Homework 5

PSTAT 131/231

Contents

Elastic Net Tuning

```
library(tidymodels)
library(MASS)
library(dplyr)
library(ISLR)
library(ISLR2)
library(tidyverse)
#install.packages("glmnet")
library(glmnet)
tidymodels_prefer()
```

For this assignment, we will be working with the file `"pokemon.csv"`, found in `/data`. The file is from Kaggle: https://www.kaggle.com/abcsds/pokemon.

The Pokémon franchise encompasses video games, TV shows, movies, books, and a card game. This data set was drawn from the video game series and contains statistics about 721 Pokémon, or "pocket monsters." In Pokémon games, the user plays as a trainer who collects, trades, and battles Pokémon to (a) collect all the Pokémon and (b) become the champion Pokémon trainer.

Each Pokémon has a primary type (some even have secondary types). Based on their type, a Pokémon is strong against some types, and vulnerable to others. (Think rock, paper, scissors.) A Fire-type Pokémon, for example, is vulnerable to Water-type Pokémon, but strong against Grass-type.

The goal of this assignment is to build a statistical learning model that can predict the **primary type** of a Pokémon based on its generation, legendary status, and six battle statistics.

Read in the file and familiarize yourself with the variables using `pokemon_codebook.txt`.

```
Poke <- read.csv(file="/Users/honchowayne/Desktop/Pokemon.csv")
Poke %>% head()
```

```
##   X.                  Name Type.1 Type.2 Total HP Attack Defense Sp..Atk
## 1  1             Bulbasaur  Grass Poison   318 45     49      49      65
## 2  2               Ivysaur  Grass Poison   405 60     62      63      80
## 3  3              Venusaur  Grass Poison   525 80     82      83     100
## 4  3 VenusaurMega Venusaur  Grass Poison   625 80    100     123     122
```

Figure 1: Fig 1. Vulpix, a Fire-type fox Pokémon from Generation 1.

```
## 5  4              Charmander   Fire            309 39      52       43       60
## 6  5              Charmeleon   Fire            405 58      64       58       80
##   Sp..Def Speed Generation Legendary
## 1      65    45          1     False
## 2      80    60          1     False
## 3     100    80          1     False
## 4     120    80          1     False
## 5      50    65          1     False
## 6      65    80          1     False
```

Exercise 1

Install and load the `janitor` package. Use its `clean_names()` function on the Pokémon data, and save the results to work with for the rest of the assignment. What happened to the data? Why do you think `clean_names()` is useful?

```
library(janitor)
Poke <- clean_names(Poke)
Poke %>% head()
```

```
##   x                  name type_1 type_2 total hp attack defense sp_atk sp_def
## 1 1             Bulbasaur  Grass Poison   318 45     49      49     65     65
## 2 2               Ivysaur  Grass Poison   405 60     62      63     80     80
## 3 3              Venusaur  Grass Poison   525 80     82      83    100    100
## 4 3 VenusaurMega Venusaur  Grass Poison   625 80    100     123    122    120
## 5 4            Charmander   Fire          309 39     52      43     60     50
## 6 5            Charmeleon   Fire          405 58     64      58     80     65
##   speed generation legendary
## 1    45          1     False
## 2    60          1     False
## 3    80          1     False
## 4    80          1     False
## 5    65          1     False
## 6    80          1     False
```

I noticed that clean.names() funciton changed each column name a bit. The original “.” turned into “ ” *and upper case turned into lower case. It is useful because resulting names are unique and consist only of the* character, numbers, and letters. Capitalization preferences can be specified using the case parameter, so it helps us to call on and manipulate data. Also, it would be useful to use on variables with spaces or periods in them so that we could use the data$variable method to get specific values.

Exercise 2

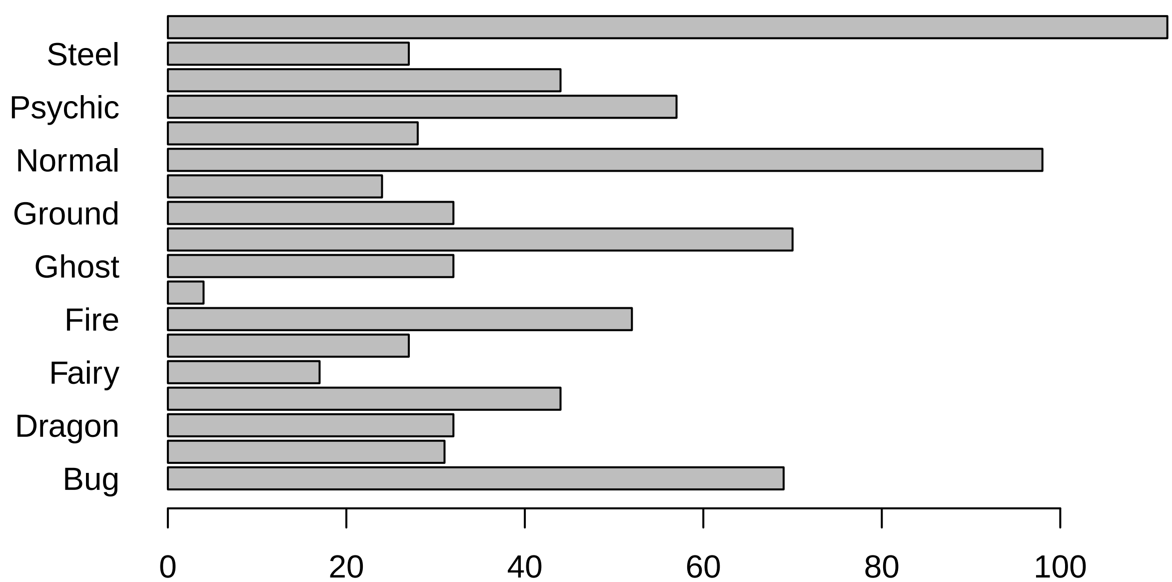
Using the entire data set, create a bar chart of the outcome variable, `type_1`.

How many classes of the outcome are there? Are there any Pokémon types with very few Pokémon? If so, which ones?

For this assignment, we'll handle the rarer classes by simply filtering them out. Filter the entire data set to contain only Pokémon whose `type_1` is Bug, Fire, Grass, Normal, Water, or Psychic.

After filtering, convert `type_1` and `legendary` to factors.

```
type_1 <- table(Poke$type_1)
barplot(type_1, horiz = TRUE, las =1)
```



From the plot, we can see that there are 18 types and the Flying type is the fewest one.

```
Poke<- filter(Poke,type_1 %in% c('Bug', 'Fire', 'Grass', 'Normal', 'Water', 'Psychic'))
Poke$type_1 <- factor(Poke$type_1)
Poke$legendary <- factor(Poke$legendary)
Poke$generation <- factor(Poke$generation)
str(Poke)
```

```
## 'data.frame':	458 obs. of  13 variables:
##  $ x         : int  1 2 3 3 4 5 6 6 6 7 ...
##  $ name      : chr  "Bulbasaur" "Ivysaur" "Venusaur" "VenusaurMega Venusaur" ...
##  $ type_1    : Factor w/ 6 levels "Bug","Fire","Grass",..: 3 3 3 3 2 2 2 2 2 6 ...
##  $ type_2    : chr  "Poison" "Poison" "Poison" "Poison" ...
##  $ total     : int  318 405 525 625 309 405 534 634 634 314 ...
##  $ hp        : int  45 60 80 80 39 58 78 78 78 44 ...
##  $ attack    : int  49 62 82 100 52 64 84 130 104 48 ...
##  $ defense   : int  49 63 83 123 43 58 78 111 78 65 ...
##  $ sp_atk    : int  65 80 100 122 60 80 109 130 159 50 ...
##  $ sp_def    : int  65 80 100 120 50 65 85 85 115 64 ...
##  $ speed     : int  45 60 80 80 65 80 100 100 100 43 ...
##  $ generation: Factor w/ 6 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ legendary : Factor w/ 2 levels "False","True": 1 1 1 1 1 1 1 1 1 1 ...
```

Exercise 3

Perform an initial split of the data. Stratify by the outcome variable. You can choose a proportion to use. Verify that your training and test sets have the desired number of observations.

Next, use *v*-fold cross-validation on the training set. Use 5 folds. Stratify the folds by `type_1` as well. *Hint: Look for a `strata` argument.* Why might stratifying the folds be useful?

```
set.seed(3435)
Poke_split <- initial_split(Poke, prop = 0.70, strata = type_1)
Poke_train <- training(Poke_split)
Poke_test <- testing(Poke_split)
```

```
dim(Poke_train)
```

```
## [1] 318  13
```

```
dim(Poke_test)
```

```
## [1] 140  13
```

```
Poke_fold <- vfold_cv(Poke_train, v=5, strata = type_1)
```

Stratification is the process of rearranging the data as to ensure each fold is a good representative of the whole. For example in a binary classification problem where each class comprises 50% of the data, it is best to arrange the data such that in every fold, each class comprises around half the instances.

Exercise 4

Set up a recipe to predict `type_1` with `legendary`, `generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def`.

- Dummy-code `legendary` and `generation`;
- Center and scale all predictors.

```
Poke_rec <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def, Po
  step_dummy(c(legendary,generation)) %>%
  step_center(all_predictors()) %>%
  step_normalize(all_predictors())
```

Exercise 5

We'll be fitting and tuning an elastic net, tuning `penalty` and `mixture` (use `multinom_reg` with the `glmnet` engine).

Set up this model and workflow. Create a regular grid for `penalty` and `mixture` with 10 levels each; `mixture` should range from 0 to 1. For this assignment, we'll let `penalty` range from -5 to 5 (it's log-scaled).

How many total models will you be fitting when you fit these models to your folded data?

we will fit 500 total models

```
ridge_spec <- multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

ridge_workflow <- workflow() %>%
  add_recipe(Poke_rec) %>%
  add_model(ridge_spec)

penalty_mix_grid <- grid_regular(penalty(range = c(-5, 5)), mixture(range = c(0,1)),levels = 10)
penalty_mix_grid %>% head()
```

```
## # A tibble: 6 x 2
##    penalty mixture
##      <dbl>   <dbl>
## 1 0.00001        0
## 2 0.000129       0
## 3 0.00167        0
## 4 0.0215         0
## 5 0.278          0
## 6 3.59           0
```

Exercise 6

Fit the models to your folded data using `tune_grid()`.

Use `autoplot()` on the results. What do you notice? Do larger or smaller values of `penalty` and `mixture` produce better accuracy and ROC AUC?

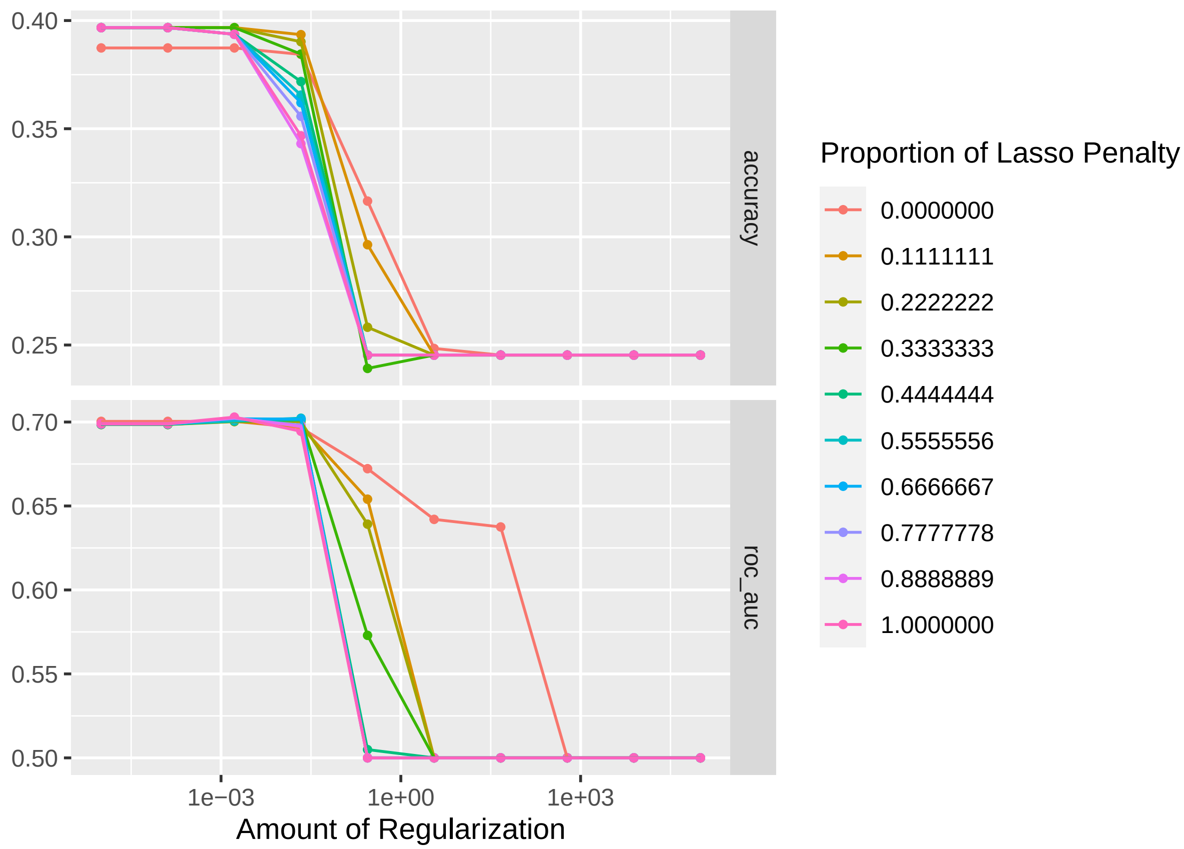
By inspection, the larger value of penalty and mixture tend to do better on accuracy and ROC AUC.

```
tune_res <- tune_grid(ridge_workflow, resamples = Poke_fold, grid = penalty_mix_grid)
tune_res
```

```
## # Tuning results
## # 5-fold cross-validation using stratification
## # A tibble: 5 x 4
```

```
##   splits           id    .metrics           .notes
##   <list>           <chr> <list>             <list>
## 1 <split [252/66]> Fold1 <tibble [200 x 6]> <tibble [0 x 3]>
## 2 <split [253/65]> Fold2 <tibble [200 x 6]> <tibble [0 x 3]>
## 3 <split [253/65]> Fold3 <tibble [200 x 6]> <tibble [0 x 3]>
## 4 <split [256/62]> Fold4 <tibble [200 x 6]> <tibble [0 x 3]>
## 5 <split [258/60]> Fold5 <tibble [200 x 6]> <tibble [0 x 3]>
```

```
autoplot(tune_res)
```



Exercise 7

Use `select_best()` to choose the model that has the optimal `roc_auc`. Then use `finalize_workflow()`, `fit()`, and `augment()` to fit the model to the training set and evaluate its performance on the testing set.

```
collect_metrics(tune_res)
```

```
## # A tibble: 200 x 8
##     penalty mixture .metric  .estimator  mean     n std_err .config
##       <dbl>   <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##   1 0.00001       0 accuracy multiclass 0.387     5  0.0133 Preprocessor1_Model~
##   2 0.00001       0 roc_auc  hand_till  0.700     5  0.0182 Preprocessor1_Model~
##   3 0.000129      0 accuracy multiclass 0.387     5  0.0133 Preprocessor1_Model~
##   4 0.000129      0 roc_auc  hand_till  0.700     5  0.0182 Preprocessor1_Model~
##   5 0.00167       0 accuracy multiclass 0.387     5  0.0133 Preprocessor1_Model~
##   6 0.00167       0 roc_auc  hand_till  0.700     5  0.0182 Preprocessor1_Model~
```

```
##   7 0.0215          0 accuracy multiclass 0.384     5  0.0114 Preprocessor1_Model~
##   8 0.0215          0 roc_auc  hand_till  0.697     5  0.0169 Preprocessor1_Model~
##   9 0.278           0 accuracy multiclass 0.317     5  0.0323 Preprocessor1_Model~
##  10 0.278           0 roc_auc  hand_till  0.672     5  0.0148 Preprocessor1_Model~
## # ... with 190 more rows
```

```r
best <- select_best(tune_res, metric = 'roc_auc')
best
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00167       1 Preprocessor1_Model093
```

```r
ridge_final <- finalize_workflow(ridge_workflow, best)
ridge_final_fit <- fit(ridge_final, data = Poke_train)

augment(ridge_final_fit, new_data = Poke_test) %>%
  accuracy(truth = type_1, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.343
```

Exercise 8

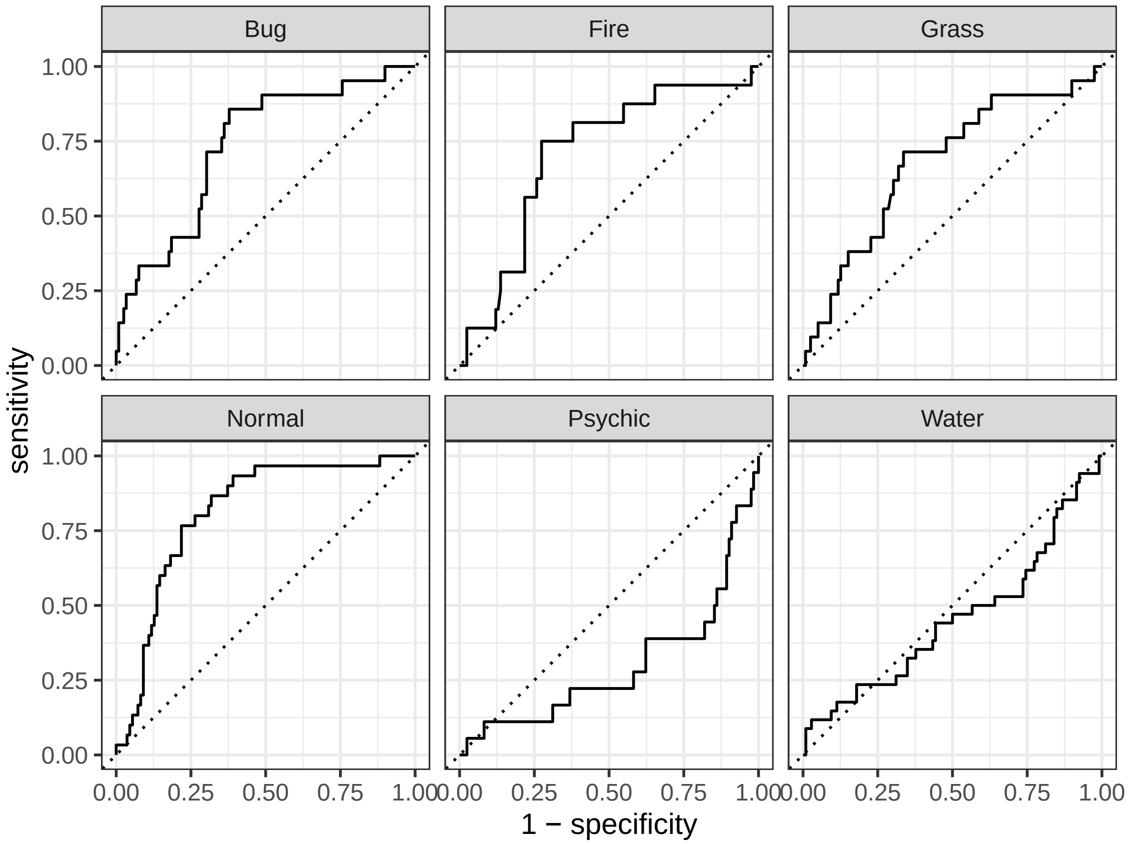
Calculate the overall ROC AUC on the testing set.

Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix.

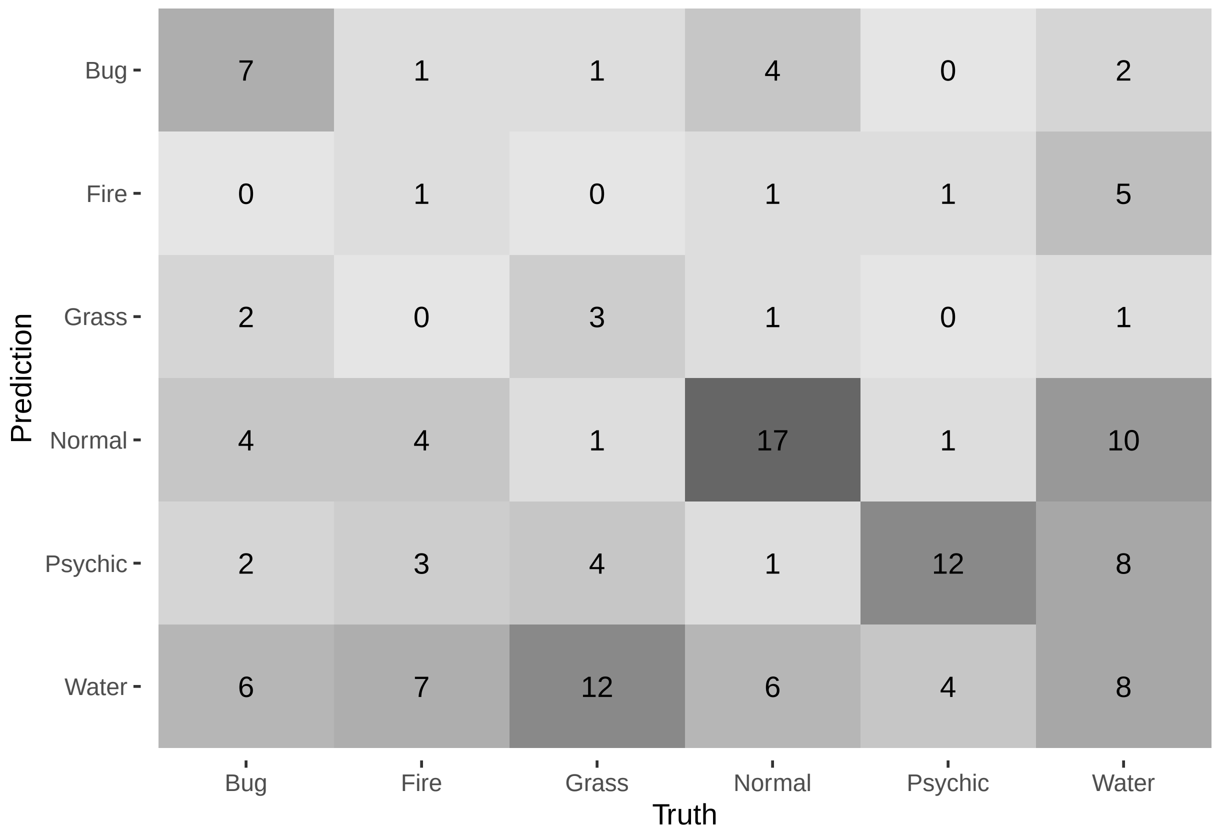
What do you notice? How did your model do? Which Pokemon types is the model best at predicting, and which is it worst at? Do you have any ideas why this might be?

I think the Normal type is the model best at predicting while Water type is the worst because the auc under Normal is the greatest while the auc under Normal is the smallest. What happened might because the data of Normal type is spread out relatively even whole Water type has too many extreme cases.

```r
final_fit <- augment(ridge_final_fit, new_data = Poke_test) %>%
  roc_curve(type_1, estimate = c(.pred_Bug, .pred_Fire, .pred_Grass,
                                 .pred_Normal, .pred_Water, .pred_Psychic))
autoplot(final_fit, type = 'heatmap')
```




```
final_fit <- augment(ridge_final_fit, new_data = Poke_test)
final_fit %>%
  conf_mat(type_1, .pred_class) %>%
  autoplot(type = 'heatmap')
```



For 231 Students

Exercise 9

In the 2020-2021 season, Stephen Curry, an NBA basketball player, made 337 out of 801 three point shot attempts (42.1%). Use bootstrap resampling on a sequence of 337 1's (makes) and 464 0's (misses). For each bootstrap sample, compute and save the sample mean (e.g. bootstrap FG% for the player). Use 1000 bootstrap samples to plot a histogram of those values. Compute the 99% bootstrap confidence interval for Stephen Curry's "true" end-of-season FG% using the quantile function in R. Print the endpoints of this interval.