us>

Description This package will create sophisticated models of training data and validate the models with an independent test set, cross validation, or in the case of Random Forest Models, with Out Of Bag (OOB) predictions on the training data. It will create graphs and tables of the model validation results. It will apply these models to GIS .img files of predictors to create detailed prediction surfaces. It will handle large predictor files for map making, by reading in the .img files in chunks, and output to the .txt file the prediction for each data chunk, before reading the next chunk of data.

License Unlimited

Repository CRAN

Date/Publication 2011-04-23 20:54:47

R topics documented:

ModelMap-package	2
build.rastLUT	4
get.test	5
model.build	6
model.diagnostics	13
model.interaction.plot	20
model.mapmake	26

ModelMap-package	<i>Modeling and Map production using Random Forest and Stochastic Gradient Boosting</i>
------------------	---

Description

This package will create sophisticated models of training data and validate the models with an independent test set, cross validation, or in the case of Random Forest Models, with Out Of Bag (OOB) predictions on the training data. It will create graphs and tables of the model validation results. It will apply these models to GIS .img files of predictors to create detailed prediction surfaces. It will handle large predictor files for map making, by reading in the .img files in chunks, and output to the .txt file the prediction for each data chunk, before reading the next chunk of data.

Details

Package: ModelMap
 Type: Package
 Version: 2.1.1
 Date: 2011-04-21
 License: Unlimited. This code was written and prepared by a U.S. Government employee on official time, and therefore it is

This package provides a push button approach to complex model building and production mapping. It contains five functions.

It contains two simple functions that can be used before beginning the model building process: [get.test](#) that can be used to randomly divide a training dataset into training and test/validation sets; and [build.rastLUT](#) that uses GUI prompts to walk a user through the process of setting up a Raster look up table to link predictors from the training data with the rasters used for map construction.

It also contains four functions used for the model building and map construction process itself: [model.build](#), [model.diagnostics](#), [model.interaction.plot](#) and [model.mapmake](#).

ModelMap can be run in a traditional R command mode, where all arguments are specified in the function call. However it can also be used in a full push button mode, where you type in the simple command such as [model.build](#), and GUI pop up windows ask questions about the type of model, the file locations of the data, etc...

Random Forest is implemented through the `randomForest` package within R. Random Forest is more user friendly than Stochastic Gradient Boosting, as it has fewer parameters to be set by the user, and is less sensitive to tuning of these parameters. A Random Forest model consists of multiple trees that vote on predictions. For each tree a random subset of the training data is used to construct the tree, with the remaining data points used to construct out-of-bag (OOB) error estimates. At each node of the tree a random selection of predictors is chosen to determine the split. The number of predictors used to select the splits is the primary user specified parameter that can affect model performance, and this parameter can be automatically optimized using the `randomForest` function `tuneRF()`. Random Forest will not over fit data, therefore the only penalty of increasing the number

of trees is computation time. Random Forest can compute variable importance, an advantage over some "black box" modeling techniques if it is important to understand the ecological relationships underlying a model (Breiman, 2001).

Stochastic gradient boosting (Friedman 2001, 2002), is related to both boosting and bagging. Many small classification or regression trees are built sequentially from "pseudo"-residuals (the gradient of the loss function of the previous tree). At each iteration, a tree is built from a random sub-sample of the dataset (selected without replacement) and an incremental improvement in the model. Using only a fraction of the training data increases both the computation speed and the prediction accuracy, while also helping to avoid over-fitting the data. An advantage of stochastic gradient boosting is that it is not necessary to pre-select or transform predictor variables. It is also resistant to outliers, as the steepest gradient algorithm emphasizes points that are close to their correct classification. Stochastic gradient boosting is implemented through the `gbm` package within R. One disadvantage of Stochastic Gradient Boosting, compared to Random Forest, is increased number of user specified parameters, and the SGB models tend to be more sensitive to these parameters. Model fitting parameters include distribution, interaction depth, bagging fraction, shrinkage rate, and training fraction. These parameters can be set in the argument list when calling `model.map()`. Values for these parameters other than the defaults can not be set by point and click in the GUI pop up windows. Friedman (2001, 2002) and Ridgeway (1999) provide guidelines on appropriate settings for model fitting options.

For Presence-Absence data, the package `PresenceAbsence` is used for model validation.

For map making, the `rgdal` is used to read `.img` files.

For interaction plots, the `fields` package is used to produce image plots.

Author(s)

Author: Elizabeth Freeman and Tracey Frescino

Maintainer: Elizabeth Freeman <eafreeman@fs.fed.us>

References

Breiman, L. (2001) Random Forests. *Machine Learning*, 45:5-32.

Elith, J., Leathwick, J. R. and Hastie, T. (2008). A working guide to boosted regression trees. *Journal of Animal Ecology*. 77:802-813.

Friedman, J.H. (2001). Greedy function approximation: a gradient boosting machine. *Ann. Stat.*, 29(5):1189-1232.

Friedman, J.H. (2002). Stochastic gradient boosting. *Comput. Stat. Data An.*, 38(4):367-378.

Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. *R News* 2(3), 18-22.

Ridgeway, G., (1999). The state of boosting. *Comp. Sci. Stat.* 31:172-181

 build.rastLUT

Build a raster Look-UP-Table for training dataset

Description

GUI prompts will help the user build a Look-Up-Table to associate predictor variable with their corresponding spatial rasters.

Usage

```
build.rastLUT(imageList=NULL, predList=NULL, qdata.trainfn=NULL, rastLUTfn=NULL, folder=NULL)
```

Arguments

imageList	Vector. A vector of character strings giving names and full paths to all raster data files used in model.
predList	Vector. A vector of character strings giving the predictor names used as headers in the model training data.
qdata.trainfn	String. The name (full path or base name with path specified by folder) of the training data file used for building the model. The file must be a comma-delimited file *.csv with column headings. qdata.trainfn can also be an R dataframe. The column headers from qdata.trainfn are used to generate a list of possible predictors for the raster Look-UP-Table.
rastLUTfn	String. The name of the file output for the Look-Up-Table. By default, if a file name is provided by the "qdatatrainfn" argument "_rastLUT.csv" appended after "qdatatrainfn". Otherwise, default filename for look-up-table is "rastLUT.csv"
folder	String. The folder used for output. Do not add ending slash to path string. If folder = NULL (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify folder = getwd().

Details

This function helps the user create a raster Look-Up-Table to be used later by `model.mapmake()`. Currently this function only works in a Windows environment.

First, if "folder" is not given, the user selects the output folder for the Look-Up-Table.

Second, if "predList" or "qdatatrainfn" are not given, the user selects the file containing the training data. The header of the file is used to generate a selection list of possible predictor variables.

Third, if "imageList" is not provided, the user selects the rasters.

Finally, the function steps through each band of each raster, and the user selects the appropriate predictor.

Value

Returns a data frame containing the raster Look-Up-Table. Also Writes a .csv file containing the raster Look-Up-Table.

Author(s)

Elizabeth Freeman

Examples

```

folder<-system.file("external", "helpexamples", package = "ModelMap")
qdata.trainfn = paste(folder, "/DATATRAIN.csv", sep="")

#build.rastLUT( qdata.trainfn=qdata.trainfn,
# folder=folder)

```

get.test

*Randomly Divide Data into Training and Test Sets***Description**

Uses random selection to split a dataset into training and test data sets

Usage

```
get.test(proportion.test, qdatafn = NULL, seed = NULL, folder=NULL, qdata.trainfn = paste(strsplit(qdatafn, "\\.csv$"), "_train.csv", sep=""))
```

Arguments

proportion.test	Number. The proportion of the training data that will be randomly extracted for use as a test set. Value between 0 and 1.
qdatafn	String. The name (basename or full path) of the data file to be split into training and test data. This data should include both response and predictor variables. The file must be a comma-delimited file (*.csv) with column headings and the predictor names in the file must match the raster layer files, if applying predictions (predict = TRUE). If NULL (the default), a GUI interface prompts user to browse to the data file.
seed	Integer. The number used to initialize randomization to randomly select rows for a test data set. If you want to produce the same model later, use the same seed. If seed = NULL (the default), a new one is created each time.
folder	String. The folder used for all output from predictions and/or maps. Do not add ending slash to path string. If folder = NULL (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify folder = getwd().
qdata.trainfn	String. The name of the file output of training data. By default, _train appended after qdatafn.
qdata.testfn	String. The name of the file output of test data. By default, _test appended after qdatafn.

Details

This function should be run once, before starting analysis to create training and test sets. If the cross validation option is to be used with RF or SGB models, or if the OOB option is to be used for RF models, then this step is unnecessary.

Value

Outputs a training data file and test data file. Unless `qdata.trainfn` or `qdata.testfn` are specified, the output will be located in the same folder as the original data file (`qdatafn`). The output will have the same rows and columns as the original data.

Author(s)

Elizabeth Freeman

Examples

```
qdatafn<-system.file("external", "helpexamples", "DATATRAIN.csv", package = "ModelMap")  
qdata<-read.table(file=qdatafn, sep="," , header=TRUE, check.names=FALSE)  
  
get.test( proportion.test=0.2,  
qdatafn=qdatafn,  
seed=42,  
folder=getwd(),  
qdata.trainfn="example.train.csv",  
qdata.testfn="example.test.csv")
```

model.build

Model Building

Description

Create sophisticated models using either Random Forest or Stochastic Gradient Boosting from training data

Usage

```
model.build(model.type = NULL, qdata.trainfn = NULL, folder = NULL, MODELfn = NULL, predList = NULL, pre
```

Arguments

model.type	String. Model type. "RF" or "SGB". (Eventually planned to include "GAM".) If model.obj is specified, the model.type will be extracted from model.obj, and the argument model.type will be ignored (with a warning).
qdata.trainfn	String. The name (full path or base name with path specified by folder) of the training data file used for building the model (file should include columns for both response and predictor variables). The file must be a comma-delimited file *.csv with column headings. qdata.trainfn can also be an R dataframe. If predictions will be made (predict = TRUE or map=TRUE) the predictor column headers must match the names of the raster layer files, or a rastLUT must be provided to match predictor columns to the appropriate raster and band. If qdata.trainfn = NULL (the default), a GUI interface prompts user to browse to the training data file.
folder	String. The folder used for all output from predictions and/or maps. Do not add ending slash to path string. If folder = NULL (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify folder = getwd().
MODELfn	String. The file name to use to save the generated model object. If MODELfn = NULL (the default), a default name is generated by pasting model.type_response.type_response.name. If the other output filenames are left unspecified, MODELfn will be used as the basic name to generate other output filenames. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by folder.
predList	String. A character vector of the predictor short names used to build the model. These names must match the column names in the training/test data files and the names in column two of the rastLUT. If predList = NULL (the default), a GUI interface prompts user to select predictors from column 2 of rastLUT. If both predList = NULL and rastLUT = NULL, then a GUI interface prompts user to browse to rasters used as predictors, and select from a generated list, the individual layers (bands) of rasters used to build the model. In this case (i.e., rastLUT = NULL), predictor column names of training data must be standard format, consisting of raster stack name followed by b1, b2, etc..., giving the band number within each stack (Example: stacknameb1, stacknameb2, stacknameb3, ...).
predFactor	String. A character vector of predictor short names of the predictors from predList that are factors (i.e categorical predictors). These must be a subset of the predictor names given in predList. Categorical predictors may have multiple categories.
response.name	String. The name of the response variable used to build the model. If response.name = NULL, a GUI interface prompts user to select a variable from the list of column names from training data file. response.name must be column name from the training/test data files.
response.type	String. Response type: "binary" or "continuous". binary response must be binary 0/1 variable with only 2 categories. All zeros will be treated as one category, and everything else will be treated as the second category.

seed	Integer. The number used to initialize randomization to build RF or SGB models. If you want to produce the same model later, use the same seed. If seed = NULL (the default), a new seed is created each run.
na.action	String. Model validation. Specifies the action to take if there are NA values in the prediction data or if there is a level or class of a categorical predictor variable in the validation test set or the production (mapping) data set, but not in the training data set. There are 2 options: (1) na.action = "na.omit" (the default) where any data point or pixel with any new levels for any of the factored predictors is returned as -9999 (the NODATA value); (2) na.action = "na.roughfix" where a missing categorical predictor for a data point or pixel is replaced with the most common category for that predictor, and a missing continuous predictor is replaced with the median for that predictor.
keep.data	Logical. RF and SGB models. Should a copy of the predictor data be included in the model object. Useful for if <code>model.interaction.plot</code> will be used later.
n.tree	Integer. RF models. The number of random forest trees for a RF model. The default is 500 trees.
m.try	Integer. RF models. Number of variables to try at each node of Random Forest trees. By default, will use the "tuneRF()" function to optimize m.try.
replace	Logical. RF models. Should sampling of cases be done with or without replacement?
strata	Factor or String. RF models. A (factor) variable that is used for stratified sampling. Can be in the form of either the name of the column in qdata or a factor or vector with one element for each row of qdata.
sampsize	Vector. RF models. Size(s) of sample to draw. For classification, if sampsize is a vector of the length the number of factor levels strata, then sampling is stratified by strata, and the elements of sampsize indicate the numbers to be drawn from each strata. If argument strata is not provided, and response.type = "binary" then sampling is stratified by presence/absence. If argument sampsize is not provided model.build() will use the default value from the randomForest package: if (replace) nrow(data) else ceiling(.632*nrow(data)).
n.trees	Integer. SGB models. The number of stochastic gradient boosting trees for an SGB model. If n.trees=NULL (the default) the model creation code will increase the number of trees 100 at a time until OOB error rate stops improving. The gbm function gbm.perf() will be used to select from the total calculated trees, the best number of trees for model predictions, with argument method="OOB". The gbm package warns that OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive.. If n.trees is given and train.fraction is less than 1, then the SGB model is built with the given number of trees, and the best number of trees is calculated with gbm method="test". In both of these cases best.iter is included in the returned model object. If n.trees is given and train.fraction = 1, then the model is built with the given number of trees, and best.iter is not calculated.
shrinkage	Numeric. SGB models. A shrinkage parameter applied to each tree in the expansion. Also known as the learning rate or step-size reduction.

<code>interaction.depth</code>	Integer. SGB models. The maximum depth of variable interactions. <code>interaction.depth = 1</code> implies an additive model, <code>interaction.depth = 2</code> implies a model with up to 2-way interactions, etc...
<code>bag.fraction</code>	Numeric. SGB models. <code>bag.fraction</code> must be a number between 0 and 1, giving the fraction of the training set observations randomly selected to propose the next tree in the expansion. This introduces randomnesses into the model fit. If <code>bag.fraction < 1</code> then running the same model twice will result in similar but different fits.
<code>train.fraction</code>	Numeric. SGB models. The first <code>train.fraction * nrows(data)</code> observations are used to fit the model and the remainder are used for computing out-of-sample estimates of the loss function.
<code>n.minobsinnode</code>	Integer. SGB models. Minimum number of observations in the trees terminal nodes. Note that this is the actual number of observations not the total weight.
<code>var.monotone</code>	String. SGB models. an optional vector, the same length as the number of predictors, indicating which variables have a monotone increasing (+1), decreasing (-1), or arbitrary (0) relationship with the outcome.

Details

This package provides a push button approach to complex model building and production mapping. It contains four functions: a simple function `get.test()` that can be used to radomly divide a training dataset into training and test/validation sets; and the workhorse functions `model.build()`, `model.diagnostics()`, and `model.mapmake()`.

These functions can be run in a traditional R command mode, where all arguments are specified in the function call. However they can also be used in a full push button mode, where you type in, for example, the simple command `model.build()`, and GUI pop up windows will ask questions about the type of model, the file locations of the data, etc...

When running the ModelMap package on non-Windows platforms, file names and folders need to be specified in the argument list, but other pushbutton selections are handled by the `select.list()` function, which is platform independent.

Random Forest is implemented through the `randomForest` package within R. Random Forest is more user friendly than Stochastic Gragient Boosting, as it has fewer parameters to be set by the user, and is less sensitive to tuning of these parameters. A Random Forest model consists of multiple trees that vote on predictions. For each tree a random subset of the training data is used to construct the tree, with the remaining data points used to construct out-of-bag (OOB) error estimates. At each node of the tree a random selection of predictors is chosen to determine the split. The number of predictors used to select the splits (argument `mtry`) is the primary user specified parameter that can affect model performance.

By default `mtry` will be automatically optimized using the `tuneRF()` function. Note that this is a stochastic process. If there is a chance that models may be combined later with the `randomForest` package `combine` function then for consistency it is important to provide the `mtry` argument rather than using the default optimization process.

Random Forest will not over fit data, therefore the only penalty of increasing the number of trees is computation time. Random Forest can compute variable importance, an advantage over some "black

box" modeling techniques if it is important to understand the ecological relationships underlying a model (Breiman, 2001).

Stochastic gradient boosting (Friedman 2001, 2002), is related to both boosting and bagging. Many small classification or regression trees are built sequentially from "pseudo"-residuals (the gradient of the loss function of the previous tree).

At each iteration, a tree is built from a random sub-sample of the dataset (selected without replacement) and an incremental improvement in the model. Using only a fraction of the training data increases both the computation speed and the prediction accuracy, while also helping to avoid overfitting the data. An advantage of stochastic gradient boosting is that it is not necessary to pre-select or transform predictor variables. It is also resistant to outliers, as the steepest gradient algorithm emphasizes points that are close to their correct classification. Stochastic gradient boosting is implemented through the `gbm` package within R.

One disadvantage of Stochastic Gradient Boosting, compared to Random Forest, is increased number of user specified parameters, and the SGB models tend to be more sensitive to these parameters. Model fitting parameter options include distribution, interaction depth, bagging fraction, shrinkage rate, and training fraction. These parameters can be set in the argument list when calling `model.map()`. Values for these parameters other than the defaults can not be set by point and click in the GUI pop up windows, and must be set in the argument list when calling `model.build()`. Friedman (2001, 2002) and Ridgeway (1999) provide guidelines on appropriate settings for model fitting options.

Also, unlike Random Forest models, in Stochastic Gradient Boosting, there is a penalty for using too many trees. The default behavior in `model.map()` is to increase the number of trees 100 at a time until the model stops improving, then call the `gbm` subfunction `gbm.perf(method="OOB")` to select the best number of iterations. Alternatively, the `model.build()` argument `ntrees` can be used to set some large number of trees to be calculated all at once and, again, the `gbm.perf(method="OOB")` function will be used to select the best number of trees. Note that the `gbm` package warns that "OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive." The `gbm` package offers two alternative techniques for calculating the best number of trees, but these are not yet implemented in the `ModelMap` package, as they require the use of a formula interface for model building.

Value

The function will return the model object. Additionally, it will write a text file to disk, in the folder specified by `folder`. This file lists the values of each argument as chosen from GUI prompts used for the function call.

Author(s)

Elizabeth Freeman and Tracey Frescino

References

- Breiman, L. (2001) Random Forests. *Machine Learning*, 45:5-32.
- Friedman, J.H. (2001). Greedy function approximation: a gradient boosting machine. *Ann. Stat.*, 29(5):1189-1232.
- Friedman, J.H. (2002). Stochastic gradient boosting. *Comput. Stat. Data An.*, 38(4):367-378.

Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. R News 2(3), 18–22.

Ridgeway, G., (1999). The state of boosting. Comp. Sci. Stat. 31:172-181

See Also

[get.test](#), [model.diagnostics](#), [model.mapmake](#)

Examples

```
#####
##### Run this set up code: #####
#####

# set seed:
seed=38

# Define training and test files:

qdata.trainfn = system.file("external", "helpexamples", "DATATRAIN.csv", package = "ModelMap")

# Define folder for all output:
folder=getwd()

#####
##### Pick one of the following sets of definitions: #####
#####

##### Continuous Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_Bio_TC"

#predictors:
predList=c("TCB", "TCG", "TCW")

#define which predictors are categorical:
predFactor=FALSE

# Response name and type:
response.name="BIO"
response.type="continuous"

##### binary Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_CONIFTYP_TC"
```

```

#predictors:
predList=c("TCB","TCG","TCW")

#define which predictors are categorical:
predFactor=FALSE

# Response name and type:
response.name="CONIFTYP"

# This variable is 1 if a conifer or mixed conifer type is present,
# otherwise 0.

response.type="binary"

##### Continuous Response, Categorical Predictors #####

# In this example, NLCD is a categorical predictor.
#
# You must decide what you want to happen if there are categories
# present in the data to be predicted (either the validation/test set
# or in the image file) that were not present in the original training data.
# Choices:
#   na.action = "na.omit"
#               Any validation datapoint or image pixel with a value for any
#               categorical predictor not found in the training data will be
#               returned as NA.
#   na.action = "na.roughfix"
#               Any validation datapoint or image pixel with a value for any
#               categorical predictor not found in the training data will have
#               the most common category for that predictor substituted,
#               and the a prediction will be made.

# You must also let R know which of the predictors are categorical, in other
# words, which ones R needs to treat as factors.
# This vector must be a subset of the predictors given in predList

#file name to store model:
MODELfn="RF_BIO_TCandNLCD"

#predictors:
predList=c("TCB","TCG","TCW","NLCD")

#define which predictors are categorical:
predFactor=c("NLCD")

# Response name and type:
response.name="BIO"
response.type="continuous"

#####

```

```
##### build model: #####
#####

### create model before batching (only run this code once ever!) ###

model.obj = model.build( model.type="RF",
                        qdata.trainfn=qdata.trainfn,
                        folder=folder,
                        MODELfn=MODELfn,
                        predList=predList,
                        predFactor=predFactor,
                        response.name=response.name,
                        response.type=response.type,
                        seed=seed
)

```

model.diagnostics

Model Predictions and Diagnostics

Description

Takes model object and makes predictions, runs model diagnostics, and creates graphs and tables of the results.

Usage

```
model.diagnostics(model.obj = NULL, qdata.trainfn = NULL, qdata.testfn = NULL, folder = NULL, MODELfn =
```

Arguments

model.obj	R model object. The model object to use for prediction. The model object must be of type RF or SGB. (Eventually planned to include "GAM".)
qdata.trainfn	String. The name (full path or base name with path specified by folder) of the training data file used for building the model (file should include columns for both response and predictor variables). The file must be a comma-delimited file *.csv with column headings. qdata.trainfn can also be an R dataframe. If predictions will be made (predict = TRUE or map=TRUE) the predictor column headers must match the names of the raster layer files, or a rastLUT must be provided to match predictor columns to the appropriate raster and band. If qdata.trainfn = NULL (the default), a GUI interface prompts user to browse to the training data file.
qdata.testfn	String. The name (full path or base name with path specified by folder) of the independent data set for testing (validating) the model's predictions. The file must be a comma-delimited file ".csv" with column headings and the column

headings must be the same as those in the training data file. `qdata.testfn` can also be an R dataframe. If `qdata.testfn = NULL` (default), a GUI interface asks user if there is a test set available, then prompts user to browse to the test data file. If no test set is desired (for example, cross-fold validation will be performed, or for RF models, Out-Of-Bag estimation, set `qdata.testfn = FALSE`. If no test set is given, and `qdata.testfn` is not set to `FALSE`, the GUI interface asks if a proportion of the data should be set aside as an independent test set. If this is desired, the user will be prompted to specify the proportion to set aside as test data, and two new data files will be generated in the out put folder. The new file names will be the original data file name with `"_train"` and `"_test"` appended to the end of the file names.

<code>folder</code>	String. The folder used for all output from predictions and/or maps. Do not add ending slash to path string. If <code>folder = NULL</code> (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify <code>folder = getwd()</code> .
<code>MODELfn</code>	String. The file name to use to save the generated model object. If <code>MODELfn = NULL</code> (the default), a default name is generated by pasting <code>model.type_response.type_response.name</code> . If the other output filenames are left unspecified, <code>MODELfn</code> will be used as the basic name to generate other output filenames. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by <code>folder</code> .
<code>response.name</code>	String. The name of the response variable used to build the model. The <code>response.name</code> must be column name from the training/test data files. If the <code>model.obj</code> was constructed in <code>ModelMap</code> with the <code>model.build()</code> function, then the <code>model.diagnostics()</code> can extract the <code>response.name</code> from the <code>model.obj</code> . If the model was constructed outside of <code>ModelMap</code> the you may need to specify the <code>response.name</code> . In particular, if a SGB model was constructed with the aid of Elith's code, it is necessary to specify the <code>response.name</code> argument, as all models constructed with this code are given a response name of <code>"y.data"</code> . If the <code>response.name</code> argument differs from the response name in the <code>model.obj</code> , the specified argument is given preference, and a warning generated.
<code>unique.rowname</code>	String. The name of the unique identifier used to identify each row in the training data. If <code>unique.rowname = NULL</code> , a GUI interface prompts user to select a variable from the list of column names from the training data file. If <code>unique.rowname = FALSE</code> , a variable is generated of numbers from 1 to <code>nrow(qdata)</code> to index each row.
<code>seed</code>	Integer. The number used to initialize randomization to build RF or SGB models. If you want to produce the same model later, use the same seed. If <code>seed = NULL</code> (the default), a new seed is created each run.
<code>prediction.type</code>	String. Prediction type. <code>"TEST"</code> , <code>"CV"</code> , <code>"OOB"</code> or <code>"TRAIN"</code> . If <code>predict = "TEST"</code> , validation predictions will be made on the test set provided by <code>qdata.testfn</code> . If <code>predict = "CV"</code> , cross validation will be used on the training data provided by <code>qdata.trainfn</code> . If <code>model.obj</code> is a Random Forest model and <code>predict = "OOB"</code> the Out-of-Bag predictions will be calculated on the training data. If <code>model.obj</code> is a Stochastic Gradient Boosting model and <code>predict = "TRAIN"</code> the predictions will be calculated on the training data, but these predictions

should be used with caution as this will lead to over optimistic estimates of model quality. A *.csv file of the unique id, observed, and predicted values is generated and put in the specified (or default) folder.

MODELpredfn	String. Model validation. A character string used to construct the output file names for the validation diagnostics, for example the prediction *.csv file, and the graphics *.jpg, *.pdf and *.ps files. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by folder. If MODELpredfn = NULL (the default), a default name is created by pasting modelfn and "_pred".												
na.action	String. Model validation. Specifies the action to take if there are NA values in the prediction data or if there is a level or class of a categorical predictor variable in the validation test set or the production (mapping) data set, but not in the training data set. There are 2 options: (1) na.action = "na.omit" (the default) where any data point or pixel with any new levels for any of the factored predictors is returned as -9999 (the NODATA value); (2) na.action = "na.roughfix" where a missing categorical predictor for a data point or pixel is replaced with the most common category for that predictor, and a missing continuous predictor is replaced with the median for that predictor.												
v.fold	Integer (or logical FALSE). Model validation. The number of cross validation folds to use when making validation predictions on the training data. Only used if prediction.type = "CV".												
device.type	String or vector of strings. Model validation. One or more device types for graphical output from model validation diagnostics. Current choices: <table> <tr> <td>"default"</td> <td>default graphics device</td> </tr> <tr> <td>"jpeg"</td> <td>*.jpg files</td> </tr> <tr> <td>"none"</td> <td>no graphics produced</td> </tr> <tr> <td>"pdf"</td> <td>*.pdf files</td> </tr> <tr> <td>"postscript"</td> <td>*.ps files</td> </tr> <tr> <td>"win.metafile"</td> <td>*.emf files</td> </tr> </table>	"default"	default graphics device	"jpeg"	*.jpg files	"none"	no graphics produced	"pdf"	*.pdf files	"postscript"	*.ps files	"win.metafile"	*.emf files
"default"	default graphics device												
"jpeg"	*.jpg files												
"none"	no graphics produced												
"pdf"	*.pdf files												
"postscript"	*.ps files												
"win.metafile"	*.emf files												
DIAGNOSTICfn	String. Model validation. Name used as base to create names for output files from model validation diagnostics. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by folder. Defaults to DIAGNOSTICfn = MODELfn followed by the appropriate suffixes (i.e. ".csv", ".jpg", etc...).												
jpeg.res	Integer. Model validation. Pixels per inch for jpeg plots. The default is 72dpi, good for on screen viewing. For printing, suggested setting is 300dpi.												
device.width	Integer. Model validation. The device width for diagnostic plots in inches.												
device.height	Integer. Model validation. The device height for diagnostic plots in inches.												
cex	Integer. Model validation. The cex for diagnostic plots.												
req.sens	Numeric. Model validation. The required sensitivity for threshold optimization for binary response model evaluation.												
req.spec	Numeric. Model validation. The required specificity for threshold optimization for binary response model evaluation.												

FPC	Numeric. Model validation. The False Positive Cost for threshold optimization for binary response model evaluation.
FNC	Numeric. Model validation. The False Negative Cost for threshold optimization for binary response model evaluation.
n.trees	Integer. SGB models. The number of stochastic gradient boosting trees for an SGB model. If n.trees=NULL (the default) the model creation code will increase the number of trees 100 at a time until OOB error rate stops improving. The gbm function gbm.perf() will be used to select from the total calculated trees, the best number of trees for model predictions, with argument method="OOB". The gbm package warns that OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive.

Details

model.diagnostics() takes model object and makes predictions, runs model diagnostics, and creates graphs and tables of the results.

model.diagnostics() can be run in a traditional R command mode, where all arguments are specified in the function call. However it can also be used in a full push button mode, where you type in the simple command model.map(), and GUI pop up windows will ask questions about the type of model, the file locations of the data, etc...

When running model.map() on non-Windows platforms, file names and folders need to be specified in the argument list, but other pushbutton selections are handled by the select.list() function, which is platform independent.

Diagnostic predictions are made by one of four methods, and a text file is generated consisting of three columns: Observation ID, observed values and predicted values. If prediction.type = "CV") a fourth column indicates which cross-fold each observation fell into.

A variable importance graph is made.

If response.type = "binary", a summary graph is made using the PresenceAbsence package and a *.csv spreadsheets are created of optimized thresholds by several methods with their associated error statistics, and predicted prevalence.

If response.type = "continuous" a scatterplot of observed vs. predicted is created with a simple linear regression line. The graph is labeled with slope and intercept of this line as well as Pearson's and Spearman's correlation coefficients.

Value

The function will return a dataframe of the row ID, and the Observed and predicted values. If prediction.type = "CV" the dataframe also includes a column indicating which cross-validation fold each datapoint was in.

Author(s)

Elizabeth Freeman and Tracey Frescino

References

- Breiman, L. (2001) Random Forests. *Machine Learning*, 45:5-32.
- Elith, J., Leathwick, J. R. and Hastie, T. (2008). A working guide to boosted regression trees. *Journal of Animal Ecology*. 77:802-813.
- Friedman, J.H. (2001). Greedy function approximation: a gradient boosting machine. *Ann. Stat.*, 29(5):1189-1232.
- Friedman, J.H. (2002). Stochastic gradient boosting. *Comput. Stat. Data An.*, 38(4):367-378.
- Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. *R News* 2(3), 18–22.
- Ridgeway, G., (1999). The state of boosting. *Comp. Sci. Stat.* 31:172-181

See Also

[get.test](#), [model.build](#), [model.mapmake](#)

Examples

```
#####
##### Run this set up code: #####
#####

# set seed:
seed=38

# Define training and test files:

qdata.trainfn = system.file("external", "helpexamples", "DATATRAIN.csv", package = "ModelMap")
qdata.testfn = system.file("external", "helpexamples", "DATATEST.csv", package = "ModelMap")

# Define folder for all output:
folder=getwd()

#####
##### Pick one of the following sets of definitions: #####
#####

##### Continuous Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_Bio_TC"

#predictors:
predList=c("TCB", "TCG", "TCW")

#define which predictors are categorical:
predFactor=FALSE
```

```

# Response name and type:
response.name="BIO"
response.type="continuous"

##### binary Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_CONIFTYP_TC"

#predictors:
predList=c("TCB","TCG","TCW")

#define which predictors are categorical:
predFactor=FALSE

# Response name and type:
response.name="CONIFTYP"

# This variable is 1 if a conifer or mixed conifer type is present,
# otherwise 0.

response.type="binary"

##### Continuous Response, Categorical Predictors #####

# In this example, NLCD is a categorical predictor.
#
# You must decide what you want to happen if there are categories
# present in the data to be predicted (either the validation/test set
# or in the image file) that were not present in the original training data.
# Choices:
#     na.action = "na.omit"
#                 Any validation datapoint or image pixel with a value for any
#                 categorical predictor not found in the training data will be
#                 returned as NA.
#     na.action = "na.roughfix"
#                 Any validation datapoint or image pixel with a value for any
#                 categorical predictor not found in the training data will have
#                 the most common category for that predictor substituted,
#                 and the a prediction will be made.

# You must also let R know which of the predictors are categorical, in other
# words, which ones R needs to treat as factors.
# This vector must be a subset of the predictors given in predList

#file name to store model:
MODELfn="RF_BIO_TcandNLCD"

#predictors:
predList=c("TCB","TCG","TCW","NLCD")

```

```

#define which predictors are categorical:
predFactor=c("NLCD")

# Response name and type:
response.name="BIO"
response.type="continuous"

#####
##### build model: #####
#####

### create model ###

model.obj = model.build( model.type="RF",
                        qdata.trainfn=qdata.trainfn,
                        folder=folder,
                        MODELfn=MODELfn,
                        predList=predList,
                        predFactor=predFactor,
                        response.name=response.name,
                        response.type=response.type,
                        seed=seed
)

#####
##### Then Run this code make validation predictions and diagnostics: #####
#####

### Define identifier for individual training and test data points ###

unique.rowname="ID"

### for Out-of-Bag predictions ###

MODELpredfn<-paste(MODELfn,"_OOB",sep="")
PRED.OOB<-model.diagnostics( model.obj=model.obj,
qdata.trainfn=qdata.trainfn,
                        folder=folder,
                        unique.rowname=unique.rowname,
                        # Model Validation Arguments
                        prediction.type="OOB",
                        MODELpredfn=MODELpredfn,
                        device.type=c("default","jpeg","pdf"),
                        na.action="na.roughfix"
)
PRED.OOB

### for Cross-Validation predictions ###

```

```

MODELpredfn<-paste(MODELfn,"_CV",sep="")
PRED.CV<-model.diagnostics( model.obj=model.obj,
  qdata.trainfn=qdata.trainfn,
  folder=folder,
  unique.rowname=unique.rowname,
  seed=seed,
  # Model Validation Arguments
  prediction.type="CV",
  MODELpredfn=MODELpredfn,
  device.type=c("default","jpeg","pdf"),
  v.fold=10,
  na.action="na.roughfix"
)
PRED.CV

### for Independant Test Set predictions ###

MODELpredfn<-paste(MODELfn,"_TEST",sep="")
PRED.TEST<-model.diagnostics( model.obj=model.obj,
  qdata.testfn=qdata.testfn,
  folder=folder,
  unique.rowname=unique.rowname,
  # Model Validation Arguments
  prediction.type="TEST",
  MODELpredfn=MODELpredfn,
  device.type=c("default","jpeg","pdf"),
  na.action="na.roughfix"
)
PRED.TEST

```

```
model.interaction.plot
```

plot of two-way model interactions

Description

Image or Perspective plot of two-way model interactions. Ranges of two specified predictor variables are plotted on X and Y axis, and fitted model values are plotted on the Z axis. The remaining predictor variable are fixed at their mean (for continuous predictors) or their most common value (for catagorical predictors).

Usage

```
model.interaction.plot(model.obj = NULL, x = NULL, y = NULL, qdata.trainfn = NULL, folder = NULL, MODEL f
```

Arguments

`model.obj` R model object. A RF or SGB model object produced by `model.build`.

x	String or Integer. Name of predictor variable to be plotted on the x axis. Alternatively, can be a number indicating a variable name from predList.
y	String or Integer. Name of predictor variable to be plotted on the y axis. Alternatively, can be a number indicating a variable name from predList.
qdata.trainfn	String. The name (full path or base name with path specified by folder) of the training data file used for building the model (file should include columns for both response and predictor variables). The file must be a comma-delimited file *.csv with column headings. qdata.trainfn can also be an R dataframe. If predictions will be made (predict = TRUE or map=TRUE) the predictor column headers must match the names of the raster layer files, or a rastLUT must be provided to match predictor columns to the appropriate raster and band. If qdata.trainfn = NULL (the default), a GUI interface prompts user to browse to the training data file.
folder	String. The folder used for all output. Do not add ending slash to path string. If folder = NULL (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify folder = getwd().
MODELfn	String. The file name to use to save the generated model object. If MODELfn = NULL (the default), a default name is generated by pasting model.type_response.type_response.name. If the other output filenames are left unspecified, MODELfn will be used as the basic name to generate other output filenames. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by folder.
pred.means	Vector. Allows specification of values for other predictor variables. If Null, other predictors are set to their mean value (for continuous predictors) or their most common value (for factored predictors).
xlab	String. Allows manual specification of the x label.
ylab	String. Allows manual specification of the y label.
x.range	Vector. Manual range specification for the x axis.
y.range	Vector. Manual range specification for the y axis.
z.range	Vector. Manual range specification for the z axis.
ticktype	Character: "simple" draws just an arrow parallel to the axis to indicate direction of increase; "detailed" (default) draws normal ticks as per 2D plots. If X or y is factored, ticks will be drawn on both axes.
theta	Numeric. Angles defining the viewing direction. theta gives the azimuthal direction.
phi	Numeric. Angles defining the viewing direction. phi gives the colatitude.
smooth	String. controls smoothing of the predicted surface. Options are "none" (default), "model" which uses a glm model to smoth the surface, and "average" which applies a 3x3 smoothing average. Note: smoothing is not appropriate if X or y is factored.
plot.type	Character. "persp" gives a 3-D perspective plot. "image" gives an image plot.
device.type	String or vector of strings. Model validation. One or more device types for graphical output from model validation diagnostics. Current choices:

```

"default"      default graphics device
"jpeg"        *.jpg files
"none"        no graphics produced
"pdf"         *.pdf files
"postscript"  *.ps files
"win.metafile" *.emf files

```

```

jpeg.res      Integer. Pixels per inch for jpeg plots. The default is 72dpi, good for on screen
              viewing. For printing, suggested setting is 300dpi.
device.width  Integer. The device width for diagnostic plots in inches.
device.height Integer. The device height for diagnostic plots in inches.
cex           Integer. The cex for diagnostic plots.
col           Vector. Color table to use for image plots ( see help file on image for details).
...          additional graphical parameters (see par).

```

Details

This function provides a diagnostic plot useful in visualizing two-way interactions between predictor variables. Two of the predictor variables from the model are used to produce a grid of possible combinations of predictor values over the range of both variables. The remaining predictor variables from the model are fixed at either their means (for continuous predictors) or their most common value (for categorical predictors). Model predictions are generated over this grid and plotted as the z axis.

This function works with both continuous and categorical predictors, though the perspective plot should be interpreted with care for categorical predictors. In particular, the smooth option is not appropriate if either of the two selected predictor variables is categorical.

Author(s)

Elizabeth Freeman

References

This function is adapted from `gbm.perspec` version 2.9 April 2007, J Leathwick/J Elith. See appendix S3 from:

Elith, J., Leathwick, J. R. and Hastie, T. (2008). A working guide to boosted regression trees. *Journal of Animal Ecology*. 77:802-813.

Examples

```

#####
##### Run this set up code: #####
#####

# set seed:
seed=38

```

```

# Define training and test files:

qdata.trainfn = system.file("external", "helpexamples", "DATATRAIN.csv", package = "ModelMap")
qdata.testfn = system.file("external", "helpexamples", "DATATEST.csv", package = "ModelMap")

# Define folder for all output:
folder=getwd()

##### Continuous Response, Categorical Predictors #####

# In this example, NLCD is a categorical predictor.
#
# You must decide what you want to happen if there are categories
# present in the data to be predicted (either the validation/test set
# or in the image file) that were not present in the original training data.
# Choices:
#     na.action = "na.omit"
#         Any validation datapoint or image pixel with a value for any
#         categorical predictor not found in the training data will be
#         returned as NA.
#     na.action = "na.roughfix"
#         Any validation datapoint or image pixel with a value for any
#         categorical predictor not found in the training data will have
#         the most common category for that predictor substituted,
#         and the a prediction will be made.

# You must also let R know which of the predictors are categorical, in other
# words, which ones R needs to treat as factors.
# This vector must be a subset of the predictors given in predList

#file name to store model:
MODELfn="RF_BIO_TCandNLCD"

#predictors:
predList=c("TCB", "TCG", "TCW", "NLCD")

#define which predictors are categorical:
predFactor=c("NLCD")

# Response name and type:
response.name="BIO"
response.type="continuous"

#####
##### build model: #####
#####

### create model ###

model.obj = model.build( model.type="RF",
                        qdata.trainfn=qdata.trainfn,

```

```

        folder=folder,
        MODELfn=MODELfn,
        predList=predList,
        predFactor=predFactor,
        response.name=response.name,
        response.type=response.type,
        seed=seed
    )

#####
##### make interaction plots: #####
#####

#####
### Perspective Plots ###
#####

### specify first and third predictors in 'predList (both continuous) ###

model.interaction.plot( model.obj,
x=1,y=3,
main=response.name,
plot.type="persp",
device.type="default")

### specify first and forth predictors in 'predList (one continuous one factored) ###

model.interaction.plot( model.obj,
x=1, y=4,
main=response.name,
plot.type="persp",
device.type="default")

### same as previous example, but specifying predictors by name ##

model.interaction.plot( model.obj,
x="TCB", y="NLCD",
main=response.name,
plot.type="persp",
device.type="default")

#####
### Image Plots ###
#####

### same as previous example, but image plot ###

l <- seq(100,0,length.out=101)
c <- seq(0,100,length.out=101)
col.ramp <- hcl(h = 120, c = c, l = l)

```

```
model.interaction.plot( model.obj,
x="TCB", y="NLCD",
main=response.name,
plot.type="image",
device.type="default",
col = col.ramp)

#####
### 3-way Interaction ###
#####

### use 'pred.means' argument to fix values of additional predictors ###

### factored 3rd predictor ###

nlcd<-levels(model.obj$predictor.data$NLCD)

for(i in nlcd){
pred.means=list(NLCD=i)

model.interaction.plot( model.obj,
x="TCG", y="TCW",
main=paste("NLCD =", i),
pred.means=pred.means,
z.range=c(0,110),
theta=290,
plot.type="persp",
device.type="default")
}

### continuous 3rd predictor ###

tcb<-seq( min(model.obj$predictor.data$TCB),
max(model.obj$predictor.data$TCB),
length=5)

tcb<-signif(tcb,2)

for(i in tcb){
pred.means=list(TCB=i)

model.interaction.plot( model.obj,
x="TCG", y="TCW",
main=paste("TCB =", i),
pred.means=pred.means,
z.range=c(0,120),
```

```

theta=290,
plot.type="persp",
device.type="default")
}

### 4-way Interesting combos ###

tcb=c(1300,2900,3400)
nlcd=c(11,90,95)

for(i in 1:3){
pred.means=list(TCB=tcb[i],NLCD=nlcd[i])

model.interaction.plot( model.obj,
x="TCG", y="TCW",
main=paste("TCB =",tcb[i],"          NLCD =",nlcd[i]),
pred.means=pred.means,
z.range=c(0,120),
theta=290,
plot.type="persp",
device.type="default")
}

```

model.mapmake

Map Making

Description

Applies models to either ERDAS Imagine image (.img) files or ESRI Grids of predictors to create detailed prediction surfaces. It will handle large predictor files for map making, by reading in the .img files in chunks, and output to the .txt file the prediction for each data chunk, before reading the next chunk of data.

Usage

```
model.mapmake(model.obj= NULL, folder = NULL, MODELfn = NULL, rastLUTfn = NULL, na.action = "na.omit", m
```

Arguments

`model.obj` R model object. The model object to use for prediction, if the model has been previously created. The model object must be of type RF or SGB. (Eventually planned to include "GAM".) If NULL (the default), a model is generated of type specified by the argument `model.type`.

folder	String. The folder used for all output from predictions and/or maps. Do not add ending slash to path string. If folder = NULL (default), a GUI interface prompts user to browse to a folder. To use the working directory, specify folder = getwd().
MODELfn	String. The file name to use to save the generated model object. If MODELfn = NULL (the default), a default name is generated by pasting model.type_response.type_response.name. If the other output filenames are left unspecified, MODELfn will be used as the basic name to generate other output filenames. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by folder.
rastLUTfn	String. The file name (full path or base name with path specified by folder) of a .csv file for a rastLUT. Alternatively, a dataframe containing the same information. The rastLUT must include 3 columns: (1) the full path and name of the raster file; (2) the shortname of each predictor / raster layer (band); (3) the layer (band) number. The shortname (column 2) must match the names predList, the predictor column names in training/test data set (qdata.trainfn and qdata.testfn, and the predictor names in model.obj. Example of comma-delimited file: <pre>C:/button_test/tc99_2727subset.img, tc99_2727subsetb1, 1 C:/button_test/tc99_2727subset.img, tc99_2727subsetb2, 2 C:/button_test/tc99_2727subset.img, tc99_2727subsetb3, 3</pre>
na.action	String. Model validation. Specifies the action to take if there are NA values in the prediction data or if there is a level or class of a categorical predictor variable in the validation test set or the production (mapping) data set, but not in the training data set. There are 2 options: (1) na.action = "na.omit" (the default) where any data point or pixel with any new levels for any of the factored predictors is returned as -9999 (the NODATA value); (2) na.action = "na.roughfix" where a missing categorical predictor for a data point or pixel is replaced with the most common category for that predictor, and a missing continuous predictor is replaced with the median for that predictor.
numrows	Integer. Map Production. The number of rows to be predicted at a time.
map.sd	Logical. Map Production. If map.sd = TRUE, maps of mean, standard deviation, and coefficient of variation of the predictions from all the trees are generated for each pixel. If map.sd = FALSE (the default), only the predicted probability map will be built. This option is only available if the model.type = "RF" the response.type = "continuous". Note: This option requires much more available memory. If you get the error "...cannot allocate vector of size...", you must reduce the value of numrow. The names of the additional maps default to: <pre>folder/model.type_response.type_response.name_mean.txt folder/model.type_response.type_response.name_stdev.txt folder/model.type_response.type_response.name_coefv.txt</pre>
asciifn	String. Map Production. Filename of output file for map production. The file-

	name can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by <code>folder</code> . If <code>asciifn = NULL</code> (the default), a name is created by pasting <code>modelfn</code> and <code>"_map.txt"</code> .
<code>asciifn.mean</code>	String. Map Production. Used if <code>map.sd = TRUE</code> and <code>response.type = "continuous"</code> . Filename of output file for mean of trees. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by <code>folder</code> . If <code>asciifn.mean = NULL</code> (the default), a name is created by pasting <code>modelfn</code> and <code>"_map_mean.txt"</code> .
<code>asciifn.stdev</code>	String. Map Production. Used if <code>map.sd = TRUE</code> and <code>response.type = "continuous"</code> . Filename of output file for standard deviation of trees. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by <code>folder</code> . If <code>asciifn.stdev = NULL</code> (the default), a name is created by pasting <code>modelfn</code> and <code>"_map_stdev.txt"</code> .
<code>asciifn.coefv</code>	String. Map Production. Used if <code>map.sd = TRUE</code> and <code>response.type = "continuous"</code> . Filename of output file for coefficient of variation of trees. The filename can be the full path, or it can be the simple basename, in which case the output will be to the folder specified by <code>folder</code> . If <code>asciifn.coefv = NULL</code> (the default), a name is created by pasting <code>modelfn</code> and <code>"_map_coefv.txt"</code> .
<code>n.trees</code>	Integer. SGB models. The number of stochastic gradient boosting trees for an SGB model. If <code>n.trees=NULL</code> (the default) the model creation code will increase the number of trees 100 at a time until OOB error rate stops improving. The <code>gbm</code> function <code>gbm.perf()</code> will be used to select from the total calculated trees, the best number of trees for model predictions, with argument <code>method="OOB"</code> . The <code>gbm</code> package warns that OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive.

Details

`model.mapmake()` can be run in a traditional R command mode, where all arguments are specified in the function call. However it can also be used in a full push button mode, where you type in the simple command `model.mapmake()`, and GUI pop up windows will ask questions about the type of model, the file locations of the data, etc...

When running `model.mapmake()` on non-Windows platforms, file names and folders need to be specified in the argument list, but other pushbutton selections are handled by the `select.list()` function, which is platform independent.

For map making, the package `rgdal` is used to read `.img` files. The data for production mapping should be in the form of pixel-based raster layers representing the predictors in the model. If there is more than one predictor or raster layer, the layers must all have the same number of columns and rows. The layers must also have the same extent, projection, and pixel size, for effective model development and accuracy. The layers must also be in either ESRI Grid or ERDAS Imagine image (single or multi-band) raster data formats, having continuous or categorical data values. The R package `rgdal` is used to read spatial rasters into R.

When creating maps of non-rectangular study regions there may be large portions of the rectangle where you have no predictors, and are uninterested in making predictions. The suggested value for the pixels outside the study area is `-9999`. These pixels will be ignored in the predictions, thus saving computing time, and will be exported as `-9999`. Any value other than `-9999` will be treated as a

legal data value and a prediction will be generated for each pixel. Note: in Imagine image files, if the specified NODATA is set as -9999, any -9999 pixels will be read into R as NA, and if `na.action = "na.roughfix"`, predictions will be attempted for these pixels. This will cause the computation time to increase, and these predictions will need to be masked out when the final map is imported back into a GIS system.

The function `model.mapmake()` outputs an ASCII grid file of map information suitable to be imported into a GIS. Small maps can also be imported back into R using the function `read.asciigrid()` from the `sp` package.

Value

The function does not return a value, instead it writes text files of map information (suitable for importing into a GIS) to the specified folder.

Author(s)

Elizabeth Freeman and Tracey Frescino

References

- Breiman, L. (2001) Random Forests. *Machine Learning*, 45:5-32.
- Friedman, J.H. (2001). Greedy function approximation: a gradient boosting machine. *Ann. Stat.*, 29(5):1189-1232.
- Friedman, J.H. (2002). Stochastic gradient boosting. *Comput. Stat. Data An.*, 38(4):367-378.
- Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. *R News* 2(3), 18-22.
- Ridgeway, G., (1999). The state of boosting. *Comp. Sci. Stat.* 31:172-181

See Also

[get.test](#), [model.build](#), [model.diagnostics](#)

Examples

```
#####
##### Run this set up code: #####
#####

# set seed:
seed=38

# Define training and test files:

qdata.trainfn = system.file("external", "helpexamples", "DATATRIN.csv", package = "ModelMap")

# Define folder for all output:
folder=getwd()
```

```
#####
##### Pick one of the following sets of definitions: #####
#####

##### Continuous Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_Bio_TC"

#predictors:
predList=c("TCB","TCG","TCW")

#define which predictors are categorical:
predFactor=FALSE

# Response name and type:
response.name="BIO"
response.type="continuous"

##### binary Response, Continuous Predictors #####

#file name to store model:
MODELfn="RF_CONIFTYP_TC"

#predictors:
predList=c("TCB","TCG","TCW")

#define which predictors are categorical:
predFactor=FALSE

# Response name and type:
response.name="CONIFTYP"

# This variable is 1 if a conifer or mixed conifer type is present,
# otherwise 0.

response.type="binary"

##### Continuous Response, Categorical Predictors #####

# In this example, NLCD is a categorical predictor.
#
# You must decide what you want to happen if there are categories
# present in the data to be predicted (either the validation/test set
# or in the image file) that were not present in the original training data.
# Choices:
#     na.action = "na.omit"
#     Any validation datapoint or image pixel with a value for any
#     categorical predictor not found in the training data will be
```

```

#           returned as NA.
#   na.action = "na.roughfix"
#           Any validation datapoint or image pixel with a value for any
#           categorical predictor not found in the training data will have
#           the most common category for that predictor substituted,
#           and the a prediction will be made.

# You must also let R know which of the predictors are categorical, in other
# words, which ones R needs to treat as factors.
# This vector must be a subset of the predictors given in predList

#file name to store model:
MODELfn="RF_BIO_TCandNLCD"

#predictors:
predList=c("TCB","TCG","TCW","NLCD")

#define which predictors are categorical:
predFactor=c("NLCD")

# Response name and type:
response.name="BIO"
response.type="continuous"

#####
##### build model: #####
#####

### create model ###

model.obj = model.build( model.type="RF",
                        qdata.trainfn=qdata.trainfn,
                        folder=folder,
                        MODELfn=MODELfn,
                        predList=predList,
                        predFactor=predFactor,
                        response.name=response.name,
                        response.type=response.type,
                        seed=seed
)

#####
##### Then Run this code to predict map pixels #####
#####

# A single model was built from the training data,
# but it will be applied to two sets of image data, one from 2001 and one from 2004

```

```
#####

### Create a list of the filenames (including paths) for the rast Look up Tables ###

rastLUTfn.2001 <- paste(system.file(package="ModelMap"),"/external/helpexamples/LUT_2001.csv",sep="")
rastLUTfn.2004 <- paste(system.file(package="ModelMap"),"/external/helpexamples/LUT_2004.csv",sep="")

### Load rast LUT tables, and add path to the filenames in column 1 ###

rastLUT.2001 <- read.table(rastLUTfn.2001,header=FALSE,sep=",",stringsAsFactors=FALSE)
rastLUT.2004 <- read.table(rastLUTfn.2004,header=FALSE,sep=",",stringsAsFactors=FALSE)

rastLUT.2001[,1] <- paste(system.file(package="ModelMap"),"external/helpexamples",rastLUT.2001[,1],sep="/")
rastLUT.2004[,1] <- paste(system.file(package="ModelMap"),"external/helpexamples",rastLUT.2004[,1],sep="/")

### Define filenames for map output ###

asciifn.2001 <- "RF_BIO_TCandNLCD_01.txt"
asciifn.2004 <- "RF_BIO_TCandNLCD_04.txt"

asciifn.2001 <- paste(folder,asciifn.2001,sep="/")
asciifn.2004 <- paste(folder,asciifn.2004,sep="/")

### Define Number of rows of raster to read in at one time ###
# if crashes with warning: "unable to assign..." lower this number

numrows=500

### Create ascii text files of predicted map data ###

model.mapmake( model.obj=model.obj,
               folder=folder,
               rastLUTfn=rastLUT.2001,
               # Model Validation Arguments
               na.action="na.roughfix",
               # Mapping arguments
               numrows = numrows,
               asciifn=asciifn.2001
               )

model.mapmake( model.obj=model.obj,
               folder=folder,
               rastLUTfn=rastLUT.2004,
               # Model Validation Arguments
               na.action="na.roughfix",
               # Mapping arguments
               numrows = numrows,
```

```

        asciifn=asciifn.2004
    )

#####
##### run this code to create maps in R (For small maps only!)#####
#####

### Define Color Ramp ###

l <- seq(100,0,length.out=101)
c <- seq(0,100,length.out=101)
col.ramp <- hcl(h = 120, c = c, l = l)

### read in map data ###

mapgrid.2001 <- read.asciigrid(asciifn.2001,as.image=TRUE)
mapgrid.2004 <- read.asciigrid(asciifn.2004,as.image=TRUE)

### create map ###

dev.new(width = 8, height = 4)
opar <- par(mfrow=c(1,2),mar=c(3,3,2,1),oma=c(0,0,3,4),xpd=NA)

zlim <- c(0,max(mapgrid.2001$z,mapgrid.2004$z,na.rm=TRUE))
legend.label<-rev(pretty(zlim,n=5))
legend.colors<-col.ramp[trunc((legend.label/max(legend.label))*100)+1]

image(mapgrid.2001, col = col.ramp,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext("2001 Imagery",side=3,line=1,cex=1.2)

image(mapgrid.2004, col = col.ramp,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext("2004 Imagery",side=3,line=1,cex=1.2)

legend( x=max(mapgrid.2004$x),y=max(mapgrid.2004$y),
legend=legend.label,
fill=legend.colors,
bty="n",
cex=1.2
)
mtext("Predictions",side=3,line=1,cex=1.5,outer=TRUE)
par(opar)

#####
##### Run this code to map predictor data in R (For small maps only!) #####
#####

### Define Color Ramps ###

l <- seq(100,0,length.out=101)
c <- seq(0,100,length.out=101)

```

```

col.ramp.1 <- hcl(h = 15, c = c, l = 1)
col.ramp.2 <- hcl(h = 70, c = c, l = 1)
col.ramp.3 <- hcl(h = 150, c = c, l = 1)

dev.new(width = 9, height = 6)
opar <- par(mfcol=c(2,3),mar=c(3,3,2,1),oma=c(0,0,3,4),xpd=NA)

#band 1
predgrid.2001=readGDAL(rastLUT.2001[1,1],band=rastLUT.2001[1,3])
predgrid.2001=as.image.SpatialGridDataFrame(predgrid.2001)
predgrid.2004=readGDAL(rastLUT.2004[1,1],band=rastLUT.2004[1,3])
predgrid.2004=as.image.SpatialGridDataFrame(predgrid.2004)

zlim <- range(predgrid.2001$z,predgrid.2004$z)

image(predgrid.2001, col = col.ramp.1,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext(rastLUT.2001[1,2],side=3,cex=1.5)
mtext("2001 Imagery",side=2,cex=1.5,line=1)

image(predgrid.2004, col = col.ramp.1,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext("2004 Imagery",side=2,cex=1.5,line=1)

#band 2
predgrid.2001=readGDAL(rastLUT.2001[2,1],band=rastLUT.2001[2,3])
predgrid.2001=as.image.SpatialGridDataFrame(predgrid.2001)
predgrid.2004=readGDAL(rastLUT.2004[2,1],band=rastLUT.2004[2,3])
predgrid.2004=as.image.SpatialGridDataFrame(predgrid.2004)

zlim <- range(predgrid.2001$z,predgrid.2004$z)

image(predgrid.2001, col = col.ramp.2,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext(rastLUT.2001[2,2],side=3,cex=1.5)

image(predgrid.2004, col = col.ramp.2,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")

#band 3
predgrid.2001=readGDAL(rastLUT.2001[3,1],band=rastLUT.2001[3,3])
predgrid.2001=as.image.SpatialGridDataFrame(predgrid.2001)
predgrid.2004=readGDAL(rastLUT.2004[3,1],band=rastLUT.2004[3,3])
predgrid.2004=as.image.SpatialGridDataFrame(predgrid.2004)

zlim <- range(predgrid.2001$z,predgrid.2004$z)

image(predgrid.2001, col = col.ramp.3,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")
mtext(rastLUT.2001[3,2],side=3,cex=1.5)

image(predgrid.2004, col = col.ramp.3,zlim=zlim,asp=1,bty="n",xaxt="n",yaxt="n")

mtext("Predictor Imagery",side=3,line=1,cex=1.5,outer=TRUE)

```

par(opar)

Index

*Topic **models**

- build.rastLUT, 4
- get.test, 5
- model.build, 6
- model.diagnostics, 13
- model.interaction.plot, 20
- model.mapmake, 26

*Topic **package**

- ModelMap-package, 2

build.rastLUT, 2, 4

get.test, 2, 5, 11, 17, 29

model.build, 2, 6, 17, 29

model.diagnostics, 2, 11, 13, 29

model.interaction.plot, 2, 8, 20

model.mapmake, 2, 11, 17, 26

ModelMap (ModelMap-package), 2

ModelMap-package, 2

par, 22