Week 4 Code Examples, CSC 400, Spring 2024

1. Ensemble Methods
   a. Adaboost
   b. GBM
   c. XGBoost
2. Naïve Bayes
   a. Gaussian
   b. Multinomial
   c. Bernoulli
3. K-mean
4. LDA (linear discriminant analysis)
5. Nearest Centroid classifier
6. Visualization

Adaboost

We'll start with an example with the iris dataset.

```
library(adabag)
library(caret)
data <- iris
head(data)
summary(data)
dim(data)
```

We'll split into testing and training sets, and then build our model.

```
parts = createDataPartition(data$Species, p = 0.8, list = F)
train = data[parts, ]
test = data[-parts, ]

model_adaboost <- boosting(Species~., data=train, boos=TRUE, mfinal=50)
summary(model_adaboost)
```

Then we can test the model and see how we did on the test set.

```
pred_test = predict(model_adaboost, test)
pred_test
accuracy_model <- (sum(diag(pred_test$confusion)))/length(test[,1])
accuracy_model
```

GBM

We'll start by installing several packages (some just for visualization), and import the data we'll be using. As with other applications, we'll create a test and train split.

```
library(rsample)
library(gbm)
library(xgboost)
library(caret)
library(h2o)
library(pdp)
library(ggplot2)
library(lime)
library(AmesHousing)
set.seed(123)
ames_split <- initial_split(AmesHousing::make_ames(), prop = .7)
ames_train <- training(ames_split)
ames_test  <- testing(ames_split)
```

When we run the model, the session will use the multiple cores your computer is designed with instead of just one. But depending on the size of your data and the speed of your machine, it may still take some time to process. Since we are modeling sale price here, our example is a regression model and so uses the Gaussian distribution. If you are doing binary classification, then choose binomial as your distribution. See the documentation for other options (for example, count data may need a poisson distribution).

```
gbm.fit <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 10000,
  interaction.depth = 1,
  shrinkage = 0.001,
  cv.folds = 5,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
)
print(gbm.fit)
```

Then we can assess the fit and the number of trees needed to optimize the model.

```
sqrt(min(gbm.fit$cv.error))
gbm.perf(gbm.fit, method = "cv")
```

Typically, these ensemble methods have several hyperparameters that will need to be tuned to get the best fit. In this next example, we'll adjust the shrinkage to be a bit larger and reduce the number of trees. Because the change in gradient is adjusted less slowly (more quickly), it requires fewer iterations to get to the optimal solution in this example.

```
gbm.fit2 <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 5000,
  interaction.depth = 3,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
)
min_MSE <- which.min(gbm.fit2$cv.error)
sqrt(gbm.fit2$cv.error[min_MSE])
gbm.perf(gbm.fit2, method = "cv")
```

Because there is more than one parameter to tune, we need to find an efficient method of testing many possible combinations.

```
hyper_grid <- expand.grid(
  shrinkage = c(.01, .1, .3),
  interaction.depth = c(1, 3, 5),
  n.minobsinnode = c(5, 10, 15),
  bag.fraction = c(.65, .8, 1),
  optimal_trees = 0,
  min_RMSE = 0
)
nrow(hyper_grid)
```

In this case, we've set four parameters with three options each, leaving $3^4 = 81$ possible combinations in this example. If you thought the last model took a long time, this one will take longer.

```
random_index <- sample(1:nrow(ames_train), nrow(ames_train))
random_ames_train <- ames_train[random_index, ]

for(i in 1:nrow(hyper_grid)) {
  #set.seed(123)
  gbm.tune <- gbm(
    formula = Sale_Price ~ .,
    distribution = "gaussian",
    data = random_ames_train,
    n.trees = 5000,
    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],
    train.fraction = .75,
    n.cores = NULL, # will use all cores by default
    verbose = FALSE
  )
  hyper_grid$optimal_trees[i] <- which.min(gbm.tune$valid.error)
  hyper_grid$min_RMSE[i] <- sqrt(min(gbm.tune$valid.error))
}
```

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

After running the model, you'll get a bunch of outputs that will allow you to narrow the range of values tested in your hypergrid. This will allow you to home in on the best combination of parameters. Repeat as necessary to get to your best model, then rerun your initial GBM model with those parameter settings.

```
gbm.fit.final <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 483,
  interaction.depth = 5,
  shrinkage = 0.1,
  n.minobsinnode = 5,
  bag.fraction = .65,
  train.fraction = 1,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
)
```

We can then test and visualize our final model.

```
par(mar = c(5, 8, 1, 1))
summary(
  gbm.fit.final,
  cBars = 10,
  method = relative.influence, # also can use permutation.test.gbm
  las = 2
)

gbm.fit.final %>%
  partial(pred.var = "Gr_Liv_Area", n.trees = gbm.fit.final$n.trees, grid.resolution = 100) %>%
  autoplot(rug = TRUE, train = ames_train) +
  scale_y_continuous(labels = scales::dollar)
```

The following method produces a really cool looking graph.

```
ice1 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
  ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1) +
  ggtitle("Non-centered") +
  scale_y_continuous(labels = scales::dollar)
```

```
ice2 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
  ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1, center = TRUE) +
  ggtitle("Centered") +
  scale_y_continuous(labels = scales::dollar)
gridExtra::grid.arrange(ice1, ice2, nrow = 1)
```

And we can make predictions with our model.

```
model_type.gbm <- function(x, ...) {
  return("regression")
}

predict_model.gbm <- function(x, newdata, ...) {
  pred <- predict(x, newdata, n.trees = x$n.trees)
  return(as.data.frame(pred))
}

local_obs <- ames_test[1:2, ]
explainer <- lime(ames_train, gbm.fit.final)
explanation <- explain(local_obs, explainer, n_features = 5)
plot_features(explanation)

pred <- predict(gbm.fit.final, n.trees = gbm.fit.final$n.trees, ames_test)
caret::RMSE(pred, ames_test$Sale_Price)
```

Our example is with regression, but this also works for classification, though you'll use different metrics for the model. The documentation for the package can help you update these settings.

XGBoost

We'll look at an XGBoost model for classification using the iris dataset.

```
library(xgboost)
library(caret)
library(e1071)

data <- iris
parts = createDataPartition(data$Species, p = 0.7, list = F)
train = data[parts, ]
test = data[-parts, ]
```

We've split into test and training sets. If you look at the data, you see that the fifth column is the categorical variable, so we split that off as our output.

```
X_train = data.matrix(train[,-5])
y_train = train[,5]
X_test = data.matrix(test[,-5])
y_test = test[,5]
```

The XGBoost package uses a matrix format for the model building, so we need to convert our data to the correct type.

```
xgboost_train = xgb.DMatrix(data=X_train, label=y_train)
xgboost_test = xgb.DMatrix(data=X_test, label=y_test)
```

Now we can build our model. We'll start with a tree-depth of 3 and 50 iterations.

```
model <- xgboost(data = xgboost_train,
                 max.depth=3, ,
                 nrounds=50)
summary(model)
```

Then we can make our predictions on the test data and find the confusion matrix.

```
pred_test = predict(model, xgboost_test)
pred_test
pred_test[(pred_test>3)] = 3
pred_y = as.factor((levels(y_test))[round(pred_test)])
print(pred_y)
conf_mat = confusionMatrix(y_test, pred_y)
print(conf_mat)
```

Naïve Bayes

We'll start by importing packages, and then our data, which we'll split into test and training data.

```
library(rsample)
library(dplyr)
library(ggplot2)
library(caret)
library(h2o)

attrition <- attrition %>%
  mutate(
    JobLevel = factor(JobLevel),
    StockOptionLevel = factor(StockOptionLevel),
    TrainingTimesLastYear = factor(TrainingTimesLastYear)
  )
set.seed(123)
split <- initial_split(attrition, prop = .7, strata = "Attrition")
train <- training(split)
test  <- testing(split)
```

Then, we can inspect the data we want to model.

```
table(train$Attrition) %>% prop.table()
table(test$Attrition) %>% prop.table()

train %>%
  filter(Attrition == "Yes") %>%
  select_if(is.numeric) %>%
  cor() %>%
  corrplot::corrplot()
```

```
train %>%
  select(Age, DailyRate, DistanceFromHome, HourlyRate, MonthlyIncome, MonthlyRate) %>%
  gather(metric, value) %>%
  ggplot(aes(value, fill = metric)) +
  geom_density(show.legend = FALSE) +
  facet_wrap(~ metric, scales = "free")
```

Next, we split the data into the inputs (x) and the outputs (y) to be predicted. We'll use cross validation to check our results.

```
features <- setdiff(names(train), "Attrition")
x <- train[, features]
y <- train$Attrition

train_control <- trainControl(
  method = "cv",
  number = 10
)
```

Then we train the model and look at the confusion matrix.

```
library(klaR)
nb.m1 <- train(
  x = x,
  y = y,
  method = "nb",
  trControl = train_control
)

confusionMatrix(nb.m1)
```

We can adjust the model and retune it by adjusting hyperparameters. Then we look at the results to see the best version of these parameters.

```
search_grid <- expand.grid(
  usekernel = c(TRUE, FALSE),
  fL = 0:5,
  adjust = seq(0, 5, by = 1)
)
```

```
nb.m2 <- train(
  x = x,
  y = y,
  method = "nb",
  trControl = train_control,
  tuneGrid = search_grid,
  preProc = c("BoxCox", "center", "scale", "pca")
)

nb.m2$results %>%
  top_n(5, wt = Accuracy) %>%
  arrange(desc(Accuracy))
```

We can also look at the results graphically.

```
plot(nb.m2)
```

Once we have the best model on the training data, we can look at the test data and compare to predictions.

```
confusionMatrix(nb.m2)

pred <- predict(nb.m2, newdata = test)
confusionMatrix(pred, test$Attrition)
```

A brief example of how we can make this a gaussian naïve bayes model is shown below with some fabricated data.

```
library(naivebayes)

cols <- 10 ; rows <- 100
M <- matrix(rnorm(rows * cols, 100, 15), nrow = rows, ncol = cols)
y <- factor(sample(paste0("class", LETTERS[1:2]), rows, TRUE, prob = c(0.3,0.7)))
colnames(M) <- paste0("V", seq_len(ncol(M)))

gnb <- gaussian_naive_bayes(x = M, y = y)
summary(gnb)
head(predict(gnb, newdata = M, type = "class"))
head(predict(gnb, newdata = M, type = "prob"))
coef(gnb)
```

We can look at a similar process for a multinomial naïve bayes model with simulated data.

```
cols <- 10 ; rows <- 100
M <- matrix(sample(0:5, rows * cols, TRUE, prob = c(0.95, rep(0.01, 5))),
            nrow = rows, ncol = cols)
y <- factor(sample(paste0("class", LETTERS[1:2]), rows, TRUE, prob = c(0.3,0.7)))
colnames(M) <- paste0("V", seq_len(ncol(M)))
laplace <- 1

mnb <- multinomial_naive_bayes(x = M, y = y, laplace = laplace)
summary(mnb)
head(predict(mnb, newdata = M, type = "class"))
head(predict(mnb, newdata = M, type = "prob"))
coef(mnb)
```

And a similar example with Bernoulli Naïve Bayes with a simulated example.

```
cols <- 10 ; rows <- 100 ; probs <- c("0" = 0.9, "1" = 0.1)
M <- matrix(sample(0:1, rows * cols,  TRUE, probs), nrow = rows, ncol = cols)
y <- factor(sample(paste0("class", LETTERS[1:2]), rows, TRUE, prob = c(0.3,0.7)))
colnames(M) <- paste0("V", seq_len(ncol(M)))
laplace <- 0

bnb <- bernoulli_naive_bayes(x = M, y = y, laplace = laplace)
summary(bnb)
head(predict(bnb, newdata = M, type = "class"))
head(predict(bnb, newdata = M, type = "prob"))
coef(bnb)
```

K-Means

While this algorithm is technically a clustering algorithm, it is often used in a semi-supervised fashion as a classification algorithm. We can check its accuracy against the original classification labels after clustering. The number of classes tells us how to set k in this context. Without those labels, we'd need to experiment with the value to k to find the best model (which we'll talk about more when we do clustering).

Our example will be on the iris dataset, and we'll remove the initial species labels and compare them to our results later on.

```
data(iris)
str(iris)
library(clusterR)
library(cluster)
iris_1 <- iris[, -5]
```

We can run the model with k=3 since there are three species. Then look the confusion matrix to see how things line up.

```
set.seed(240)
kmeans.re <- kmeans(iris_1, centers = 3, nstart = 20)
kmeans.re
kmeans.re$cluster
cm <- table(iris$Species, kmeans.re$cluster)
cm
```

One issue here is that the numbering of the clusters may not line up optimally with the named labels, so we'll have to match the ones that produce the best fit with the data.

We want to visualize the data and the data centers (against two of the variables).

```
plot(iris_1[c("Sepal.Length", "Sepal.Width")])
plot(iris_1[c("Sepal.Length", "Sepal.Width")],
     col = kmeans.re$cluster)
plot(iris_1[c("Sepal.Length", "Sepal.Width")],
     col = kmeans.re$cluster,
     main = "K-means with 3 clusters")

kmeans.re$centers
kmeans.re$centers[, c("Sepal.Length", "Sepal.Width")]


points(kmeans.re$centers[, c("Sepal.Length", "Sepal.Width")],
       col = 1:3, pch = 8, cex = 3)
y_kmeans <- kmeans.re$cluster
clusplot(iris_1[, c("Sepal.Length", "Sepal.Width")],
         y_kmeans,
         lines = 0,
         shade = TRUE,
         color = TRUE,
         labels = 2,
         plotchar = FALSE,
         span = TRUE,
         main = paste("Cluster iris"),
         xlab = 'Sepal.Length',
         ylab = 'Sepal.Width')
```

Linear Discriminant Analysis (LDA)

This algorithm should not be confused with the other LDA—Latent Dirichlet Allocation. To use this algorithm, we'll run it on the same iris dataset as above.

```
library(klaR)
library(psych)
library(MASS)
library(devtools)

pairs.panels(iris[1:4],
             gap = 0,
             bg = c("red", "green", "blue")[iris$Species],
             pch = 21)
```

Create a test and training set and run the model on the training data.

```
set.seed(123)
ind <- sample(2, nrow(iris),
              replace = TRUE,
              prob = c(0.6, 0.4))
training <- iris[ind==1,]
testing <- iris[ind==2,]

linear <- lda(Species~., training)
linear
```

One of the graphs we want to use needs to be installed a little differently.

```
options(repos = c(
    fawda123 = 'https://fawda123.r-universe.dev',
    CRAN = 'https://cloud.r-project.org'))

# Install ggord
install.packages('ggord')
```

```
library(ggord)
ggord(linear, training$Species, ylim = c(-10, 10))
```

And we can make some partition plots.

```
partimat(Species~., data = training, method = "lda")
partimat(Species~., data = training, method = "qda")
```

Let's also look at the confusion matrix and accuracy for both the training data and the test data.

```
p1 <- predict(linear, training)$class
tab <- table(Predicted = p1, Actual = training$Species)
tab

sum(diag(tab))/sum(tab)

p2 <- predict(linear, testing)$class
tab1 <- table(Predicted = p2, Actual = testing$Species)
tab1

sum(diag(tab1))/sum(tab1)
```

Nearest Centroid Classifier

For our example, we'll import libraries and create some simulated data to model. We'll also plot the data to take a look at it before we model.

```
library(lolR)
library(ggplot2)
library(MASS)
n=400
d=2

testdat <- lol.sims.mean_diff(n, d)
X <- testdat$X
Y <- testdat$Y

data <- data.frame(x1=X[,1], x2=X[,2], y=Y)
data$y <- factor(data$y)
ggplot(data, aes(x=x1, y=x2, color=y)) +
  geom_point() +
  xlab("x1") +
  ylab("x2") +
  ggtitle("Simulated Data") +
  xlim(-4, 6) +
  ylim(-4, 4)
```

We model the data with the nearest centroid classifier. We plot the results afterwards.

```
classifier <- lol.classify.nearestCentroid(X, Y)

data <- cbind(data, data.frame(size=1))
data <- rbind(data, data.frame(x1=classifier$centroids[,1],
                               x2=classifier$centroids[,2], y="center", size=5))
ggplot(data, aes(x=x1, y=x2, color=y, size=size)) +
  geom_point() +
  xlab("x1") +
  ylab("x2") +
  ggtitle("Data with estimated Centers") +
  guides(size=FALSE) +
  xlim(-4, 6) +
  ylim(-4, 4)
```

Then we make predictions based on the centers. We can look at how the data is classified from this process.

```
Yhat <- predict(classifier, X)
data$y[1:(length(data$y) - 2)] <- Yhat
ggplot(data, aes(x=x1, y=x2, color=y, size=size)) +
  geom_point() +
  xlab("x1") +
  ylab("x2") +
  ggtitle("Data with Predictions") +
  guides(size=FALSE) +
  xlim(-4, 6) +
  ylim(-4, 4)
```

From here, we'd want to compare the predictions to the original classifications with a confusion matrix. The nearest centroid classifier tends to be more accurate when the classes have greater separation between them, as one might expect.

Resources:

1. https://www.projectpro.io/recipes/apply-adaboost-or-classification-r
2. https://rpubs.com/praveen_jalaja/adaboosting
3. https://uc-r.github.io/gbm_regression
4. https://www.projectpro.io/recipes/apply-xgboost-for-classification-r
5. https://www.r-bloggers.com/2021/04/naive-bayes-classification-in-r/
6. https://uc-r.github.io/naive_bayes
7. https://search.r-project.org/CRAN/refmans/naivebayes/html/gaussian_naive_bayes.html
8. https://search.r-project.org/CRAN/refmans/naivebayes/html/multinomial_naive_bayes.html
9. https://search.r-project.org/CRAN/refmans/naivebayes/html/bernoulli_naive_bayes.html
10. https://www.datacamp.com/tutorial/k-means-clustering-r
11. https://www.r-bloggers.com/2021/05/linear-discriminant-analysis-in-r/
12. https://fawda123.github.io/ggord/
13. https://cran.r-hub.io/web/packages/loIR/vignettes/nearestCentroid.html