Random Forests, Boosting, and Stacking

ACM

Load in the packages:

```r
library(printr)
library(randomForest)
library(tidyverse)
library(gbm)
library(caret)
library(MASS)
```

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

```r
# Load the data and remove NAs
PimaIndiansDiabetes3 <- read_csv("pima.indians.diabetes3.csv",
                                 show_col_types = FALSE)
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
0	177	60	29	34.6	1.072	21	diabetic
0	124	56	13	21.8	0.452	21	normal
10	101	86	37	45.6	1.136	38	diabetic

```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

```r
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.

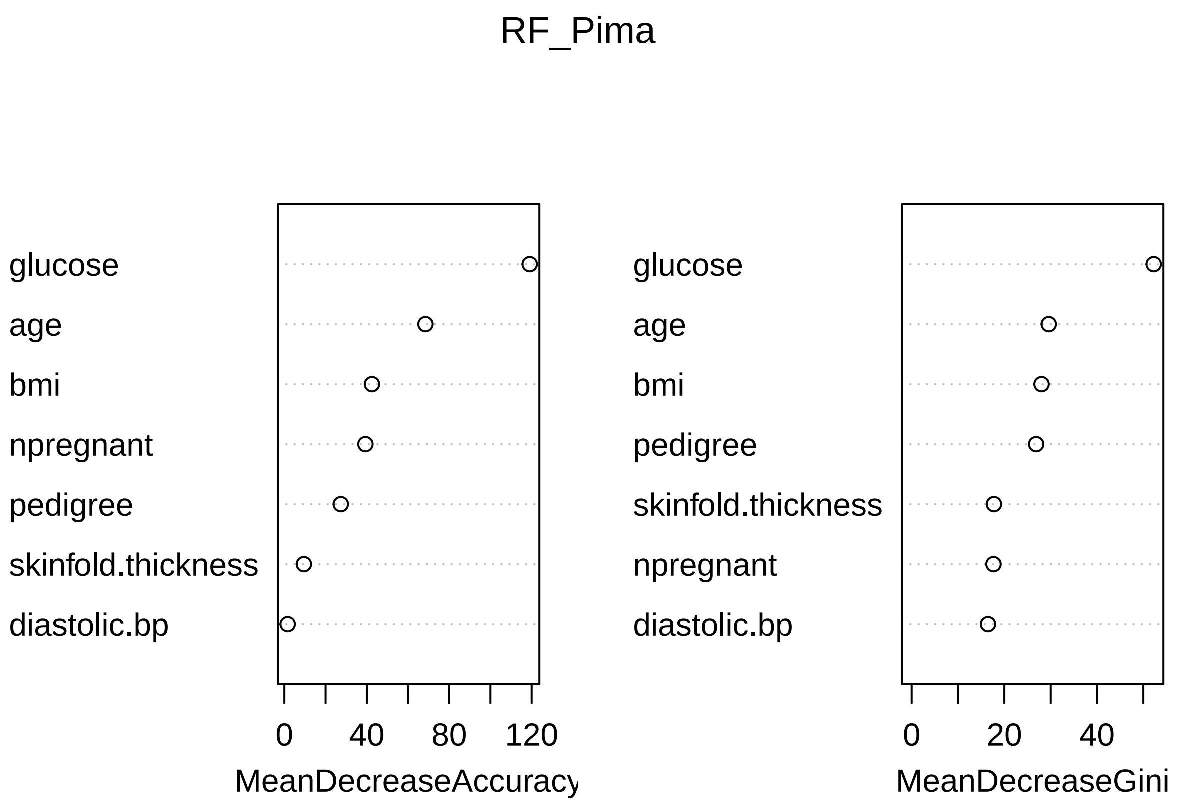
```r
RF_Pima = randomForest(diabetes ~., data = train.data, ntree=5000,
                         mtry = M, importance=TRUE)
 yhat.bag = predict( RF_Pima, newdata = test.data)
 mean(yhat.bag==test.data$diabetes)
```

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

```r
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

```r
varImpPlot(RF_Pima)
```



Boosting

```
?caret::train
```

```
Fit Predictive Models over Different Tuning Parameters

Description:

     This function sets up a grid of tuning parameters for a number of
     classification and regression routines, fits each model and
     calculates a resampling based performance measure.

Usage:

     train(x, ...)

     ## Default S3 method:
     train(
       x,
       y,
       method = "rf",
       preProcess = NULL,
       ...,
       weights = NULL,
       metric = ifelse(is.factor(y), "Accuracy", "RMSE"),
       maximize = ifelse(metric %in% c("RMSE", "logLoss", "MAE", "logLoss"), FALSE, TRUE),
       trControl = trainControl(),
       tuneGrid = NULL,
       tuneLength = ifelse(trControl$method == "none", 1, 3)
     )

     ## S3 method for class 'formula'
     train(form, data, ..., weights, subset, na.action = na.fail, contrasts = NULL)

     ## S3 method for class 'recipe'
     train(
       x,
       data,
       method = "rf",
       ...,
       metric = ifelse(is.factor(y_dat), "Accuracy", "RMSE"),
       maximize = ifelse(metric %in% c("RMSE", "logLoss", "MAE"), FALSE, TRUE),
       trControl = trainControl(),
       tuneGrid = NULL,
       tuneLength = ifelse(trControl$method == "none", 1, 3)
     )
```

Arguments:

```
       x: For the default method, 'x' is an object where samples are in
          rows and features are in columns. This could be a simple
          matrix, data frame or other type (e.g. sparse matrix) but
          must have column names (see Details below). Preprocessing
          using the 'preProcess' argument only supports matrices or
          data frames. When using the recipe method, 'x' should be an
          unprepared 'recipe' object that describes the model terms
          (i.e. outcome, predictors, etc.) as well as any
          pre-processing that should be done to the data. This is an
          alternative approach to specifying the model. Note that, when
          using the recipe method, any arguments passed to 'preProcess'
          will be ignored. See the links and example below for more
          details using recipes.

     ...: Arguments passed to the classification or regression routine
          (such as 'randomForest'). Errors will occur if values for
          tuning parameters are passed here.

       y: A numeric or factor vector containing the outcome for each
          sample.

  method: A string specifying which classification or regression model
          to use. Possible values are found using
          'names(getModelInfo())'. See
          <http://topepo.github.io/caret/train-models-by-tag.html>. A
          list of functions can also be passed for a custom model
          function. See
          <http://topepo.github.io/caret/using-your-own-model-in-train.html>
          for details.

preProcess: A string vector that defines a pre-processing of the
          predictor data. Current possibilities are "BoxCox",
          "YeoJohnson", "expoTrans", "center", "scale", "range",
          "knnImpute", "bagImpute", "medianImpute", "pca", "ica" and
          "spatialSign". The default is no pre-processing. See
          'preProcess' and 'trainControl' on the procedures and how to
          adjust them. Pre-processing code is only designed to work
          when 'x' is a simple matrix or data frame.

 weights: A numeric vector of case weights. This argument will only
          affect models that allow case weights.

  metric: A string that specifies what summary metric will be used to
          select the optimal model. By default, possible values are
          "RMSE" and "Rsquared" for regression and "Accuracy" and
          "Kappa" for classification. If custom performance metrics are
          used (via the 'summaryFunction' argument in 'trainControl',
          the value of 'metric' should match one of the arguments. If
          it does not, a warning is issued and the first metric given
          by the 'summaryFunction' is used. (NOTE: If given, this
          argument must be named.)

maximize: A logical: should the metric be maximized or minimized?

trControl: A list of values that define how this function acts. See
          'trainControl' and
          <http://topepo.github.io/caret/using-your-own-model-in-train.html>.
          (NOTE: If given, this argument must be named.)

tuneGrid: A data frame with possible tuning values. The columns are
          named the same as the tuning parameters. Use 'getModelInfo'
          to get a list of tuning parameters for each model or see
          <http://topepo.github.io/caret/available-models.html>. (NOTE:
          If given, this argument must be named.)

tuneLength: An integer denoting the amount of granularity in the tuning
          parameter grid. By default, this argument is the number of
          levels for each tuning parameters that should be generated by
          'train'. If 'trainControl' has the option 'search =
          "random"', this is the maximum number of tuning parameter
          combinations that will be generated by the random search.
          (NOTE: If given, this argument must be named.)

    form: A formula of the form 'y ~ x1 + x2 + ...'

    data: Data frame from which variables specified in 'formula' or
          'recipe' are preferentially to be taken.

  subset: An index vector specifying the cases to be used in the
          training sample. (NOTE: If given, this argument must be
          named.)
```

```
na.action: A function to specify the action to be taken if NAs are
          found. The default action is for the procedure to fail. An
          alternative is 'na.omit', which leads to rejection of cases
          with missing values on any required variable. (NOTE: If
          given, this argument must be named.)

contrasts: A list of contrasts to be used for some or all the factors
          appearing as variables in the model formula.

Details:

     'train' can be used to tune models by picking the complexity
     parameters that are associated with the optimal resampling
     statistics. For particular model, a grid of parameters (if any) is
     created and the model is trained on slightly different data for
     each candidate combination of tuning parameters. Across each data
     set, the performance of held-out samples is calculated and the
     mean and standard deviation is summarized for each combination.
     The combination with the optimal resampling statistic is chosen as
     the final model and the entire training set is used to fit a final
     model.

     The predictors in 'x' can be most any object as long as the
     underlying model fit function can deal with the object class. The
     function was designed to work with simple matrices and data frame
     inputs, so some functionality may not work (e.g.  pre-processing).
     When using string kernels, the vector of character strings should
     be converted to a matrix with a single column.

     More details on this function can be found at
     <http://topepo.github.io/caret/model-training-and-tuning.html>.

     A variety of models are currently available and are enumerated by
     tag (i.e. their model characteristics) at
     <http://topepo.github.io/caret/train-models-by-tag.html>.

     More details on using recipes can be found at
     <http://topepo.github.io/caret/using-recipes-with-train.html>.
     Note that case weights can be passed into 'train' using a role of
     '"case weight"' for a single variable. Also, if there are
     non-predictor columns that should be used when determining the
     model's performance metrics, the role of '"performance var"' can
     be used with multiple columns and these will be made available
     during resampling to the 'summaryFunction' function.
```

For the following code I used `results='hide'` to process the rmarkdown because this function prints 100's of pages of output (with no clear way how to silence).

```r
gbm_grid <- expand.grid(shrinkage = c(0.01, 0.1, 0.2),
                        interaction.depth = c(2,3,4),
                        n.trees = 2000,
                        n.minobsinnode = 10)
time <- system.time(model <- train(diabetes ~. , data = train.data,
                 method = "gbm",
                  distribution = "bernoulli",
               tuneGrid = gbm_grid,
                trControl = trainControl("cv", number = 10)))
```

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

```
time
```

```
##    user  system elapsed
##  25.119   0.269  25.505
```

```
model$bestTune
```

n.trees	interaction.depth	shrinkage	n.minobsinnode
2000	2	0.01	10

Now let's look at the test accuracy and compare to random forests.

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

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

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

```
## [1] 0.7383178
```

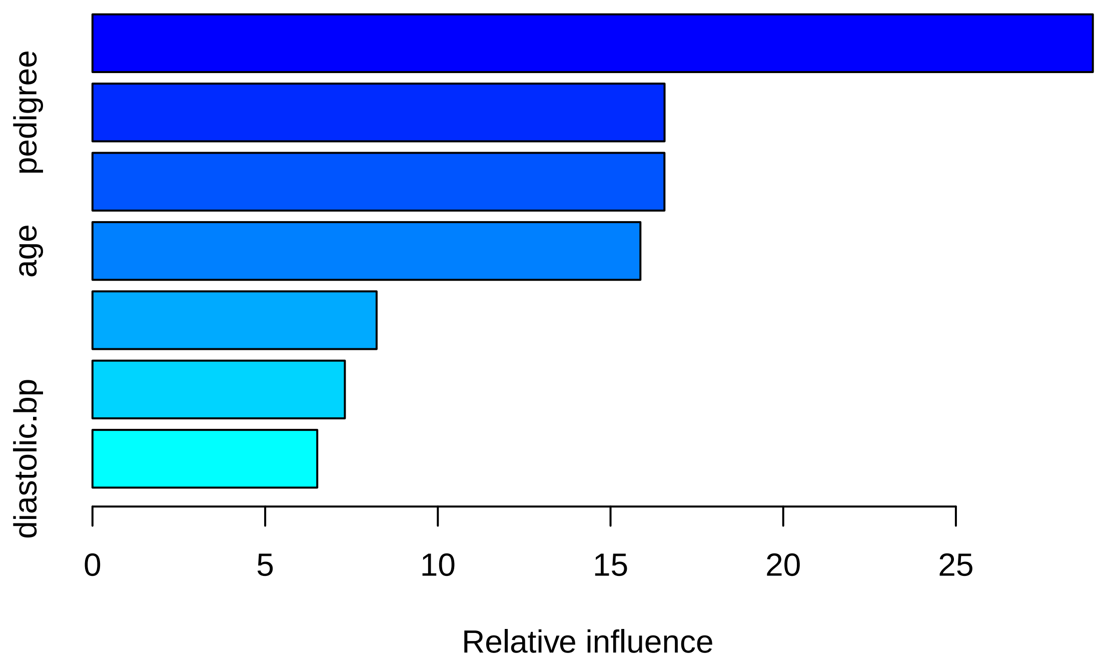
Now let's check out some summaries of the model. Note that `gbm` *requires* numeric $\{0, 1\}$ outcome, while `train` (used above) *requires* a factor outcome for classification.

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.

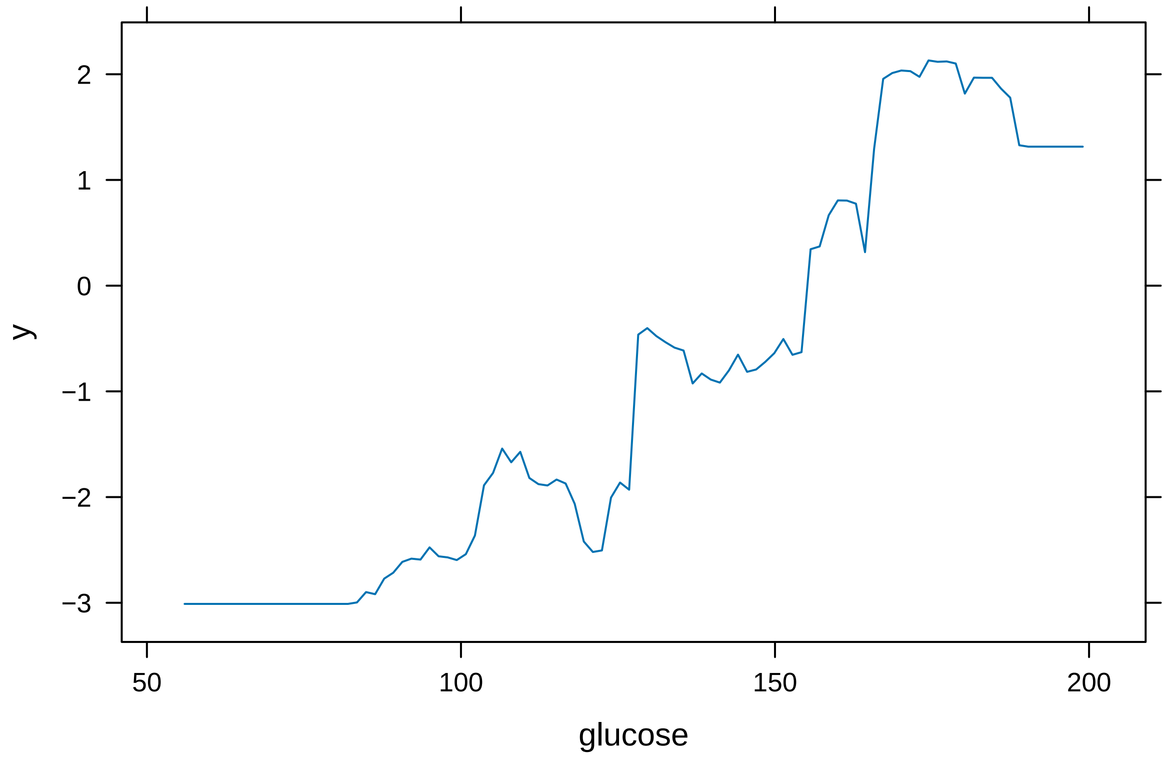
```
Pima <- Pima %>% mutate(diabetes_num = 1*I(diabetes=="diabetic"))

boost.Pima=gbm(diabetes_num ~.-diabetes, data=Pima[training.samples,],
                  distribution = "bernoulli",
                   n.trees = 2000, interaction.depth = 4,
                   shrinkage = 0.01, n.minobsinnode = 10)
summary(boost.Pima) # we could use summary(model) and get the same thing.
```

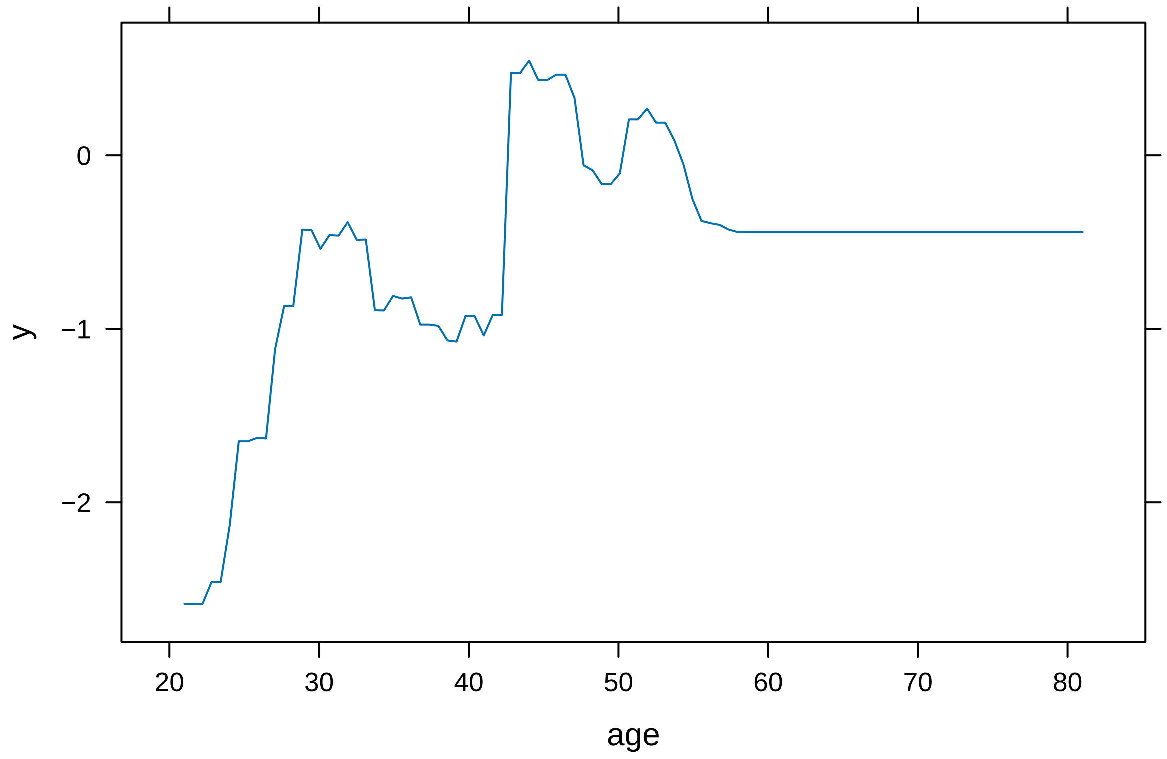


	var	rel.inf
glucose	glucose	28.965738
pedigree	pedigree	16.564268
bmi	bmi	16.561508
age	age	15.864936
skinfold.thickness	skinfold.thickness	8.227297
npregnant	npregnant	7.307896
diastolic.bp	diastolic.bp	6.508358

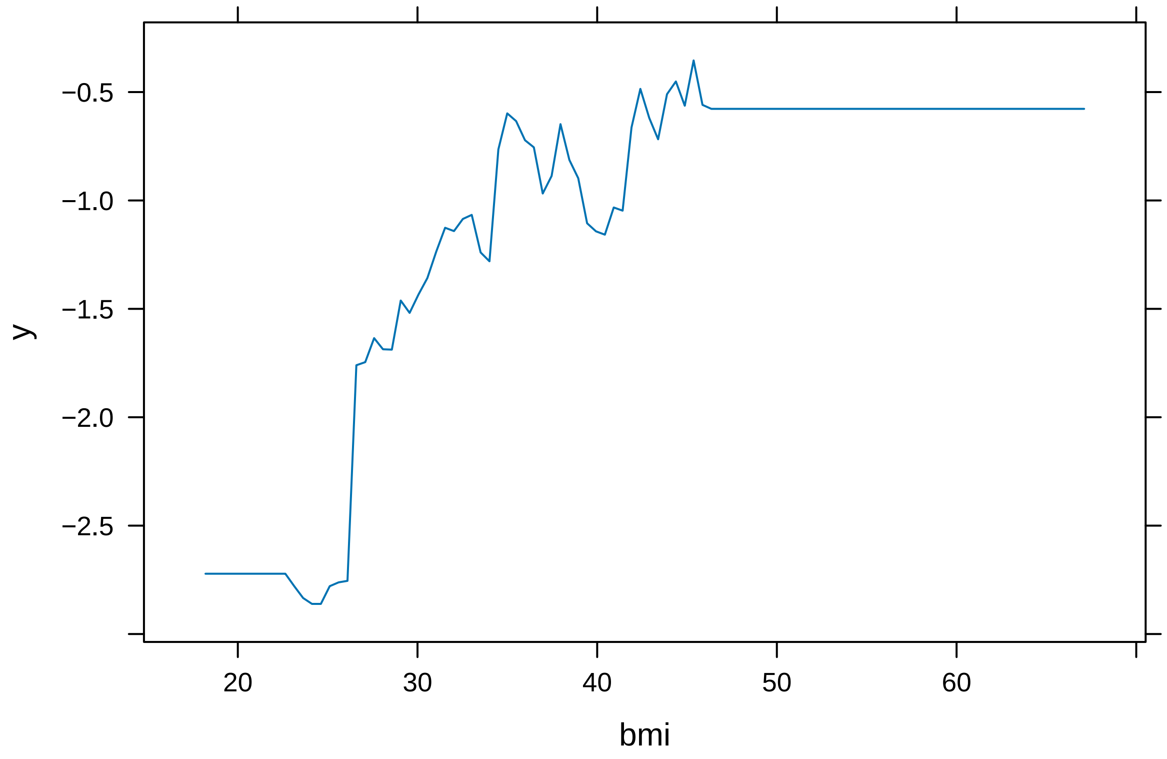
```r
par(mfrow=c(2,2))
plot(boost.Pima, i="glucose")
```



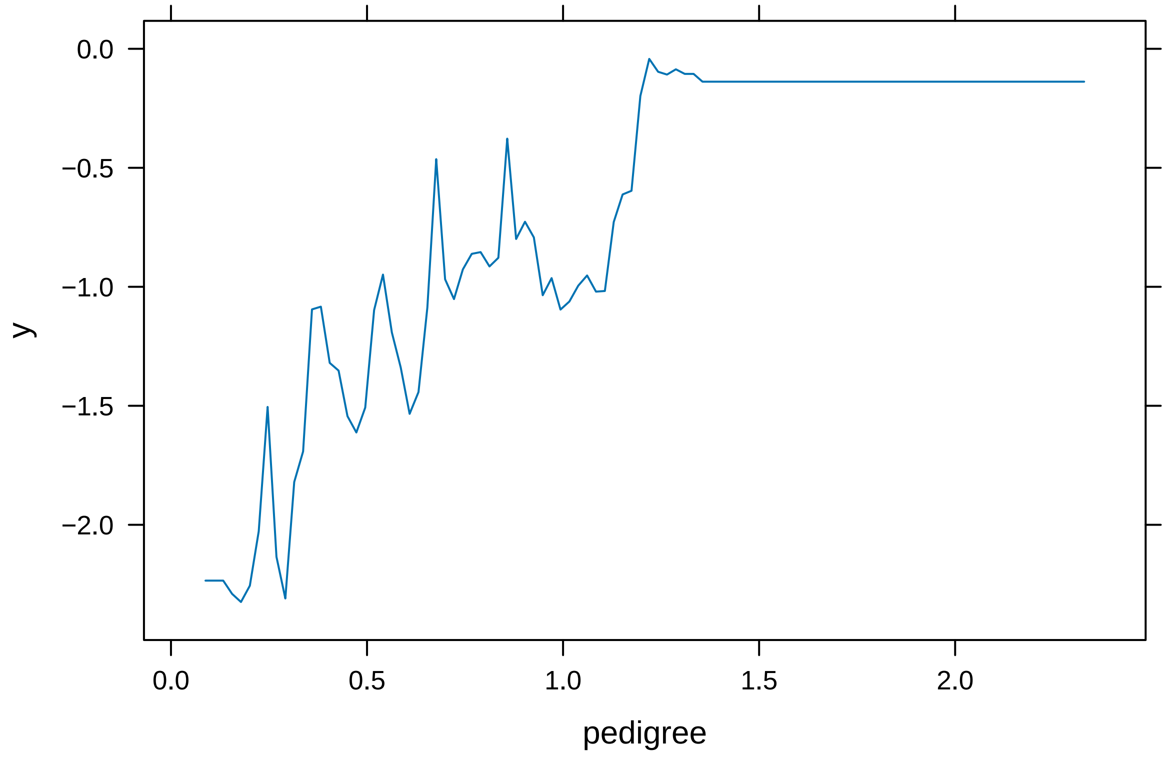
```r
plot(boost.Pima, i="age")
```



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



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



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`). We're not going to use a meta-learner to start.

```
# 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
```

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

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

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

```r
library(e1071)
```

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

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

```r
# 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] 0.7383178
```

Now let's repeat with linear regression (`regr.lm`) as the meta-learner.

```r
# 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.