ACTL4305/5305 Actuarial Data Analytic Application

Week 8: Gradient Boosting Machine

Learning Objectives

In this week's lab, we continue our exploration of the credit data introduced in Week 7, with a focus on fitting Gradient Boosting Machines. We will use the same illustrative subset of 1000 observations. However, feel free to explore the complete dataset or a larger sample after the lab for more comprehensive result comparisons.

- We guide you through the fundamental process of training and fine-tuning gradient boosting machines using the `caret` and `gbm` package in R.
- We will also examine various interpretation tools, including feature importance and partial dependence plots, while also acknowledging their inherent limitations.

1 Gradient Boosting Machine

Gradient boosting machine (GBM) is the process of iteratively adding basis functions in a greedy fashion so that each additional basis function further reduces the selected loss function. The main idea of boosting is to add new models to the ensemble sequentially. At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learned so far. Gradient boosting optimizes an arbitrary differentiable cost function such as squared error.

Gradient boosting differs from both bagging and random forest. Bagging uses equal weights for all learners we include in the model. Boosting is different as it employs non-uniform weights. Whereas random forests build an ensemble of deep independent trees, GBMs build an ensemble of shallow and weak successive trees with each tree learning and improving on the previous. When combined, these many weak successive trees produce a powerful "committee" that is often hard to beat with other algorithms.

The `gbm` R package is the original R implementation of gradient boosting machine.

```
# load packages
library(dplyr)
library(gbm)
library(caret)
library(pROC)
library(ROCR)
library(tidyr)
library(PRROC)
library(doParallel)
library(glmnet)
library(pdp)
library(ggplot2)
library(gridExtra)

# load data
```

```r
credit <- read.csv("credit.csv")%>% dplyr::select(-X, -ID)

payamt_colsnames <- paste0("PAY_", c(1, 2:6))

credit <- credit%>%dplyr::mutate_at(vars(EDUCATION, MARRIAGE,SEX, default,
                                         payamt_colsnames), funs(factor))

credit$default <- as.factor(ifelse(credit$default == 1, "Yes", "No"))

#Extract a sample from the training set to speed up the computation
set.seed(310)
credit <- credit[sample(nrow(credit),size = 1000, replace = FALSE),]

# reproduciblity

set.seed(123)

# data splitting

index <- createDataPartition(credit$default, p = 0.7, list = FALSE)
train <- credit[index, ]; test <- credit[-index, ]
```

2 Tuning a GBM

In the `train` function, the tuning parameters for the `gbm` method include:

- **Number of trees** (`n.trees`): The total number of trees to fit. GBMs often require many trees; however, unlike random forests GBMs can overfit (think about why) so the goal is to find the optimal number of trees that minimize the loss function of interest with cross validation.
- **Depth of trees** (`interaction.depth`): The number d of splits in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a stump consisting of a single split. More commonly, d is greater than 1 but it is likely $d < 10$ will be sufficient.
- **Learning rate** (`shrinkage`): Controls how quickly the algorithm proceeds down the gradient descent. Smaller values reduce the chance of overfitting but also increases the time to find the optimal fit. This is also called shrinkage.
- **Min. terminal node size** (`n.minobsinnode`): an integer specifying the minimum number of observations in the terminal nodes of the trees.

In the original `gbm` package, you can also fine-tune:

- **Subsampling** (`bag.fraction`): Controls whether or not you use a fraction of the available training observations. Using less than 100% of the training observations means you are implementing stochastic gradient descent. This can help to minimize overfitting and keep from getting stuck in a local minimum or plateau of the loss function gradient (i.e., this is known as **Stochastic Gradient Boosting**, which uses a randomly selected subset of the data to fit each tree, adding a stochastic element to the boosting process. In contrast, standard GBM uses the entire dataset to fit each tree. In the `gbm` package, the default is stochastic GBM (`bag.fraction = 1` for standard GBM and `<1` for stochastic GBM).). Default is 0.5.

2.1 Training and Tuning with Caret

```
#Define control parameters for train
fitcontrol <- trainControl(method = "cv",
                           number = 5,
                           savePredictions = TRUE,
                           classProbs = TRUE,
                           summaryFunction = twoClassSummary,
                           allowParallel = TRUE)

####GBM####
set.seed(515)
gbm1 <- train(default ~ ., data = train, method="gbm", distribution = "bernoulli",
              metric = "ROC", trControl = fitcontrol, verbose = FALSE)
print(gbm1)
```

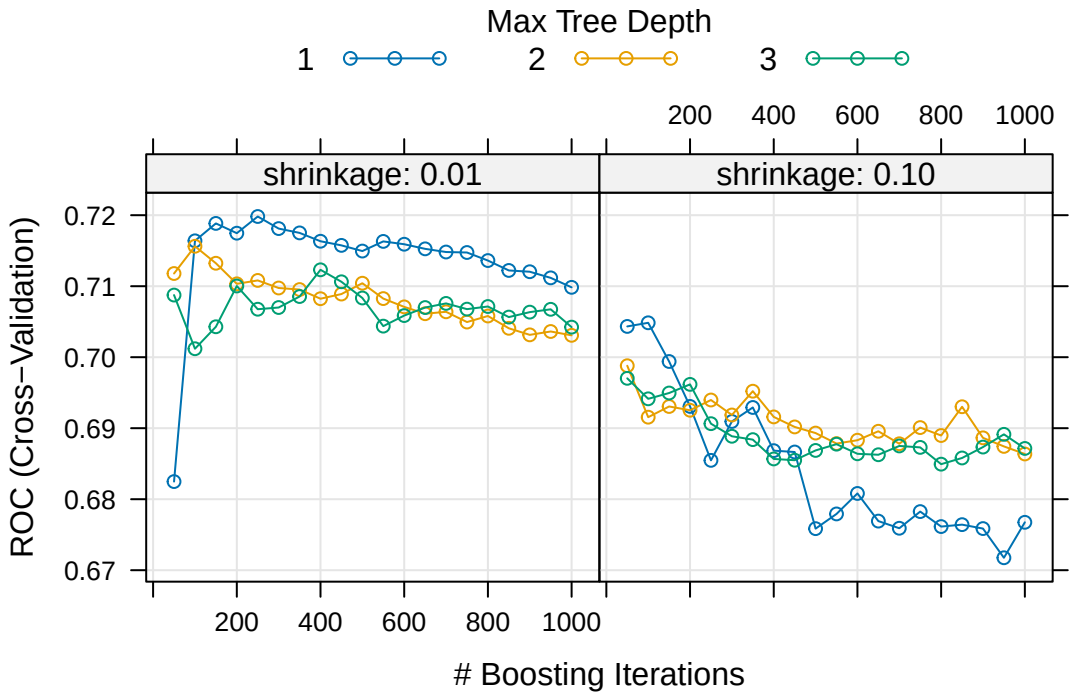
```
## Stochastic Gradient Boosting
##
## 701 samples
##  23 predictor
##   2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 561, 560, 561, 561, 561
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  ROC        Sens       Spec
##   1                   50      0.7132251  0.9630989  0.2580645
##   1                  100      0.7020809  0.9557255  0.2897177
##   1                  150      0.6953663  0.9501699  0.2961694
##   2                   50      0.7196405  0.9538906  0.2897177
##   2                  100      0.7168283  0.9372749  0.3338710
##   2                  150      0.7162165  0.9317193  0.3274194
##   3                   50      0.6946928  0.9483011  0.2830645
##   3                  100      0.6833573  0.9298675  0.2770161
##   3                  150      0.6842283  0.9280326  0.2957661
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 50, interaction.depth =
##  2, shrinkage = 0.1 and n.minobsinnode = 10.
```

```
gbm1_pred <- predict(gbm1, newdata = test[,-ncol(credit)], type = "raw")
gbm1_conf <- confusionMatrix(gbm1_pred,  test[,ncol(credit)], positive="Yes")

gbm1_probpred <- predict(gbm1, newdata = test[,-ncol(credit)], type = "prob")
gbm1_auc <- roc.curve(scores.class0 = gbm1_probpred$Yes,
                      weights.class0 = as.numeric(test$default)-1, curve = T)
gbm1_auc$auc
```

```
## [1] 0.6919712
```

Based on the displayed outputs, it's evident that default grid values for `interaction.depth` and `n.trees` were used and selected, even though we didn't input any specific tuning parameter grid values. Now, let's take the next step and define our own tuning grid.

```
#Retrain GBM by performing grid search
parameters <- expand.grid(n.trees =  (1:20)*50,
                          interaction.depth = c(1, 2, 3),
                          shrinkage = c(0.01, 0.1),
                          n.minobsinnode = 10)

cl <- makeCluster(detectCores())
registerDoParallel(cl)

set.seed(385)
gbm_gridfit <- train(default ~ .,
               data = train,
               method = "gbm",
               distribution = "bernoulli",
               metric = "ROC",
               trControl = fitcontrol,
               tuneGrid = parameters,
               verbose = FALSE)
stopCluster(cl)

#print(gbm_gridfit)
plot(gbm_gridfit)
```



Based on the outcomes of the five-fold cross-validation, the optimal values selected for the model were `n.trees` = 250, `interaction.depth` = 1, `shrinkage` = 0.01, and `n.minobsinnode` = 10. The plotted graph demonstrates

the relationship between `n.tree` and `shrinkage`, highlighting that a smaller learning rate typically necessitates a larger number of trees to achieve better performance (a higher AUC in this case).

> **Exercise:** For further exploration, the code provided below offers an expanded grid of tuning parameters. Feel free to adjust and experiment with the code to observe any potential changes in optimal values. For instance, you can also consider using a larger training sample size to gauge the impact of data size on the optimal values.

```
# This code chunk won't be executed but will be displayed
#Exercise
parameters <- expand.grid(n.trees = (1:30)*50,
                          interaction.depth = c(1, 3, 5, 7, 9),
                          shrinkage = c(0.01, 0.05, 0.1),
                          n.minobsinnode = c(3, 5, 10, 15))

set.seed(629)

cl <- makeCluster(detectCores())
registerDoParallel(cl)

gbm_gridfit2 <- train(default ~ .,
                  data = train,
                  method = "gbm",
                  distribution = "bernoulli",
                  metric = "ROC",
                  trControl = fitcontrol,
                  tuneGrid = parameters,
                  verbose = FALSE)
stopCluster(cl)

print(gbm_gridfit2)
plot(gbm_gridfit2)
```

Using the provided code above (without further modifications), the final model is trained as follows:

```
#Final Model
set.seed(875)
fitControl_final <- trainControl(method = "none", classProbs = TRUE)

gbm1_best <- train(default ~ .,
                  data = train,
                  method = "gbm",
                  distribution = "bernoulli",
                  metric = "ROC",
                  trControl = fitControl_final,
                  ## Only a single model can be passed to the
                  ## function when no resampling is used:
                  tuneGrid = data.frame(interaction.depth = 1,
                                        n.trees = 150,
                                        shrinkage = 0.01,
                                        n.minobsinnode = 5),
                  verbose = FALSE)

gbm1_best_pred <- predict(gbm1_best, newdata = test[,-ncol(credit)], type = "raw")
```

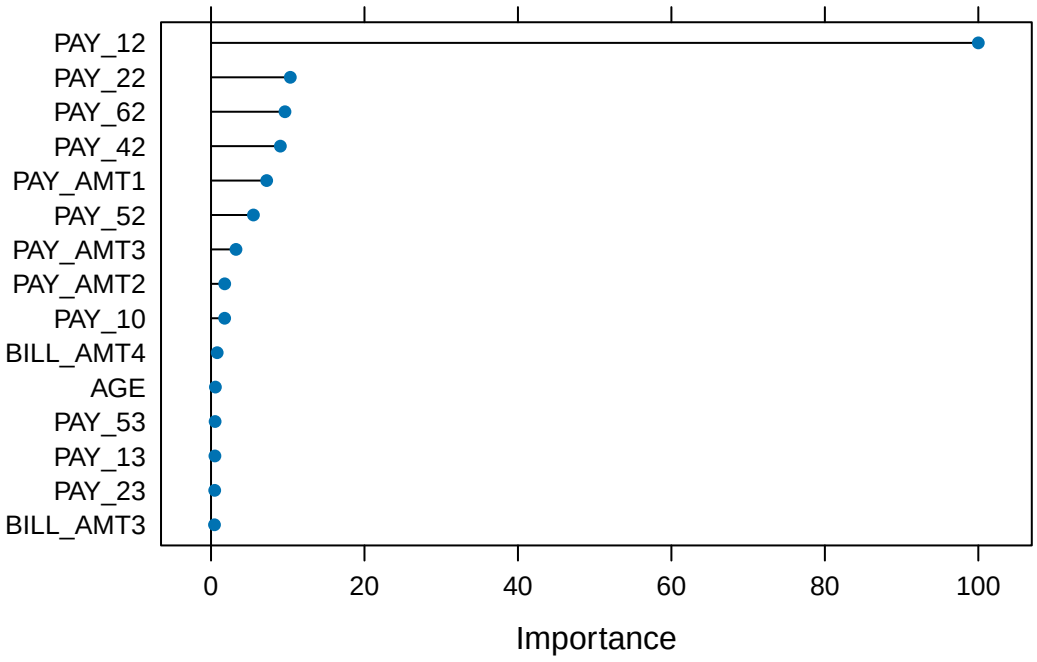
```
gbm1_best_conf <- confusionMatrix(gbm1_best_pred,  test[,ncol(credit)], positive="Yes")

gbm1_best_probpred <- predict(gbm1_best, newdata = test[,-ncol(credit)], type = "prob")
gbm1_best_auc <- roc.curve(scores.class0 = gbm1_best_probpred$Yes,
                     weights.class0 = as.numeric(test$default)-1, curve = T)
gbm1_best_auc$auc
```

```
## [1] 0.7156459
```

When comparing the performance of `gbm1_best` and other models from Week 7 on the same hold-out test set, the results are as follows: the accuracy of `gbm1_best` is 0.7792642, the AUC is 0.7156459, the sensitivity is 0.1641791, the specificity is 0.9568966, and the F-score is 0.25.

Below, we use the `varImp` function in `caret` to visualize the feature importance of the tuned GBM model (and later, we will use the corresponding functions in the gbm package).

```
####Feature Importance####
#varImp(gbm1_best)
plot(varImp(gbm1_best), top = 15)
```



2.2 Training and Tuning with GBM

Now, let's employ the gbm package to train and fit a GBM model. When using `gbm` directly, there are additional arguments that can be controlled.

```
#Adjust the format of Y for gbm
credit2 <- credit
credit2$default <- as.numeric(credit2$default)-1
```

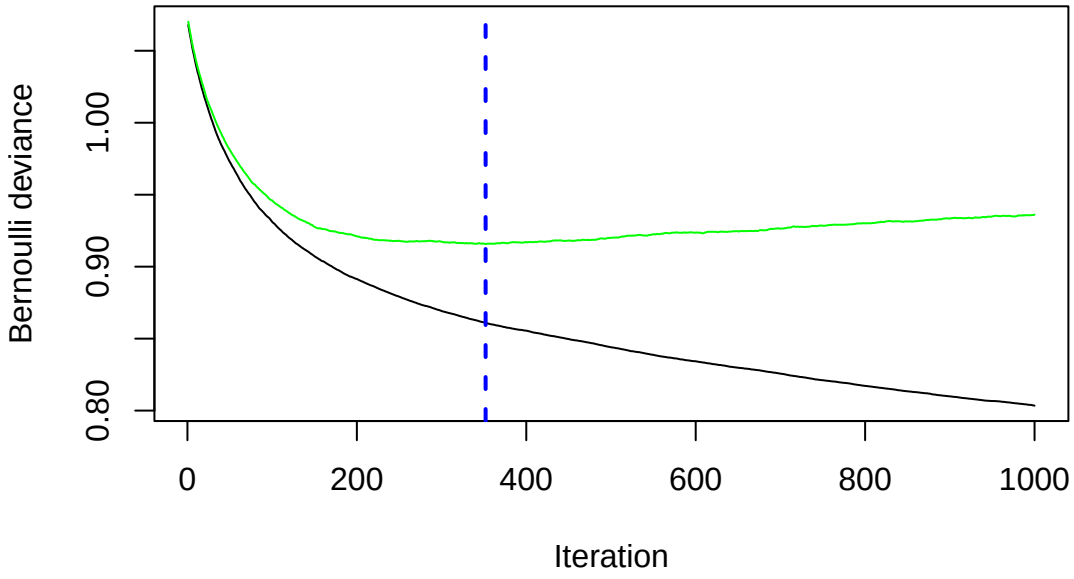
```r
train2 <- credit2[index, ]; test2 <- credit2[-index, ] #apply the same split

set.seed(904)
gbm2 <- gbm(formula = default ~ .,
               data = train2,
               distribution = "bernoulli",
               n.trees = 1000,
               interaction.depth = 1,
               shrinkage = 0.01,
               cv.folds = 5,
               n.cores = NULL, # will use all cores by default
               verbose = FALSE)
# print results
print(gbm2)
```

```
## gbm(formula = default ~ ., distribution = "bernoulli", data = train2,
##     n.trees = 1000, interaction.depth = 1, shrinkage = 0.01,
##     cv.folds = 5, verbose = FALSE, n.cores = NULL)
## A gradient boosted model with bernoulli loss function.
## 1000 iterations were performed.
## The best cross-validation iteration was 352.
## There were 23 predictors of which 20 had non-zero influence.
```

```r
# get the minimum Bernoulli deviance
min(gbm2$cv.error)
```

```
## [1] 0.9157407
```

```r
# plot loss function as a result of n trees added to the ensemble
gbm.perf(gbm2, method = "cv")
```




```
## [1] 352
```

```
#The black curve is training error;
#the green curve is CV error;
#the vertical blue dashed line represents the optimal number of trees=352.
```

> **Question:** Can you infer the types of errors represented by the black and green lines, even without the presence of a legend?

The default settings in `gbm` include a learning rate (`shrinkage`) of 0.1 (here, we adjusted it to 0.01), and `gbm` uses a default number of trees of 100 (here, we adjusted it to 1000). The default depth of each tree (`interaction.depth`) is 1, which means we are ensembling a bunch of stumps. Lastly, I have included `cv.folds` to perform a 5-fold cross-validation.

From the generated plot, a clear distinction emerges: the black line represents the training error, which decreases with more iterations (or trees). Meanwhile, the green line signifies the cross-validation error, optimized at 352 trees. Both errors are measured in terms of Bernoulli deviance when the distribution is Bernoulli.

Now, we would like to perform a manual grid search and a template is provided below. Note that you can introduce stochastic gradient descent by allowing `bag.fraction` < 1.

> **Exercise:** Use the provided code below to create your grid of hyperparameter combinations. Feel free to modify and experiment with the code to observe potential variations in the optimal values of tuning parameters. Consider the following questions:
>
> - Will a larger learning rate (e.g., 0.3) lead to an optimal or sub-optimal fit?
> - Do the top models predominantly consist of stumps (`interaction.depth` = 1)? What if you introduce more training data?
> - Can the inclusion of a stochastic component (`bag.fraction`) enhance the performance?

```r
# This code chunk won't be executed but will be displayed
# Exercise
# create hyperparameter grid
hyper_grid <- expand.grid(shrinkage = c(0.01),
                          interaction.depth =  c(1, 3, 5),
                          n.minobsinnode = c(5, 10, 15),
                          bag.fraction = c(0.8, 1),
                          optimal_trees = 0, # a place to dump results
                          min_error = 0 # a place to dump results
                          )

# total number of combinations
nrow(hyper_grid)

# grid search
for(i in 1:nrow(hyper_grid)) {

  # reproducibility
  set.seed(123)

  # train model
  gbm.tune <- gbm(
    formula = default ~ .,
    distribution = "bernoulli",
    data = train2,
    n.trees = 2000,
    interaction.depth = hyper_grid$interaction.depth[i],
    shrinkage = hyper_grid$shrinkage[i],
    n.minobsinnode = hyper_grid$n.minobsinnode[i],
    bag.fraction = hyper_grid$bag.fraction[i],
    cv.folds = 5,
    n.cores = NULL, # will use all cores by default
    verbose = FALSE
  )

  # add min training error and trees to grid
  hyper_grid$optimal_trees[i] <- which.min(gbm.tune$cv.error)
  hyper_grid$min_error[i] <- sqrt(min(gbm.tune$cv.error))
}

hyper_grid %>%
  dplyr::arrange(min_error) %>%
  head(10)
```

The provided code below represents the final model using the template grid values without any modifications.

```r
# train GBM model
set.seed(456)

gbm2_best <- gbm(
  formula = default ~ .,
  distribution = "bernoulli",
  data = train2,
  n.trees = 240,
  interaction.depth = 3,
```

```
  shrinkage = 0.01,
  n.minobsinnode = 10,
  bag.fraction = 0.8,
  train.fraction = 1,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
)

gbm2_best_probpred <- predict(gbm2_best, newdata = test2[,-ncol(credit)], type = "response")
gbm2_best_auc <- roc.curve(scores.class0 = gbm2_best_probpred,
                     weights.class0 = test2$default, curve = T)
gbm2_best_auc$auc
```

```
## [1] 0.7233016
```

```
gbm2_best_pred <- ifelse(gbm2_best_probpred > 0.5, 1, 0)
gbm2_best_conf <- confusionMatrix(as.factor(gbm2_best_pred),
                                  as.factor(test2[,ncol(credit)]), positive="1")
```

When comparing the performance of `gbm2_best` to `gbm1_best` and other models from Week 7 on the same hold-out test set, the results are as follows: the accuracy of `gbm2_best` is 0.7959866, the AUC is 0.7233016, the sensitivity is 0.2985075, the specificity is 0.9396552, and the F-score is 0.3960396.

3 Feature Importance in gbm

3.1 The Credit Data

The `summary` method for `gbm` produces a data frame and a plot that reveal the most influential variables. The parameter `cBars` allows you to adjust the number of variables displayed in terms of influence, following the order of importance. By default, the method used is `relative.influence`.

```
# Set up the layout and plot margins
par(mfcol = c(1, 2))
par(mar = c(5, 7, 1, 1))

# Plot for relative.influence
invisible(summary(gbm2_best, cBars = 15, method = relative.influence, las = 2))
#Note that invisible() function to prevent the R output from being displayed

# Plot for permutation.test.gbm
invisible(summary(gbm2_best, cBars = 15, method = permutation.test.gbm,las = 2))
```

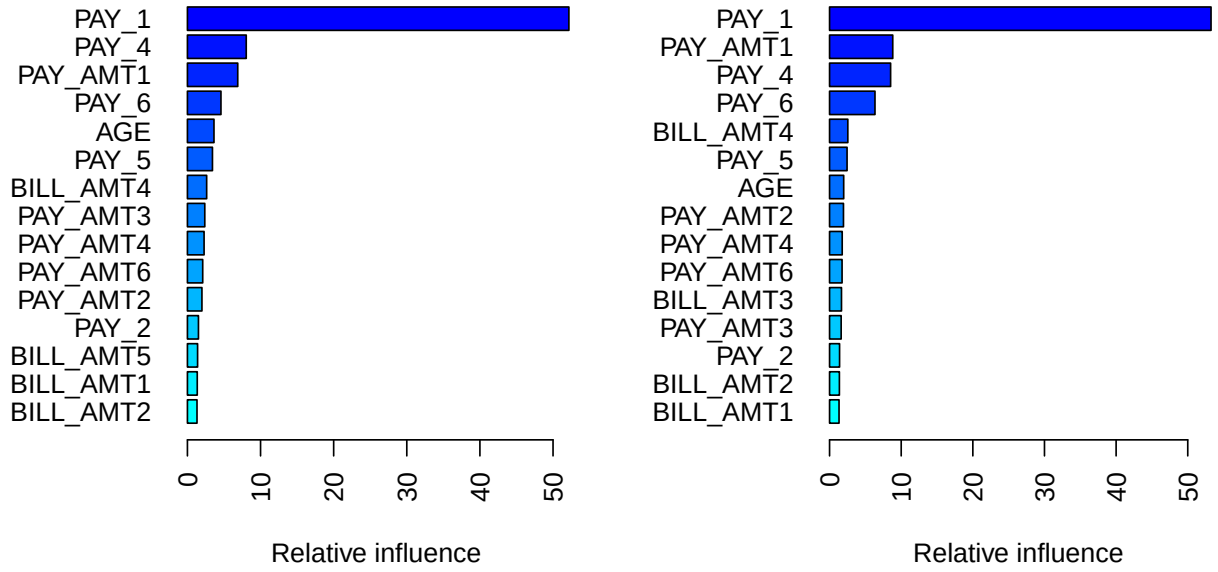


Figure 1: Feature Importance Using Two Methods: relative.influence (lhs) and permutation.test.gbm (rhs)

```
# Reset the layout and plot margins to the default
par(mfcol = c(1, 1))
par(mar = c(5.1, 4.1, 4.1, 2.1))
```

- When `method = relative.influence`: This approach aligns with Friedman (2001). At each split in each tree, `gbm` calculates the improvement in the split criterion for a specific metric (e.g., Gini impurity and MSE). This improvement is then averaged for each variable across all the trees that the variable is used. The variables with the largest average decrease in the given metric are considered most important.
- When `method = permutation.test.gbm`: For each tree, the entire training sample is passed down the tree and the prediction accuracy is recorded. Then the values for each variable (one at a time) are ran-

domly permuted and the accuracy is again computed. The decrease in accuracy as a result of this randomly "shaking up" of variable values is averaged over all the trees for each variable. The variables with the largest average decrease in accuracy are considered most important.

3.2 Limitations of Permutation Feature Importance (Recall from Week 7)

Permutation feature importance is a technique used to assess the importance of a feature by calculating the increase in a model's prediction error after randomly permuting that feature. Its primary advantage lies in its model-agnostic nature, which makes it applicable to a wide array of machine learning algorithms. However, it does have several limitations, as outlined by the Society of Actuaries for actuaries (SOA, 2021).

- Permuting correlated features may result in the creation of unrealistic observations. This may result in bias in the feature importance metrics.
- Correlated features may have their importance reduced when they contain similar information.

4 Partial Dependence Plots

4.1 The Credit Data

Partial Dependence (PD) plots illustrate the marginal effect that one or two features have on the predicted outcome of a machine learning model. This type of plot is a global method, considering all instances and offering insights into the global relationship between a feature and the predicted outcome. In the context of classification, where the machine learning model produces probabilities, the PD plot displays the probability for a certain class given different values for the target feature S. For additional information, you can refer to Molnar (2023).

```
pd_age <- partial(gbm2_best, pred.var = "AGE", plot = TRUE, prob = TRUE,
        n.trees = gbm2_best$n.tree, plot.engine = "ggplot2", lwd = 2, col = "blue")
#pd_age$data
pd_age
```

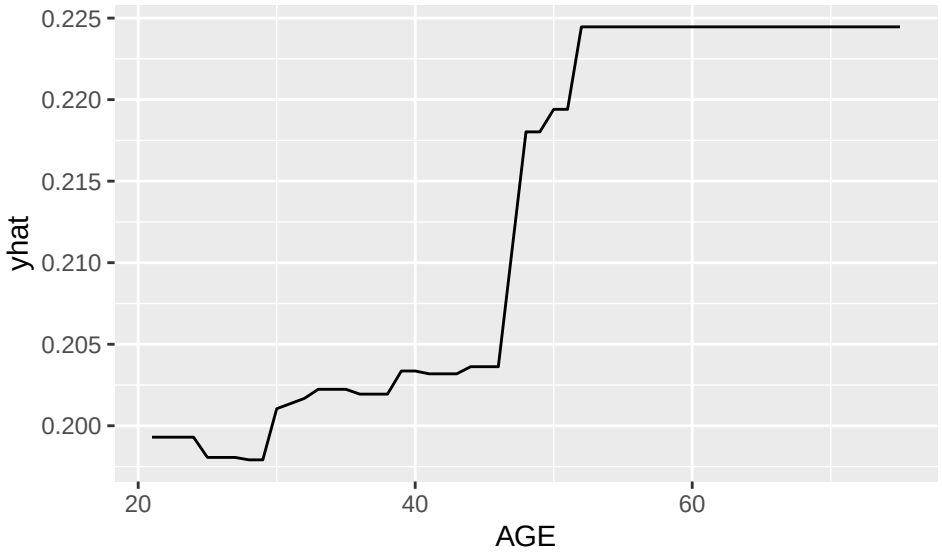


Figure 2: Partial Dependence Plot for Age

From the PD plot above for age, it's evident that a general increasing trend is observed. The risk of default becomes higher as people get older.

```
pd_pay1 <- partial(gbm2_best, pred.var = "PAY_1", plot = TRUE, prob = TRUE,
                   n.trees = gbm2_best$n.tree, plot.engine = "ggplot2", lwd = 2, col = "blue")

pd_pay1_plot <- ggplot(data = pd_pay1$data, aes(x = PAY_1, y = yhat)) +
  geom_bar(stat = "identity", fill = "blue") +
  labs(title = "Partial Dependence Plot for PAY_1",
       x = "PAY_1",
       y = "Predicted Probability") +
  theme_minimal()

print(pd_pay1_plot)
```

The PD plot is also applicable to categorical features. Calculating PD for categorical features is straightforward. For each category, a PD estimate is obtained by setting all data instances to have the same category.

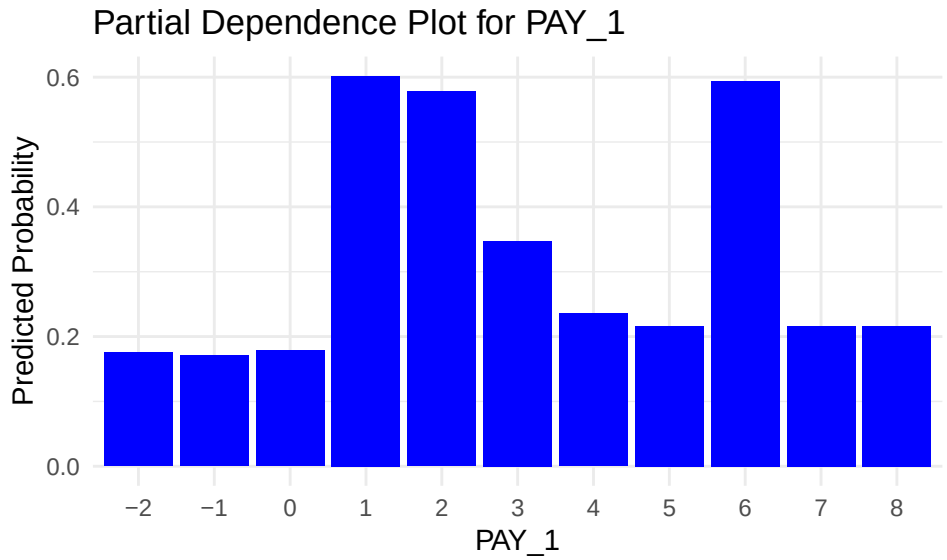


Figure 3: Partial Dependence Plot for PAY 1

`PAY_1` is the most influential feature as identified by earlier by feature importance. This observation is not surprising, given that `PAY_1` represents the repayment status in September 2005, the latest repayment status in the dataset (which includes past monthly payment records from April to September 2005). To recap, these categories correspond to: -2: No consumption; -1: Paid in full; 0: The use of revolving credit; 1 = payment delay for one month; 2 = payment delay for two months; and so on up to 8 = payment delay for eight months; 9 = payment delay for nine months and above. The above PD plot reveals that our fitted GBM model effectively discerns individuals with payment delays as being at higher risk, particularly within categories 1, 2, 3, and 6.

4.2 Limitations of PD Plots

PD plots are a popular interpretation tool due to their calculation using a concise algorithm that is both easy to implement and comprehend. Nevertheless, PD plots have several limitations, as summarized by the Society of Actuaries for actuaries (SOA, 2021).

- PD plots can be computationally expensive as it recalculates predictions on the entire dataset per point on a single feature plot.
- PD plots assumes independence between features, so correlation between features can cause potentially misleading interpretations. A review of feature correlation values is recommended prior to running a PD plot.
- The calculation in the PD plots is an average and is, therefore, susceptible to any skewed data or outliers.