Performing Fractional Polynomials in R

Lena Kristin Bache-Mathiesen

Contents

Introduction

This document is intended as a guide for modeling the relationship between a measure of training load and injury in sports research, using fractional polynomials (FP). The examples will go through standard logistic regression, and later a mixed effects logistic regression model.However, the steps to model training load using FP in a Poisson model and other regression models are the same.

Preparation

First, load required packages.

```r
library(tidyverse) # for creating figures, and for a self-made function we'll need later on
library(rlang) # for creating functions with tidyverse syntax
library(mfp) # the mfp package has functions for automatic determination of FP
library(lme4) # package for mixed model functions
library(merTools) # a sister-package to lme4 for extra functions on mixed models
```

Next step is to load your data. At the minimum, your data should have

- one column for load, measured in any metric (such as sRPE, GPS measures or ACWR)
- one column for injury, must either be a logical variable (TRUE/FALSE) or coded (0 for no injury, 1 for injury) to work in our logistic regression model.
- one column for athlete ID, coupling the load values and injuries to the right person

```r
# here, we used the d_example_guide.rds data, simulated from football data, for the example to be repro
# replace the object "d" with your own data.
d = readRDS("d_example_guide.rds")
```

Standard Logistic Regression

For a regular logistic regression model, we can write:

```
fit_logistic = glm(injury ~ load, data = d)
```

Running `fit_logistic` or `summary(fit_logistic)` will provide us the results and information from the fit.

Fitting load with fractional polynomial terms, we can run (note that it may take a little time):

```
fit_fp = mfp(injury ~ fp(load), data = d, family = "binomial")
```

The function `fp()` is from the `mfp`package. The model searches for the best fit of a number of possible polynomial transformations. It uses a backwards selection process.

Running `summary()` of the fit will provide information of how many polynomials terms were added (of 1 or 2), and to what power they were chosen. Estimate is the beta-value and represents the logodds. The p-value indicates significance of each polynomial term.

By saving the object like this:

```
fit_summary = summary(fit_fp)
```

It's possible to access different data from the fit For example, the aic, or the coefficients, to use them later:

```
fit_summary$aic
```

```
## [1] 14777.15
```

```
fit_summary$coefficients
```

```
##                            Estimate    Std. Error   z value     Pr(>|z|)
## (Intercept)            3.3551343943 0.0437397223  76.70681 0.000000e+00
## I(((load + 1)/100)^1) -0.4935883599 0.0110508208 -44.66531 0.000000e+00
## I(((load + 1)/100)^3)  0.0009313095 0.0000840651  11.07843 1.596439e-28
```

Visualization

We can interpret our results numerically, but it may still be useful to create a figure of the predictions to see the relationship shape.

In a case where multiple variables have been included in the dataset, a new dataset needs to be created where the other variables have been set to a constant parameter, such as using the mean age.

We make a new object name for this dataset here, so we don't overwrite our original data object:

```
pred_data = d
pred_data$age = 17
```

We then add our data to the argument, `newdata` in the `predict()` command. The argument `type = "response"` means we will receive probability of injury instead of logodds of injury.

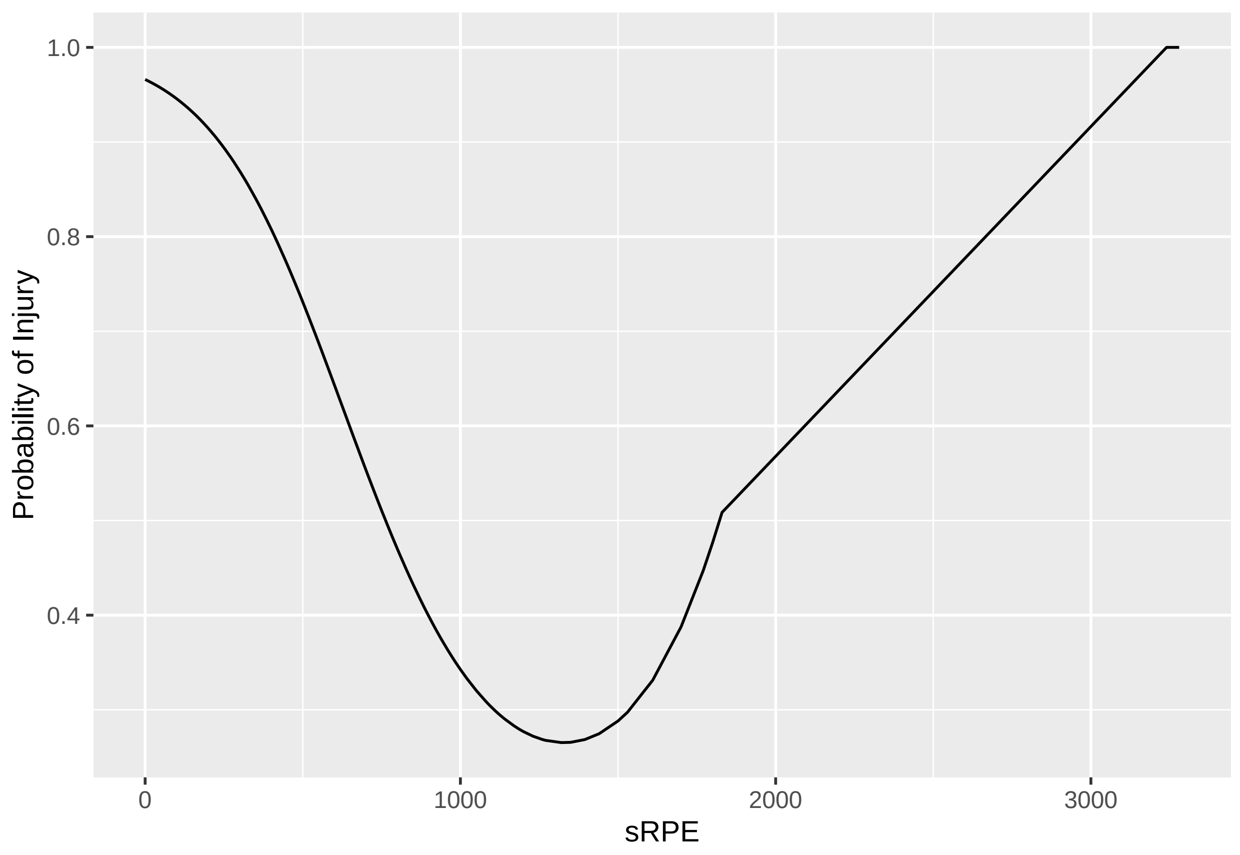
```
pred_load = predict(fit_fp, type = "response", newdata = pred_data)
```

To create a figure from our predictions, we add the predictions to our dataset with the load values used for predictions:

```
pred_data$yhat = pred_load
```

We loaded the `tidyverse` package, which includes the `ggplot2` package. A `ggplot2`-plot of the simplest, default form:

```
ggplot(pred_data, aes(x = load, y = yhat)) +
  geom_line() +
  xlab("sRPE") +
  ylab("Probability of Injury")
```



Mapping the load values to the predicted probabilities (figure above) showed that the polynomials modelled a U-shaped relationship for the example data.

Mixed Effects Regression Model

A mixed effects logistic regression model is a bit more complicated most notably, because running a general linear mixed model (GLMM) isn't part of base R. Here, we use the `lme4` package. Not everything in base R or other packages is compatible with the `lme4` package. Most critically, the `mfp()` package for running fractional polynomials are not compatible with the `lme4` package.

For a mixed model with binomial distribution and random intercept per athlete one can run:

```
fit_mixed = glmer(injury ~ load + (1 | p_id), family = "binomial", data = d)
```

For a mixed model with binomial distribution, random intercept and random slope per athlete

```
fit_mixed_slope = glmer(injury ~ load + (load | p_id), family = "binomial", data = d)
```

Since running the `mfp()`-function won't work inside `glmer()`, we need another solution. One solution would be manually model every possible incarnation of an FP2 model and manually determine the best fit.

Here we've made an automatized solution

Below is a function, glmer_fp() that searches for the best FP fit in a standard regression model with the mfp-function we used earlier. The FP-formula that was found to be the best one is then extracted from the standard model and run in a random effects model using glmer(). the default is a logistic regression (binomial distribution of outcome measure)

The function was created using `tidyverse`, `rlang` (which is why we loaded these at the top), the `lme4` and `mfp` package.

```
# the function has the following arguments:
# d                  The dataset with the load and injury variable
# injury             The variable used to denote injury.
#                    Can be heealth problems or any other definition of injury.
# load               The load variable in the dataset.
# rdm_effect         The random effect term.
#                    Must be surrounded by quotes "". Examples are "(1|your_id_variable)"
#                    for a random intercept and "(load|your_id_variable)"
#                    for a random slope + random intercept.
# family             Determine the model family. The defualt is binomial,
#                    meaning the models will run logistic regression.
#                    can be sett to "poisson" or other alternatives, see ?glmer()
# for a multivariable model, extra covariates will have to be added
# using + after fp(!!load), i.e. !!injury ~ fp(!!load) + sex + age,
# but the names must match those in the data
# and the function will no longer be general
glmer_fp = function(d, injury, load, rdm_effect, family = "binomial"){
  injury = enexpr(injury)
  load = enexpr(load)

  # run mfp to find best FP terms
  prox_fit = eval_bare(expr(mfp(!!injury ~ fp(!!load), data = d, family = family)))
  fp_form = prox_fit$formula

  # automatically use that formula in a random effects model
  formula_start = paste0("!!",fp_form[2], " ", fp_form[1])
  formula = paste0(formula_start, " ", fp_form[3], " + ",rdm_effect,"")
  glmm_fit = eval_bare(expr(glmer(as.formula(formula), family = family, data = d)))
  glmm_fit
}
```

An example of using the function with a random intercept and random slope:

```
fit_mixed = glmer_fp(d, injury, load, "(load|p_id)")
```

An example of using the function with a random intercept only:

```
fit_mixed_intercept = glmer_fp(d, injury, load, "(1|p_id)")
```

If both were able to converge, we can determine best fit with AIC:

```
AIC(fit_mixed)
```

```
## [1] 14744.94
```

```
AIC(fit_mixed_intercept)
```

```
## [1] 14742.65
```

As seen above, in the example data, the random intercept model had better fit

Visualization

Since we now have the random effect, we must set the example data to a fixed athlete as our example. we make a new object name for this dataset here, so we don't overwrite our original data object:

```
pred_data_mixed = d
pred_data_mixed$p_id = 1
```

For GLMM, we use the `predictInterval()` function that also predict the confidence intervals. This is why we loaded the `merTools` package. `predictInterval()` can't be used on anything but a `glmer()`-created object. Note that estimating the confidence intervals might take some time. Code below:

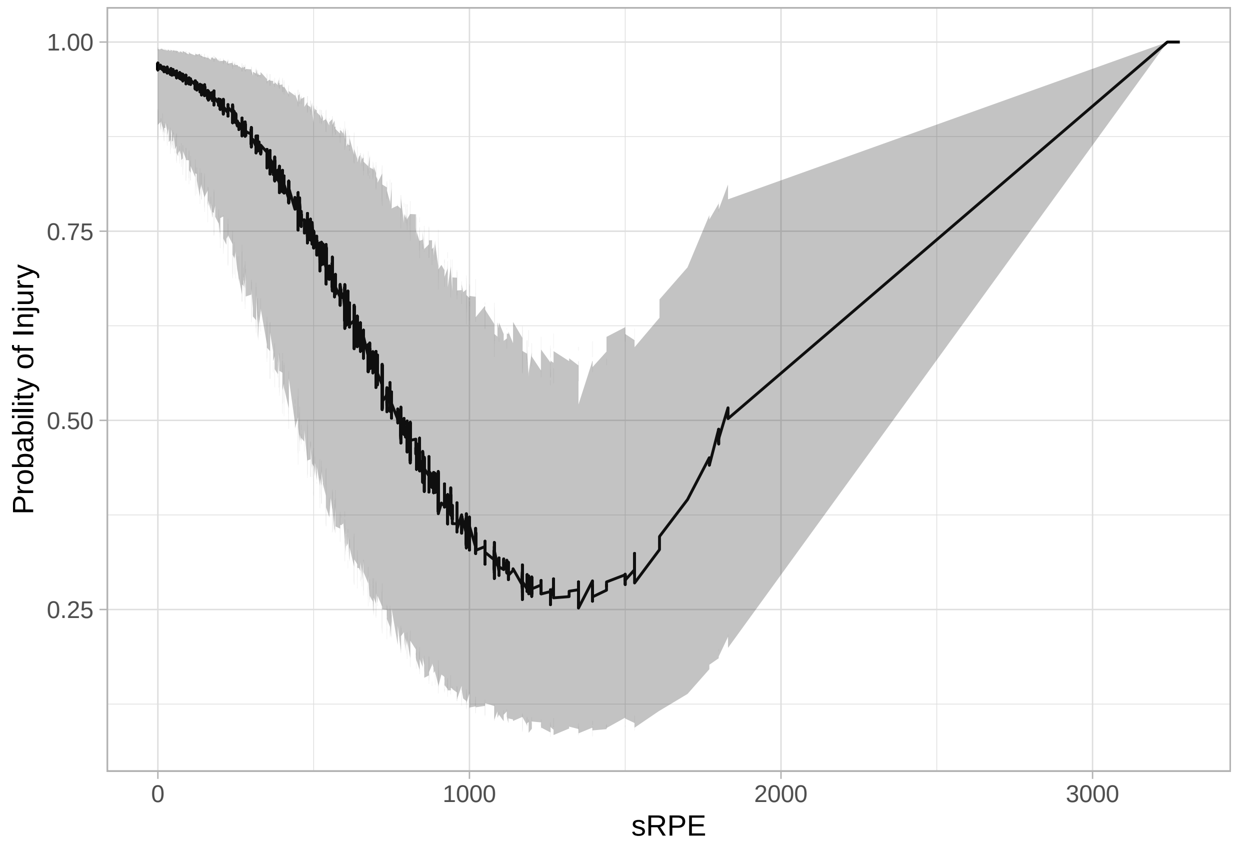
```
pred_load_mixed_ci = predictInterval(fit_mixed_intercept, ignore.fixed.terms = 1, type = "probability",
```

We add the load data we used for predictions to our predicted values:

```
pred_load_mixed_ci$load = pred_data_mixed$load
```

For a simple `ggplot2`-plot with confidence intervals:

```
ggplot(pred_load_mixed_ci, aes(x = load, y = fit, min = lwr, max = upr)) +
  geom_line() +
  geom_ribbon(alpha = 0.3) + # alpha is for transparency
  theme_light() + # a simple way to clean up
  xlab("sRPE") +
  ylab("Probability of Injury")
```



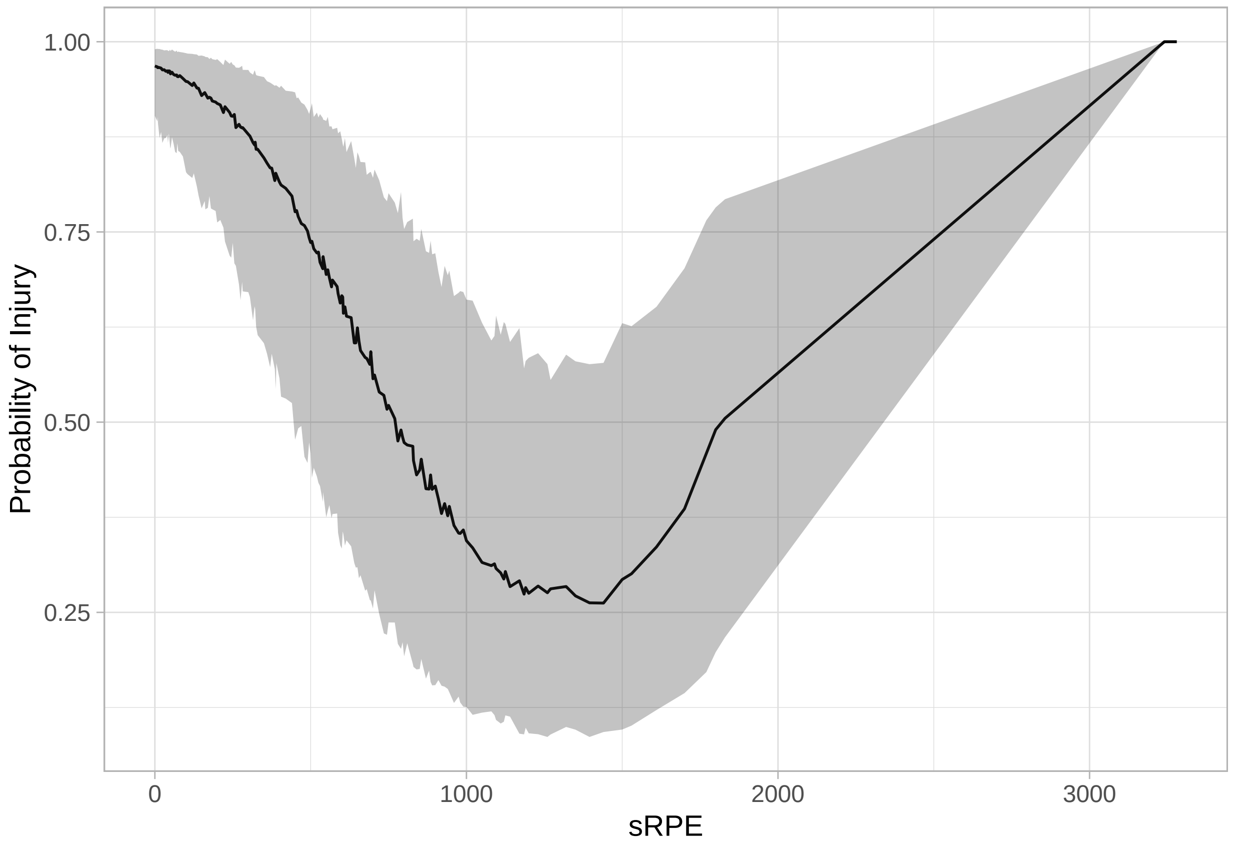
If you don't want multiple predictions per load value, you can predict based on a distinct set of values. Code below:

```
pred_data_distinct = as.data.frame(unique(d$load))
pred_data_distinct$p_id = 1
# the names need to be exactly the same as the dataset used to fit the model
names(pred_data_distinct)[1] = "load"
```

```
pred_load_mixed_distinct = predictInterval(fit_mixed_intercept, ignore.fixed.terms = 1, type = "probabil
pred_load_mixed_distinct$load = pred_data_distinct$load
```

Final plot.

```
ggplot(pred_load_mixed_distinct, aes(x = load, y = fit, min = lwr, max = upr)) +
  geom_line() +
  geom_ribbon(alpha = 0.3) +
  theme_light() +
  xlab("sRPE") +
  ylab("Probability of Injury")
```


Probability of Injury
sRPE