M5PrimeLab

M5' regression tree, model tree, and tree ensemble toolbox for Matlab/Octave

ver. 1.7.0

Gints Jekabsons

E-mail: gints.jekabsons@rtu.lv
http://www.cs.rtu.lv/jekabsons/

User's manual

August, 2016

CONTENTS

1. INTRODUCTION

What is M5PrimeLab

M5PrimeLab is a Matlab/Octave toolbox for building regression trees and model trees using M5' method (Wang & Witten, 1997; Quinlan, 1992) as well as building ensembles of M5' trees using Bagging (Breiman, 1996), Random Forests (Breiman, 2001; Breiman, 2002), and Extremely Randomized Trees (also known as Extra-Trees) (Geurts et al., 2006). The built trees can also be linearized into decision rules either directly or using the M5'Rules method (Holmes et al., 1999).

With this toolbox you can build trees and decision rules either individually or in ensembles and test them on separate test sets or using Cross-Validation, use them for prediction, assess variable importance, decompose their predictions into input variable contributions, as well as print and plot the structure. M5PrimeLab accepts input variables to be continuous, binary, and categorical, as well as manages missing values.

Note that regardless of which ensembling algorithm one chooses, M5PrimeLab builds the individual trees according to the M5' method, i.e., usage of the Standard Deviation Reduction criterion as well as how the categorical input variables are dealt with and other details specific to the method are not reconfigurable. Also see the note on Extra-Trees below.

This user's manual provides overview of the functions available in the M5PrimeLab.

M5PrimeLab can be downloaded at http://www.cs.rtu.lv/jekabsons/.

A note on Extremely Randomized Trees implementation in M5PrimeLab

The current implementation of Extra-Trees in M5PrimeLab deals with categorical variables with more than two categories differently from how the standard Extra-Trees method does it.

In standard Extra-Trees (Geurts et al., 2006), if a categorical variable is chosen for splitting, two random subsets of categories are drawn – one from the list of categories that reached the node and the other from the list that did not reach the node. The splitting point is then the union of those two subsets versus all the other categories.

In the M5PrimeLab implementation of Extra-Trees, such categorical variables are automatically replaced with synthetic binary variables in accordance with the M5' method before any building of trees is even started. The number of the synthetic binary variables is equal to the number of categories minus one. If such a categorical variable is chosen for splitting, the splitting point is then defined using one of those synthetic binary variables, chosen randomly.

The more such categorical variables are in the data, the potentially more different the results from the standard Extra-Trees.

However, note that continuous and binary variables are dealt with exactly in the same way as in standard Extra-Trees.

Feedback

For any feedback on the toolbox including bug reports feel free to contact me via the email address given on the title page of this user's manual.

Citing the M5PrimeLab toolbox

Jekabsons G., M5PrimeLab: M5' regression tree, model tree, and tree ensemble toolbox for Matlab/Octave, 2016, available at http://www.cs.rtu.lv/jekabsons/

2. AVAILABLE FUNCTIONS

M5PrimeLab toolbox provides the following list of functions:

- `m5pbuild` – builds M5' regression tree, model tree, or ensemble of trees; the tree can also be linearized into decision rules; for tree ensembles, can also assess input variable importances as well as provide data for ensemble interpretation;
- `m5pparams`, `m5pparams2` – creates configuration for building M5' trees or decision rules;
- `m5pparamsensemble`, `m5pparamsensemble` – creates configuration for building ensembles of trees;
- `m5ppredict` – makes predictions using M5' tree, decision rule set, or ensemble of trees; can also compute each input variable's contribution to each prediction;
- `m5ptest` – tests M5' tree, decision rule set, or ensemble of trees on a test data set;
- `m5pcv` – tests M5' performance using Cross-Validation;
- `m5pprint` – prints M5' tree or decision rule set in a human-readable form;
- `m5pplot` – plots M5' tree.

2.1. Function `m5pbuild`

Purpose:

Builds M5' regression tree, model tree, or ensemble of trees.
The trees can also be linearized into decision rules.

Call:

```
[model, time, ensembleResults] = m5pbuild(Xtr, Ytr, trainParams, isBinCat, trainParamsEnsemble, keepNodeInfo, verbose)
```

All the input arguments, except the first two, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`Xtr, Ytr` : `Xtr` is a matrix with rows corresponding to observations and columns corresponding to input variables. `Ytr` is a column vector of response values. Input variables can be continuous, binary, as well as categorical (indicated using `isBinCat`). All values must be numeric. Categorical variables with more than two categories will be automatically replaced with synthetic binary variables (in accordance with the M5' method). Missing values in `Xtr` must be indicated as `NaN`.

`trainParams` : A structure of training parameters for the algorithm. If not provided, defaults will be used (see function `m5pparams` for details).

`isBinCat` : A vector of flags indicating type of each input variable – either continuous (`false`) or categorical (`true`) with any number of categories, including binary. The vector should be of the same length as the number of columns in `Xtr`. `m5pbuild` then detects all the actually possible values for categorical variables from the training data. Any new values detected later, i.e., during prediction, will be treated as `NaN`. By default, the vector is created with all values equal to `false`, meaning that all the variables are treated as continuous.

`trainParamsEnsemble` : A structure of parameters for building ensemble of trees. If not provided, a single tree is built. See function `m5pparamsensemble` for

details. This can also be useful for variable importance assessment. See user's manual for examples of usage.
Note that the ensemble building algorithm employs random number generator for which you can set seed before calling `m5pbuild`.

`keepNodeInfo` : Whether to keep models (in model trees) and response values (in regression trees) in interior nodes of trees. And whether to keep indices of training observations that reached each node and standard deviation of each node. These are useful for further analysis and plotting. Default value = `true`. If set to `false`, the information is removed from the trees so that the structure takes up less memory. Note that interior nodes of smoothed trees will not contain models or response values regardless of the value of this parameter because only the models in the leaves are smoothed. Also note that the standard deviations are saved before doing smoothing.

`verbose` : Whether to output additional information to console. (default value = `true`)

Output:

`model` : A single M5' tree (or a decision rule set) or a cell array of M5' trees (or decision rule sets) if an ensemble is built. A structure defining one tree (or a decision rule set) has the following fields:

- `binCat` : Information regarding original (continuous / binary / categorical) input variables, transformed (synthetic binary) input variables, possible values for categorical input variables and other information.
- `trainParams` : A structure of training parameters for the algorithm (updated if any values are chosen automatically).
- `tree, rules, outcomes` : Structures and arrays defining either the built tree or the generated decision rules.

`time` : Algorithm execution time (in seconds).

`ensembleResults` : A structure of results for ensembles of trees or decision rules. Not available for Extra-Trees. The structure has the following fields:

- `OOBError` : Out-of-bag estimate of prediction Mean Squared Error of the ensemble after each new tree is built. The number of rows is equal to the number of trees built. `OOBError` is available only if `getOOBError` in `trainParamsEnsemble` is set to `true`. Note that it's possible to calculate mean out-of-bag predictions (and therefore out-of-bag errors for each individual training data observation) by summing the columns of `OOBContrib`.
- `OOBIndices` : Logical matrix. For each tree (column) indicates which observations were out-of-bag (and thus used in computing `OOBError`). The number of rows in `OOBIndices` is equal to the number of rows in `Xtr` and `Ytr`. `OOBIndices` is available only if `getOOBError` or `getOOBContrib` in `trainParamsEnsemble` is set to `true`.
- `OOBNum` : Number of times observations were out-of-bag (and thus used in computing `OOBError`). The length of `OOBNum` is equal to the number of rows in `Xtr` and `Ytr`. `OOBNum` is available only if `getOOBError` or `getOOBContrib` in `trainParamsEnsemble` is set to `true`.
- `OOBContrib` : A matrix allowing interpreting ensembles in accordance with the Forest Floor methodology (Welling et al., 2016). See also example of usage in Section 3.2.
  It is a matrix of contributions of each input variable to the response for each `Xtr` row in terms of response value changes along the prediction path of a tree (averaged over the whole ensemble) so that `Yoob` = *in-bag_mean* +

x_1_*contribution* + x_2_*contribution* + ... + x_n_*contribution*, where `Yoob` is prediction of response for out-of-bag observation. `OOBContrib` has the same number of columns as `Xtr` plus one, the last column being the in-bag response mean. The sum of columns of `OOBContrib` is equal to `Yoob` of the whole ensemble for each row of `Xtr`.

`OOBContrib` is available only if `getOOBContrib` in `trainParamsEnsemble` is set to `true`.

Note that it's also possible to compute contributions and explain predictions for new data (including with single trees) – see function `m5ppredict`.

`varImportance` : Variable importance assessment. Calculated when out-of-bag data of a variable is permuted. A matrix with four rows and as many columns as there are columns in `Xtr`. First row is the average increase of out-of-bag Mean Absolute Error (MAE), second row is standard deviation of the average increase of MAE, third row is the average increase of out-of-bag Mean Squared Error (MSE), fourth row is standard deviation of the average increase of MSE. The final variable importance estimate is often calculated by dividing each MAE or MSE by the corresponding standard deviation. Bigger values then indicate bigger importance of the corresponding variable. See user's manual for example of usage. `varImportance` is available only if `getVarImportance` in `trainParamsEnsemble` is > 0.

Remarks:

M5' method builds a tree in two phases: growing phase and pruning phase. In the first phase the algorithm starts with one leaf node and recursively tries to split each leaf node so that intra-subset variation in the response variable's values down each branch is minimized (i.e., Standard Deviation Reduction (SDR) is maximized).

At the end of the first phase we have a large tree that typically overfits the data, and so a pruning phase is engaged. In this phase, the tree is pruned back from each leaf until an estimate of the expected error that will be experienced at each node cannot be reduced any further.

Finally, the tree is smoothed (optionally). In M5PrimeLab, smoothing is done in `m5pbuild`, right after the pruning phase (instead of doing it only at the moment of prediction, i.e., in `m5ppredict`), by incorporating regression models of the interior nodes into regression models of each leaf. Smoothing can increase accuracy of predictions but it also makes the trees more difficult to interpret, as smoothed trees have more complex models at their leaves (they can even look as if dropping of terms never occurred).

2.2. Function `m5pparams`

Purpose:

Creates configuration for building M5' trees or decision rules. The output structure is for further use with `m5pbuild` and `m5pcv` functions.

Call:

```
trainParams = m5pparams(modelTree, minLeafSize, minParentSize, prune, smoothingK, splitThreshold, aggressivePruning, maxDepth, eliminateTerms, vanillaSDR, extractRules)
```

All the input arguments of this function are optional. Empty values are also accepted (the corresponding defaults will be used).

It is quite possible that the default values for `minLeafSize` and `minParentSize` will be far from optimal for your data.

For a typical configuration of ensembles of regression trees (whether Bagging, Random Forests, or Extra-Trees), call `trainParams = m5pparams(false, 1, 5, false, 0, 1E-6);`

Input:

`modelTree` : Whether to build a model tree (`true`) or a regression tree (`false`) (default value = `true`). Model trees combine a conventional regression tree with the possibility of linear regression functions at the leaves. However, note that whether a leaf node actually contains more than just a constant, depends on pruning and smoothing (if both are disabled, a model tree will not differ from a regression tree).

`minLeafSize` : The minimum number of training observations a leaf node may represent. If `prune` = `true`, allowed minimum is 2. Otherwise, allowed minimum is 1. Default value = 2 (Wang & Witten, 1997). If built trees contain too many too small leaves (especially in the top layers of the tree), consider increasing this number. This will also result in smaller trees that are less sensitive to noise (but can also be underfitted).

`minParentSize` : The minimum number of observations a node must have to be considered for splitting, i.e., the minimum number of training observations an interior node may represent. Default value = `minLeafSize` × 2 (Wang & Witten, 1997). Values lower than that are not allowed. If built trees are too large or overfit the data, consider increasing this number – this will result in smaller trees that are less sensitive to noise (but can also be underfitted). For ensembles of unpruned trees, the typical value is 5 (with `minLeafSize` = 1) (Breiman, 2001).

`prune` : Whether to prune the tree (default value = `true`). Pruning is done by eliminating leaves and subtrees in regression trees and model trees as well as, if `eliminateTerms` = `true`, by eliminating terms in models of model trees (using sequential backward selection algorithm), if doing so improves the estimated error.

`smoothingK` : Smoothing parameter. Set to 0 to disable smoothing (default). Smoothing is usually not recommended for regression trees but can be useful for model trees. It tries to compensate for sharp discontinuities occurring between adjacent nodes of the tree. The larger the value compared to the number of observations reaching the nodes, the more pronounced is the smoothing. In case studies by Quinlan, 1992, as well as Wang & Witten, 1997, this almost always had a positive effect on model trees (with `smoothingK` = 15). Smoothing is performed after tree building and pruning, therefore this parameter does not influence those processes. Unfortunately, smoothed trees are harder to interpret.

`splitThreshold` : A node is not split if the standard deviation of the values of response variable at the node is less than `splitThreshold` of the standard deviation of response variable for the entire training data (default value = 0.05 (i.e., 5%) (Wang & Witten, 1997)).

`aggressivePruning` : By default, pruning is done as proposed by Quinlan, 1992, and Wang & Witten, 1997, but you can also employ more aggressive pruning, similar to the one that is implemented in Weka's version of M5' (Hall et al., 2009). Simply put, in the aggressive pruning version, while estimating error of a subtree, one penalizes not only the number of parameters of regression models at its leaves but also its total number of splits. Aggressive pruning produces smaller trees that are less sensitive to noise and potentially easier to interpret. However, this can also result in underfitting. (default value = `false`)

`maxDepth` : Maximum depth of a tree. Controlling tree complexity using this parameter is not typical for M5' trees. (default value = `Inf`, i.e., no limitation)

`eliminateTerms` : Whether to eliminate terms of models in model trees using the sequential backward selection algorithm (default value = `true`). The parameter has no effect if `prune` = `false`.

`vanillaSDR` : Whether to calculate the Standard Deviation Reduction criterion using the "vanilla" formula from Quinlan, 1992 or the updated formula from Wang & Witten, 1997 (default value = `false`, i.e., the updated formula is used). The difference between those two formulas is that the updated one tries to take into account the number of missing values for each input variable, as well as the number of categories for each categorical input variable. See equations (1) and (5) in Wang & Witten, 1997. Note that the updated formula is actually fairly ad hoc and not well studied.
This parameter has no effect if the data has no missing values and no categorical variables with more than two categories.

`extractRules` : M5' trees can also be used for generating decision rules. M5PrimeLab provides two methods for doing it. Set `extractRules` = 1 to extract rules from one tree directly. Each leaf is made into a rule by making a conjunction of all the tests encountered on the path from the root to that leaf. This produces rules that are unambiguous in that it doesn't matter in what order they are executed. The rule set always makes exactly the same predictions as the original tree, even with unknown values and smoothing.
Set `extractRules` = 2 to use the M5'Rules method (Holmes et al., 1999). With this method, the rules are generated iteratively. In each iteration, a new tree is built using the training data and one leaf that has the largest data coverage is made into a rule. Then the tree is discarded and all observations covered by the rule are removed from the data. The process is repeated until the data is empty. M5'Rules produces smaller rule sets than the simple extraction method, however it cannot use the M5' smoothing technique (parameter `smoothingK` is ignored).
(default value = 0, i.e., no rules are extracted)

Output:

`trainParams` : A structure of parameters for further use with `m5pbuild` and `m5pcv` functions containing the provided values (or defaults, if not provided).

2.3. Function `m5pparams2`

Purpose:

Creates configuration for building M5' trees or decision rules. The output structure is for further use with `m5pbuild` and `m5pcv` functions.

This function is an alternative to function `m5pparams` for supplying parameters as name/value pairs.

Call:

```
trainParams = m5pparams2(varargin)
```

Input:

`varargin` : Name/value pairs for the parameters. For the list of the names, see description of function `m5pparams`.

Output:

`trainParams` : A structure of parameters for further use with `m5pbuild` and `m5pcv` functions containing the provided values (or defaults, if not provided).

2.4. Function `m5pparamsensemble`

Purpose:

Creates configuration for building ensembles of M5' trees using Bagging, Random Forests, or Extra-Trees. The output structure is for further use with `m5pbuild` and `m5pcv` functions.

Call:

```
trainParamsEnsemble = m5pparamsensemble(numTrees, numVarsTry, withReplacement, inBagFraction, extraTrees, getOOBError, getVarImportance, getOOBContrib, verboseNumIter)
```

All the input arguments of this function are optional. Empty values are also accepted (the corresponding defaults will be used). The first five arguments control the behaviour of the ensemble building method. The last four arguments enable getting additional information.

The default values are prepared for building Random Forests. Changes required for a Bagging configuration: `numVarsTry` = 0. Changes required for a typical Extra-Trees configuration: `numVarsTry` = 0, `extraTrees` = true.

Remember to configure how individual trees are built for the ensemble (see description of `m5pparams`). See Section 3.2 for examples of usage.

Input:

`numTrees` : Number of trees to build (default value = 100). Should be set so that every data observation gets predicted at least a few times.

`numVarsTry` : Number of input variables randomly sampled as candidates at each split in a tree. Set to -1 (default) to automatically sample one third of the variables (typical for Random Forests in regression). Set to 0 to use all variables (typical for Bagging and Extra-Trees in regression). Set to a positive integer if you want some other number of variables to sample. To select a good value for `numVarsTry` in Random Forests, Leo Breiman suggests trying the default value and trying a value twice as high and half as low (Breiman, 2002).
In Extra-Trees, this parameter is also called attribute selection strength (Geurts et al., 2006).
Note that while using this parameter, function `m5pbuild` takes the total number of input variables directly from supplied training data set, before any synthetic binary variables are made (from categorical variables with more than two categories). Also note that `m5pbuild` will always round the `numVarsTry` value down.

`withReplacement` : Should sampling of in-bag observations for each tree be done with (`true`) or without (`false`) replacement? Both, Bagging and Random Forests typically use sampling with replacement. (default value = `true`)

`inBagFraction` : The fraction of the total number of observations to be sampled for in-bag set. Default value = 1, i.e., the in-bag set will be the same size as the original data set. This is the typical setting for both, Bagging and Random Forests. Note that for sampling without replacement `inBagFraction` should be lower than 1 so that out-of-bag set is not empty.

`extraTrees` : Set to true to build Extra-Trees (default = `false`). If enabled, parameters `withReplacement`, `inBagFraction`, `getOOBError`, and `getVarImportance` are ignored. This is because Extra-Trees method does not use out-of-bag data, i.e., all trees are build using the whole available training data set.

`getOOBError` : Whether to perform out-of-bag error calculation to estimate prediction error of the ensemble (default value = `true`). The result will be stored in the output argument `ensembleResults` of function `m5pbuild`. Disable for speed.

`getVarImportance` : Whether to assess importance of input variables (by calculating the average increase in error when out-of-bag data of a variable is permuted) and how many times the data is permuted per tree for the assessment. Default value = 1. Set to 0 to disable and gain some speed. Numbers larger than 1 can give slightly more stable estimate, but the process is even slower. The result will be stored in the output argument `ensembleResults` of function `m5pbuild`.

`getOOBContrib` : Whether to compute input variable contributions in out-of-bag data according to the Forest Floor methodology. Available only for ensembles of unsmoothed regression trees and only if `m5pbuild` is called with `keepNodeInfo` = `true`. The result will be stored in the output argument `ensembleResults` of function `m5pbuild`. For details, see description of `OOBContrib`.

`verboseNumIter` : Set to some positive integer to print progress every `verboseNumIter` trees. Set to 0 to disable. (default value = 50)

Output:

`trainParamsEnsemble` : A structure of parameters for further use with `m5pbuild` and `m5pcv` functions containing the provided values (or defaults, if not provided).

Remarks:

See the note in Section 1 on the most important difference between the implementation of Extra-Trees in M5PrimeLab and standard Extra-Trees.

2.5. Function `m5pparamsensemble2`

Purpose:

Creates configuration for building ensembles of M5' trees using Bagging, Random Forests, or Extra-Trees. The output structure is for further use with `m5pbuild` and `m5pcv` functions.

This function is an alternative to function `m5pparamsensemble` for supplying parameters as name/value pairs.

Call:

```
trainParamsEnsemble = m5pparamsensemble2(varargin)
```

Input:

`varargin` : Name/value pairs for the parameters. For the list of the names, see description of function `m5pparamsensemble`.

Output:

`trainParamsEnsemble` : A structure of parameters for further use with `m5pbuild` and `m5pcv` functions containing the provided values (or defaults, if not provided).

2.6. Function `m5ppredict`

Purpose:

Predicts response values for the given query points `Xq` using M5' tree, decision rule set, or ensemble of trees. For unsmoothed regression trees (whether in ensembles or as individual trees), can also provide a matrix of input variable contributions to the response value for each row of `Xq`.

Call:

```
[Yq, contrib] = m5ppredict(model, Xq)
```

Input:

`model` : M5' model, decision rule set, or a cell array of M5' models, if ensemble of trees is to be used.

`Xq` : A matrix of query data points. Missing values in `Xq` must be indicated as `NaN`.

Output:

`Yq` : A column vector of predicted response values. If `model` is an ensemble, `Yq` is a matrix whose rows correspond to `Xq` rows (i.e., observations) and columns correspond to each ensemble size (i.e., the increasing number of trees), the values in the very last column being the values for a full ensemble.

`contrib` : Available only for unsmoothed regression trees (whether in ensembles or as single trees) and only if `m5pbuild` was called with `keepNodeInfo = true`.

A matrix of contributions of each input variable to the response for each `Xq` row in terms of response value changes along the prediction path of a tree so that `Yq` = *training_set_mean* + x_1*_contribution* + x_2*_contribution* + ... + x_n*_contribution*. `contrib` has the same number of columns as `Xq` plus one, the last column being the training set response mean for single trees or in-bag response mean for ensembles. The sum of columns of `contrib` is equal to `Yq` (or to the last column of `Yq` for ensembles).

This allows interpreting single trees and whole ensembles as well as explaining their predictions. The implemented method is sometimes also called "feature contribution method" (Saabas, 2014/2015 (see this for the simplest explanation); Welling et al., 2016; Kuz'min et al., 2011; Palczewska et al., 2013). See also examples of usage in Section 3.

Note that this function does not decompose contributions according to the Forest Floor methodology (Welling et al., 2016) as `contrib` is computed using the given `Xq`, instead of the out-of-bag data. For decomposition in accordance with Forest Floor, see output argument `ensembleResults.OOBContrib` of function `m5pbuild`.

Remarks:

1. If the data contains categorical variables with more than two categories, they are transformed into synthetic binary variables in exactly the same way as `m5pbuild` does it.
2. Any previously unseen values of binary or categorical variables are treated as `NaN`.

2.7. Function `m5ptest`

Purpose:

Tests M5' tree, decision rule set, or ensemble of trees on a test data set (`Xtst`, `Ytst`).

Call:

```
results = m5ptest(model, Xtst, Ytst)
```

Input:

`model`	: M5' model, decision rule set, or a cell array of M5' models (if ensemble of trees is to be tested).
`Xtst, Ytst`	: `Xtst` is a matrix with rows corresponding to testing observations, and columns to corresponding input variables. `Ytst` is a column vector of response values. Missing values in `Xtst` must be indicated as `NaN`.

Output:

`results`	: A structure of different error measures calculated on the test data set. The structure has the following fields (if the `model` is an ensemble, the fields are column vectors with one (cumulative) value for each ensemble size, the very last value being error for a full ensemble):
`MAE`	: Mean Absolute Error.
`MSE`	: Mean Squared Error.
`RMSE`	: Root Mean Squared Error.
`RRMSE`	: Relative Root Mean Squared Error.
`R2`	: Coefficient of Determination.

2.8. Function `m5pcv`

Purpose:

Tests M5' performance using *k*-fold Cross-Validation.

Call:

```
[results, residuals] = m5pcv(X, Y, trainParams, isBinCat, k, shuffle, nCross,
trainParamsEnsemble, verbose)
```

All the input arguments, except the first two, are optional. Empty values are also accepted (the corresponding defaults will be used).

Note that, if parameter `shuffle` is set to `true`, this function employs random number generator for which you can set seed before calling the function.

Input:

`X, Y`	: The data. Missing values in `X` must be indicated as `NaN`. (see function m5pbuild for details)
`trainParams`	: A structure of training parameters. If not provided, defaults will be used (see function `m5pparams` for details).
`isBinCat`	: See description of function `m5pbuild`.
`k`	: Value of *k* for k-fold Cross-Validation. The typical values are 5 or 10. For Leave-One-Out Cross-Validation set *k* equal to *n*. (default value = 10)
`shuffle`	: Whether to shuffle the order of observations before performing Cross-Validation. (default value = `true`)

`nCross` : How many times to repeat Cross-Validation with different data partitioning. This can be used to get more stable results. Default value = 1, i.e., no repetition. Useless if `shuffle` = `false`.

`trainParamsEnsemble` : A structure of parameters for building ensembles of trees. If not provided, a single tree is built. See function `m5pparamsensemble` for details.

`verbose` : Whether to output additional information to console. (default value = `true`)

Output:

`resultsTotal` : A structure of results averaged over Cross-Validation folds. For tree ensembles, the structure contains fields that are column vectors with one value for each ensemble size, the very last value being value for a full ensemble.

`resultsFolds` : A structure of row vectors of results for each Cross-Validation fold. For tree ensembles, the structure contains matrices whose rows correspond to Cross-Validation folds while columns correspond to each ensemble size, the very last value being a value for a full ensemble.

Both structures have the following fields:

`MAE` : Mean Absolute Error.

`MSE` : Mean Squared Error.

`RMSE` : Root Mean Squared Error.

`RRMSE` : Relative Root Mean Squared Error. Not reported for Leave-One-Out Cross-Validation.

`R2` : Coefficient of Determination. Not reported for Leave-One-Out Cross-Validation.

`nRules` : Number of rules in tree. For ensembles of trees, this field is omitted.

`nVars` : Number of input variables included in tree. This counts original variables (not synthetic ones that are automatically made). For ensembles of trees, this field is omitted.

2.9. Function `m5pprint`

Purpose:

Prints M5' tree or decision rule set in a human-readable form. Does not work with ensembles.

Call:

```
m5pprint(model, showNumCases, precision, dealWithNaN)
```

All the input arguments, except the first one, are optional. Empty values are also accepted (the corresponding defaults will be used).

Input:

`model` : M5' model or decision rule set.

`showNumCases` : Whether to show the number of training observations corresponding to each leaf (default value = `true`).

`precision` : Number of digits used for any numerical values shown (default value = 5).

`dealWithNaN` : Whether to display how the tree deals with missing values (`NaN`, displayed as '?') (default value = `false`).

Remarks:

1. For smoothed M5' trees / decision rule sets, the smoothing process is already done in `m5pbuild`, therefore if you want to see unsmoothed versions (which are usually easier to interpret) you should build trees with smoothing disabled.
2. If the training data has categorical variables with more than two categories, the corresponding synthetic binary variables are shown.

2.10. Function `m5pplot`

Purpose:

Plots M5' tree. Does not work with ensembles or decision rule sets.

Call:

```
m5pplot(model, varargin)
```

In the plotted tree, left child of a node corresponds to outcome 'true' and right child to outcome 'false'.

All the input arguments, except the first one, are optional.

Input:

`model` : M5' model.

`varargin` : Name/value pairs of parameters:

- `showNumCases` : Whether to show the number of training observations corresponding to each node. Set to `'all'` to show it for all nodes. Set to `'leaves'` to show it for leaves only. Set to `'off'` or any other value to turn it off (default value = `'all'`).
- `showSD` : Whether to show standard deviation values corresponding to each node (default value = `false`). These values can also be interpreted as Root Mean Squared Error of each node in its corresponding partition of the training dataset. Note that the information is available only if the tree was built using `keepNodeInfo` = `true`.
- `precision` : Number of digits used for any numerical values shown (default value = 5).
- `dealWithNaN` : Whether to display how the tree deals with missing values (`NaN`, displayed as '?') (default value = `false`).
- `layout` : Graph layout algorithm and tree style. Set to `'oblique'` for semi-optimized layout of a tree with edges that form oblique angles. Set to `'right'` for unoptimized layout of a tree with edges that form right angles. Set to `'old'` for the old version of the graph layout algorithm (default value = `'oblique'`).
- `widthMult` : Edge width multiplier (default value = 1).
- `variableWidth` : Whether edge width should reflect the number of training observations (default value = `false`).
- `colorize` : Whether to colorize nodes and edges according to the response values (default value = `false`). Not available for model trees or if `layout` = `'old'`. Complete colorization is available only if the regression tree was built using `keepNodeInfo` = `true`.
- `fontSize` : Font size for text (default value = 10).

Remarks:

1. For smoothed M5' trees, the smoothing process is already done in `m5pbuild`, therefore if one wants to see unsmoothed versions (which are usually easier to interpret), the trees should be built with smoothing disabled.
2. If the training data has categorical variables with more than two categories, the corresponding synthetic binary variables are shown.
3. For unsmoothed regression trees, if they were built using `keepNodeInfo = true`, the plot will show predicted values at interior nodes as well.

3. EXAMPLES OF USAGE

3.1. Growing regression trees, model trees, and decision rules

We start by creating a dataset using a three-dimensional function with one continuous variable, one binary variable, and one categorical variable with four categories. The data consists of randomly uniformly distributed 100 observations.

```
X = [rand(100,1) rand(100,1)<0.5 floor(rand(100,1)*4)];
Y = X(:,1).*(X(:,3)==0) + X(:,2).*(X(:,3)==1) - ...
    2*X(:,1).*(X(:,3)==2) + 3*(X(:,3)==3) + 0.02*randn(100,1);
```

First let's try to grow a model tree. All the parameters will be left to their defaults. We won't use smoothing because our data has sharp discontinuities and we don't want to loose them.

We will supply `isBinCat` vector indicating that the first input variable is continuous, the second is binary, and the third is categorical with four categories (detected automatically). M5' tree is grown by calling `m5pbuild`.

```
params = m5pparams2('modelTree', true);
model = m5pbuild(X, Y, params, [false true true]);
```

As the tree growing process ends, we can examine the structure of the grown tree using function `m5pprint`. First we see synthetic variables (automatically made if the data contains at least one categorical variable with more than two categories) and then the tree itself. Each leaf of a model tree contains either a constant or a linear regression model. Number of training data observations for each leaf is shown in parentheses.

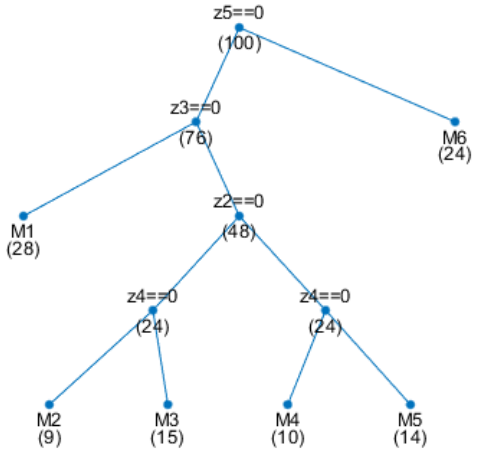
```
m5pprint(model);

Synthetic variables:
z1 = x1
z2 = x2
z3 = 1, if x3 is in {0, 1, 3} else = 0
z4 = 1, if x3 is in {1, 3} else = 0
z5 = 1, if x3 is in {3} else = 0
The tree:
if z5 == 0
 if z3 == 0
  y = -0.0039976 -2.0036*z1 (28)
 else
  if z2 == 0
   if z4 == 0
    y = -0.0055303 +1.0162*z1 (9)
   else
    y = -0.0012792 (15)
  else
   if z4 == 0
    y = -0.0083598 +1.0228*z1 (10)
   else
    y = 1.0004 (14)
else
 y = 2.9962 (24)
Number of rules: 6
Number of original input variables used: 3 (x1, x2, x3)
```

If the tree is too large or overfits, consider increasing `minLeafSize` and/or `minParentSize` or setting `aggressivePruning` to true. You can use function `m5pcv` to test different configurations.

Now let’s plot the tree using the function `m5pplot`. Left child of a node corresponds to outcome 'true' and right child to 'false'.

```
m5pplot(model);
```




```
Synthetic variables:
z1 = x1
z2 = x2
z3 = 1 if x3 is in {0, 1, 3} else = 0
z4 = 1 if x3 is in {1, 3} else = 0
z5 = 1 if x3 is in {3} else = 0
Models:
M1 = -0.0039976 -2.0036*z1
M2 = -0.0055303 +1.0162*z1
M3 = -0.0012792
M4 = -0.0083598 +1.0228*z1
M5 = 1.0004
M6 = 2.9962
```

We can evaluate performance of this M5' configuration on the data using Cross-Validation (10 folds by default). This is done using function `m5pcv`. Note that for more stable results one should consider repeating Cross-Validation several times (see description of the argument `nCross` of function `m5pcv`).

```
rng(1);
results = m5pcv(X, Y, params, [false true true])

results =
       MAE: 0.0175
       MSE: 5.7272e-04
      RMSE: 0.0226
     RRMSE: 0.0178
        R2: 0.9996
    nRules: 6.1000
     nVars: 3
```

Now, let's try doing the same but instead of model tree we will grow a regression tree. In a regression tree, each leaf predicts the output using just a simple constant.

```
params = m5pparams2('modelTree', false);
model = m5pbuild(X, Y, params, [false true true]);
m5pprint(model);

Synthetic variables:
z1 = x1
z2 = x2
z3 = 1, if x3 is in {0, 1, 3} else = 0
z4 = 1, if x3 is in {1, 3} else = 0
z5 = 1, if x3 is in {3} else = 0
The tree:
if z5 == 0
 if z3 == 0
  if z1 <= 0.54819
   if z1 <= 0.24359
    if z1 <= 0.16427
     y = -0.14619 (2)
    else
     y = -0.41656 (2)
   else
    if z1 <= 0.3558
     y = -0.6216 (5)
    else
     y = -0.83133 (4)
  else
   if z1 <= 0.77846
```

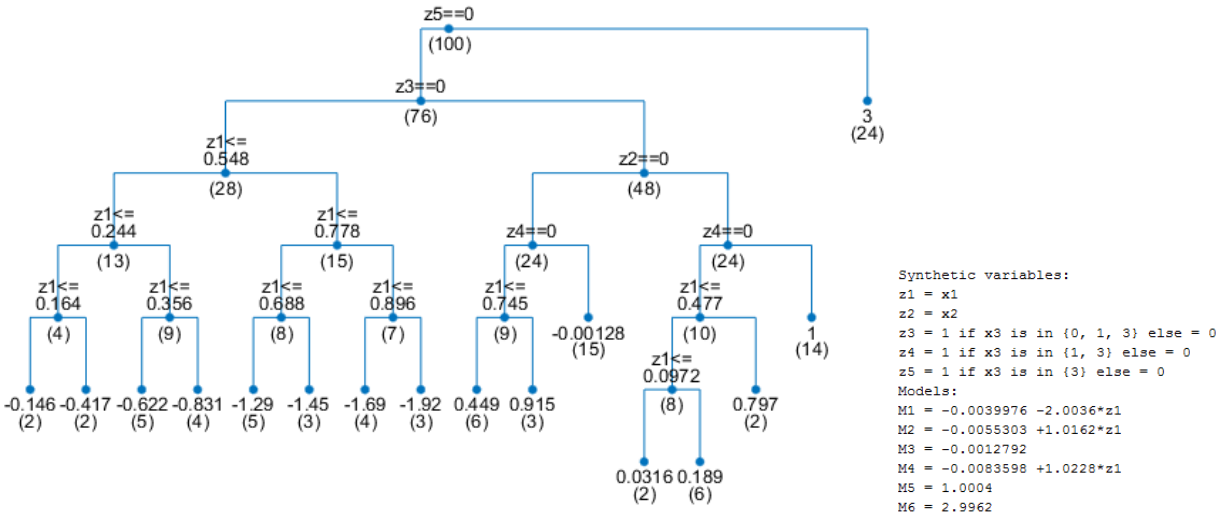
```
    if z1 <= 0.68792
     y = -1.2857 (5)
    else
     y = -1.4471 (3)
   else
    if z1 <= 0.89632
     y = -1.685 (4)
    else
     y = -1.9157 (3)
 else
  if z2 == 0
   if z4 == 0
    if z1 <= 0.74506
     y = 0.44942 (6)
    else
     y = 0.91532 (3)
   else
    y = -0.0012792 (15)
  else
   if z4 == 0
    if z1 <= 0.47721
     if z1 <= 0.097205
      y = 0.031616 (2)
     else
      y = 0.18881 (6)
    else
     y = 0.79682 (2)
   else
    y = 1.0004 (14)
else
 y = 2.9962 (24)
Number of rules: 16
Number of original input variables used: 3 (x1, x2, x3)

rng(1);
results = m5pcv(X, Y, params, [false true true])

results =
       MAE: 0.0754
       MSE: 0.0178
      RMSE: 0.1233
     RRMSE: 0.1029
        R2: 0.9842
    nRules: 14.9000
     nVars: 3
```

Let's plot the tree. But this time we'll also try a different visualization style.

```
m5pplot(model, 'precision', 3, 'layout', 'right');
```



To use the tree for predictions, just call `m5ppredict`. Moreover, for regression trees, we can also compute how much each input variable contributed to the predicted response, i.e., by how much each variable "pulled" the response away from its mean.

```
[Yq, contrib] = m5ppredict(model, [0.5 0 2]);
fprintf('Prediction: %f\n', Yq(1));
fprintf('Training set mean: %f\n', contrib(1,end));
fprintf('Input variable contributions:\n');
[~, idx] = sort(abs(contrib(1,1:end-1)), 'descend');
for i = idx
    fprintf('x%d: %f\n', i, contrib(1,i));
end
```

```
Prediction: -0.831327
Training set mean: 0.633115
Input variable contributions:
x3: -1.733675
x1: 0.269233
x2: 0.000000
```

Or in other words: *prediction* = -0.831327 = 0.633115 (training set response mean) - 1.733675 (loss from x_3) + 0.269233 (gain from x_1). This can be viewed as a breakdown of the prediction in terms of response value changes along the prediction path, together with input variable names that "caused" these changes due to being the split variables in the path. For simple explanation of the "feature contribution method", see Saabas 2014/2015.

Next, let's try generating decision rules from M5' model trees using the M5'Rules method. This is an iterative method. In each iteration it grows a tree, selects the rule that covers the most data observations, discards the tree, and removes all observations from the data that were covered by the selected rule. The method is slower than the direct rule extraction method but it usually produces fewer rules.

```
params = m5pparams2('modelTree', true, 'extractRules', 2);
model = m5pbuild(X, Y, params, [false true true]);
m5pprint(model);
```

```
Synthetic variables:
z1 = x1
z2 = x2
z3 = 1, if x3 is in {0, 1, 3} else = 0
z4 = 1, if x3 is in {1, 3} else = 0
z5 = 1, if x3 is in {3} else = 0
The decision rules:
if z5 == 0 and z3 == 0 then y = -0.0039976 -2.0036*z1 (28)
if z5 == 1 then y = 2.9962 (24)
if z2 == 0 and z4 == 1 then y = -0.0012792 (15)
if z4 == 0 then y = -0.0074507 +1.0195*z1 (19)
y = 1.0004 (14)
Number of rules: 5
Number of original input variables used: 3 (x1, x2, x3)
```

These decision rules actually perfectly capture the function that generated our dataset.

Let's evaluate performance of this M5'Rules configuration on the data using 10-fold Cross-Validation.

```
rng(1);
results = m5pcv(X, Y, params, [false true true])
```

```
results =
       MAE: 0.0161
       MSE: 3.9249e-04
      RMSE: 0.0196
     RRMSE: 0.0156
        R2: 0.9997
    nRules: 5
     nVars: 3
```

We can see that for our dataset on average this method produces fewer rules than there were in model trees above while the predictive performance is similar.

3.2. Growing ensembles of trees

For this example we will use Housing dataset available at the UCI repository (http://archive.ics.uci.edu/ml/). The data has 506 observations and 13 input variables. One input variable is binary, all others are continuous.

We will grow a Random Forest for this data.

First, we must create a configuration for growing individual trees in the ensemble. We will create a configuration for ensembles of regression trees: the minimum number of observations a node must have to be considered for splitting will be 5, the minimum number of training observations a leaf node may represent will be 1, the trees will not be pruned, no smoothing will be applied, and we will set `splitThreshold` equal to 1E-6.

```
params = m5pparams(false, 1, 5, false, 0, 1E-6);
```

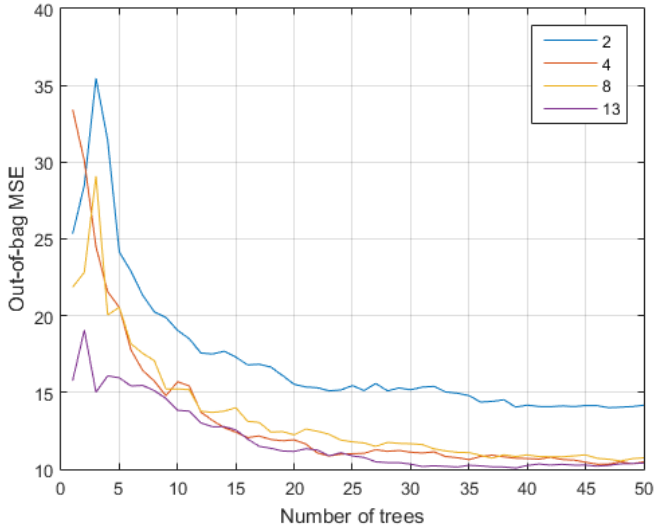
Next, we must create a configuration for growing the ensemble. This is done using function `m5pparamsensemble`. The default parameters in this function are already prepared to grow Random Forests but let's try to find a better value for `numVarsTry` (this is the number of input variables randomly sampled as candidates at each split while growing a tree). By default the value is -1 which means that `m5pbuild` will automatically set it to one third of the number of input variables (typical for Random Forests in regression problems). For our data, the value is floor(13 / 3) = 4. But let's try also a value twice as high and half as low (as suggested by Breiman, 2002) as well as all variables (which is the value for Bagging), i.e., 2, 4, 8, and 13. Of course, alternatively, we could instead try different values for `minLeafSize` (together with `minParentSize`), `maxDepth`, or compare some other configurations.

For faster processing, we will grow only 50 trees. Later, when the "best" value is found, we will grow a bigger ensemble.

We will also enable `getOOBError` because we need out-of-bag error estimates and disable `getVarImportance` and `getOOBContrib` because we don't yet need those.

An ensemble is grown by calling `m5pbuild`. We will supply `isBinCat` vector indicating that one variable is binary and the rest are continuous. By supplying the fifth argument to `m5pbuild` (`paramsEnsemble`), we are indicating that an ensemble should be created instead of just one tree.

```
paramsEnsemble = m5pparamsensemble(50, [], [], [], [], true, 0, false);
isBinCat = [false(1,3) true false(1,9)];
numVarsTry = [2 4 8 13];
figure; hold on;
for i = 1:4
    paramsEnsemble.numVarsTry = numVarsTry(i);
    [~, ~, ensembleResults] = m5pbuild(X, Y, params, isBinCat, paramsEnsemble);
    plot(ensembleResults.OOBError(:,1));
end
grid on;
xlabel('Number of trees');
ylabel('Out-of-bag MSE');
legend({'2' '4' '8' '13'}, 'Location', 'NorthEast');
```

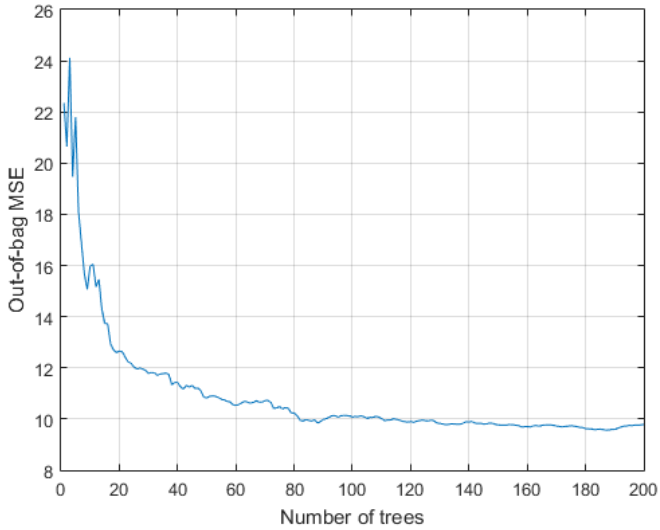


We can see that curves for values 4, 8, and 13 are very similar and are better than curve for 2. So let's say we choose `numVarsTry` to be the default, 4. Now let's build a larger ensemble consisting of say 200 trees and, while we're at it, use it to estimate input variable importance (leave `getVarImportance` to its default, 1) and compute input variable contributions (leave `getOOBContrib` to its default, `true`).

```
paramsEnsemble = m5pparamsensemble(200);
[model, time, ensembleResults] = m5pbuild(X, Y, params, isBinCat, paramsEnsemble);
```

Now we can inspect the out-of-bag error curve again.

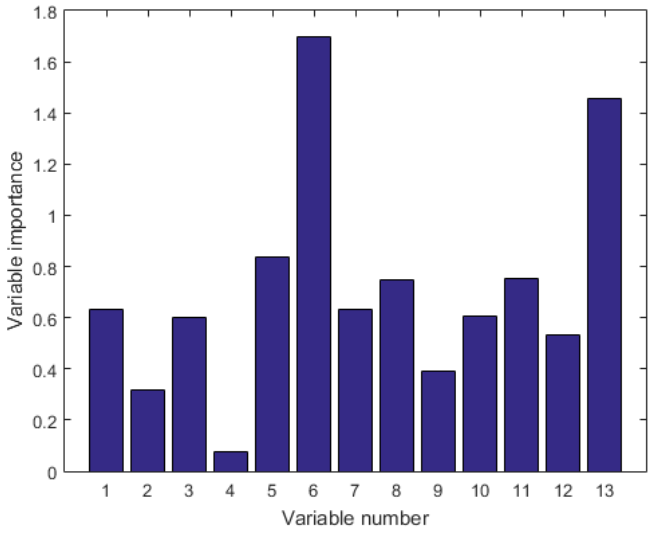
```
figure;
plot(ensembleResults.OOBError(:,1));
grid on;
xlabel('Number of trees');
ylabel('Out-of-bag MSE');
```



We can see that the prediction error estimate becomes quite stable. The values in last row of `ensembleResults.OOBError` show us that the ensemble of 200 trees estimates its prediction error to be MSE = 9.8.

Now let's plot variable importances. For that we will use the third and the forth row of `ensembleResults.varImportance`. The third row is the average increase of out-of-bag MSE when out-of-bag data of a variable is permuted. The fourth row is standard deviation of the average increase of the MSE. The importance estimate is often calculated by dividing the increase by its standard deviation. Bigger values then indicate bigger importance of the corresponding variable (we could also express these values as percent of the maximum importance).

```
figure;
bar(ensembleResults.varImportance(3,:) ./ ensembleResults.varImportance(4,:));
xlabel('Variable number');
ylabel('Variable importance');
```



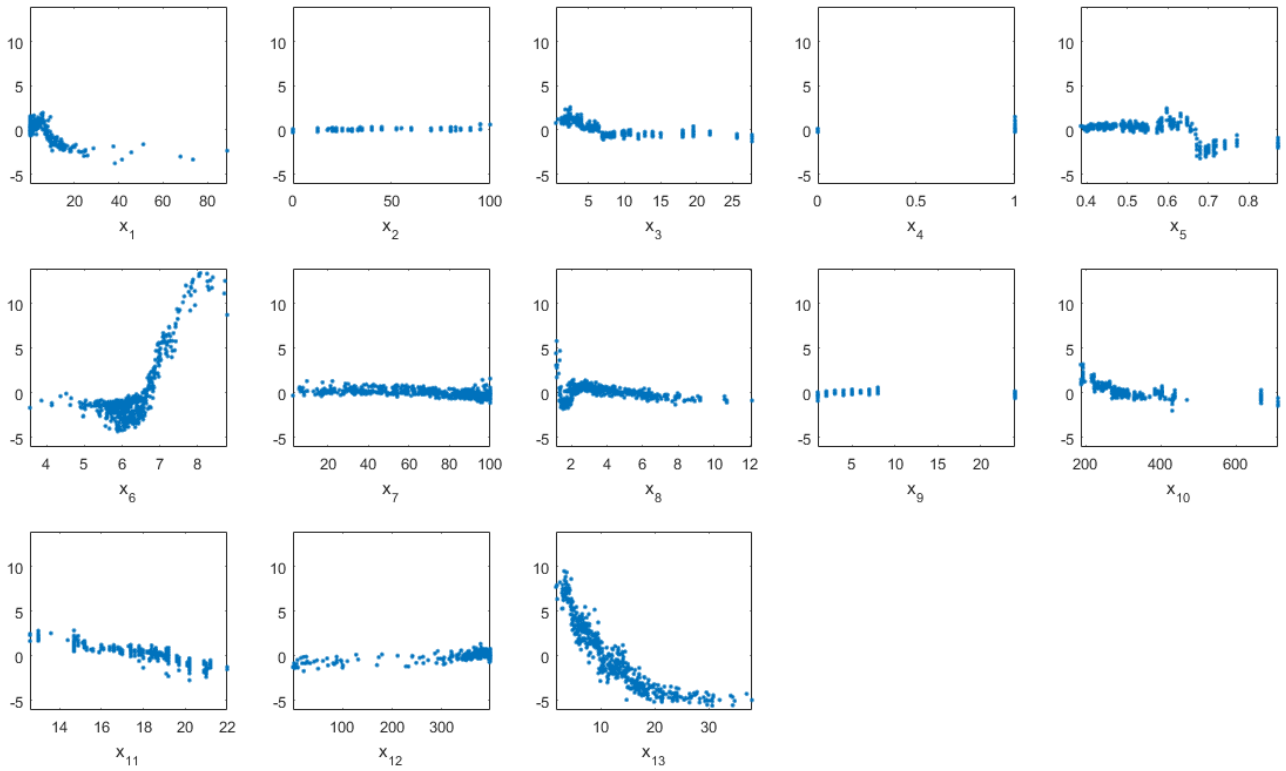
We can see that the 6th and 13th variables are estimated to be the most important ones while the 2nd and 4th are estimated to be the least important ones.

Now, let's take a look at input variable contributions to predicted responses, i.e., by how much each variable "pulled" responses away from their means. This can be done by making Forest Floor main effect plots (Welling et a., 2016).

```
figure;
contrib = ensembleResults.OOBContrib;
cminmax = [min(min(contrib(:,1:(end-1))))-0.5 max(max(contrib(:,1:(end-1))))+0.5];
for i = 1 : size(X,2)
    subplot(3,5,i);
    scatter(X(:,i), contrib(:,i), 50, '.');
    ylim(cminmax); xlim([min(X(:,i)) max(X(:,i))]);
    xlabel(['x_{' num2str(i) '}']); box on;
end
```

Now we can see not only that the 6th and 13th variables are very important but also how their contributions change depending on their values.

Note that similarly one can also make Forest Floor interaction plots (Welling et a., 2016).



While `ensembleResults.OOBContrib` allows finding out input variable contributions only for out-of-bag data from the training, computing contributions for any data can be done using function `m5ppredict`.

```
[Yq, contrib] = m5ppredict(model, [0.1 45 3 0 0.5 6.7 30 7 5 400 15 390 5]);
fprintf('Prediction: %f\n', Yq(1));
fprintf('In-bag mean: %f\n', contrib(1,end));
fprintf('Input variable contributions:\n');
[~, idx] = sort(abs(contrib(1,1:end-1)), 'descend');
for i = idx
    fprintf('x%d: %f\n', i, contrib(1,i));
end
```

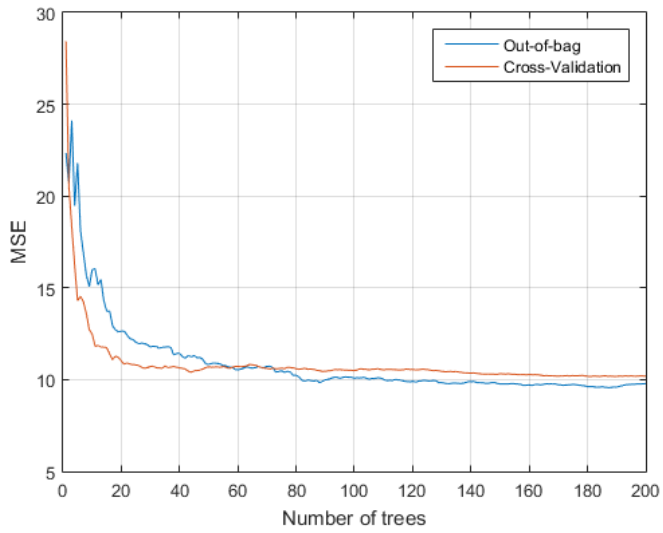
```
Prediction: 29.800000
In-bag mean: 22.547387
Input variable contributions:
x13: 3.970270
x3: 1.942738
x11: 1.666809
x6: -1.169694
x8: -0.657689
x5: 0.604105
x10: -0.459337
x7: 0.172198
x1: 0.157650
x2: 0.140998
x9: 0.044719
x12: -0.021819
x4: -0.016336
```

We can see that, even though in general the 6th variable is among the two most important ones, in this particular case, the 13th, 3rd, and 11th variables contributed to the response more than the 6th. We can also see that they all contributed to the increase of response value, while the 6th variable contributed to the decrease.

Finally, if needed, we can also evaluate the predictive performance of the ensemble using Cross-Validation or a separate test data set. This is done using `m5pcv` and `m5ptest`. The output arguments of both these functions contain matrices with errors calculated at each ensemble size (as well as, in case of `m5pcv`, at each fold).

```
rng(1);
resultsCV = m5pcv(X, Y, params, isBinCat, 10, [], [], paramsEnsemble);

figure;
plot(ensembleResults.OOBError(:,1));
hold on;
plot(resultsCV.MSE);
grid on;
xlabel('Number of trees');
ylabel('MSE');
legend({'Out-of-bag' 'Cross-Validation'}, 'Location', 'NorthEast');
```



We can see that in our case Cross-Validation gives us error estimate that is very similar to the previously calculated out-of-bag estimate.

4. REFERENCES

1. Breiman L. Bagging predictors. Machine Learning 24 (2), 1996, pp. 123-140.
2. Breiman L. Random forests. Machine Learning, 45 (1), 2001, pp. 5-32.
3. Breiman L. Manual on setting up, using, and understanding random forests v4.0. Statistics Department University of California Berkeley, CA, USA, 2002
4. Geurts P., Ernst D., Wehenkel L. Extremely randomized trees. Machine Learning 63 (1), 2006, pp. 3-42.
5. Hall M., Frank E., Holmes G., Pfahringer B., Reutemann P., Witten I. H. The WEKA data mining software: an update, SIGKDD Explorations, 11 (1), 2009
6. Holmes G., Hall M., Frank E. Generating rule sets from model trees. 12th Australian Joint Conference on Artificial Intelligence, 1999, pp. 1-12.
7. Kuz'min V. E., Polishchuk P. G., Artemenko A. G., Andronati S. A. Interpretation of qsar models based on random forest methods. Molecular Informatics, 30 (6-7), 2011, pp. 593-603.
8. Palczewska A., Palczewski J., Robinson R. M., Neagu D. Interpreting random forest models using a feature contribution method. Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on Information Reuse and Integration, San Francisco, CA, USA, IEEE, 2013, pp. 112-119.
9. Quinlan J. R. Learning with continuous classes. Proceedings of 5th Australian Joint Conference on Artificial Intelligence, World Scientific, Singapore, 1992, pp. 343-348.
10. Saabas A., Interpreting random forests, 2014/2015, http://blog.datadive.net/interpreting-random-forests/ and http://blog.datadive.net/random-forest-interpretation-with-scikit-learn/ (accessed August 5, 2016)
11. Wang Y. & Witten I. H. Induction of model trees for predicting continuous classes. Proceedings of the 9th European Conference on Machine Learning Poster Papers, Prague, 1997, pp. 128-137.
12. Welling S. H., Refsgaard H. H. F., Brockhoff P. B., Clemmensen L. H. Forest Floor Visualizations of Random Forests. ArXiv e-prints, July 2016