Demo - Forecasting

ER Deyle

Fall 2023; Marine Semester Block 3

The purpose of this demo is to run through the steps of making model predictions for a set of data. The inspiration is the Ye et al. 2015 paper we've looked at previous, which compares parametric foreacsting and machine learning forecasting for sockeye salmon in the Frasier River system.

Data Wrangling

We load in the data from Ye et al. 2015 PNAS paper which describes stock and recruitment of 9 sockeye salmon stocks in the Frasier River system of British Columbia.

```
install.packages("readxl")
library(readxl)
```

```
data_sockeye <- readxl::read_xls("./DATA/pnas.1417063112.sd02.xls",na = "NA")
```

The analysis in the paper spans all 9 stocks; however the model fits of the Seymour stock specifically are the focus of illustration. To examine just the Seymour stock, we identify and extract the subset of rows of the original data frame that correspond to the Seymour stock.

```
I_seymour_rows <- which(data_sockeye$stk == "Seymour")
print(I_seymour_rows)
```

```
##  [1] 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344
## [20] 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363
## [39] 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382
## [58] 383 384 385 386 387 388 389 390
```

```
data_sockeye_seymour <- data_sockeye[I_seymour_rows,]
```

Note the inclusion of the `which()`. This command takes a vector of `TRUE` and `FALSE` values and returns the indices with `TRUE`. This can be pretty handy. On the other hand, it's not actually necessary in this case. If you use a logical vector (vector of `TRUE` and `FALSE` values) to index another vector `my_vector`, R will return only the elements of `my_vector` that line up with the `TRUE` values.

```
index_vector <- c(TRUE,FALSE,FALSE,TRUE)
my_vector <- 1:4

my_vector[index_vector]
```

```
## [1] 1 4
```

So with the Frasier river Sockeye data frame, we could have written instead

```
I_seymour_rows <- (data_sockeye$stk == "Seymour")
data_sockeye_seymour <- data_sockeye[I_seymour_rows,]
```

If you really want to save space, this can even go on one line.

```
data_sockeye_seymour <- data_sockeye[data_sockeye$stk == "Seymour",]
```

```
data_sockeye_seymour <- subset(data_sockeye, stk == "Seymour")
```

We also note that rows after 2006 don't include values for several variables.

```
tail(data_sockeye_seymour,10)
```

```
## # A tibble: 10 x 7
##    stk        yr     rec4      rec5      rec      eff   juv
##    <chr>   <dbl>    <dbl>     <dbl>    <dbl>    <dbl> <dbl>
##  1 Seymour  2003  0.0112   0.00087   0.0121  0.0185      NA
##  2 Seymour  2004  0.00667  0.000246  0.00692 0.000762    NA
##  3 Seymour  2005  0.00781  0         0.00781 0.00233     NA
##  4 Seymour  2006 NA       NA        NA       0.0578      NA
##  5 Seymour  2007 NA       NA        NA       0.00590     NA
##  6 Seymour  2008 NA       NA        NA       0.000311    NA
##  7 Seymour  2009 NA       NA        NA       0.00309     NA
##  8 Seymour  2010 NA       NA        NA       0.287       NA
##  9 Seymour  2011 NA       NA        NA       0.00799     NA
## 10 Seymour  2012 NA       NA        NA       0.000264    NA
```

So we make one more modification on the rows.

```
data_sockeye_seymour <- data_sockeye[data_sockeye$stk == "Seymour",]
data_sockeye_seymour <- data_sockeye[data_sockeye$yr < 2006,]
```

Then, if you dig through the supplemental code, you will see that the data frame column names get reworked at some point.

```
data_sockeye_seymour <- data_sockeye[data_sockeye$stk == "Seymour",]
data_sockeye_seymour <- data_sockeye_seymour[data_sockeye_seymour$yr < 2006,]
data_sockeye_seymour <- data_sockeye_seymour[,c("yr","rec","eff")]
names(data_sockeye_seymour) <- c("year","recruits","spawners")
```

The Ye et al. paper point out the large differences in parametric model fit that result from looking at just half the data at a time. How would we go about plotting this? There's shorter ways to write all this, but let's go about it systematically. We can create a new column in the data frame to designate the two "halves" of the data. Just like R will let us store a value in a vector position that doesn't exist yet, it will also let us store data in a data frame column that doesn't exist yet. We could do something very simple like store "FALSE" for pre-1977 rows and "TRUE for 1977 and later:

```
data_sockeye_seymour$data_segment <- data_sockeye_seymour$year > 1976
head(data_sockeye_seymour)
```

```
## # A tibble: 6 x 4
##    year recruits spawners data_segment
##   <dbl>    <dbl>    <dbl> <lgl>
## 1  1948   0.0297  0.00128 FALSE
## 2  1949   0.0264  0.00348 FALSE
## 3  1950   0.162   0.00470 FALSE
## 4  1951   0.0688  0.0115  FALSE
## 5  1952   0.0112  0.00278 FALSE
## 6  1953   0.0450  0.00291 FALSE
```

But since we want to make a similar plot to the PNAS paper, we will go a bit fancier and instead use `ifelse()` to assign two different character strings.

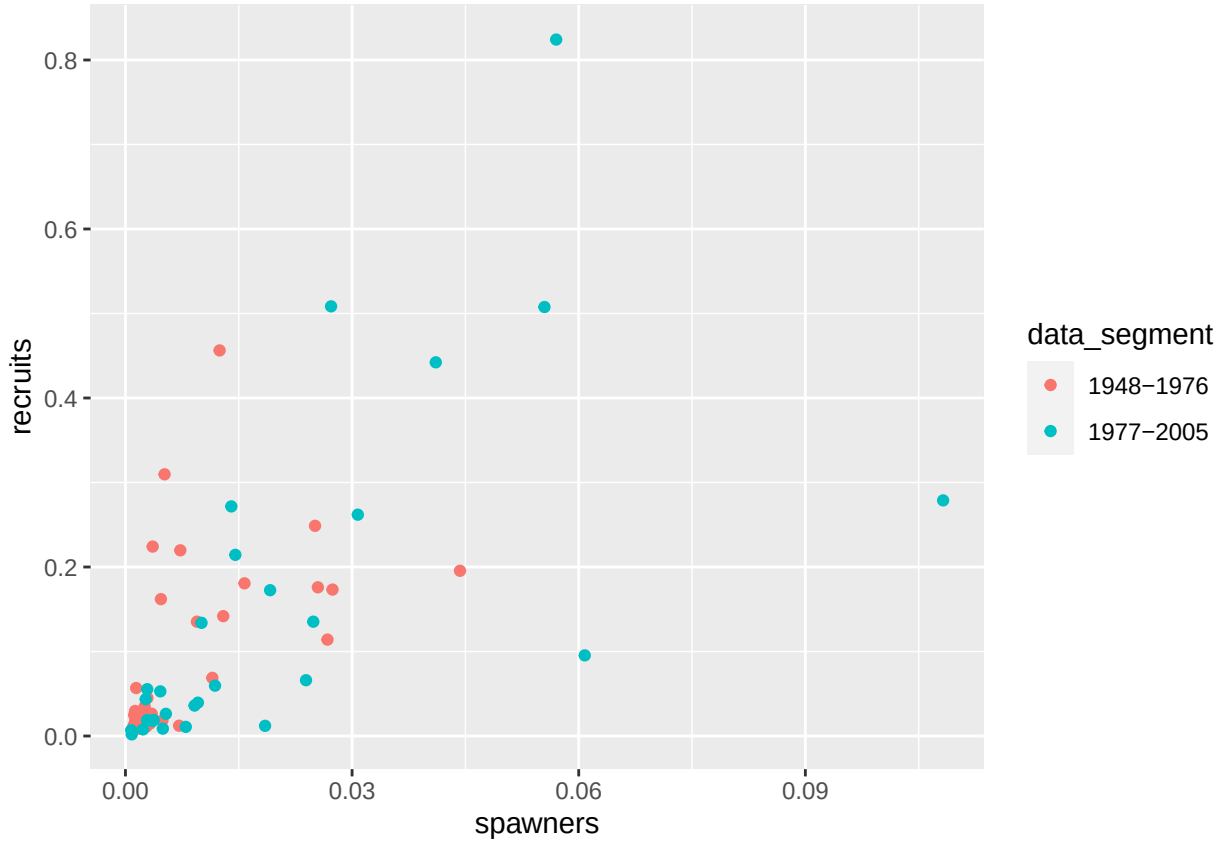
```
data_sockeye_seymour$data_segment <- ifelse(data_sockeye_seymour$year > 1976,"1977-2005","1948-1976")
head(data_sockeye_seymour)
```

```
## # A tibble: 6 x 4
##    year recruits spawners data_segment
##   <dbl>    <dbl>    <dbl> <chr>
## 1  1948   0.0297  0.00128 1948-1976
## 2  1949   0.0264  0.00348 1948-1976
## 3  1950   0.162   0.00470 1948-1976
## 4  1951   0.0688  0.0115  1948-1976
## 5  1952   0.0112  0.00278 1948-1976
## 6  1953   0.0450  0.00291 1948-1976
```

Plotting

Now we can use `ggplot`. Recall that once we tell ggplot which data frame we are starting with, we can refer to columns just by their name.

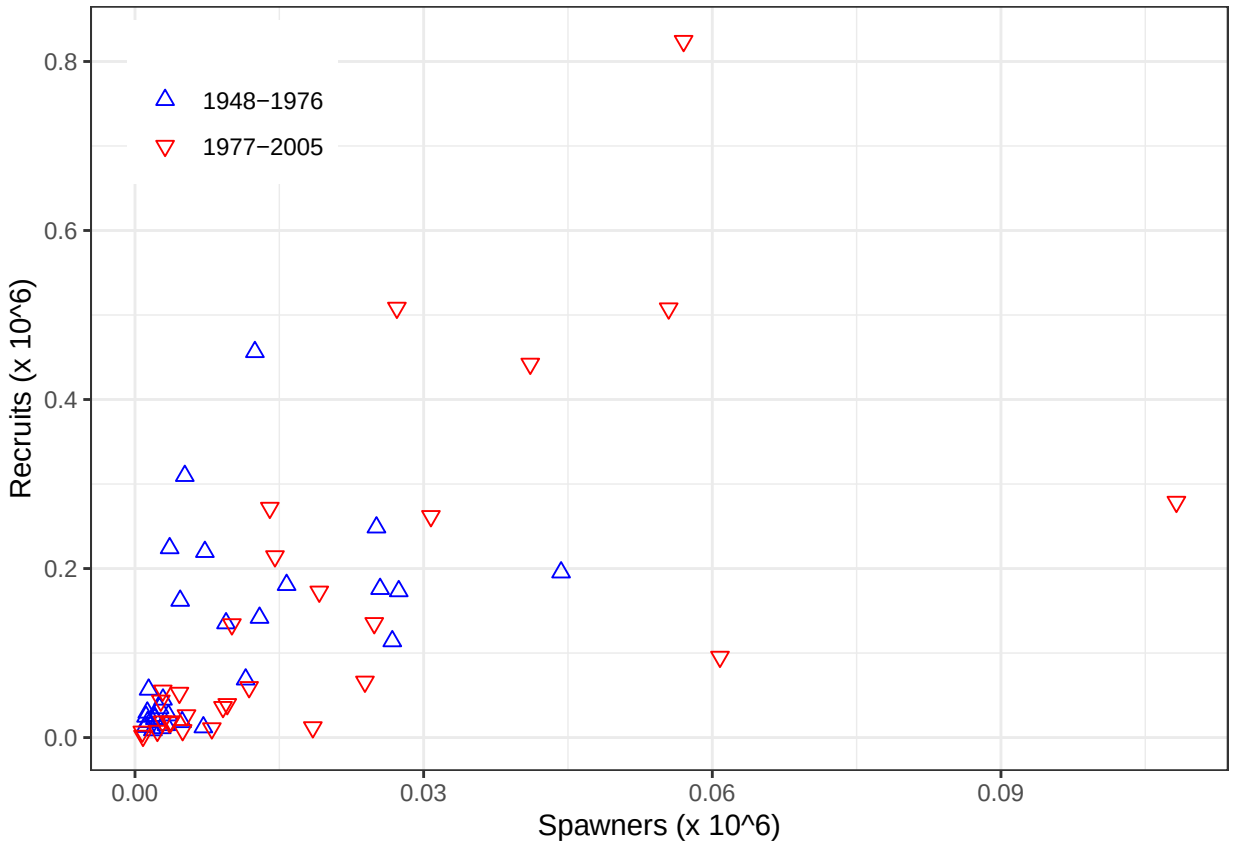
```
library(ggplot2)
ggplot(data=data_sockeye_seymour,aes(x=spawners,y=recruits)) +
  geom_point(aes(color=data_segment))
```



Several things. The points are too small; they are the same shape; they are the wrong shapes; the colors are also wrong!

```
g_1 <- ggplot(data=data_sockeye_seymour,aes(x=spawners,y=recruits)) +
  geom_point(aes(color=data_segment,pch=data_segment),size=2) +
  scale_color_manual(values=c("blue","red"),breaks=c("1948-1976","1977-2005")) +
```

```
  scale_shape_manual(values=c(2,6),breaks=c("1948-1976","1977-2005")) +
  labs(x="Spawners (x 10^6)",y="Recruits (x 10^6)",color="",shape="") +
  theme_bw() + theme(legend.position = c(0.125, 0.875))

print(g_1)
```



Model fitting.

Now comes the fitting part. In the original paper, Ye et al. used a Gibbs sampler package for fitting the Ricker model and implemented their model comparison using 4-fold cross-validations. They included R code. Here it is.

```
ricker_ret_model_4 <- function(df, p4 = 0.95, num_iter = 30000, num_burnin = 20000,
                               model_file = "ricker_model.txt")
{
    # setup data
    pred_recruits <- vector("list", NROW(df))
    jags.params <- c("alpha", "beta")

    lib_segments <- matrix(NA, nrow = 4, ncol = 2)
    segment_size <- NROW(df)/4
    start_index <- 1
    for(i in 1:4)
    {
        lib_segments[i,1] <- floor(start_index)
        end_index <- start_index - 1 + segment_size
        lib_segments[i,2] <- floor(end_index)
```

```
        start_index <- end_index+1
    }
    for(i in 1:4)
    {
        # setup lib and pred
        lib <- lib_segments[-i,]
        pred <- lib_segments[i,]
        lib_index <- do.call(c, lapply(1:NROW(lib), function(x) {lib[x,1]:lib[x,2]}))
        pred_index <- pred[1]:pred[2]

        jags.data <- list(recruits = df$recruits[lib_index], spawners = df$spawners[lib_index])

        # get estimates for params
        min_S_index <- which(jags.data$spawners == min(jags.data$spawners))
        alpha_hat <- log(jags.data$recruits[min_S_index] / jags.data$spawners[min_S_index])
        max_R_index <- which(jags.data$recruits == max(jags.data$recruits))
        beta_hat <- jags.data$spawners[max_R_index]

        # use param estimates to set initial values for chains
        jags.inits <- list(list(alpha = alpha_hat / 2.71828, beta = beta_hat / 2.71828,
                                .RNG.seed = 1234, .RNG.name = "base::Super-Duper"),
                           list(alpha = alpha_hat / 2.71828, beta = beta_hat * 2.71828,
                                .RNG.seed = 1234, .RNG.name = "base::Super-Duper"),
                           list(alpha = alpha_hat * 2.71828, beta = beta_hat / 2.71828,
                                .RNG.seed = 1234, .RNG.name = "base::Super-Duper"),
                           list(alpha = alpha_hat * 2.71828, beta = beta_hat * 2.71828,
                                .RNG.seed = 1234, .RNG.name = "base::Super-Duper"))

        # run jags
        my_model <- jags.model(file = model_file, data = jags.data, inits = jags.inits,
                               n.chains = length(jags.inits))
        update(my_model, n.iter = num_burnin)
        jags_output <- jags.samples(my_model, jags.params, n.iter = num_iter)
        alpha <- as.vector(jags_output[["alpha"]][1,,])
        beta <- as.vector(jags_output[["beta"]][1,,])

        # make prediction
        for(k in pred_index)
        {
            pred_recruits[[k]] <- df$spawners[k] * exp(alpha - beta * df$spawners[k])
        }
    }

    r_pred <- rep.int(NaN, NROW(df))
    for(k in 2:NROW(df))
    {
        r_pred[k] <- median(pred_recruits[[k]] * p4 +
                                pred_recruits[[k-1]] * (1-p4))
    }
    return(r_pred)
}
```

A lot to digest. One thing that will make that hard to follow is the use of a coupled package "JAGS" ("Just Another Gibbs Sampler"). It's usage is a bit weird, and also we have not discussed Gibbs samplers in the

course. It is a variety of optimization that's in the same branch as the Simulated Annealing, "SANN", functionality of R's `optim()`.

Briefly, I'm going to violate the spirit of the class and throw a chunk of pre-written R code without full explanation. This is the "use someone else's package" solution to parameter fitting. Although the "someone else's package" is the companion R-package to the Bolker textbook, `bbmle`, and the fitting tool, `mle2()`, is really a wrapper for the base R `optim()`. To use `mle2()` to fit a model, you can express the model as a function or as a formula for a statistical model. To write statistical models, it's useful to have a reference for R's distributions.

```
?Distributions
```

R describes "The functions for the density/mass function, cumulative distribution function, quantile function and random variate generation are named in the form dxxx, pxxx, qxxx and rxxx respectively." In this case, we use a log-normal distribution because it matches well with the exponential form of the Ricker equation.

```
mle2_seymour_all <- bbmle::mle2(recruits ~ dlnorm(meanlog=a - b*spawners + log(spawners),sdlog=tau_r),
                    data=data_sockeye_seymour,
                    start = list(a=2.2,b=10,tau_r=.5))

mle2_seymour_first <- bbmle::mle2(recruits ~ dlnorm(meanlog=a - b*spawners + log(spawners),sdlog=tau_r)
                    data=data_sockeye_seymour[data_sockeye_seymour$data_segment=="1948-1976",],
                    start = list(a=2.2,b=10,tau_r=.5))
```

```
## Warning in dlnorm(x = c(0.029658, 0.02635, 0.16199, 0.06882, 0.011249, 0.04502,
## : NaNs produced
```

```
mle2_seymour_second <- bbmle::mle2(recruits ~ dlnorm(meanlog=a - b*spawners + log(spawners),sdlog=tau_r
                    data=data_sockeye_seymour[data_sockeye_seymour$data_segment=="1977-2005",],
                    start = list(a=2.2,b=10,tau_r=.5))
```

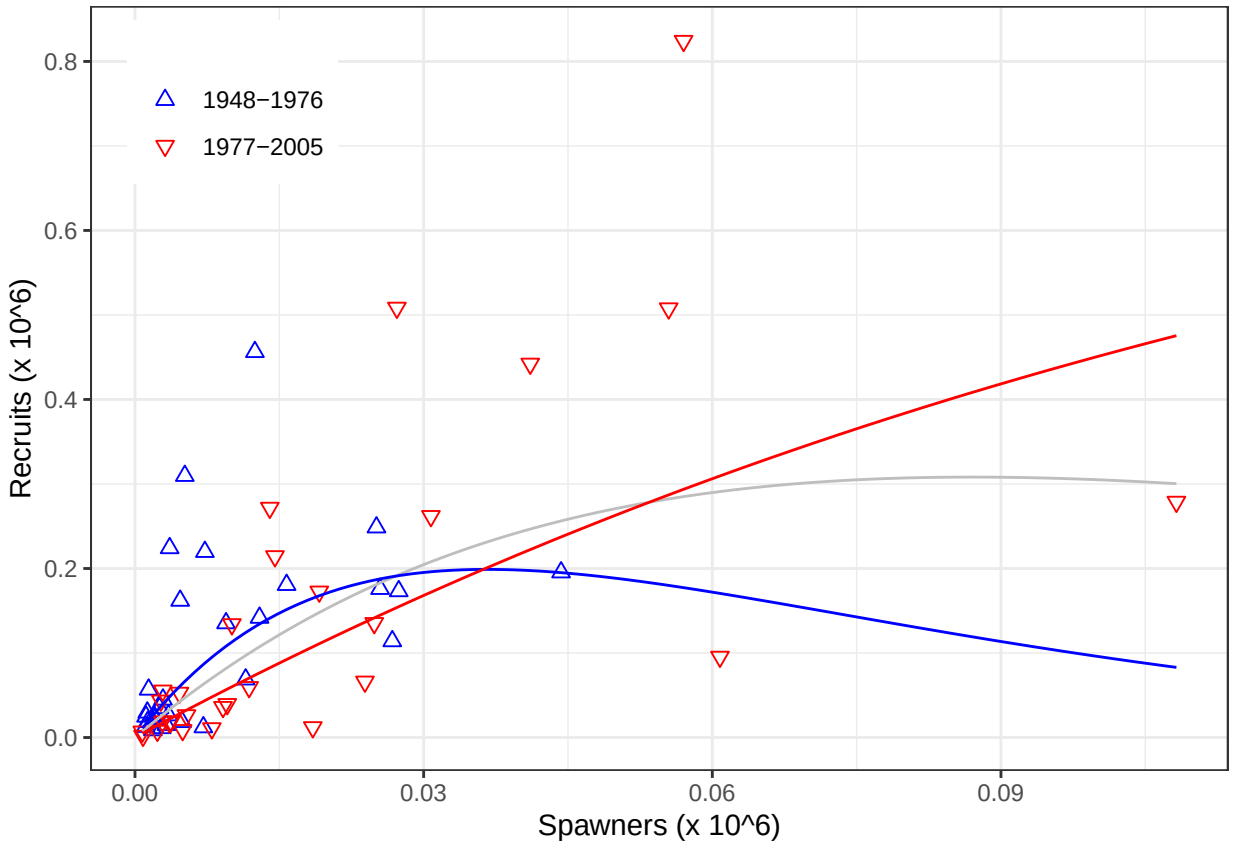
```
## Warning in dlnorm(x = c(0.055414, 0.261925, 0.1353, 0.052848, 0.026268, : NaNs
## produced
```

We can use these to generate costum functions that reflect each specific parameterizations of the Ricker model.

```
f_Ricker <- function(S,a,b) S*exp(a - b*S)

# f_Ricker_fit_all <- function(S) f_Ricker(S,a=bbmle::coef(mle2_seymour_all)[1],b=bbmle::coef(mle2_seym


f_Ricker_fit_all <- function(S) f_Ricker(S,a=2.2647134,b=11.4945836)

f_Ricker_fit_first <- function(S) f_Ricker(S,a=2.693686,b=27.342560)

f_Ricker_fit_second <- function(S) f_Ricker(S,a=1.816421,b=3.107657)
```

Then we can use the ggplot object we saved before with the initial plot and add these functions on top to get a very close reproduction of the figure in Ye et al.

```
g_1 +
  geom_function(fun=f_Ricker_fit_all,color="grey") +
  geom_function(fun=f_Ricker_fit_first,color="blue") +
  geom_function(fun=f_Ricker_fit_second,color="red")
```



However, the real point here was to look at the out-of-sample forecast skill. If we want to predict the stock-recruitment behavior in the second half from the fit in the first half, we already have a function to do just that! It is the function we just plotted `f_Ricker_fit_first()`. To organize our out-of-sample prediction, we make a new data frame to store the observed and predicted values. The observed values are already in hand, of course.

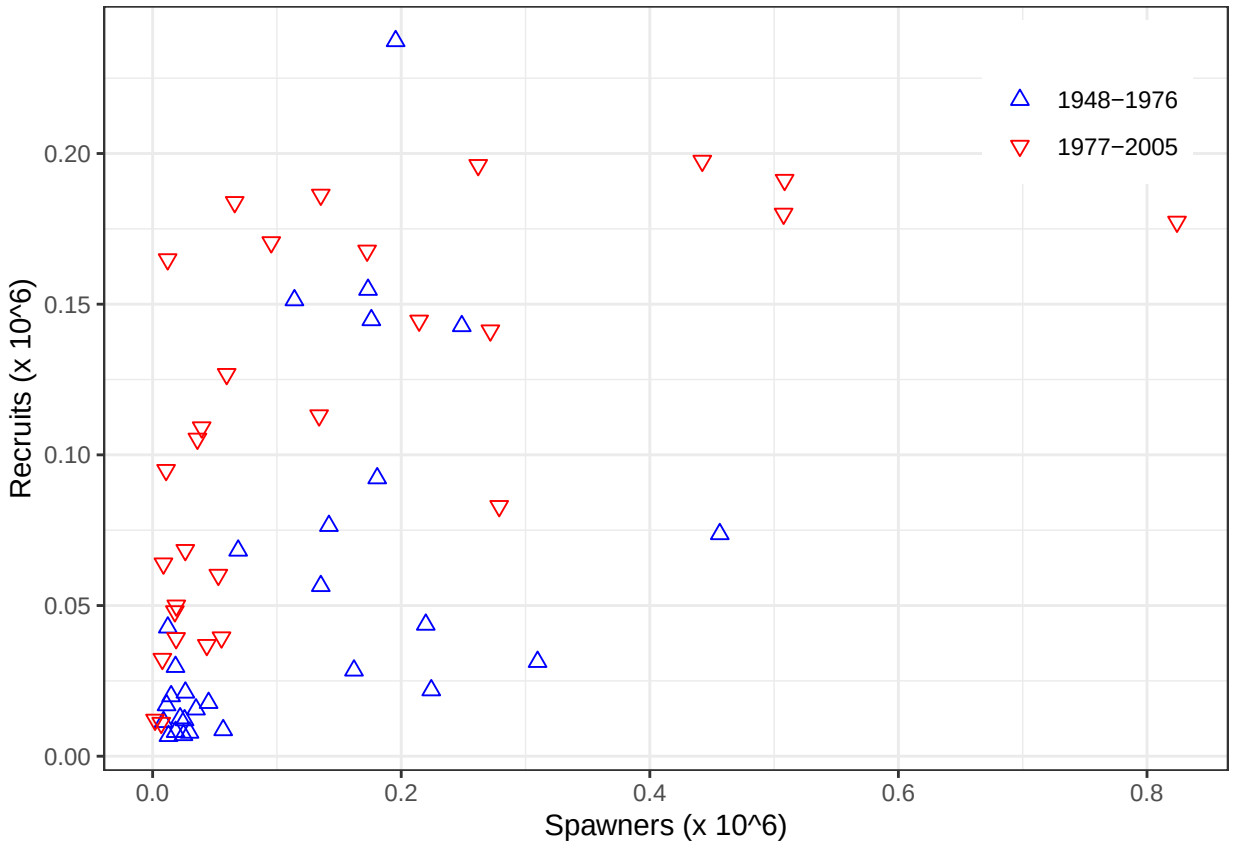
```
modeled_seymour_recruits <- data.frame(year=data_sockeye_seymour$year,
                                       observed=data_sockeye_seymour$recruits,
                                         predicted=NA,
                                         data_segment=data_sockeye_seymour$data_segment)
```

Now we assign the `predicted` values for each half, calling the corresponding custom function.

```
modeled_seymour_recruits$predicted <- c(
  f_Ricker_fit_second(data_sockeye_seymour$spawners[data_sockeye_seymour$data_segment=="1948-1976"]),
  f_Ricker_fit_first(data_sockeye_seymour$spawners[data_sockeye_seymour$data_segment=="1977-2005"])
)
```

And we can plot the observed versus predicted values to get a sense for the accuracy. In fact, we can use almost the exact same plot as above, but we replace the data frame and remap the `aes()` so that `x` is observed recruitment and `y` is predicted recruitment.

```
ggplot(modeled_seymour_recruits,aes(x=observed,y=predicted)) +
  geom_point(aes(color=data_segment,pch=data_segment),size=2) +
  scale_color_manual(values=c("blue","red"),breaks=c("1948-1976","1977-2005")) +
  scale_shape_manual(values=c(2,6),breaks=c("1948-1976","1977-2005")) +
  labs(x="Spawners (x 10^6)",y="Recruits (x 10^6)",color="",shape="") +
  theme_bw() + theme(legend.position = c(0.875, 0.875))
```


1948−1976
1977−2005
Recruits (x 10^6)
Spawners (x 10^6)
0.00
0.05
0.10
0.15
0.20
0.0
0.2
0.4
0.6
0.8