Regularized Greedy Forest in C++ Version 1.2: User Guide

1 Introduction

Regularized greedy forest (RGF) [2] is a tree ensemble learning method. The purpose of this document is to guide you through the software package of RGF implementation. [2] should be referred to in order to understand RGF. With the code provided in the package, you can do the following:
Using the stand-alone executable:

- Conduct RGF training for regression tasks and binary classification tasks.
- Apply trained models to new data.

It is possible to call RGF functions from your code, but documentation for this purpose is not included in this guide.

This guide is organized as follows. Section 2 quickly goes through the functionality. Data formats are described in Section 3. Section 4 explains all the parameters.

2 Get started: A quick tour of functionality

The first thing to do is to download the package, extract the content, and create the executable. Please follow the online instructions and `README`.

The top directory of the extracted content is `rgf1.2`. Make sure that the executable `rgf` (or `rgf.exe` on Windows) is at `rgf1.2/bin`. To go through the examples in this section, always set the current directory to `rgf1.2/test` because the resource files configured for using sample toy data assume that the current directory is `rgf1.2/test`. For the same reason, the path expressions in this section should be understood to be relative to `rgf1.2/test` unless it begins with `rgf1.2`.

2.1 Get started: `train` – training

To get started, set the current directory to `rgf1.2/test` and enter in the command line:

```
perl  call_exe.pl  ../bin/rgf  train  sample/train
```

If successful, the last several lines of the screen should look like:

```
Generated 5 model file(s):
output/sample.model-01
output/sample.model-02
output/sample.model-03
output/sample.model-04
output/sample.model-05

Sat Dec 10 12:17:39 2011:  Done ...
elapsed:  0.115
```

What happened is: the Perl script `call_exe.pl` read parameters in the configuration file `sample/train.inp` and passed them to the executable `../bin/rgf` along with the action to take (`train`); as a result, training was conducted and five models were saved to files. Here is the configuration file `sample/train.inp` that `call_exe.pl` was given:

```
#### sample input to "train" ####
train_x_fn=sample/train.data.x  # Training data points
train_y_fn=sample/train.data.y  # Training targets
model_fn_prefix=output/sample.model
reg_L2=1                        # Regularization parameter
algorithm=RGF
loss=LS                         # Square loss
test_interval=100               # Save models every time 100 leaves are added.
max_leaf_forest=500             # Stop training when #leaf reaches 500
Verbose
```

It essentially says: conduct training with the training data points in `sample/train.data.x` and the training targets in `sample/train.data.y` and save the trained models to the files. Any texts from "`#`" to the end of line are comments. The line "`model_fn_prefix...`" indicates that the system should generate model path names using the string "`output/sample.model`" as a prefix and attaching sequential numbers "`-01`", "`-02`", $\cdots$ to it. It also says: training should proceed until the number of leaf nodes in the forest reaches 500 (`max_leaf_forest=500`); and every time another 100 leaf nodes are added to the forest, the system should simulate the end of training and save the model for later testing (`test_interval=100`). As a result, you should obtain five models each of which contains approximately 100, 200, 300, 400, or 500 leaf nodes. We delay explanation of other parameters until Section 4 where complete lists of parameters are given. The format of training data files is described in Section 3.1.

2.2 Models: Why one call of training results in several model files

As seen above, one call of RGF training typically results in several model files. This fact may deserve some explanation as it differs from typical boosting methods.

For example, suppose that you conducted Gradient Boosting [1] training using regression trees as base learners; stopped training when you obtained 500 trees; and saved them to a file for later testing. Then in principle, using these 500 trees, you can test 500 distinct models, each of which consists of the first k trees where $k = 1, 2, \cdots, 500$, by simply changing the number of trees to be used for making predictions. In principle, you do not have to save 500 models individually. This is because Gradient Boosting does not change the previously-generated trees; it only adds new trees as it proceeds. The same can be said about other typical boosting methods such as AdaBoost.

By contrast, RGF performs *fully-corrective update of weights*, which updates the weights of *all* the leaf nodes of *all* the trees, in the designated interval and at the end of training. For this reason, if we save the model of, for example, 500 trees, then these 500 trees can be used *only* for testing the additive model of 500 trees. Unlike the Gradient Boosting example above, the first k trees of these 500 trees do not constitute a meaningful model. If we stopped training when k trees were obtained, the weights assigned to the nodes of the k trees would be totally different from those of the first k trees of the 500 trees.

It might be simpler if the system let a user only specify when to stop training and only return one model, but it would be very inefficient to train several models of different sizes this way. For efficiency, our implementation trains several models of different sizes in one call by *simulating the end of training* in the interval designated by `test_interval`. More precisely, training branches into two in the designated interval. One continues training as if nothing happened, and the other ends training, which triggers weight optimization (if it has not been triggered by the designated optimization interval), and tests or saves the model. That is how one call of training produces several models.

2.3 `predict`: To apply a model to new data

The next example reads a model from one of the five model files generated in Section 2.1 and applies the model to new data. Set the current directory to `rgf1.2/test` and enter:

```
perl  call_exe.pl  ../bin/rgf  predict  sample/predict
```

If successful, after parameters are displayed, something similar to the following should be displayed:

```
output/sample.pred:  output/sample.model-03,#leaf=301,#tree=73
Sat Dec 10 13:20:54 2011:  Done ...
```

which indicates that the prediction values were saved to `output/sample.pred`; the model was read from the file `output/sample.model-03` and it contained 301 leaf nodes and 73 trees.

The configuration file `sample/predict.inp` we used is:

```
#### sample input to "predict"
test_x_fn=sample/test.data.x       # Test data points
model_fn=output/sample.model-03    # Model file
prediction_fn=output/sample.pred   # Where to write prediction values
```

It says: read the model from `output/sample.model-03`; apply it to the data points in `sample/test.data.x`; and save the prediction values to `output/sample.pred`. The format of the prediction file is described in Section 3.3.

2.4 Executable `rgf` and Perl script `call_exe.pl`

The executable `rgf`, called through the Perl script in the examples above, takes two arguments:

`rgf` *action parameters*

action	`train | predict | train_test | train_predict`	
	`train`	Conduct training and save the trained models to files. Input: training data; Output: models.
	`predict`	Apply a model to new data. Input: a model and test data; Output: predictions
	`train_test`	Train and test the models in one call. Input: training data and test data; Output: performance results. Optional output: models.
	`train_predict`	Train and apply the models to new data in one call. Input: training data and test data; Output: predictions, model information, and models.
parameters	Parameters are in the form of: *keyword1=value1*, *keyword2=value2*, *Option1*,$\cdots$ Example: `algorithm=RGF,train_x_fn=data.x,train_y_fn=data.y,...`	

Although what is done by `train_test` or `train_predict` can also be done by combining `train` and `predict`, use of `train_test` or `train_predict` has advantages in some situations as discussed in Sections 2.5 and 2.6.

To get help on parameters, call `rgf` with *action* but without *parameters*, for example:

```
rgf train
rgf predict
```

Since parameters could be long and tedious to type in, the Perl script `call_exe.pl` introduced above is provided to ease the job. It essentially reads parameters from a configuration file and concatenates them with delimiter "`,`" to pass to `rgf`. The syntax is as follows:

`perl call_exe.pl` *executable action config_pathname*

executable	Typically, `../bin/rgf`, i.e., `rgf1.2/bin/rgf`.
action	`train | predict | train_test | train_predict`
config_pathname	Path name to the configuration file without extension. The extension of configuration files must be "`.inp`".

In the configuration files, any text from "`#`" to the end of line is considered to be a comment.

Additionally, `call_exe.pl` provides an interface to perform several runs in one call with one configuration file. This is convenient, for example, for testing different degrees of regularization with other parameters fixed. `sample/regress_train_test.inp` provides a self-explaining example.

2.5 `train_test`: train, apply, and evaluate models

`train_test` performs training and test in one call. What `train_test` does can also be done by combining `train` and `predict` and writing an evaluation routine by yourself. One advantage of `train_test` other than convenience is that it can save disk space by not having to write the models to files.

To try the example configuration for `train_test`, set the current directory to `rgf1.2/test`, and enter:

```
perl  call_exe.pl  ../bin/rgf  train_test  sample/train_test
```

If successful, the last several lines of the screen should look like:

```
Generated 5 model file(s):
output/m-01
output/m-02
output/m-03
output/m-04
output/m-05

Sat Dec 10 10:17:50 2011:  Done ...
elapsed:  0.135
```

The configuration file `sample/train_test.inp` is:

```
#### sample input to "train_test" ####
train_x_fn=sample/train.data.x  # Training data points
train_y_fn=sample/train.data.y  # Training targest
test_x_fn=sample/test.data.x    # Test data points
test_y_fn=sample/test.data.y    # Test targest
evaluation_fn=output/sample.evaluation
                                # Where to write evaluation results
model_fn_prefix=output/m        # Save models.  This is optional.
algorithm=RGF
reg_L2=1                        # Regularization parameter
loss=LS                         # Square loss
test_interval=100               # Test at every 100 leaves
max_leaf_forest=500             # Stop training when 500 leaves are added
Verbose
```

It is mostly the same as the configuration file for `train` in Section 2.1 except that test data is specified by `test_x_fn` (data points) and `test_y_fn` (targets) and `evaluation_fn` indicates where the performance evaluation results should be written. In this example, model files are saved to files, as `model_fn_prefix` is specified. If `model_fn_prefix` is omitted, the models are not saved.

Now check the evaluation file (`output/sample.evaluation`) that was just generated. It should look like the following except that the items following `cfg` are omitted here:

```
#tree,29,#leaf,100,acc,0.61,rmse,0.9886,sqerr,0.9773,#test,100,cfg,...,output/m-01
#tree,52,#leaf,200,acc,0.66,rmse,0.9757,sqerr,0.952,#test,100,cfg,...,output/m-02
#tree,73,#leaf,301,acc,0.66,rmse,0.9824,sqerr,0.9651,#test,100,cfg,...,output/m-03
#tree,94,#leaf,400,acc,0.69,rmse,0.9767,sqerr,0.9539,#test,100,cfg,...,output/m-04
#tree,115,#leaf,501,acc,0.67,rmse,0.985,sqerr,0.9702,#test,100,cfg,...,output/m-05
```

Five lines indicate that five models were trained and tested. For example, the first line says: a model with 29 trees and 100 leaf nodes was applied to 100 data points and classification accuracy was found to be 61%, and the model was saved to `output/m-01`.

The evaluation file format is described in Section 3.5. The format of training data and test data files is described in Section 3.1.

2.6 `train_predict`: train and apply the models and save predictions

The primary function of `train_predict` is to perform training; apply the trained models to test data; and write predictions to files. What `train_predict` does can also be done by combining `train` and predict. One advantage of train_predict is that it can save disk space by not having to write model files. (Typically, predictions take up much less disk space than models.) In particular, train_predict can be used for one-vs-all training for multi-class categorization, whereas train_test cannot since testing (evaluation) of predictions has to wait until training of all the K one-vs-all models for K classes is done.

Note that by default all the models are written to files, and to save disk space as discussed above, the option switch SaveLastModelOnly needs to be turned on. With this switch on, only the last (and largest) model is written to the file to enable future warm-start. (Warm-start resumes training from where the training stopped before, which is also explained in 4.3.1.)

Model information such as sizes is also written to files. The original purpose is to save information that would be disposed of otherwise with SaveLastModelOnly on. But for simplicity, train_predict always generates model information files irrespective of on/off of the switch. The provided sample configuration file for train_predict, sample/train_predict.inp is as follows.

```
#### sample input to "train_predict" ####
train_x_fn=sample/train.data.x  # Training data points
train_y_fn=sample/train.data.y  # Training targets
test_x_fn=sample/test.data.x    # Test data points
model_fn_prefix=output/m
SaveLastModelOnly               # Only the last (largest) model will be saved.
:
test_interval=100               # Test every time 100 leaves are added.
max_leaf_forest=500             # Stop training when #leaf reaches 500
```

In this example, the model path names will be output/m-01, $\cdots$, output/m-05, but the only last one output/m-05 is actually written to the file, as SaveLastModelOnly is turned on. The path names for saving the predictions and model information are generated by attaching .pred and .info to the model path names, respectively. Therefore, after entering the following in the command line,

```
perl  call_exe.pl  ../bin/rgf  train_predict  sample/train_predict
```

we should have the following 11 files at the output directory:

- Five prediction files: m-01.pred, $\cdots$, m-05.pred
- Five model information files m-01.info, $\cdots$, m-05.info
- One model file m-05

The data format is described in Section 3.1.

3 Input/output file format

This section describes the format of input/output files.

3.1 Data file format

3.1.1 Data points

The data points (or feature vectors) should be given in a plain text file of the following format. Each line represents one data point. In each line, values should be separated by one or more white space characters.

All the lines should have exactly the same number of values. The values should be in the format that is recognized as valid floating-point number expressions by `atof` of C libraries. The following example represents three data points of five dimensions.

```
0.3     -0.5  1  0    2
1.555   0     0  2.8  0
0       0     0  3    0
```

(NOTE) Currently, there is no support for categorical values. All the values must be numbers. This means that categorical attributes, if any, need to be converted to indicator vectors in advance.

Alternative data format for sparse data points For *sparse* data which has many zero components (e.g., bag-of-word data), the following format can be used instead. The first line should be "sparse *d*" where *d* is the feature dimensionality. Starting from the second line, each line represents one data point. In each line, non-zero components should be specified as *feature#*:*value* where *feature#* begins from 0 and goes up to $d-1$. For example, the three data points above can be expressed as:

```
sparse    5
0 : 0.3    1 : -0.5   2 : 1   4 : 2
0 : 1.555  3 : 2.8
3 : 3
```

3.1.2 Targets

The target values should be given in a plain text file of the following format. Each line contains the target value of one data point. The order must be in sync with the data point file. If the data is for the classification task, the values must be in $\{1, -1\}$, for example:

```
+1
-1
-1
```

If paired with the data point file example above, this means that the target value of the first data point $[0.3, -0.5, 1, 0, 2]$ is 1 and the target value of the second data point $[1.555, 0, 0, 2.8, 0]$ is -1, and so on.

For regression tasks, the target values could be any real values, for example:

```
0.35
1.23
-0.0028
```

3.2 Data point weight file format

As introduced later, training optionally takes the user-specified weights of data points as input. The data point weights should be given in a plain text file of the same format as the target file. That is, each line should contain the user-specified weight of one data point, and the order must be in sync with the data point file of training data.

3.3 Prediction file format

`predict` and `train_predict` output prediction values to a file. The prediction file is a plain text file that contains one prediction value per line. The order of the values is in sync with the data point file of test data.

3.4 Model information file format

`train_predict` outputs model information to files. The model information file is a plain text file that has one line, for example:

```
#tree,378,#leaf,5000,sign,-___-_RGF_,cfg,reg_L2=0.1;loss=LS...
```

This example means that the model consists of 378 trees and 5000 leaf nodes; and the model was trained with RGF with the parameter setting following "`cfg`".

3.5 Evaluation file format

`train_test` outputs performance evaluation results to a file in the CSV format. Here is an example:

```
#tree,115,#leaf,500,acc,0.64,rmse,0.9802,sqerr,0.9607,#test,100,cfg,...
#tree,213,#leaf,1000,acc,0.65,rmse,0.9721,sqerr,0.945,#test,100,cfg,...
```

In the evaluation file each line represents the evaluation results of one model. In each line, each value is preceded by its descriptor; e.g., "`#tree,115`" indicates that the number of trees is 115 in the tested model. In the following, y_i and p_i are the target value and prediction value of the i-th data point, respectively; $\mathcal{I}(x)$ is the indicator function so that $\mathcal{I}(x) = 1$ if x is true and 0 otherwise; and m is the number of test data points.

Descriptor	Meaning
`#tree`	Number of trees in the model
`#leaf`	Number of leaf nodes in the model
`acc`	Accuracy regarding the task as a classification task. $\sum_{i=1}^{m} \mathcal{I}(y_i \cdot p_i > 0)/m$.
`rmse`	RMSE regarding the task as a regression task. $\sqrt{\sum_{i=1}^{m}(y_i - p_i)^2/m}$
`sqerr`	Square error. RMSE$\times$RMSE.
`#test`	Number of test data points m.
`cfg`	Some of training parameters.

In addition, if models were saved to files, the last item of each line will be the model path name.

(NOTE) Although performances are shown in several metrics, depending on the task some are obviously meaningless and should be ignored, e.g., accuracy should be ignored on the regression task; RMSE and square error should be ignored on the classification task especially when exponential loss is used.

3.6 Model files

The model files generated by `train` or `train_test` are binary files. Caution is needed *if* you wish to share model files between the environments with different *endianness*. By default the code assumes *little-endian*. To share model files between environments with different endians the executable used in the *big-endian* environment needs to be compiled in a certain way; see `README` for detail.

4 Parameters

4.1 Overview of RGF training

Since many of the parameters are for controlling training, let us first give a brief overview of RGF training, focusing on the things that can be controlled via parameters. [2] should be referred to for more precise and complete definition.

Suppose that we are given n training data points $\mathbf{x}_1, \cdots, \mathbf{x}_n$ and targets $y_1, \cdots, y_n$. The additive model obtained by RGF training is in the form of: $h_{\mathcal{F}}(\mathbf{x}) = \sum_v \alpha_v \cdot g_v(\mathbf{x})$, where v goes through all the leaf nodes in the forest $\mathcal{F}$, $g_v(\mathbf{x})$ is the *basis function* associated with node v, and α_v is its *weight* or coefficient. Initially, we have an empty forest with $h_{\mathcal{F}}(\mathbf{x}) = 0$. As training proceeds, the forest $\mathcal{F}$ obtains more and more nodes so the model $h_{\mathcal{F}}(\mathbf{x})$ obtains more and more basis functions. The training objective of RGF is to find the model that minimizes regularized loss, which is the sum of loss and a regularization penalty term:

$$\frac{1}{n}\sum_{i=1}^{n} \ell(h_{\mathcal{F}}(\mathbf{x}_i), y_i) + \mathcal{G}(\mathcal{F}) , \tag{1}$$

where ℓ is a loss function; and $\mathcal{G}(\mathcal{F})$ is the regularization penalty term. RGF grows the forest with greedy search so that regularized loss is minimized, while it performs fully-corrective update of weights to minimize the regularized loss in the designated interval. The loss function ℓ and the interval of weight optimization can be specified by parameters.

There are three methods of regularization discussed in [2]. One is L_2 regularization on leaf-only models in which the regularization penalty term $\mathcal{G}(\mathcal{F})$ is:

$$\lambda \cdot \sum_v \alpha_v^2/2 ,$$

where λ is a constant. This is equivalent to standard L_2 regularization and penalizes larger weights. The other two are called *min-penalty regularizers*. Their definition of the regularization penalty term over each tree is in the form of:

$$\lambda \cdot \min_{\{\beta_v\}} \left\{ \sum_v \gamma^{d_v} \beta_v^2/2 \;:\; \text{some conditions on } \{\beta_v\} \right\} ,$$

where d_v is the depth of node v; and λ and γ are constants. While [2] should be consulted for precise definition of min-penalty regularizers, one thing to note here is that a larger $\gamma > 1$ penalizes deeper nodes (corresponding to more complex basis functions) more severely. Parameters are provided to choose the regularizer or to specify the degree of regularization through λ or γ.

Optionally, it is possible to have different L_2 regularization parameters for the process of growing a forest and the process of weight correction.

On the regression tasks, it is sensible to normalize targets so that the average becomes zero since regularization shrinks weights towards zero. An option switch `NormalizeTarget` is provided for this purpose. When it is turned on, the model is fitted to the normalized targets $[y_i - \bar{y}]_{i=1}^n$ where $\bar{y} = \sum_{i=1}^n y_i/n$ and the final model is set to $h_{\mathcal{F}}(\mathbf{x}) + \bar{y}$.

The regularized loss in (1) can be customized not only by specifying a loss function but also by specifying user-defined weights. Let $w_i > 0$ be the user-defined weight assigned to the i-th data point. Then instead of (1) the system will minimize the following:

$$\frac{1}{\sum_{i=1}^n w_i}\sum_{i=1}^{n} w_i \cdot \ell(h_{\mathcal{F}}(\mathbf{x}_i), y_i) + \mathcal{G}(\mathcal{F}) . \tag{2}$$

Finally, in our implementation, fully-corrective weight update is done by coordinate descent as follows:

```
for $j = 1$ to $J$ do
    foreach leaf node $v$ do
        // Update $\alpha_v$ by the Newton step with step size $\eta$ to minimize regularized loss $\mathcal{Q}$.
        $\alpha_v \leftarrow \alpha_v - \eta \cdot \frac{\partial \mathcal{Q}/\partial \delta_v|_{\delta_v=0}}{\partial^2 \mathcal{Q}/\partial \delta_v^2|_{\delta_v=0}}$    // $\delta_v$ is the additive change to $\alpha_v$.
    end
end
```

The constants J and η above can be changed using the parameters described below, although, in our experiments, we never used them as the default values worked fine on a number of datasets.

4.2 Naming conventions and notation

There are two types of parameters: keyword-value pairs and options. The keywords begin with a lower-case letter and should be followed by =*value*, e.g., "`loss=LS`". The options begin with an upper-case letter, e.g., "`Verbose`", and indicate to turn on corresponding option switches, which are off by default.

In the lists below, '`*`' in front of the keyword indicates that the designated keyword-value pair is required and cannot be omitted.

4.3 Parameters for `train`

The parameters described in this section are used by the `train` function. `train` trains models and saves them to files.

4.3.1 Parameters to control input and output for `train`

A pair of training data files (one contains the data points and the other contains the target values) are required. Another required parameter is the prefix of model path names, which is used to generate model path names by attaching to it sequential numbers "-01", "-02", and so on. The reason why one call of training typically produces multiple model files is explained in Section 2.2.

Optionally, training can be resumed from the point where training was ended last time, which we call *warm-start*. To do warm-start, the model file from which training should be resumed needs to be specified. Also optionally, user-defined weights of training data points can be specified through `train_w_fn`. They are used as in (2).

Required parameters to control input and output for `train`		
*	`train_x_fn=`	Path to the data point file of training data.
*	`train_y_fn=`	Path to the target file of training data.
*	`model_fn_prefix=`	To save models to files, path names are generated by attaching "-01", "-02",$\cdots$ to this value.
Optional parameters to control input and output for `train`		
	`train_w_fn`	Path to the file of user-defined weights assigned to training data points.
	`model_fn_for_warmstart=`	Path to the model file from which training should do warm-start.

4.3.2 Parameters to control training

In the list below, the first group of parameters are most important in the sense that they would affect either accuracy of the models or speed of training directly, and they were actually used in the experiments reported in [2]. The second group of parameters never needed to be specified in our experiments, as the default values worked fine on a number of datasets, but they may be useful in some situations. The third group is for displaying information and specifying the memory allocation policy.

The variables below refer to the corresponding variables in the overview in Section 4.1.

Parameters to control training		
	`algorithm=`	`RGF|RGF_Opt|RGF_Sib` (Default: `RGF`) `RGF`: RGF with L_2 regularization on leaf-only models `RGF_Opt`: RGF with min-penalty regularization `RGF_Sib`: RGF with min-penalty regularization with the sum-to-zero sibling constraints.
	`loss=`	Loss function $\ell(p, y)$. `LS|Expo|Log` (Default: `LS`) `LS`: square loss $(p - y)^2/2$; `Expo`: exponential loss $\exp(-py)$; `Log`: logistic loss $\log(1 + \exp(-py))$
	`max_leaf_forest=`	Training will be terminated when the number of leaf nodes in the forest reaches this value. It should be large enough so that a good model can be obtained at some point of training, whereas a smaller value makes training time shorter. Appropriate values are data-dependent and in [2] varied from 1000 to 10000. (Default:10000)
	`NormalizeTarget`	If turned on, training targets are normalized so that the average becomes zero. It was turned on in all the regression experiments in [2].
*	`reg_L2=`	λ. Used to control the degree of L_2 regularization. Crucial for good performance. Appropriate values are data-dependent. Either 1, 0.1, or 0.01 often produces good results though with exponential loss (`loss=Expo`) and logistic loss (`loss=Log`) some data requires smaller values such as 1e-10 or 1e-20.
	`reg_sL2=`	λ_g. Override L_2 regularization parameter λ for the process of growing the forest. That is, if specified, the weight correction process uses λ and the forest growing process uses λ_g. If omitted, no override takes place and λ is used throughout training. On some data, $\lambda/100$ works well.
	`reg_depth=`	γ. Must be no smaller than 1. Meant for being used with `algorithm=RGF_Opt|RGF_Sib`. A larger value penalizes deeper nodes more severely. (Default: 1)
	`test_interval=`	Test interval in terms of the number of leaf nodes. For example, if `test_interval=500`, every time 500 leaf nodes are newly added to the forest, end of training is simulated and the model is tested or saved for later testing. For efficiency, it must be either multiple or divisor of the optimization interval (`opt_interval`: default 100). If not, it may be changed by the system automatically. (Default:500)
Parameters that are probably rarely used		
	`min_pop=`	Minimum number of training data points in each leaf node. Smaller values may slow down training. Too large values may degrade model accuracy. (Default:10)

`num_iteration_opt=`	J. Number of iterations of coordinate descent to optimize weights. (Default:10 for square loss; 5 for exponential loss and the likes)
`num_tree_search=`	Number of trees to be searched for the nodes to split. The most recently-grown trees are searched first. (Default:1)
`opt_interval=`	Weight optimization interval in terms of the number of leaf nodes. For example, if `opt_interval=100`, weight optimization is performed every time approximately 100 leaf nodes are newly added to the forest. (Default:100)
`opt_stepsize=`	η. Step size of Newton updates used in coordinate descent to optimize weights. (Default:0.5)
Other parameters	
`Verbose`	Print information during training.
`Time`	Measure and display elapsed time for node search and weight optimization.
`memory_policy=`	`Conservative|Generous`. (Default:`Generous`)

4.4 Parameters for `predict`

`predict` reads a model saved by `train`, `train_test`, or `train_predict`, applies it to new data, and saves prediction values to a file.

Parameters for `predict`	
* `test_x_fn`	Path to the data point file of test data.
* `model_fn`	Path to the model file.
* `prediction_fn`	Path to the prediction file to write prediction values to.

4.5 Parameters for `train_test`

`train_test` trains models with training data and evaluates them on test data in one call.

4.5.1 Parameters to control input and output for `train_test`

`train_test` requires a pair of training data files (one contains the data points and the other contains the target values) and a pair of test data files.

Optionally, the models can be saved to files by specifying `model_fn_prefix`. The value specified with `model_fn_prefix` is used to generate model path names by attaching to it sequential numbers “-01”, “-02”, and so on. The reason why one call of training typically produces multiple model files is explained in Section 2.2. If `SaveLastModelOnly` is turned on, only the last (and largest) model will be saved, which enables warm-start later on. Other things that can be done optionally are the same as `train`. That is, optionally, training can be resumed from the point where training was ended last time (*warm-start*). Also optionally, user-defined weights of training data points can be specified through `train_w_fn`; see Section 4.1 for how they are used.

	Parameters to control input and output for `train_test`	
*	`train_x_fn=`	Path to the data point file of training data.
*	`train_y_fn=`	Path to the target file of training data.
*	`test_x_fn=`	Path to the data point file of test data.
*	`test_y_fn=`	Path to the target file of test data.
	`evaluation_fn`	Path to the file to write performance evaluation results to. If omitted, the results are written to stdout.
	`Append_evaluation`	Open the file to write evaluation results to with the append mode.
	`model_fn_prefix=`	If omitted, the models are not saved to files. Model path names are generated by attaching ”-01”, ”-02”,... to this value to save models.
	`train_w_fn`	Path to the file of user-defined weights assigned to training data points.
	`model_fn_for_warmstart=`	Path to the input model file from which training should do warm-start.

4.5.2 Parameters to control training

The parameters to control training for `train_test` are the same as those for `train`; see Section 4.3.2.

4.6 Parameters for `train_predict`

`train_predict` trains models with training data; applies the models to test data; and saves the obtained predictions and model information to files in one call. Model files are also saved to files, but whether all the models should be saved or only the last one is obtional.

4.6.1 Parameters to control input and output for `train_predict`

`train_predict` requires a pair of training data files (one contains the data points and the other contains the target values) and a test data file that has data points. The target values of test data is not required.

The value specified with `model_fn_prefix` is used to generate model path names by attaching to it sequential numbers “-01”, “-02”, and so on. The reason why one call of training typically produces multiple model files is explained in Section 2.2. To write predictions to files, the path names are generated by attaching “`.pred`” to the corresponding model path names.

When the `SaveLastModelOnly` switch is turned on, only the last (and largest) model is written to a file. This option is useful for reducing the amount of disk space needed while enabling warm-start later on. See Section 2.6 for more on the situations `train_predict` is suitable.

Information on models such as sizes are also written to files, and their path names are generated by attaching “`.info`” to the model path names. The original purpose is to save information that would be disposed of otherwise with `SaveLastModelOnly` on. But for simplicity, `train_predict` always generates model information files irrespective of on/off of `SaveLastModelOnly`.

Parameters to control input and output for `train_predict`		
*	`train_x_fn=`	Path to the data point file of training data.
*	`train_y_fn=`	Path to the target file of training data.
*	`test_x_fn=`	Path to the data point file of test data.
*	`model_fn_prefix=`	Model path names are generated by attaching "-01", "-02",... to this value to save models. Prediction path names and model information path names are generated by attaching "`.pred`" and "`.info`" to the model path names, respectively.
	`SaveLastModelOnly`	If turned on, only the last model is saved to the file.
	`train_w_fn`	Path to the file of user-defined weights assigned to training data points.
	`model_fn_for_warmstart=`	Path to the input model file from which training should do warm-start.

4.6.2 Parameters to control training

The parameters to control training for `train_predict` are the same as those for `train`; see Section 4.3.2.

References

[1] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29, 2001.

[2] Rie Johnson and Tong Zhang. Learning nonlinear functions using regularized greedy forest. Technical report, Tech Report: arXiv:1109.0887, 2011.