Class 10 Demo: Random Forests

Load packages and read in data

First, let's read in the bbb dataset. Note that we transform the buyer variable to be a factor, i.e., a categorical variable. This is required by the `rf` function that we will use to fit a random forest so that it knows we are dealing with a classification problem (i.e., a discrete outcome). Note, in general it is good practice to explicitly turn transform *all* categorical variables into factors so we will do that here and going forward.

Another final point is that we should always set a random number "seed" before fitting a RF. This is because the bootstrap samples used for each try are random so setting a seed guarantees you will be able to exactly replicate your results. We can do this via the `set.seed()` and specify a number, here 5678 (note: it is good practice to do this and henceforth we will do so at the beginning of all of our code):

```
### Install new packages:
#install.packages("ranger")

### Load packages:
#library(knitr)
#library(janitor)
#library(haven)
#library(readxl)
#library(psych)
#library(statar)
library(tidyverse)
library(mktg482)
library(ranger)

# Clear environment of datasets
rm(list=ls())

# Read in and transform data
set.seed(5678)
load("../../Data/bbb.Rdata")
bbb <- bbb %>%
  mutate(buyer=factor(buyer), gender=factor(gender))
```

Split dataset into training and test samples

Now, we split the data in a training (or calibration) sample and a test (or validation) sample:

```
bbb.training <- bbb %>% filter(training==1)
bbb.test <- bbb %>% filter(training==0)
```

Random Forest

We now estimate (or fit or train) a random forest, but only on the training data. The command says that we should use the `bbb.training` dataset and that we want a "probability"" prediction for each customer. A random forest has two tuning parameters (`mtry` and `min.node.size`) which we make explicit in the last line of the code and which we discuss below.

```
rf <- ranger(buyer ~ gender + last + total + child + youth +
             cook + do_it + reference + art + geog,
             data=bbb.training, probability=TRUE, mtry=3, min.node.size=1)
```

Next, we create predictions for the test data. There are two somewhat odd things here in the `predict` function. First, we need to specify `data` instead of `newdata` as we did for logistic regression; this is simply how the author of the `ranger` package decided to create the predict function. Second, we need the `[[1]][,2]` after predict; see the end of this document for an explanation if you are interested.

```
pred_rf_test <- predict(rf, data=bbb.test, type="response")[[1]][,2]
```

Logistic Regression

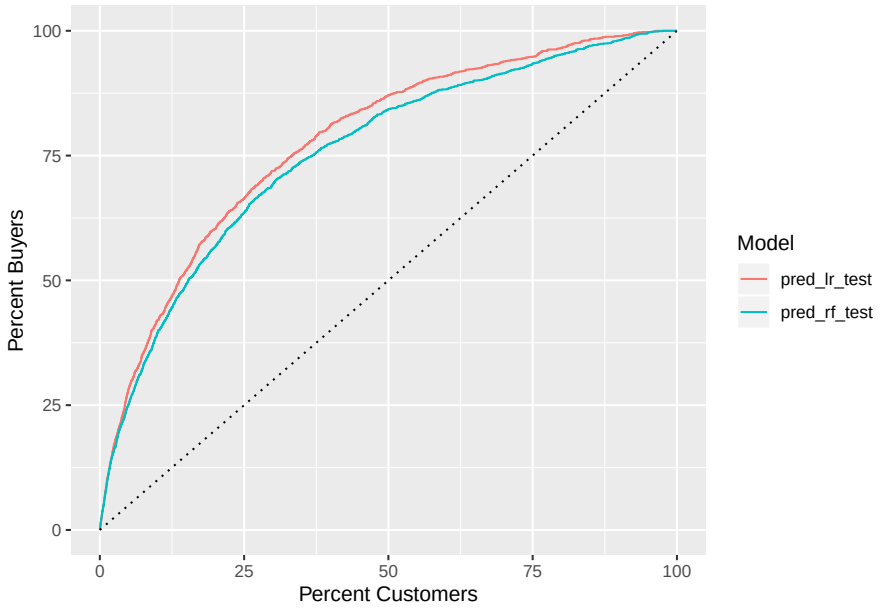
We also fit a logistic regression as in the assignment, but now estimate it only on the training sample. We also create a prediction for the test sample again using the `newdata` argument in the `predict` function.

```
lr <- glm(buyer ~ gender + last + total + child + youth +
          cook + do_it + reference + art + geog,
          family=binomial, data=bbb.training)
pred_lr_test <- predict(lr, newdata=bbb.test, type="response")
```

Model performace comparison

Now, we compare the performance on the test sample.

```
gainsplot(pred_lr_test, pred_rf_test, label.var = bbb.test$buyer)
```



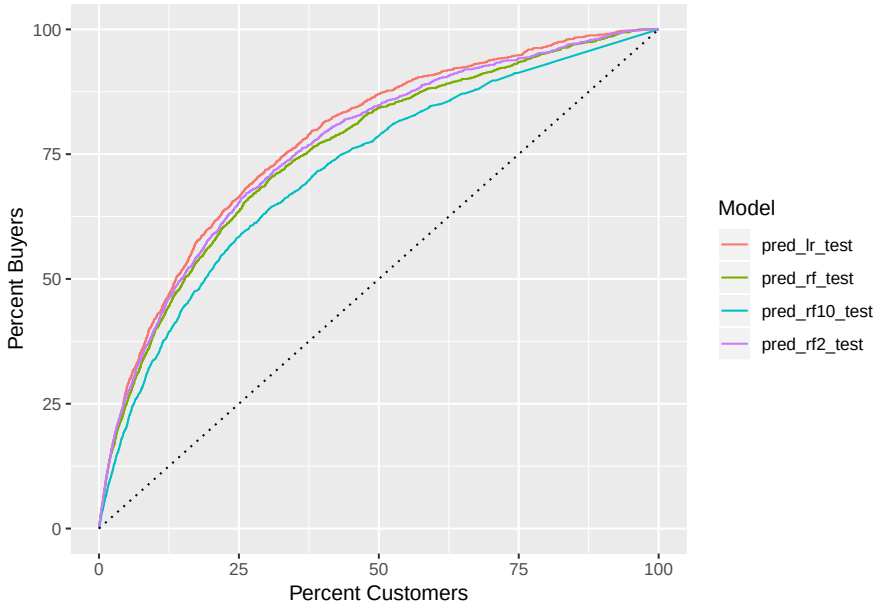

```
# A tibble: 2 x 2
  model          auc
  <chr>        <dbl>
1 pred_lr_test 0.806
2 pred_rf_test 0.781
```

The RF does not do as well as the logistic regression.

As noted above, random forests (like neural networks and most other machine learning techniques) requires selecting some tuning parameters. They key one is the `mtry` parameter, the number of randomly chosen variables at each split. Be default, `ranger` picks the square root of the number predictor variables (rounded down). Since there are ten predictor variables, in this case `ranger` would pick `mtry=3` (which we explicitly specified above). Let's try `mtry=2` and `mtry=10` for comparison (note, the latter tries every variable at each split which is likely to lead to overfitting). We will leave the other tuning parameter `min.node.size` fixed at one (in general, setting `min.node.size` to one is good practice).

```
rf2 <- ranger(buyer ~ gender + last + total + child + youth +
              cook + do_it + reference + art + geog,
              data=bbb.training, probability=TRUE, mtry=2, min.node.size=1)
rf10 <- ranger(buyer ~ gender + last + total + child + youth +
               cook + do_it + reference + art + geog,
               data=bbb.training, probability=TRUE, mtry=10, min.node.size=1)
pred_rf2_test <- predict(rf2, data = bbb.test, type="response")[[1]][,2]
pred_rf10_test <- predict(rf10, data = bbb.test, type="response")[[1]][,2]

gainsplot(pred_lr_test, pred_rf_test, pred_rf2_test, pred_rf10_test,
          label.var = bbb.test$buyer)
```




```
# A tibble: 4 x 2
  model            auc
  <chr>          <dbl>
1 pred_lr_test   0.806
2 pred_rf_test   0.781
3 pred_rf2_test  0.792
4 pred_rf10_test 0.739
```

As one can be seen, `mtry=2` is the best of the RFs and is slightly worse than logistic regression on this data whereas `mtry=10` does indeed overfit.

Caution: Random Forests Training Set Predictions

In general, we do not need to look at performance on the training data to select our model–only the test data. However, it is especially the case that we do not look at the preductions for a random forest in particular on

the training data. The reason for this is that, due to the nature of the random forest algorithm, it fits nearly perfectly on the training data (i.e., because each observation appears in about 2/3 of the bootstrap samples and therefore about 2/3 of the trees produce nearly perfect predictions for the observation). Consequently, if use code like the below

```
pred_rf_training <- predict(rf, data=bbb.training, type="response")[[1]][,2]
```

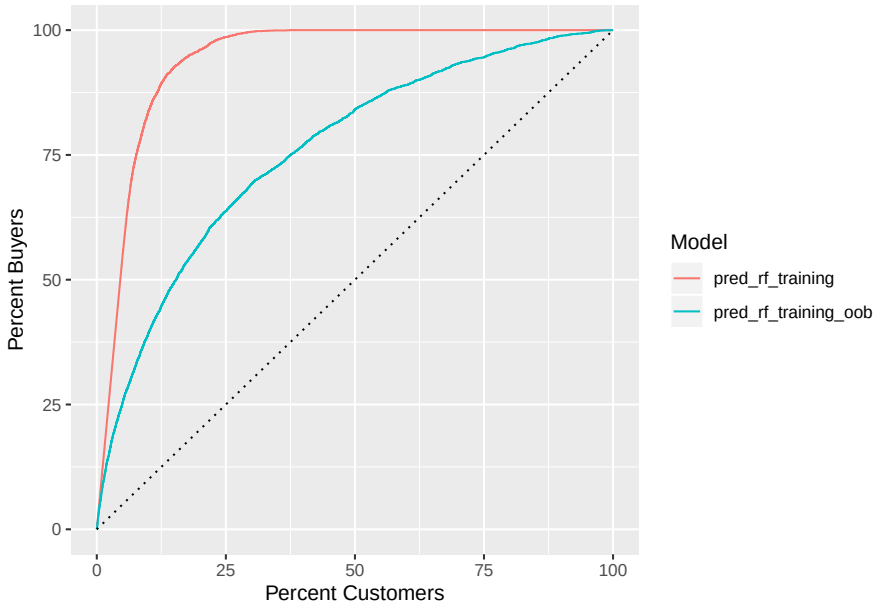
we would get nearly perfect predictions. If for some reason, we absolutely need to get predictions on the training sample, we can use what are known as "out-of-bag" predictions. These predictions use only the roughly 1/3 of trees for which each observation did not appear in the bootstrap sample for the prediction. You can obtain these via:

```
pred_rf_training_oob <- predictions(rf)[,2]
```

Note: the `predictions` command returns two columns, one for each outcome ("0" and "1"). We want only the probability of "1" so we pick the second column.

One nice feature of out-of-bag predictions is that they are reasonably resilient to overfitting (i.e., because they are based on only the roughly 1/3 of trees for which each observation did not appear in the bootstrap sample). We illustrate the points discussed above by making a gains curve for the training data:

```
gainsplot(pred_rf_training, pred_rf_training_oob, label.var = bbb.training$buyer)
```




```
# A tibble: 2 x 2
  model                  auc
  <chr>                <dbl>
1 pred_rf_training     0.984
2 pred_rf_training_oob 0.786
```

Optional: Why do we need `[[1]][,2]` to pull out predictions in `ranger`?

The `predict()` function is a general one that creates predictions for all sorts of models. The author of each model is responsible for what data the `predict()` function returns. The prediction function associated with the `glm` function, what we use to estimate a logistic regression, simply returs a vector of predictions:

```
str(pred_lr_test)
```

```
 Named num [1:15000] 0.0155 0.0777 0.0473 0.0635 0.3595 ...
 - attr(*, "names")= chr [1:15000] "1" "2" "3" "4" ...
```

However, the prediction function associated with the `ranger` function returns a more complicated object:

```
pred_rf_raw <- predict(rf, data = bbb.test, type="response")
str(pred_rf_raw)
```

```
List of 5
 $ predictions              : num [1:15000, 1:2] 0.98 0.963 0.943 0.971 0.68 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:2] "0" "1"
 $ num.trees                : num 500
 $ num.independent.variables: num 10
 $ num.samples              : int 15000
 $ treetype                 : chr "Probability estimation"
 - attr(*, "class")= chr "ranger.prediction"
```

This is what in R is called a list, which is a collection of different objects. You can see that there are 5 objects which describe various aspects of the model, with the first looking like the predictions. In a list we can grab the first object with `[[1]]`:

```
str(pred_rf_raw[[1]])
```

```
 num [1:15000, 1:2] 0.98 0.963 0.943 0.971 0.68 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:2] "0" "1"
```

You can see that the results come in two columns, the first being the probability of `0` (or no purchase) and the second being the probability of `1` (or purchase). We are interested in the probability of `1`. As a result, we want to second column. We get that by appending `[,2]` where the lack of anything before the comma means "all rows" and the 2 after the comma means "column 2". Let's try this:

```
str(pred_rf_raw[[1]][,2])
```

```
 num [1:15000] 0.0201 0.0366 0.0565 0.0294 0.3204 ...
```

This explains why above I appended `[[1]][,2]` and wrote

```
pred_rf_test <- predict(rf, data = bbb.test, type="response")[[1]][,2]
```