Random Forests, Boosting, and Stacking

ACM

```
library(printr)
library(randomForest)
library(tidyverse)
library(gbm)
library(caret)
library(MASS)
options(repos = c(CRAN = "https://cran.rstudio.com/"))
library(printr)
```

Data set: PimaIndiansDiabetes2, used in one of your homeworks, for predicting the probability of being diabetes positive based on multiple clinical variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

Random Forests

```
?randomForest
```

```
Classification and Regression with Random Forest

Description:

     'randomForest' implements Breiman's random forest algorithm (based
     on Breiman and Cutler's original Fortran code) for classification
     and regression.  It can also be used in unsupervised mode for
     assessing proximities among data points.

Usage:

     ## S3 method for class 'formula'
     randomForest(formula, data=NULL, ..., subset, na.action=na.fail)
     ## Default S3 method:
     randomForest(x, y=NULL,  xtest=NULL, ytest=NULL, ntree=500,
                  mtry=if (!is.null(y) && !is.factor(y))
                  max(floor(ncol(x)/3), 1) else floor(sqrt(ncol(x))),
                  weights=NULL,
                  replace=TRUE, classwt=NULL, cutoff, strata,
                  sampsize = if (replace) nrow(x) else ceiling(.632*nrow(x)),
                  nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
                  maxnodes = NULL,
                  importance=FALSE, localImp=FALSE, nPerm=1,
                  proximity, oob.prox=proximity,
                  norm.votes=TRUE, do.trace=FALSE,
```

```
                keep.forest=!is.null(y) && is.null(xtest), corr.bias=FALSE,
                keep.inbag=FALSE, ...)
     ## S3 method for class 'randomForest'
     print(x, ...)
```

Arguments:

data	an optional data frame containing the variables in the model. By default the variables are taken from the environment which 'randomForest' is called from.
subset	an index vector indicating which rows should be used. (NOTE: If given, this argument must be named.)
na.action	A function to specify the action to be taken if NAs are found. (NOTE: If given, this argument must be named.)
x, formula	a data frame or a matrix of predictors, or a formula describing the model to be fitted (for the 'print' method, an 'randomForest' object).
y	A response vector. If a factor, classification is assumed, otherwise regression is assumed. If omitted, 'randomForest' will run in unsupervised mode.
xtest	a data frame or matrix (like 'x') containing predictors for the test set.
ytest	response for the test set.
ntree	Number of trees to grow. This should not be set to too small a number, to ensure that every input row gets predicted at least a few times.
mtry	Number of variables randomly sampled as candidates at each split. Note that the default values are different for classification (sqrt(p) where p is number of variables in 'x') and regression (p/3)
weights	A vector of length same as 'y' that are positive weights used only in sampling data to grow each tree (not used in any other calculation)
replace	Should sampling of cases be done with or without replacement?
classwt	Priors of the classes. Need not add up to one. Ignored for regression.
cutoff	(Classification only) A vector of length equal to number of classes. The `winning' class for an observation is the one with the maximum ratio of proportion of votes to cutoff. Default is 1/k where k is the number of classes (i.e., majority vote wins).

strata: A (factor) variable that is used for stratified sampling.

sampsize: Size(s) of sample to draw. For classification, if sampsize is a vector of the length the number of strata, then sampling is stratified by strata, and the elements of sampsize indicate the numbers to be drawn from the strata.

nodesize: Minimum size of terminal nodes. Setting this number larger causes smaller trees to be grown (and thus take less time). Note that the default values are different for classification (1) and regression (5).

maxnodes: Maximum number of terminal nodes trees in the forest can have. If not given, trees are grown to the maximum possible (subject to limits by 'nodesize'). If set larger than maximum possible, a warning is issued.

importance: Should importance of predictors be assessed?

localImp: Should casewise importance measure be computed? (Setting this to 'TRUE' will override 'importance'.)

nPerm: Number of times the OOB data are permuted per tree for assessing variable importance. Number larger than 1 gives slightly more stable estimate, but not very effective. Currently only implemented for regression.

proximity: Should proximity measure among the rows be calculated?

oob.prox: Should proximity be calculated only on ``out-of-bag'' data?

norm.votes: If 'TRUE' (default), the final result of votes are expressed as fractions. If 'FALSE', raw vote counts are returned (useful for combining results from different runs). Ignored for regression.

do.trace: If set to 'TRUE', give a more verbose output as 'randomForest' is run. If set to some integer, then running output is printed for every 'do.trace' trees.

keep.forest: If set to 'FALSE', the forest will not be retained in the output object. If 'xtest' is given, defaults to 'FALSE'.

corr.bias: perform bias correction for regression? Note: Experimental. Use at your own risk.

keep.inbag: Should an 'n' by 'ntree' matrix be returned that keeps track of which samples are ``in-bag'' in which trees (but not how many times, if sampling with replacement)

...: optional parameters to be passed to the low level function 'randomForest.default'.

```r
# Load the data and remove NAs
PimaIndiansDiabetes3 <- read_csv("pima.indians.diabetes3.csv")
```

```
Rows: 532 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): class
dbl (8): npregnant, glucose, diastolic.bp, skinfold.thickness, bmi, pedigree...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
Pima <- na.omit(PimaIndiansDiabetes3) %>%
  mutate(diabetes = factor(class)) %>%
  dplyr::select(-classdigit, -class)
# Inspect the data
sample_n(Pima, 3)
```

npregnant	glucose	diastolic.bp	skinfold.thickness	bmi	pedigree	age	diabetes
1	109	60	8	25.4	0.947	21	normal
3	173	82	48	38.4	2.137	25	diabetic
2	122	52	43	36.2	0.816	28	normal

```r
# Split the data into training and test set
set.seed(123)
training.samples <- sample(1:nrow(Pima), floor(nrow(Pima)*0.8) )
train.data  <- Pima[training.samples, ]
test.data <- Pima[-training.samples, ]
```

```r
num_vars <- 1:7
tst.acc <- NULL
for(m in num_vars){
  set.seed(2)
  RF_Pima=randomForest(diabetes ~., data = train.data,
                       ntree=5000, mtry=m)
  yhat.oob = RF_Pima$predicted
  tst.acc <- c(tst.acc,
               mean(yhat.oob==train.data$diabetes))
}
data.frame(num_vars,tst.acc)
```

num_vars	tst.acc
1	0.7741176
2	0.7858824
3	0.7811765
4	0.7788235
5	0.7788235
6	0.7835294
7	0.7764706

```
M <- num_vars[which.max(tst.acc)]
```

The oob samples gave an $M = 2$ splits per tree as best with an error of 0.786.

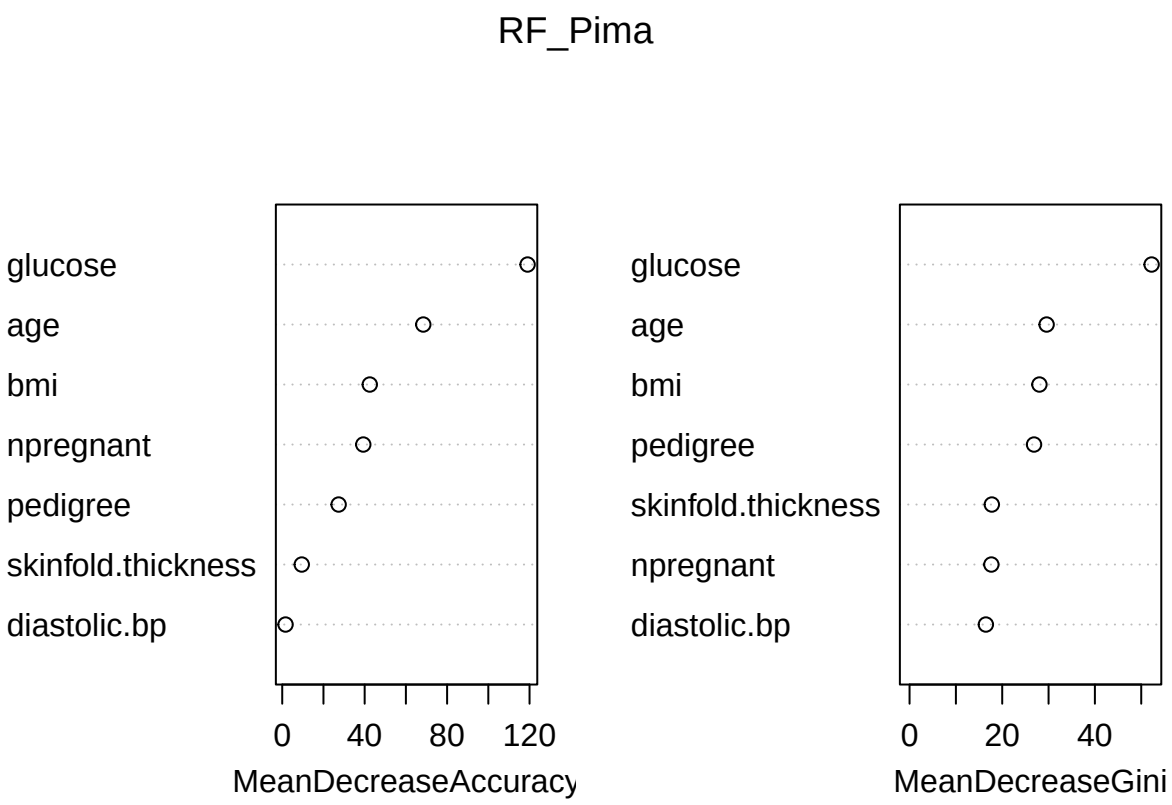
```
RF_Pima = randomForest(diabetes ~., data = train.data, ntree=5000,
                       mtry = M, importance=TRUE)
 yhat.bag = predict( RF_Pima, newdata = test.data)
 mean(yhat.bag==test.data$diabetes)
```

```
## [1] 0.7757009
```

```
importance(RF_Pima)
```

	diabetic	normal	MeanDecreaseAccuracy	MeanDecreaseGini
npregnant	6.7672373	41.968529	39.323869	17.65604
glucose	95.3698582	86.685480	119.060177	52.24134
diastolic.bp	0.5046798	1.666763	1.602164	16.47065
skinfold.thickness	0.5091445	10.889772	9.460268	17.75373
bmi	38.1559555	22.599573	42.483979	28.03013
pedigree	21.3278192	18.575821	27.369628	26.86745
age	37.6645215	55.391440	68.440583	29.57959

```
varImpPlot(RF_Pima)
```



Fitting Regression Trees

Here we'll fit regression trees to the Boston housing dataset.

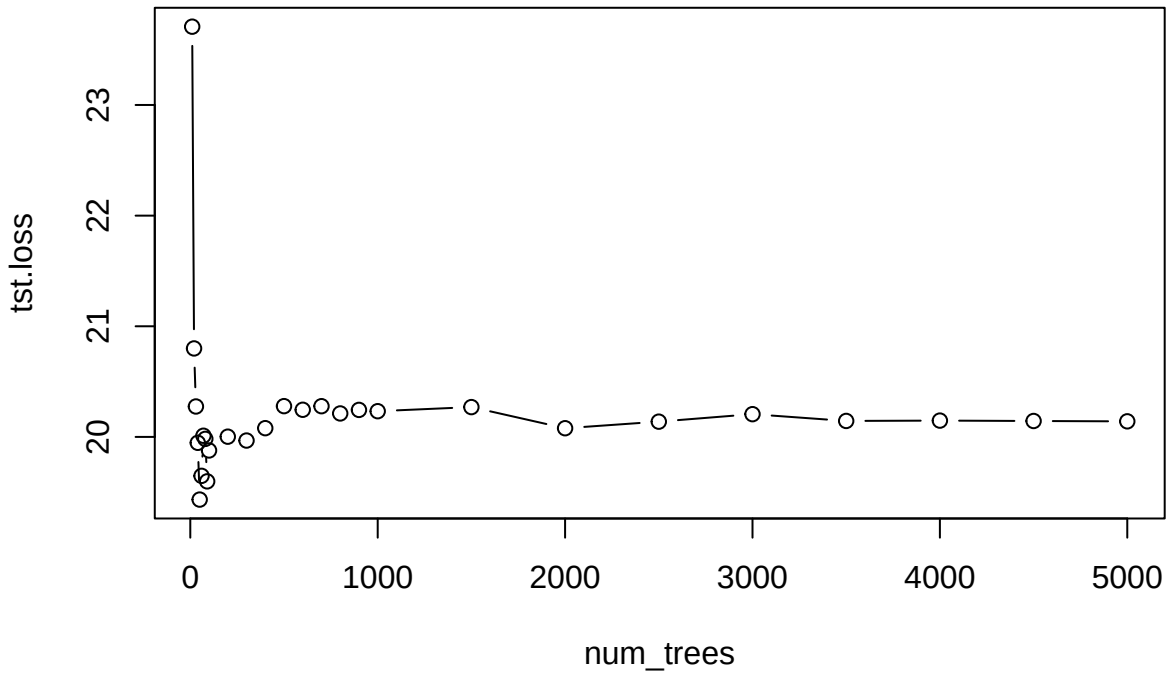
```
?Boston
```

```
Housing Values in Suburbs of Boston

Description:

     The 'Boston' data frame has 506 rows and 14 columns.

Usage:

     Boston
```

```
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)

num_vars <- 1:10
tst.loss <- NULL
for(m in num_vars){
  set.seed(1)
  bag.boston=randomForest(medv~.,data=Boston, subset = train,
                          mtry = m, ntree = 1000)
  yhat.bag = predict(bag.boston,newdata=Boston[-train,])
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
}
data.frame(num_vars,tst.loss)
```

num_vars	tst.loss
1	25.84646
2	19.57372
3	18.58931
4	18.72469
5	19.03545
6	19.37815
7	20.23230
8	20.77523
9	21.37730
10	22.16682

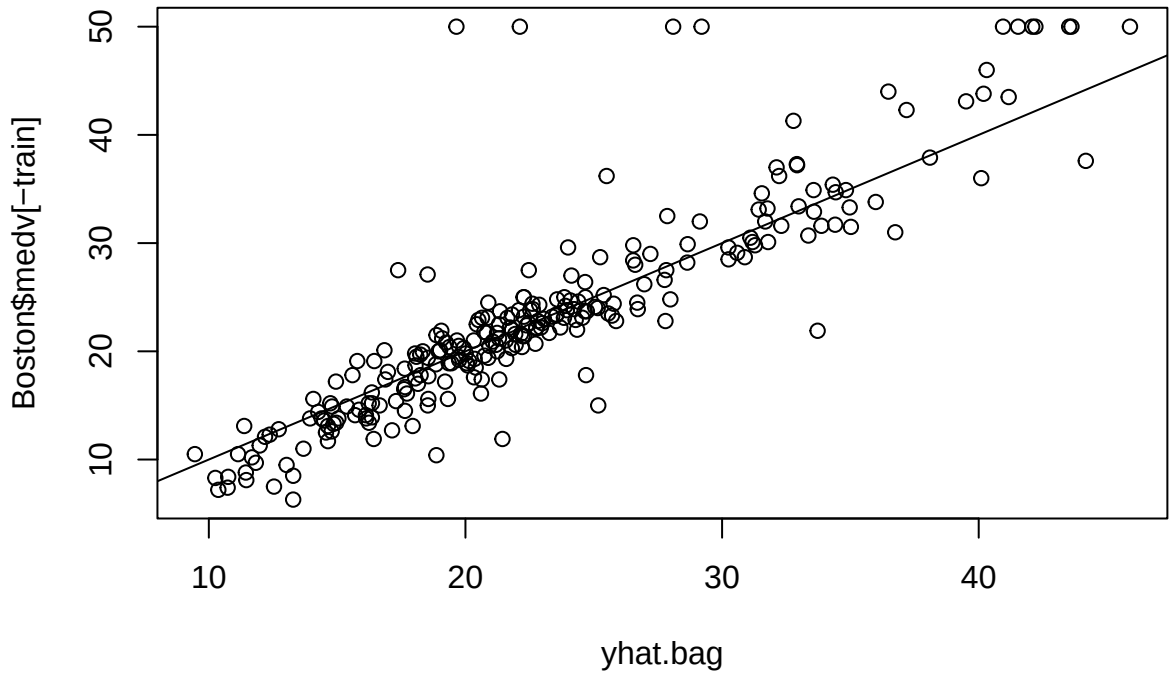
```
M <- num_vars[ which.min(tst.loss)]

num_trees <- c(seq(10,90,10), seq(100,1000,100), seq(1500, 5000, 500))
tst.loss <- NULL
for(m in num_trees){
  set.seed(1)
  bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=7,ntree=m)
  yhat.bag = predict(bag.boston,newdata=Boston[-train,])
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
}
plot(num_trees,tst.loss,type = "b")
```




```
T <- num_trees[ which.min(tst.loss)]


set.seed(1)
bag.boston=randomForest(medv~.,data=Boston,subset=train,
                        mtry=M, ntree=T, importance=TRUE)
bag.boston
```

```
Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = M, ntree = T,      importance = TRUE, subset = t
               Type of random forest: regression
                     Number of trees: 50
No. of variables tried at each split: 3

          Mean of squared residuals: 11.91747
                    % Var explained: 84.5
```

```
yhat.bag = predict(bag.boston,newdata=Boston[-train,])
plot(yhat.bag, Boston$medv[-train])
abline(0,1)
```



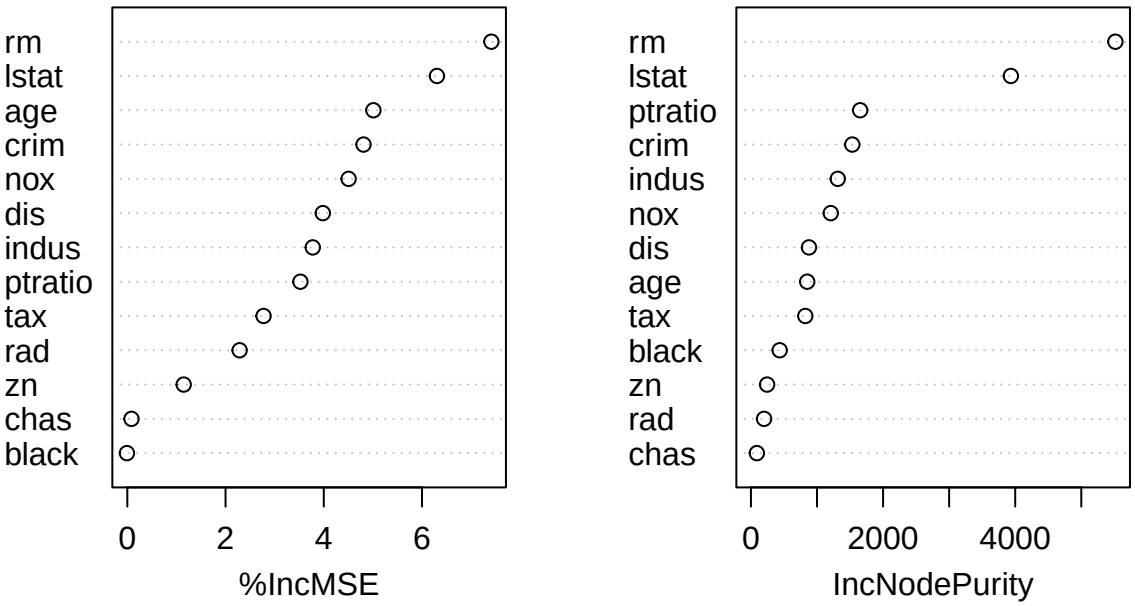

```
mean((yhat.bag-Boston$medv[-train])^2)
```

```
[1] 19.87878
```

```
importance(bag.boston)
```

	%IncMSE	IncNodePurity
crim	4.8090615	1533.67818
zn	1.1476379	244.71330
indus	3.7756917	1314.17840
chas	0.0845674	89.61804
nox	4.5055961	1208.35732
rm	7.4123996	5516.08528
age	5.0102894	851.67309
dis	3.9807567	878.43951
rad	2.2865442	198.88599
tax	2.7738072	822.75109
ptratio	3.5242156	1653.30955
black	-0.0083146	435.18696
lstat	6.3044846	3934.15080

```
varImpPlot(bag.boston)
```



Now let's take a look at some other RF's with different tuning parameters

```
bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=13,ntree=1000)
yhat.bag = predict(bag.boston,newdata=Boston[-train,])
mean((yhat.bag-Boston$medv[-train])^2)
```

```
## [1] 23.57299
```

Boosting

```
?gbm
```

```
Generalized Boosted Regression Modeling (GBM)

Description:

     Fits generalized boosted regression models. For technical details,
     see the vignette: 'utils::browseVignettes("gbm")'.

Usage:

     gbm(
       formula = formula(data),
       distribution = "bernoulli",
       data = list(),
       weights,
       var.monotone = NULL,
       n.trees = 100,
       interaction.depth = 1,
```

```
       n.minobsinnode = 10,
       shrinkage = 0.1,
       bag.fraction = 0.5,
       train.fraction = 1,
       cv.folds = 0,
       keep.data = TRUE,
       verbose = FALSE,
       class.stratify.cv = NULL,
       n.cores = NULL
     )

Arguments:

 formula: A symbolic description of the model to be fit. The formula
          may include an offset term (e.g. y~offset(n)+x). If
          'keep.data = FALSE' in the initial call to 'gbm' then it is
          the user's responsibility to resupply the offset to
          'gbm.more'.

distribution: Either a character string specifying the name of the
          distribution to use or a list with a component 'name'
          specifying the distribution and any additional parameters
          needed. If not specified, 'gbm' will try to guess: if the
          response has only 2 unique values, bernoulli is assumed;
          otherwise, if the response is a factor, multinomial is
          assumed; otherwise, if the response has class '"Surv"', coxph
          is assumed; otherwise, gaussian is assumed.

          Currently available options are '"gaussian"' (squared error),
          '"laplace"' (absolute loss), '"tdist"' (t-distribution loss),
          '"bernoulli"' (logistic regression for 0-1 outcomes),
          '"huberized"' (huberized hinge loss for 0-1 outcomes),
          classes), '"adaboost"' (the AdaBoost exponential loss for 0-1
          outcomes), '"poisson"' (count outcomes), '"coxph"' (right
          censored observations), '"quantile"', or '"pairwise"'
          (ranking measure using the LambdaMart algorithm).

          If quantile regression is specified, 'distribution' must be a
          list of the form 'list(name = "quantile", alpha = 0.25)'
          where 'alpha' is the quantile to estimate. The current
          version's quantile regression method does not handle
          non-constant weights and will stop.

          If '"tdist"' is specified, the default degrees of freedom is
          4 and this can be controlled by specifying 'distribution =
          list(name = "tdist", df = DF)' where 'DF' is your chosen
          degrees of freedom.

          If "pairwise" regression is specified, 'distribution' must be
          a list of the form
          'list(name="pairwise",group=...,metric=...,max.rank=...)'
          ('metric' and 'max.rank' are optional, see below). 'group' is
          a character vector with the column names of 'data' that
          jointly indicate the group an instance belongs to (typically
```

a query in Information Retrieval applications). For training, only pairs of instances from the same group and with different target labels can be considered. 'metric' is the IR measure to use, one of

list("conc") Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

: Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

list("mrr") Mean reciprocal rank of the highest-ranked positive instance

: Mean reciprocal rank of the highest-ranked positive instance

list("map") Mean average precision, a generalization of 'mrr' to multiple positive instances

: Mean average precision, a generalization of 'mrr' to multiple positive instances

list("ndcg:") Normalized discounted cumulative gain. The score is the weighted sum (DCG) of the user-supplied target values, weighted by log(rank+1), and normalized to the maximum achievable value. This is the default if the user did not specify a metric.

'ndcg' and 'conc' allow arbitrary target values, while binary targets 0,1 are expected for 'map' and 'mrr'. For 'ndcg' and 'mrr', a cut-off can be chosen using a positive integer parameter 'max.rank'. If left unspecified, all ranks are taken into account.

Note that splitting of instances into training and validation sets follows group boundaries and therefore only approximates the specified 'train.fraction' ratio (the same applies to cross-validation folds). Internally, queries are randomly shuffled before training, to avoid bias.

Weights can be used in conjunction with pairwise metrics, however it is assumed that they are constant for instances from the same group.

For details and background on the algorithm, see e.g. Burges (2010).

data: an optional data frame containing the variables in the model. By default the variables are taken from 'environment(formula)', typically the environment from which 'gbm' is called. If 'keep.data=TRUE' in the initial call to 'gbm' then 'gbm' stores a copy with the object. If 'keep.data=FALSE' then subsequent calls to 'gbm.more' must

```
          resupply the same dataset. It becomes the user's
          responsibility to resupply the same data at this point.

 weights: an optional vector of weights to be used in the fitting
          process. Must be positive but do not need to be normalized.
          If 'keep.data=FALSE' in the initial call to 'gbm' then it is
          the user's responsibility to resupply the weights to
          'gbm.more'.

var.monotone: an optional vector, the same length as the number of
          predictors, indicating which variables have a monotone
          increasing (+1), decreasing (-1), or arbitrary (0)
          relationship with the outcome.

 n.trees: Integer specifying the total number of trees to fit. This is
          equivalent to the number of iterations and the number of
          basis functions in the additive expansion. Default is 100.

interaction.depth: Integer specifying the maximum depth of each tree
          (i.e., the highest level of variable interactions allowed). A
          value of 1 implies an additive model, a value of 2 implies a
          model with up to 2-way interactions, etc. Default is 1.

n.minobsinnode: Integer specifying the minimum number of observations
          in the terminal nodes of the trees. Note that this is the
          actual number of observations, not the total weight.

shrinkage: a shrinkage parameter applied to each tree in the expansion.
          Also known as the learning rate or step-size reduction; 0.001
          to 0.1 usually work, but a smaller learning rate typically
          requires more trees. Default is 0.1.

bag.fraction: the fraction of the training set observations randomly
          selected to propose the next tree in the expansion. This
          introduces randomnesses into the model fit. If 'bag.fraction'
          < 1 then running the same model twice will result in similar
          but different fits. 'gbm' uses the R random number generator
          so 'set.seed' can ensure that the model can be reconstructed.
          Preferably, the user can save the returned 'gbm.object' using
          'save'. Default is 0.5.

train.fraction: The first 'train.fraction * nrows(data)' observations
          are used to fit the 'gbm' and the remainder are used for
          computing out-of-sample estimates of the loss function.

cv.folds: Number of cross-validation folds to perform. If 'cv.folds'>1
          then 'gbm', in addition to the usual fit, will perform a
          cross-validation, calculate an estimate of generalization
          error returned in 'cv.error'.

keep.data: a logical variable indicating whether to keep the data and
          an index of the data stored with the object. Keeping the data
          and index makes subsequent calls to 'gbm.more' faster at the
          cost of storing an extra copy of the dataset.
```

```
 verbose: Logical indicating whether or not to print out progress and
          performance indicators ('TRUE'). If this option is left
          unspecified for 'gbm.more', then it uses 'verbose' from
          'object'. Default is 'FALSE'.

class.stratify.cv: Logical indicating whether or not the
          cross-validation should be stratified by class. Defaults to
          'TRUE' for 'distribution = "multinomial"' and is only
          implemented for '"multinomial"' and '"bernoulli"'. The
          purpose of stratifying the cross-validation is to help
          avoiding situations in which training sets do not contain all
          classes.

 n.cores: The number of CPU cores to use. The cross-validation loop
          will attempt to send different CV folds off to different
          cores. If 'n.cores' is not specified by the user, it is
          guessed using the 'detectCores' function in the 'parallel'
          package. Note that the documentation for 'detectCores' makes
          clear that it is not failsafe and could return a spurious
          number of available cores.

Details:

     'gbm.fit' provides the link between R and the C++ gbm engine.
     'gbm' is a front-end to 'gbm.fit' that uses the familiar R
     modeling formulas. However, 'model.frame' is very slow if there
     are many predictor variables. For power-users with many variables
     use 'gbm.fit'. For general practice 'gbm' is preferable.

     This package implements the generalized boosted modeling
     framework. Boosting is the process of iteratively adding basis
     functions in a greedy fashion so that each additional basis
     function further reduces the selected loss function. This
     implementation closely follows Friedman's Gradient Boosting
     Machine (Friedman, 2001).

     In addition to many of the features documented in the Gradient
     Boosting Machine, 'gbm' offers additional features including the
     out-of-bag estimator for the optimal number of iterations, the
     ability to store and manipulate the resulting 'gbm' object, and a
     variety of other loss functions that had not previously had
     associated boosting algorithms, including the Cox partial
     likelihood for censored data, the poisson likelihood for count
     outcomes, and a gradient boosting implementation to minimize the
     AdaBoost exponential loss function.
```

```r
model <- train( diabetes ~. , data = train.data,
                method = "gbm",
                  distribution = "bernoulli",
                trControl = trainControl("cv", number = 10))
```

Let's take a look at the best tuning parameters:

```
model$bestTune
```

n.trees	interaction.depth	shrinkage	n.minobsinnode
50	1	0.1	10

If you actually look into the output, the "shrinkage" and "n.min" parameters were not varied. Now let's look at the test error and compare to random forests.

```
## Random forests test error
mean(yhat.bag==test.data$diabetes)
```

```
## Warning in `==.default`(yhat.bag, test.data$diabetes): longer object length is
## not a multiple of shorter object length
```

```
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple of
## shorter object length
```

```
## [1] 0
```

```
## GBM test error
predicted.classes <- model %>% predict(test.data)
mean(predicted.classes==test.data$diabetes)
```

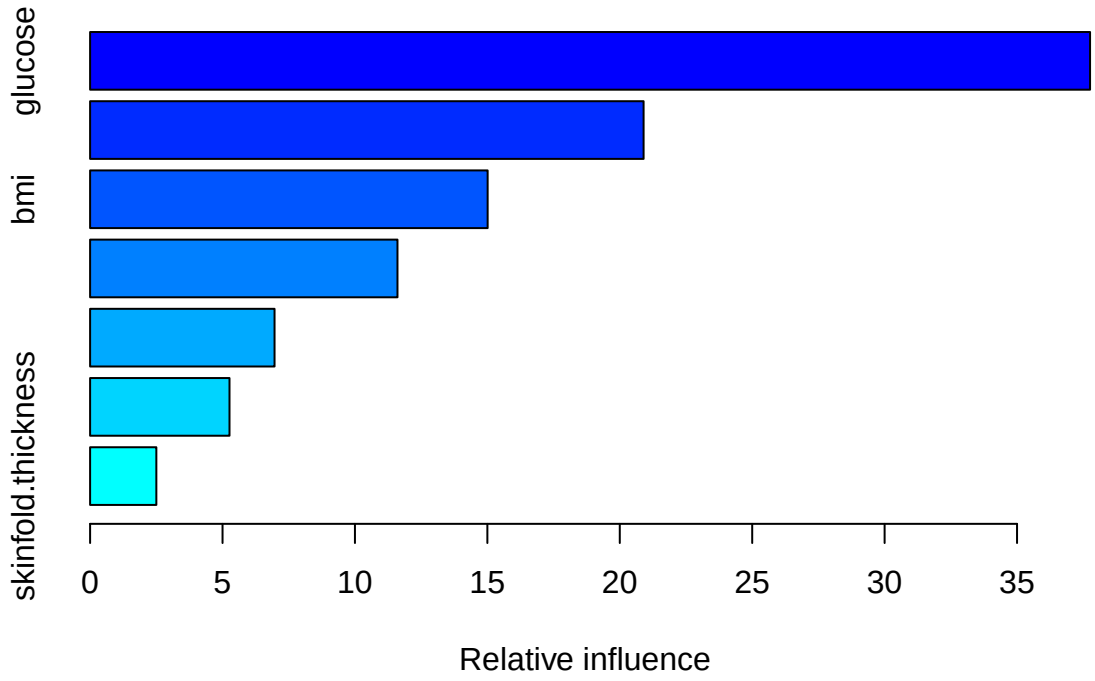
```
## [1] 0.7476636
```

Now let's check out some summaries of the model. Note that `gbm` *requires* numeric $\{0, 1\}$ outcome, while `train` (used above) *requires* a factor outcome for classification.

How would you know this? There's really no way to know unless you use these functions frequently. However, if your an experienced R user you know that outcome type is something that is often different between packages.

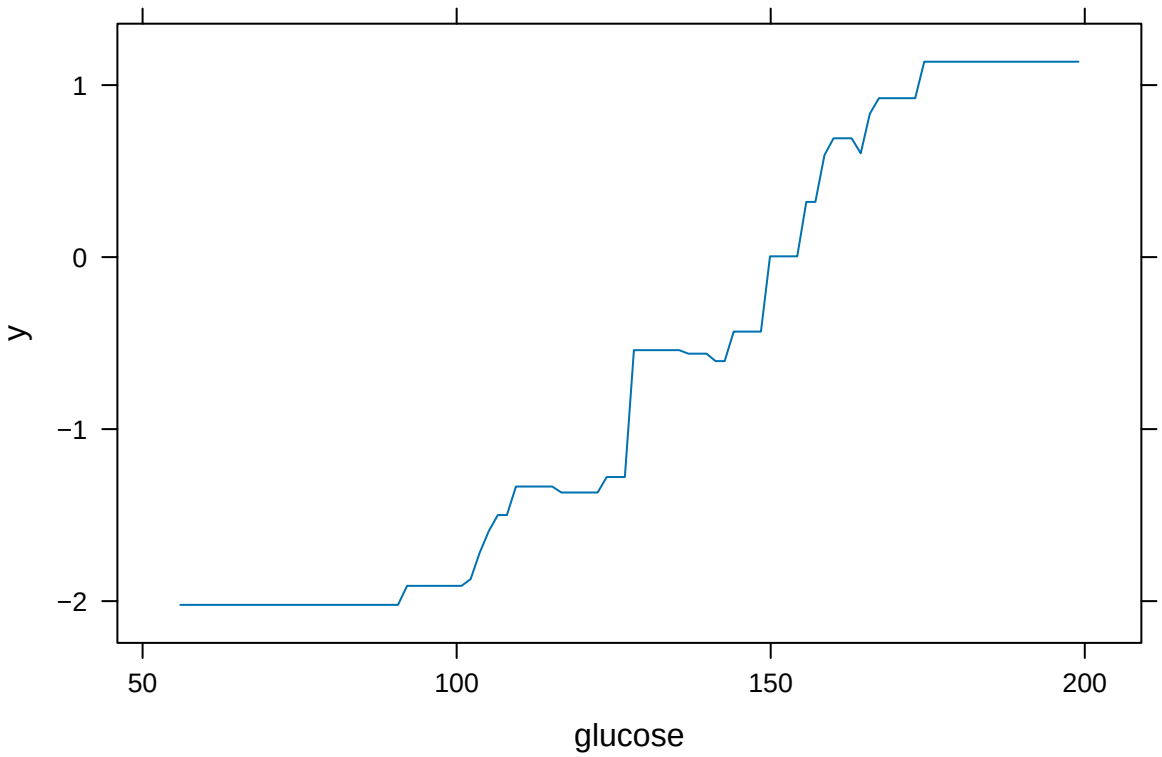
Here, I did not know that these two functions required different outcome types. When I first ran the `train` function I kept getting errors. I simply copied those errors stuck them into Google and found the solution pretty quickly.

```
Pima <- Pima %>% mutate(diabetes_num = 1*I(diabetes=="diabetic"))

boost.Pima=gbm(diabetes_num ~.-diabetes, data=Pima[training.samples,],
                  distribution = "bernoulli",
                   n.trees = 50, interaction.depth = 3,
                   shrinkage = 0.1, n.minobsinnode = 10)
summary(boost.Pima) # we could use summary(model) and get the same thing.
```

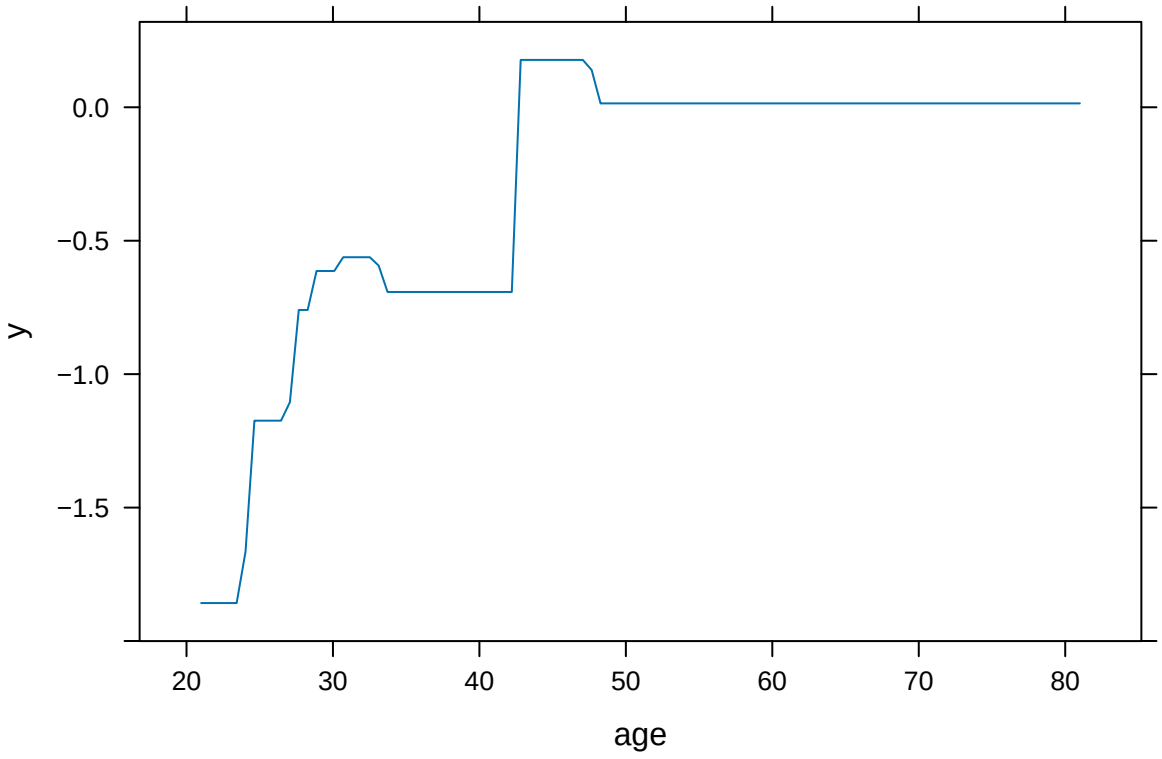


	var	rel.inf
glucose	glucose	37.756891
age	age	20.898787
bmi	bmi	15.010271
pedigree	pedigree	11.605983
npregnant	npregnant	6.964290
diastolic.bp	diastolic.bp	5.263002
skinfold.thickness	skinfold.thickness	2.500776

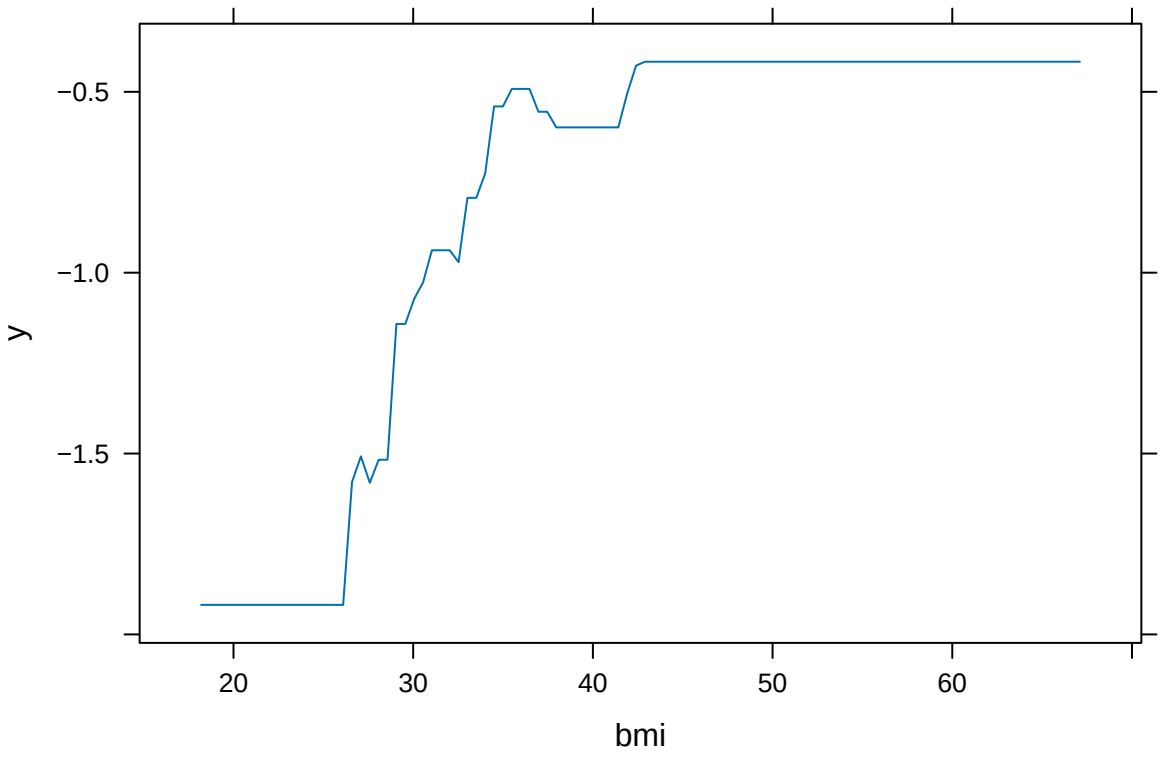
```
par(mfrow=c(2,2))
plot(boost.Pima, i="glucose")
```



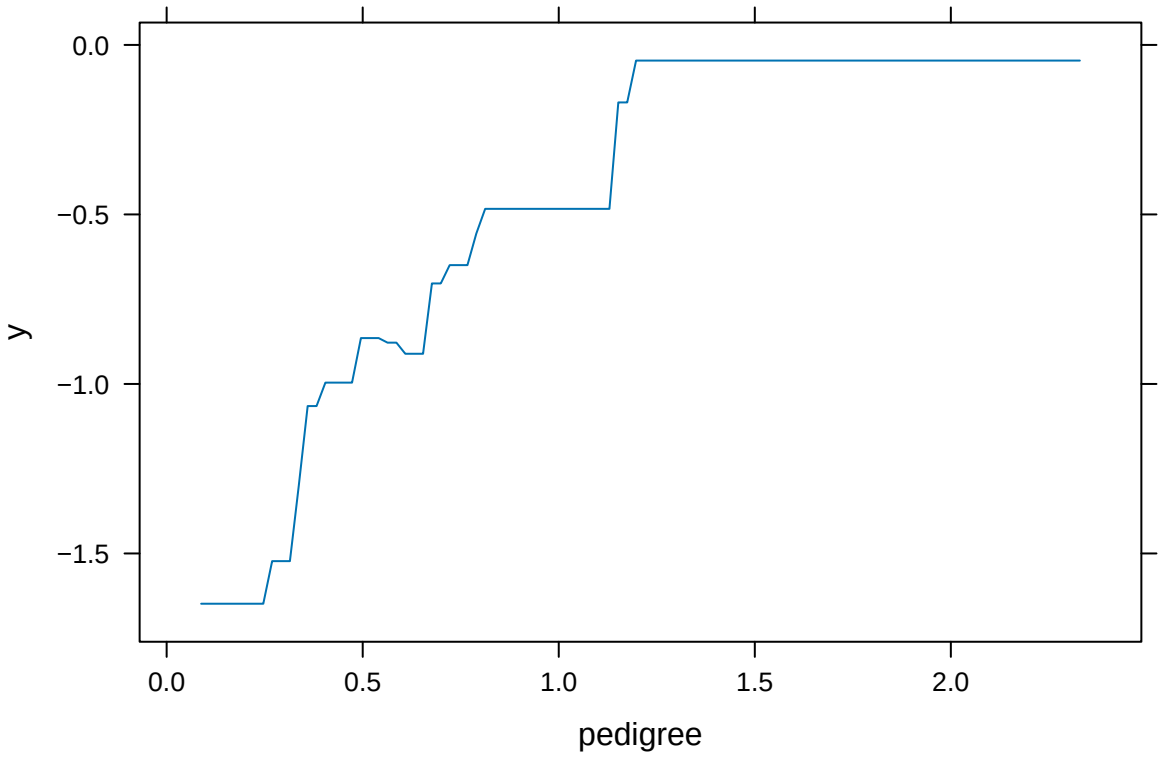

```
plot(boost.Pima, i="age")
```




```
plot(boost.Pima, i="bmi")
```



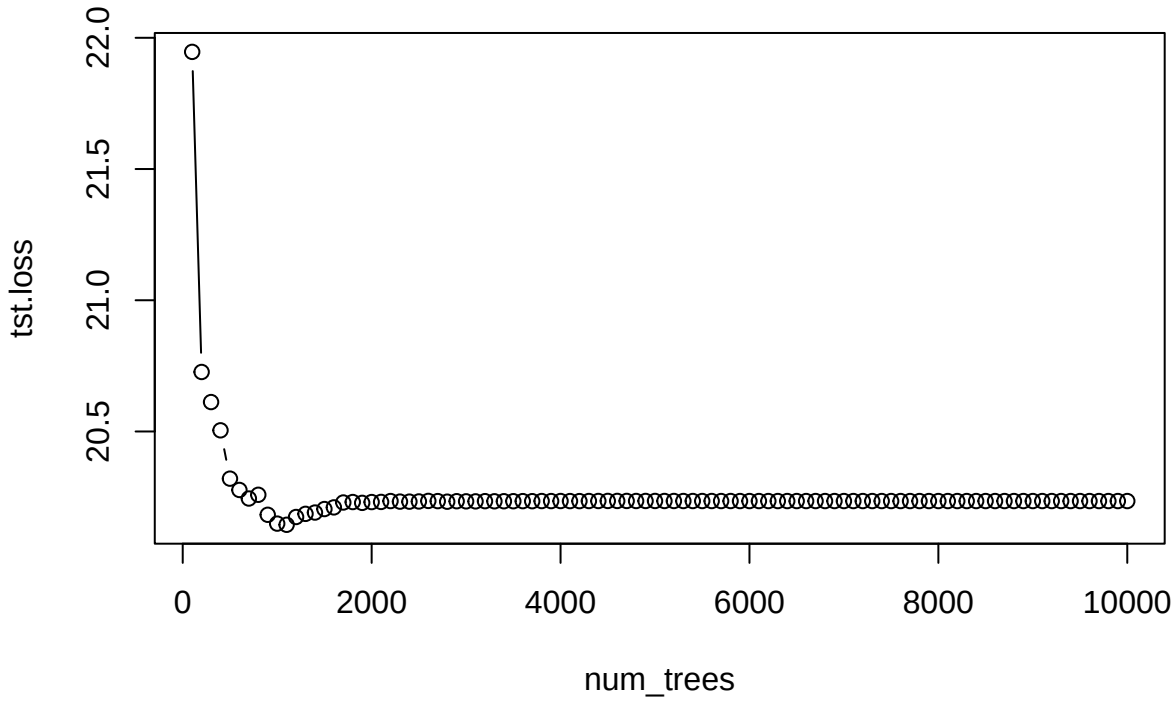

```
plot(boost.Pima, i="pedigree")
```



Now we'll look at the Boston housing dataset.

```
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)
```

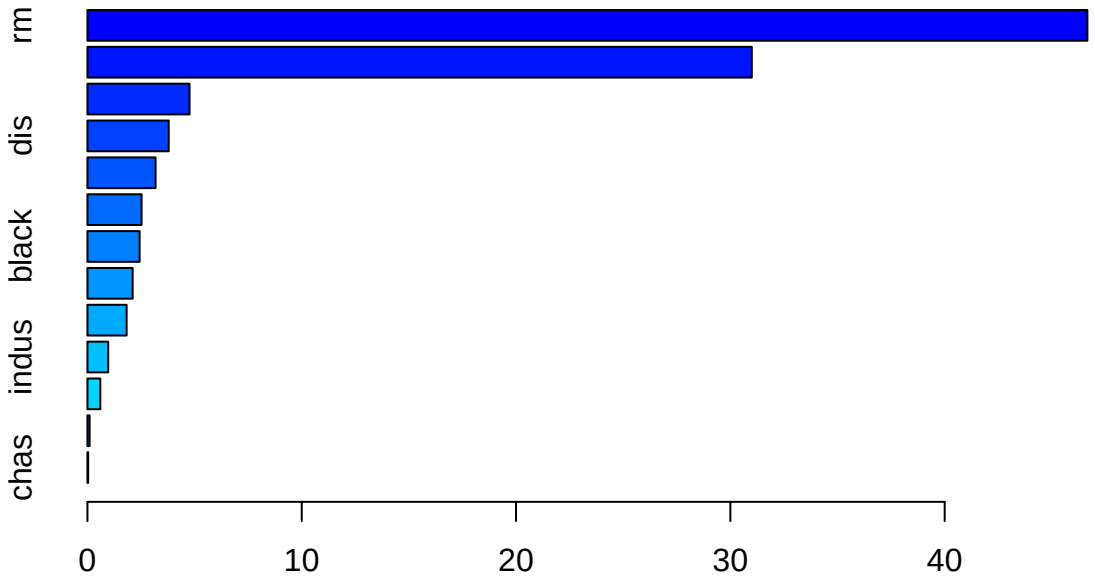
```r
set.seed(1)

num_trees <- floor(seq(100,10000,length.out = 100))
tst.loss <- NULL
boost.boston=gbm( medv~., data=Boston[train,], distribution="gaussian",
                  n.trees=max(num_trees), interaction.depth=5)
for(m in num_trees){
  yhat.bag = predict(boost.boston,newdata=Boston[-train,],n.trees = m,
                     type = "response")
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
}
plot(num_trees,tst.loss,type = "b")
```




```r
M <- num_trees[which.min(tst.loss)]
M
```

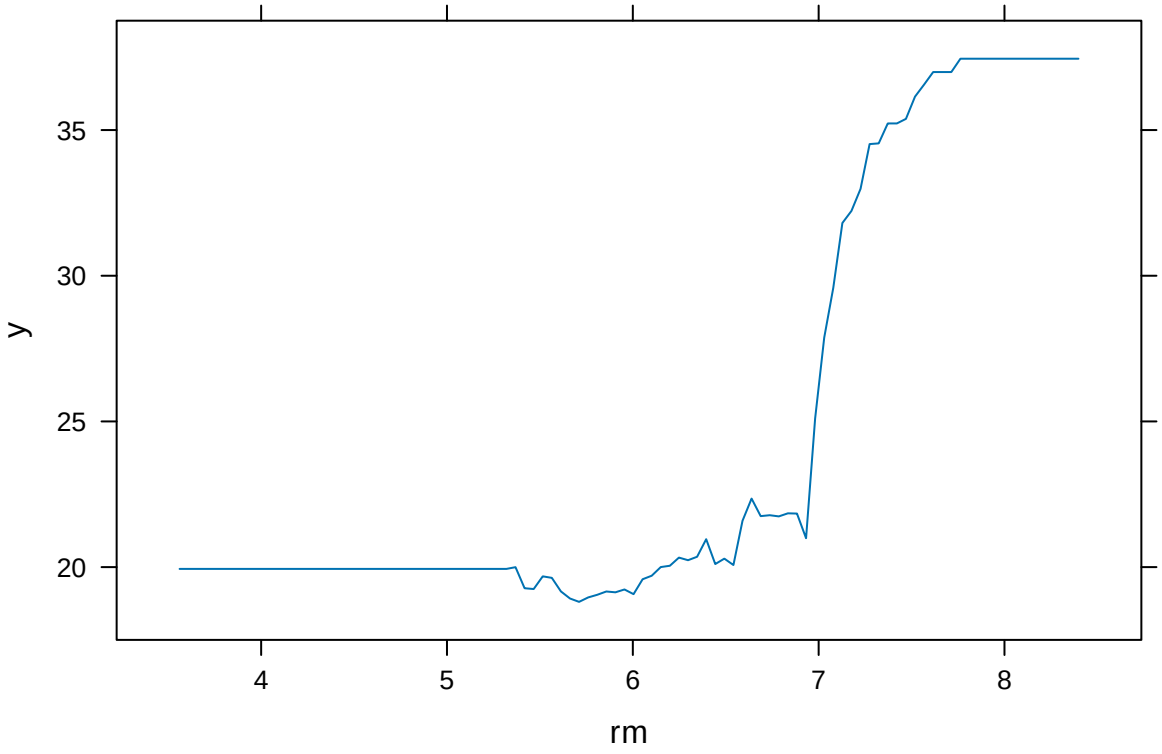
```
## [1] 1100
```

```r
boost.boston=gbm(medv~., data=Boston[train,], distribution="gaussian",
                 n.trees=M, interaction.depth=5)
summary(boost.boston)
```

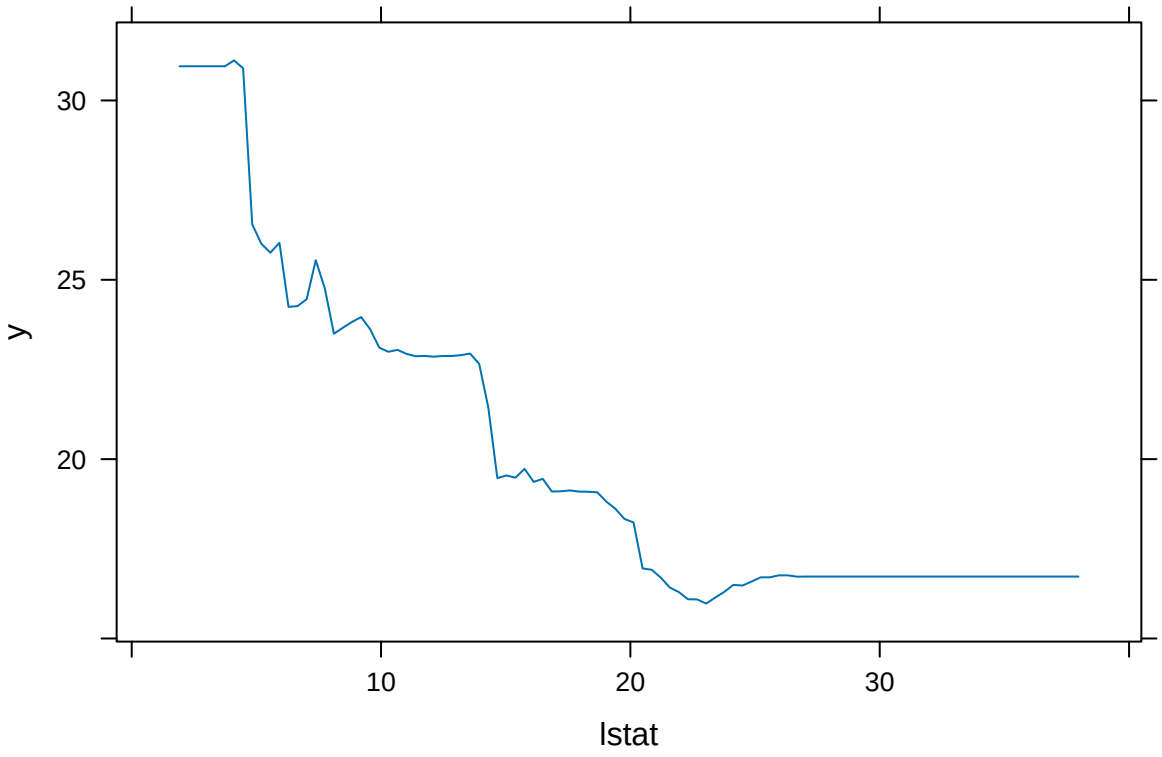


	var	rel.inf
rm	rm	46.6601217
lstat	lstat	31.0014372
crim	crim	4.7628832
dis	dis	3.7922972
nox	nox	3.1805425
age	age	2.5260908
black	black	2.4348971
ptratio	ptratio	2.1110133
tax	tax	1.8287830
indus	indus	0.9675263
rad	rad	0.6007242
zn	zn	0.1004767
chas	chas	0.0332068

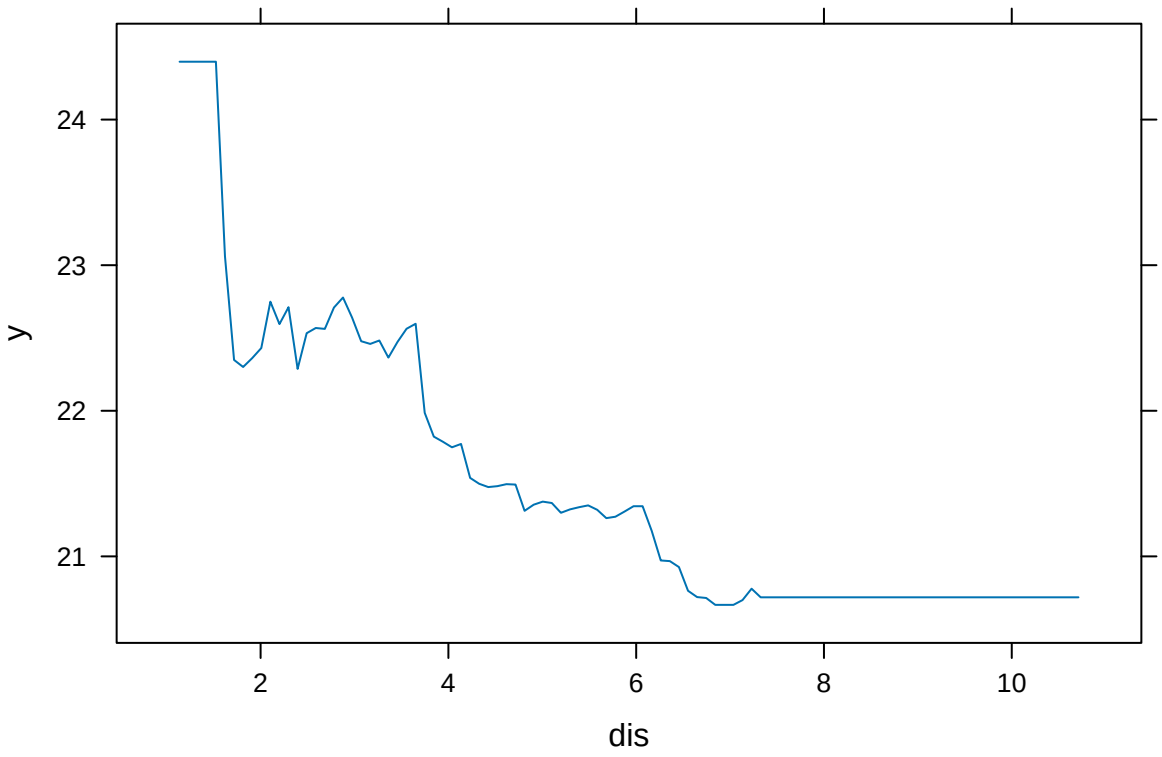
```
par(mfrow=c(2,2))
plot(boost.boston,i="rm")
```



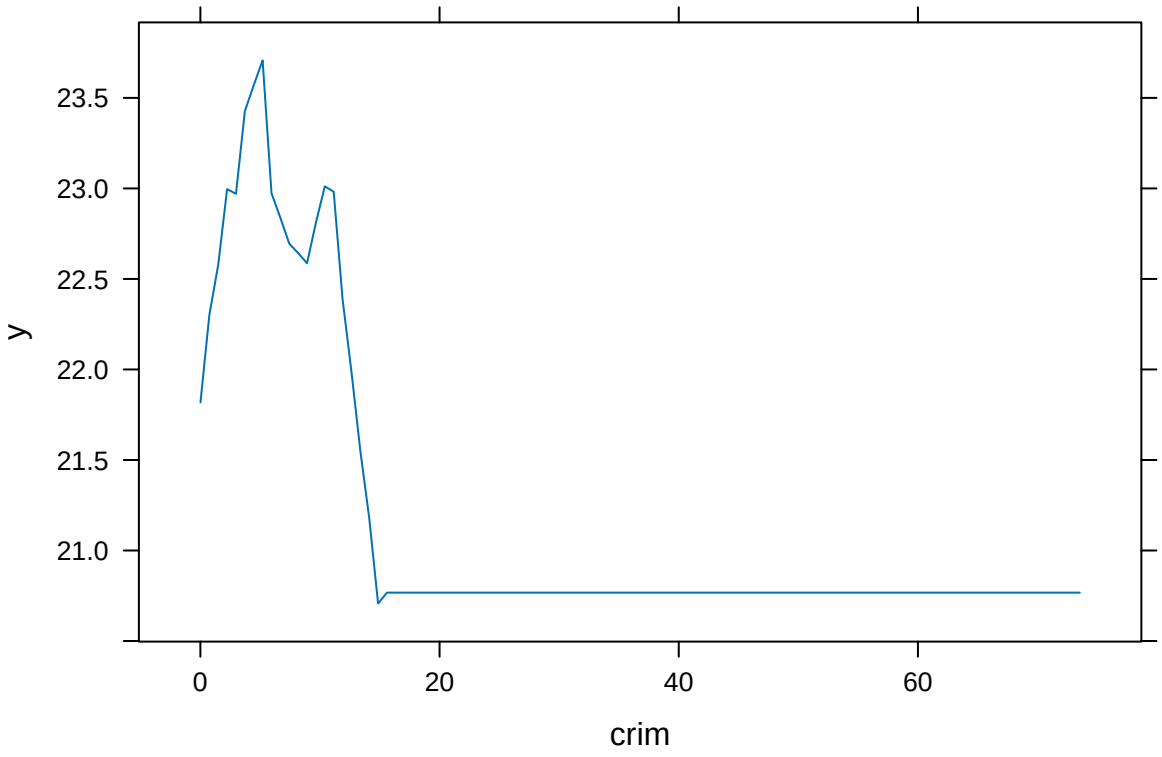

```
plot(boost.boston,i="lstat")
```




```
plot(boost.boston,i="dis")
```




```
plot(boost.boston,i="crim")
```




```
yhat.boost=predict(boost.boston,newdata=Boston[-train,], n.trees = M,
                   type = "response")
mean((yhat.boost-Boston$medv[-train])^2)
```

```
## [1] 19.19356
```

```
boost.boston=gbm(medv~.,data=Boston[train,], distribution="gaussian",
                 n.trees=M, interaction.depth=5, shrinkage=0.2, verbose=F)
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=M,type = "response")
mean((yhat.boost-Boston$medv[-train])^2)
```

```
## [1] 19.69013
```

Stacking

Here's an example of how to perform stacking using the `mlr` package in R:

We'll use the `Pima` data set to start.

In this example, we'll create a stacking model with three base learners (`rpart`, `lda`, and `svm`) and a linear regression (`regr.lm`) as the meta-learner.

```
# Install and load required packages
# install.packages(c("mlr", "rpart", "kknn", "e1071"))
library(mlr)
```

```
## Loading required package: ParamHelpers
```

```
## Warning message: 'mlr' is in 'maintenance-only' mode since July 2019.
## Future development will only happen in 'mlr3'
## (<https://mlr3.mlr-org.com>). Due to the focus on 'mlr3' there might be
## uncaught bugs meanwhile in {mlr} - please consider switching.
```

```
##
## Attaching package: 'mlr'
```

```
## The following object is masked from 'package:caret':
##
##     train
```

```
library(rpart)
library(kknn)
```

```
##
## Attaching package: 'kknn'
```

```
## The following object is masked from 'package:caret':
##
##     contr.dummy
```

```
library(e1071)
```

```
##
## Attaching package: 'e1071'
```

```
## The following object is masked from 'package:mlr':
##
##     impute
```

```
# Set up the task
tsk = makeClassifTask(data = data.frame(train.data), target = "diabetes")
test_tsk = makeClassifTask(data = data.frame(test.data), target = "diabetes")

# Define base learners
```

```
base = c("classif.rpart", "classif.lda", "classif.svm")

# Create learner object.
lrns = lapply(base, makeLearner)

# Set the type of predictions the learner should return.
lrns = lapply(lrns, setPredictType, "prob")

# We're not going to use a meta-learner here
# The average would be used for predict.type = "prob"
# The max will be used for predict.type = "response"

# Create a stacked learner
m = makeStackedLearner(base.learners = lrns,
                       predict.type = "prob",
                       method = "hill.climb")

# Train the model
tmp = train(m, tsk)

# Predict
res_train = predict(tmp, tsk)
pred_test = predict(tmp, test_tsk)

# Evaluate
mean(pred_test$data$truth == pred_test$data$response)
```

```
## [1] 0.7757009
```

Now let's repeat with a meta learner

```
# Define base learners
base = c("classif.rpart", "classif.lda", "classif.svm")

# Create learner object.
lrns = lapply(base, makeLearner)

# Set the type of predictions the learner should return.
lrns = lapply(lrns, setPredictType, "prob")

# Define the meta-learner
meta_learner <- makeLearner("classif.logreg",
                            predict.type = "response")
# Create a stacked learner
m = makeStackedLearner(base.learners = lrns,
                       super.learner = meta_learner,
                       method = "stack.nocv")
m
```

```
## Learner stack from package rpart,MASS,e1071
## Type: classif
## Name: ; Short name:
## Class: StackedLearner
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: response
```

```
## Hyperparameters: classif.rpart.xval=0
```

```r
# Train the model
tmp = train(m, tsk)

# Predict
res_train = predict(tmp, tsk)
pred_test = predict(tmp, test_tsk)

# Evaluate
mean(pred_test$data$truth == pred_test$data$response)
```

```
## [1] 0.7570093
```

Remember, this is a basic example for illustration purposes. In real applications, you'd tune hyperparameters for individual models, and evaluate model performance with measures like accuracy, AUC, etc.

You can further customize and optimize the stacking process using the functions and tools provided by `mlr` or other related R packages.