

Package ‘ModelMap’

June 14, 2009

Type Package

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

Version 1.1.13

Date 2009-06-11

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

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 2009-06-14 18:50:29

R topics documented:

ModelMap-package	2
get.test	3
model.map	5

Index	18
--------------	-----------

ModelMap-package *Modeling and Map production using Random Forest and Stochastic Gradient Boosting*

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: 1.1.13

Date: 2009-06-11

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 two functions: a simple function `get.test()` that can be used to randomly divide a training dataset into training and test/validation sets; and the workhorse, "do every thing" function nick named "The Button", and called with `model.map()`.

`model.map()` 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 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 `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 (Brieman, 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.

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.

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

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 =
```

Arguments

<code>proportion.test</code>	Number. The proportion of the training data that will be randomly extracted for use as a test set. Value between 0 and 1.
<code>qdatafn</code>	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 (<code>predict = TRUE</code>). If <code>NULL</code> (the default), a GUI interface prompts user to browse to the data file.
<code>seed</code>	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 <code>seed = NULL</code> (the default), a new one is created each time.
<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>qdata.trainfn</code>	String. The name of the file output of training data. By default, <code>_train</code> appended after <code>qdatafn</code> .
<code>qdata.testfn</code>	String. The name of the file output of test data. By default, <code>_test</code> appended after <code>qdatafn</code> .

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

See Also

[model.map](#)

Examples

```
qdatafn=paste(system.file(package="ModelMap"), "/external/DATATRAIN.csv", sep="")
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.map

Model Building and Map making

Description

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.

Usage

```
model.map(model.obj = NULL, model.type = NULL, qdata.trainfn = NULL, qdata.testfn =
```

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`.

`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.

`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.

`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`. 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`.

`rastLUT` Dataframe. A data frame of raster information used to make a map. This data frame can be an R object or read in from a comma-delimited file using the example code below. The `rastLUT` must be given if a map is desired (`map = TRUE`) or if a model is being generated and `predList = NULL`. 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 if `model.obj` is already built, the predictor names in this model object.

Set `rastLUT = FALSE` if no raster LUT is available and you do not want to be propted by a gui interface. In this case, if a map is to be produced, column headers in `qdata` MUST be of the following format: `[name of rast]b[layer number]`. See column 2 of the example below.

Example of comma-delimited file:

```
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
```

Example code for reading in file:

```
read.table(file="C:/predict_button/testdata/rastLUT_example.csv",
sep="," ,header=FALSE, check.names=F, stringsAsFactors=F)
```

`rastLUTfn` String. The file name (full path or base name with path specified by `folder`) of a `.csv` file for a `rastLUT`. This file must follow the format described above for `rastLUT`. It is not necessary to specify both `rastLUT` and `rastLUTfn`. If both are given, `model.map()` will use `rastLUT`.

<code>rastnmVector</code>	Vector of character Strings. The file names and paths (for Imagine Image files) or folder names and paths (for ArcInfo Grids) of rasters to be used to generate a selection list of possible predictors. Only needed if you are creating a model, and <code>predList</code> and <code>rastLUT</code> and <code>rastLUTfn</code> are all NULL
<code>predList</code>	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 <code>rastLUT</code> . If <code>predList = NULL</code> (the default), a GUI interface prompts user to select predictors from column 2 of <code>rastLUT</code> . If both <code>predList = NULL</code> and <code>rastLUT = NULL</code> , 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., <code>rastLUT = NULL</code>), predictor column names of training data must be standard format, consisting of raster stack name followed by <code>b1</code> , <code>b2</code> , etc..., giving the band number within each stack (Example: <code>stacknameb1</code> , <code>stacknameb2</code> , <code>stacknameb3</code> , ...).
<code>predFactor</code>	String. A character vector of predictor short names of the predictors from <code>predList</code> that are factors (i.e categorical predictors). These must be a subset of the predictor names given in <code>predList</code> . Categorical predictors may have multiple categories.
<code>response.name</code>	String. The name of the response variable used to build the model. If <code>response.name = NULL</code> , a GUI interface prompts user to select a variable from the list of column names from training data file. <code>response.name</code> must be column name from the training/test data files.
<code>response.type</code>	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.
<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>predict</code>	Logical. Model validation. If <code>predict = TRUE</code> , validation predictions will be made. If <code>predict = FALSE</code> , no validation predictions will be made. If <code>predict = TRUE</code> , a <code>*.csv</code> file of the unique id, observed, and predicted values is generated and put in the specified (or default) folder. If a test set is provided, the predictions will be made on the test set. If <code>v.fold</code> equals a number greater than 1, cross validation will be used on the training data. If no test set is provided and <code>v.fold = FALSE</code> , predictions will be made on the training set. For Random Forest Models, Out of Bag (OOB) predictions will be made on the training data. For Stochastic Gradient Boosting (SGB) models, predictions on the training data will lead to over optimistic estimates of model quality.

MODELpredfn	String. Model validation. The name of the validation prediction *.csv file. Only used if predict = TRUE. 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.csv".										
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. If set to v.fold = FALSE and no test data is supplied, validation predictions will be made on the training data. For RF models, if v.fold = FALSE, Out-of-Bag (OOB) predictions will be made on the training data, for SGB models, predictions will still be made on the training data, but they will give an over optimistic view of model quality. If a test set is supplied, v.fold will default to FALSE and validation predictions will be made on the test data.										
diagnostics	Logical. Model validation. If diagnostics = TRUE, the following diagnostic statistics and graphs will be generated for validation predictions: A variable importance graph is made. If response.type = "binary", a summary graph is made using the PresenceAbsence package and a *.csv spreadsheet is created of optimized thresholds by several methods with their associated error statistics. 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. Note: diagnostics will only be performed if predict = TRUE.										
device.type	String or vector of strings. Model validation. One or more device types for graphical output from model validation diagnostics. Current choices: <table border="0" style="margin-left: 40px;"> <tr> <td>"default"</td> <td>default graphics device</td> </tr> <tr> <td>"jpeg"</td> <td>*.jpg files</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	"pdf"	*.pdf files	"postscript"	*.ps files	"win.metafile"	*.emf files
"default"	default graphics device										
"jpeg"	*.jpg files										
"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 speci-										

	<p>defined by <code>folder</code>. Defaults to <code>DIAGNOSTICfn = MODELfn</code> followed by the appropriate suffixes (i.e. ".csv", ".jpg", etc...).</p>
<code>jpeg.res</code>	Integer. Model validation. Pixels per inch for jpeg plots. The default is 72dpi, good for on screen viewing. For printing, suggested setting is 300dpi.
<code>device.width</code>	Integer. Model validation. The device width for diagnostic plots in inches.
<code>device.height</code>	Integer. Model validation. The device height for diagnostic plots in inches.
<code>cex</code>	Integer. Model validation. The cex for diagnostic plots.
<code>req.sens</code>	Numeric. Model validation. The required sensitivity for threshold optimization for binary response model evaluation.
<code>req.spec</code>	Numeric. Model validation. The required specificity for threshold optimization for binary response model evaluation.
<code>FPC</code>	Numeric. Model validation. The False Positive Cost for threshold optimization for binary response model evaluation.
<code>FNC</code>	Numeric. Model validation. The False Negative Cost for threshold optimization for binary response model evaluation.
<code>ntree</code>	Integer. RF models. The number of random forest trees for a RF model. The default is 500 trees.
<code>mtry</code>	Integer. RF models. Number of variables to try at each node of Random Forest trees. By default, will use the " <code>tuneRF()</code> " function to optimize <code>mtry</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> package warns that OOB generally underestimates the optimal number of iterations although predictive performance is reasonably competitive. Regardless of the value of the <code>n.trees</code> argument, 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> .
<code>shrinkage</code>	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 randomness 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 * nrow(data)</code> observations are used to fit the model and the remainder are used for computing out-of-sample estimates of the loss function.

n.minobsinnode	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.
map	Logical. Map Production. If <code>map = TRUE</code> , predictions will be made across the extent of the raster layers. If <code>map = FALSE</code> , no predictions will be made. If <code>map = NULL</code> (the default), a GUI window will prompt you to select whether to create a map.
numrows	Integer. Map Production. The number of rows to be predicted at a time.
map.sd	Logical. Map Production. If <code>map.sd = TRUE</code> , maps of mean, standard deviation, and coefficient of variation of the predictions from all the trees are generated for each pixel. If <code>map.sd = FALSE</code> (the default), only the predicted probability map will be built. This option is only available if the <code>model.type = "RF"</code> the <code>response.type = "continuous"</code> . Note: This option requires much more available memory. If you get the error <code>".cannot allocate vector of size..."</code> , you must reduce the value of <code>numrow</code> . 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 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 = NULL</code> (the default), a name is created by pasting <code>modelfn</code> and <code>"_map.txt"</code> .
asciifn.mean	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> .
asciifn.stdev	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> .
asciifn.coefv	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> .

Details

This package provides a push button approach to complex model building and production mapping. It contains two functions: a simple function `get.test()` that can be used to randomly divide a

training dataset into training and test/validation sets; and the workhorse, "do every thing" function nick named "The Button", and called with `model.map()`.

`model.map()` 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.

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 (argument `mtry`) is the primary user specified parameter that can affect model performance. By default this parameter will be automatically optimized using the `tuneRF()` function. 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 (Brieman, 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 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.map()`. 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.map()` 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.

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

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.map()` 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 will return the model object. Additionally, depending on the options selected, it may also write several different things to disk, in the folder specified by `folder`. These include:

```
this-is-escaped-codenormal-bracket326bracket-normal
    the R model object
this-is-escaped-codenormal-bracket330bracket-normal
    .csv file of observed and predicted values
this-is-escaped-codenormal-bracket333bracket-normal
    variable importance and summary graphs of file type specified by device.type
    (i.e. .jpg, .ps, or .emf files)
this-is-escaped-codenormal-bracket337bracket-normal
    .csv file of thresholds optimized by multiple different criteria
this-is-escaped-codenormal-bracket340bracket-normal
    .txt files of map information suitable to be imported into GIS
this-is-escaped-codenormal-bracket343bracket-normal
    .txt file giving the values of each argument as chosen from GUI propts 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](#)

Examples

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

# set seed:
seed=38

# Define training and test files:

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

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

# Create a list of the filenames (including paths) for the rast Look up Tables:
rastLUTfn=list( paste(system.file(package="ModelMap"), "/external/LUT_2001.csv", sep=""),
                paste(system.file(package="ModelMap"), "/external/LUT_2004.csv", sep=""))

# Load rast LUT tables, and add path to the filenames in column 1:
rastLUT<-lapply(rastLUTfn, function(x){ y <- read.table(x,header=FALSE,sep=",",stringsAsFact
                                     y[,1] <- paste(system.file(package="ModelMap"), "exte
                                     return(y) })

# Define identifier for individual training and test data points:
unique.rowname="ID"

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

numrows=500

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

```
#####

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

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

#file name for validation predictions:
MODELpredfn="RF_Bio_TC_PRED.csv"

#names from column 2 of rastLUT:
predList=c("TCB","TCG","TCW")

#define which predictors are categorical:
predFactor=FALSE

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

# Map name:
asciifn<-c("RF_Bio_TC_01.txt","RF_Bio_TC_04.txt")
asciifn<-paste(folder,asciifn,sep="/")

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

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

#file name for validation predictions:
MODELpredfn="RF_CONIFTYP_TC.csv"

#names from column 2 of rastLUT:
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"

# Map name:
asciifn<-c("RF_CONIFTYP_TC_01.txt","RF_CONIFTYP_TC_04.txt")
asciifn<-paste(folder,asciifn,sep="/")

##### 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"

#file name for validation predictions:
MODELpredfn="RF_BIO_TCandNLCD_PRED.csv"

#names from column 2 of rastLUT:
predList=c("TCB", "TCG", "TCW", "NLCD")

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

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

# Map name:
asciifn<-c("RF_BIO_TCandNLCD_01.txt", "RF_BIO_TCandNLCD_04.txt")
asciifn<-paste(folder, asciifn, sep="/")

#####
##### Then run this code to building model: #####
#####

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

model.obj = model.map( model.obj=NULL,
                       model.type="RF",
                       qdata.trainfn=qdata.trainfn,
                       qdata.testfn=qdata.testfn,
                       folder=folder,
                       MODELfn=MODELfn,
                       rastLUT=rastLUT[[1]],
                       predList=predList,

```

```

        predFactor=predFactor,
        response.name=response.name,
        response.type=response.type,
        unique.rowname=unique.rowname,
        seed=seed,
    # Model Validation Arguments
        predict=FALSE,
    # Mapping arguments
        map=FALSE
)

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

model.obj = model.map( model.obj=model.obj,
    qdata.trainfn=qdata.trainfn,
    qdata.testfn=qdata.testfn, #set qdata.testfn=FALSE to use OOB on tr
    folder=folder,
    MODELfn=MODELfn,
    rastLUT=rastLUT[[1]],
    predList=predList,
    predFactor=predFactor,
    response.name=response.name,
    response.type=response.type,
    unique.rowname=unique.rowname,
    seed=seed,
    # Model Validation Arguments
        predict=TRUE,
        diagnostics=TRUE,
        DIAGNOSTICfn=MODELfn,
        device.type=c("jpeg","pdf"),
        MODELpredfn=MODELpredfn,
        v.fold=FALSE,
        na.action="na.roughfix",
    # Mapping arguments
        map=FALSE
)

#####
##### Then Run this code to create maps: #####
#####

### button - Batch (must have already created model) ###

load(paste(folder,"/",MODELfn,sep=""))

for(i in 1:length(rastLUTfn)){

    print("#####")
    print(paste("Starting",asciifn[i]))
    print("#####")

```

```
model.obj = model.map( model.obj=model.obj,
                       folder=folder,
                       rastLUT=rastLUT[[i]],
                       seed=seed,
                       # Model Validation Arguments
                       predict=FALSE,
                       na.action="na.roughfix",
                       # Mapping arguments
                       map=TRUE,
                       numRows = numRows,
                       asciifn=asciifn[i]
                       )
}
```

Index

*Topic **models**

`get.test`, [3](#)

`model.map`, [5](#)

*Topic **package**

`ModelMap-package`, [2](#)

`get.test`, [3](#), [13](#)

`model.map`, [4](#), [5](#)

`model.run` (*model.map*), [5](#)

`ModelMap` (*ModelMap-package*), [2](#)

`ModelMap-package`, [2](#)