Machine Learning

INFO 640 | Pratt Institute | McSweeney

```
#install.packages("factoextra")
#install.packages("rpart") #to make decision trees
#install.packages("rpart.plot") #to visualize decision trees
#install.package("rattle")
#install.packages(RColorBrewer) #nice colors

library(tidyverse)
library(rpart)
library(rpart.plot)
library(rattle)
library(RColorBrewer)
library(class)
library(factoextra)
```

This is meant to be a very high level overview of what you can do with machine learning in R. We are going to use the packages that come preloaded, and in the most straightforward applications. However, as you will see, you can do a lot with the basics. As an introduction, the primary purpose is to prepare you to learn more on your own, and introduce what is possible. As you will see, you have already been using some of these techniques in the other labs.

We will work with numerical datasets in 3 ways: classification, regression, and clustering. Some major methods are omitted from this lab (Naive Bayes, PCA, neural nets, SVMs, Logistic Regression, etc.).

Regression

The goal of regression is to learn a pattern and then situate a point within the pattern. Regression requires a large amount of pre-labeled data to learn the pattern. Regression works best with numerical variables. You can always transform a categorical variable to make it numerical, but if you have categorical variables, you may want to look into classification instead.

Basic Linear Regression

You have already done this in the regression lab, but we will go through a review here. Basic Linear Regression is one of the most productive machine learning algorithms. Linear Regression forms the foundation of Multivariate Regression (more than 1 input variable), which forms the foundation for classical approaches to a wide variety of predictive models.

We will look at data about housing sales in Texas. We want to know when the best time to list a house is. Based on personal observation and our understanding of supply and demand, we suspect that the more houses are available on the market, the less we will get for our house when we sell it.

The Texas housing marker dataset is preloaded in R as 'txhousing'. Remember, it is always best practice to inspect your data before working with it.

```
data("txhousing")
?txhousing
summary(txhousing)
head(txhousing)
```

We would like to build a model representing median sale price as a function of the number of listings. Remember that when building linear regression models, we always pass the outcome variable (y) first and the predictor variable (x) second.

```
lm_housing <- lm(median ~ listings, data=txhousing)
```

Let's say we see that there are 799 houses for sale, we want to know if we should sell ours, too. Let's see how much we could get for it if there are 800 houses for salle.

We'll use our linear model to predict the median sale price with 800 listings in the area using the 'predict' function with our new, unseen dataset that has a column named the exact same way as our other predictor variables.

```
unseen <- data.frame(listings = 800)
predict(lm_housing, unseen)
```

Great, we have a prediction of the median value of a house if there are 800 houses on the market. But we have NO IDEA how good or bad that model is. Let's check it out by calling summary on our model to review the R-squared value

```
summary(lm_housing)
```

Question

Should we trust this model? What other variables do you have available to you? Try to see if another variable is a better predictor of median housing price. Hint: Some variables are not cadidates for regression - why?

```
lm_housing1 <- lm(median ~ ____, data=txhousing)
summary(lm_housing1)
```

Multiple Regression

Clearly our linear model was much too narrow. It could be the case that the number of listings affects the sale price, but it's more likely that factors such as square footage, condition, etc. affect the sale price more. To take all of these factors into consideration, we would need multiple regression. Unfortuantely Texas Housing doesn't have the information we need, so let's go to the internet and get some data from Streeteasy.

We're looking at NYC now, so we'll shift our focus to looking at rent rather than sale price. We'd like to know which variables are most predictive of the sale price, so let's train our model on all of our numerical variables.

```
street <- read_csv("../Datasets/streeteasy.csv")
summary(street)
```

```
street_select6 <- street %>% select(rent, bedrooms,bathrooms, size_sqft, min_to_subway,floor

lm_street_select6 <- lm(rent ~ ., data=street_select6)
summary(lm_street_select6)
```

If we look at this summary, it also tells us how much each variable influences the model. The lower the p value (Pr(>|t|)), the more the variable influences the model. Most of these variables influence the model equally, but it seems that minutes to the subway and the building age are less predictive of the rent than other variables.

Because adding more variables adds more complexity, which hurts the model, it is better to asses the quality of the model with the adjusted R-squared, which accounts for this complexity.

Now let's compare this model with 2 others. First, a model that only takes square footage into account and a model that takes everything except neighborhood, submarket, borough. Remember that if you want ALL of your variables, you can use a .

```
lm_street_size <- lm(___ ~ ___, data=___)
___(___)

street_select16 <- street[,1:17]
#inspect your data
str(___)

lm_street_select16 <- lm(___ ~ ___, data=___)
___(___)
```

Which model is the best predictor of the rent of an apartment? In that model, which variable(s) contribute the most to the output?

Classification: Decision Trees

Decision trees require some randomness. To make it reproducible, we will set a seed for our classifier to refer to. It's the rpart function that needs the seed.

```
set.seed(1234)
```

Let's use the Titanic dataset. We will use the version distributed by Kaggle that has already been broken into training and testing sets. https://www.kaggle.com/c/titanic/data

```
train <- read.csv("Desktop/INFO640-Labs/Datasets/titanic/train.csv", header=TRUE)
test <- read.csv("Desktop/INFO640-Labs/Datasets/titanic/test.csv", header=TRUE)
```

Inspect our sets

```
str(train)
str(test)
```

That's a lot of variables, we don't need or want all of those.

```
train <- train %>%
  select(Pclass, Sex, Age, Survived)

test <- test %>%
  select(Pclass, Sex, Age, Survived)
```

We will use the rpart library to build a decision tree model to predict who survived. Since all of our varaibles are factors (they are categorical), we will use the 'Class' function. For more functions, see the rpart documentation. To select all of the columns, use a . We could replace Pclass + Sex + Age with . to select all of our variables.

'`tree <- rpart(Survived ~ Pclass + Sex + Age , train, method = "class") fancyRpartPlot(tree)`

Follow this tree to see if a Female in 3rd class over age 39 would have likely survived.

Now that we've learned our tree, let's test it. We'll use the predict function to create a table of predictions with our test set.

```
pred <- predict(tree, test, type = "class")
```

To understand the outcome of our predictions, we need to make a confusion matrix. This is a table of the Real and Predicted outcomes.

```
conf <- table(test$Survived, pred)
conf
```

Wow! It looks like our decision tree has remarkable accuracy! It only got 13 wrong.Let's calculate the True Positives/Negatives and Falso Positives/Neagatives. We will use these to calculate the accuracy, precision, and recall. To calculate the accuracy, we can sum what we got correct (represented along the diagonal) over everything. Precision is the ratio of True Positives to everything labeled positive. Recall is the ratio of True Positives to everything actually positive.

```
# Assign TP, FN, FP and TN using conf
TP <- conf[1, 1] # this will be 212
FN <- conf[1, 2] # this will be 78
FP <- conf[2,1] # fill in
TN <- conf[2,2] # fill in

# Calculate and print the accuracy: acc
acc <- sum(TP,TN)/sum(TP,TN,FP,FN)
acc

# Calculate and print out the precision: prec
prec <- TP/sum(TP,FP)
prec

# Calculate and print out the recall: rec
rec <- TP/sum(TP,FN)
rec
```

There is a shortcut to calculate accuracy:

```
acc <- sum(diag(conf))/sum(conf)
acc
```

That's a pretty amazing result. The thing is that we already knew the outcome here. What about if we didn't know.

We might end up making a really specific model. Overfitting means that it works for the instances it was trained on but may not generalize to a larger, noisier environment. This is a common problem in machine learning, and results in highly interpretable but unreliable datasets. The model below has been overfit to the data.

```
overfit_tree <- rpart(Survived ~ Pclass + Sex + Age , train, method = "class", control = rpa
fancyRpartPlot(overfit_tree)
```

There are too many nodes and decisions. This could probably be simpler and therefore more robust. Let's prune our tree with the prune function and see if it gets more generalizable.

```
pruned <- prune(tree, cp=.01)
fancyRpartPlot(pruned)
```

Classification: KNN

Let's turn our attention to Classification Algorithms. We will use k Nearest Neighbors to classify our data.

Let's start by making vectors (lists) of just the outcome variables for both our train and test sets. First we'll drop any na values since they will cause problems in our algorithms

```
train <- drop_na(train)
test <- drop_na(test)
train_labels <- train$Survived
test_labels <- test$Survived
```

We want to drop the answers, so we will drop the "Survived" column. First, make new dataframes of our train and test sets so that we can get back to our original data if we need to. Then remove the answers ('Survived') from our knn_train and knn_test sets since we are trying to predict who survives, not train on who survives.

```
knn_train <- train
knn_test <- test

knn_train$Sex <- as.factor(knn_train$Sex)
knn_test$Sex <- as.factor(knn_test$Sex)

knn_train$Sex <- as.factor(gsub("male", "1", knn_train$Sex))
knn_train$Sex <- as.factor(gsub("female", "0", knn_train$Sex))

knn_test$Sex <- as.factor(gsub("male", "1", knn_test$Sex))
knn_test$Sex <- as.factor(gsub("female", "0", knn_test$Sex))

knn_train$Sex <- as.numeric(knn_train$Sex)
knn_test$Sex <- as.numeric(knn_test$Sex)

knn_train$Survived <- NULL
knn_test$Survived <- NULL
```

The point of kNN is to situate each datapoint on a multi-dimensional plane and figure out the shortest distance to all the other points. For this to work, all of the variables must be on the same scale.

Class is a factor that is not easily comparable to something like Age since they are on incompatible scales. We will need to normalize our variables. The idea is to put each variable on a scale relevant to itself, so we need to find the minimum and maximum values for each variable and transform each datum into a proportion of that scale.

Normalize Pclass

```
min_class <- min(knn_train$Pclass)
max_class <- max(knn_train$Pclass)
knn_train$Pclass <- (knn_train$Pclass - min_class) / (max_class - min_class)
knn_test$Pclass <- (knn_test$Pclass - min_class) / (max_class - min_class)
```

Normalize Age

```
min_age <- min(knn_train$Age)
max_age <- max(knn_train$Age)
knn_train$Age <- (knn_train$Age - min_age) / (max_age - min_age)
knn_test$Age <- (knn_test$Age - min_age)/(max_age - min_age)
```

Now we can make our predictions based on the k-nearest neighbors. We will use 5. It's best practice to use an odd number, and you will rarely need more than 7. We'll make a new variable, 'pred' and call the knn function on it. The knn function requires: training set (knn_train) testing set (knn_test) vector of true classifications of the training set (cl) number of neighbors to consider (k)

```
k_pred <- knn(train=knn_train, test=knn_test, cl=train_labels, k=5)
k_pred
```

Now let's make a confusion table of the results with our test_labels as the true values, and the pred as the predicted.

```
conf <- table(test_labels, k_pred)
conf
```

knn is sometimes refered to as lazy. The only way to apply our classifier to a new dataset is to treat that dataset as a test variable. So we will make a new dataframe. I have PRENORMALIZED this data for the sake of brevity. If you had a new dataset, you would have to put it through the same normalization process you did before to normalize all of your variables

```
my_titanic <- data.frame("Age" = .6, "Pclass" = .5, "Sex" = 0)
new_k_pred <- knn(train=knn_train, test=my_titanic, cl=train_labels, k=5)
new_k_pred
```

My example person is predicted to have survived!

There are lots more clustering algorithms: Naive Bayes, Logistic Regression, etc. For more on these, take Machine Learning next semester :)

Clustering

Basic Clustering - Unsupervised

We'll start with kmeans clustering on a dataset that we already know the answer to: the iris dataset. We will separate the descriptor variables from the Species, and use the descriptors (Sepal Length & Width, Petal Length & Width) along with the kmeans algorithm to cluster our datapoints. If we did not have the

Species label, we could only know that we have 3 groups. We would need domain knowledge to know what those 3 groups are.

```
set.seed(1234)
data(iris)
```

To select our dataset, we use bracket notation. The call is [rows,columns]. We want all of the rows, so leave the first argument blank, and we want columns in the 1st position through the 4th. We also want the species as a separate dataset.

```
my_iris <- iris[,1:4]
species <- iris$Species
```

We already know that we should have 3 groups, and will ask R to restart 10 times to get the average answer from all 10 of these trials. We will show later how to figure out how many clusters you need.

```
kmeans_iris <- kmeans(my_iris, centers=3, nstart=10)
kmeans_iris
```

This object tells us a lot of useful information about our clusters, such as the centroid for each variable/dimension and how many objects are in each group.

We can use the predicted values to see how well our clustering performed. We'll use the table function to calculate the confusion matrix.

```
table(species, kmeans_iris$cluster)
```

Not too bad - it looks like the versicolor was miscategorized once, and the virginica was miscategorized 14 times. We should probably look closer at what distinguishes the two, because clearly only petal & sepal length & width is not enough.

Let's plot the result of our kmeans clustering to illustrate how our classifier performed. We are going to only plot the Petal variables because we only have 2 dimensions to work with here.

```
plot(Petal.Length ~ Petal.Width, data = my_iris, col = kmeans_iris$cluster)
```

That looks great, but we have a much better way to visualize these clusters - with the factoextra package.

```
fviz_cluster(kmeans_iris, data = my_iris)
```

This tells us not only the boundaires of each category, but how much each dimension is playing a role and the labels for each item.

Clustering Unsupervised Unknown Clusters

If we didn't know how many clusters to ask for, we could try a lot by iterating through a for loop. This look will allow us to try out values of k from 1 to 15. It will add each result in a list. Cleverly, the number of k is the position in the list, so we can just add to our empty list. We want to save the value of the Within Sum of Squares (wss) as a proportion of the Total Sum of Squares. We are looking for a value around .2

```
# Initialize total within sum of squares error: wss
wss_tot <- 0
for (i in 1:15) {
  km_out <- kmeans(my_iris, centers = i, nstart = 10)
  # Save total within sum of squares to wss variable
  wss_tot[i] <- km_out$tot.withinss/km_out$totss
}
```

Now to find the optimal k, we will plot our wss_tot ratio and look for the elbow. The elbow is the optimum number of clusters.

```
plot(1:15, wss_tot, type = "b",
     xlab = "Number of Clusters",
     ylab = "WSS/TSS")
```

Notably, it looks like 2 would be equally good of a choice. This is reflected in our data. It was clear that there was one cluster that was very different from all the other clusters. However, the other 2 clusters had some overlap, so we may have gotten a similarly acceptable numerical result based on these 4 variables with 2 clusters as we got with 3.

Now it's your turn. Use the US Arrests dataset (loaded below)

You are going to 1. loop through the numbers 1:15 to get the wss/tot for each data point 2. Identify the best k for this data 3. Use that k to build a new kmeans cluster 4. plot the result

```
data("USArrests")
str(USArrests)

#1
wss_tot <- 0
for (i in ___) {
  km_out <- kmeans(___, centers = ___, nstart = 10)

  # Save total within sum of squares to wss variable
  wss_tot[___] <- km_out$___/km_out$___
}

#2
plot(1:15, wss_tot, type = "b",
     xlab = "Number of Clusters",
     ylab = "WSS/TSS")

#3
```

```
kmeans_arrests <- kmeans(___, centers= ___, nstart=10)
kmeans_arrests

#4
plot(Murder ~ Assault, data = ___, col = kmeans_arrests$___)
fviz_cluster(___, data = ___)
```