Wheat Yield exercise

Jordan Richards

2023-01-22

Dataset

- Exercise developed by Chris Wikle and Dan Pagendam (2019).
- Dataset created by Dan Pagendam and Josh Bowden at CSIRO.
- The data consists of simulated wheat yields from a farm in Dalby, Queensland using the model APSIMX.
- The dataset was created from 10,000 simulations of wheat growth under randomly generated meteorological conditions and management (planting and fertiliser application).

Predictors

- The predictors are:
  - summary statistics for the amount of rainfall in each of 52 weeks in the year.
  - degree days in each of the 52 weeks in the year.
  - cumulative evaporation in each of the 52 weeks of the year.
  - the "thermal time" of the wheat at each of the 52 weeks in the year.
  - the amount of Nitrogen fertiliser applied at planting.
  - the amount of Nitrogen fertiliser applied as top-up.
  - the day of year that the crop was planted.
  - the planting density of seed.

Response

- There are four response variables:
  - wheat grain yield.
  - grain size.
  - grain protein content.
  - wheat total weight.

Load dataset

```r
#For those using Colab run: install.packages("keras")
load("../Data/wheat.RData")
```

- The dataset contains both training and test data, with response denoted by Y and predictors denoted by $\mathbf{X}$
- The training and test sets were determined by randomly allocating approximately 10% of the simulations to the test/validation set and the remaining 90% to the training set.

```r
dim(trainData_X)
```

```
## [1] 8993  212
```

```r
dim(testData_X)
```

```
## [1] 1007  212
```

Data normalisation

- We first scale the input data to improve the numerical stability of training

```
rescaleCols <- function(rowX, colMins, colMaxs)
{
  r <- (rowX - colMins)/(colMaxs - colMins)
  r[is.nan(r)] <- 0
  return(r)
}
```

```
colMinsX <- apply(trainData_X, 2, min)
colMaxsX <- apply(trainData_X, 2, max)
```

```
trainData_X_scaled <- t(apply(trainData_X, 1, rescaleCols,
                              colMinsX, colMaxsX))
testData_X_scaled <- t(apply(testData_X, 1, rescaleCols,
                             colMinsX, colMaxsX))
```

Response

- We focus on modelling a single output response Y: Wheat total weight

```
trainData_Y = matrix(trainData_Y[, "wheatTotalWeight"],
                     ncol = 1)
testData_Y = matrix(testData_Y[, "wheatTotalWeight"],
                    ncol = 1)
```

- We will begin by 'predicting' Y given $\mathbf{X}$, i.e., we estimate the expectation $\mathbb{E}[Y|\mathbf{X}]$

Building a Keras prediction model

- We will build a feed-forward neural network with densely-connected layers and ReLU activations
- Begin by defining the input layer:

```
library(keras)
input.lay <- layer_input(shape = ncol(trainData_X_scaled),
      name = 'input_layer')
```

Hidden layers

- The model will have three hidden layers, each with 64 nodes

```
hidden.lay<- input.lay %>%
  layer_dense(units = 64,   activation = "relu") %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 64, activation = "relu")
```

- The output layer has a single unit; we estimate one value, i.e., the expectation, for each input vector

```
output.lay <- hidden.lay %>% layer_dense(units = 1)
```

Define a Keras model

- Model definition and summary

```
model <- keras_model(
  inputs = c(input.lay),
  outputs = c(output.lay)
)
```

```
summary(model)
```

Model compiltation

- We now compile the model with a loss function and an optimiser
- Here we use the MSE loss, which targets the mean of the response Y. If we wanted to target the median, we could use MAE instead
- We will use the RMSProp optimisation algorithm

```
model %>% compile(
  loss = "mse",
  optimizer = optimizer_rmsprop(learning_rate = 0.0001)
)
```

Fitting the model

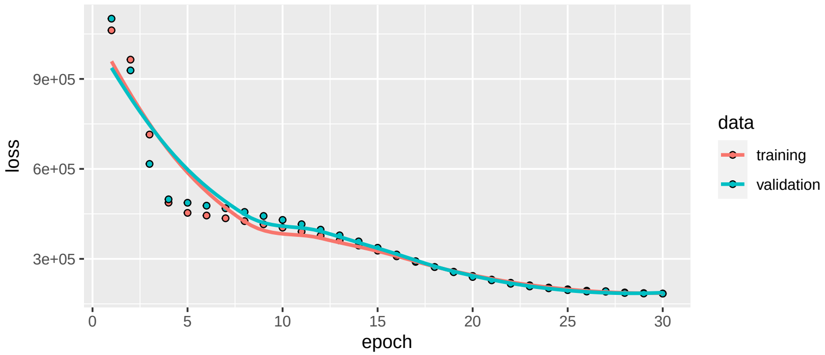
- We are now ready to fit the model.
- We define the training data, but also out-of-sample validation data that we can use to check how well the model generalises.
- When training the neural network, we use batches of data (32 samples here) to estimate the gradient of the loss function w.r.t the parameters.
- We also need to specify the number of epochs.
- Because we chose a small learning rate, we'll probably need to use more epochs to find the optimal fit.

```r
history <- model %>% fit(
  x = trainData_X_scaled, y = trainData_Y,
  epochs = 30, batch_size = 32,
  validation_data = list(testData_X_scaled, testData_Y)
)
```

Fitting the model

- Check for overfitting

```
plot(history)
```

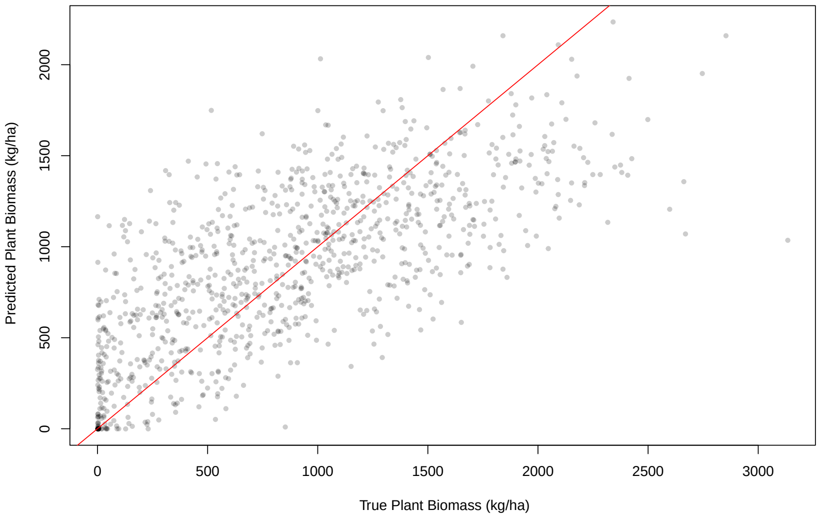


Predictions

- Check predictions

```r
pred_test <- predict(model, testData_X_scaled)
plot(testData_Y, pred_test,
     xlab = " True Plant Biomass (kg/ha)",
     ylab = " Predicted Plant Biomass (kg/ha)",
     pch = 20,
     col = adjustcolor("black", 0.2))
abline(0, 1, col = "red")
```

Predictions



Improvements

- We just trained a model using the build in “mse” loss function. This minimises mean squared error and so is analogous to linear regression where we assumed the error to be homoscedastic.
- Instead, let’s fit a log-normal regression model with varying scale parameter, i.e., a heteroscedastic model
- Do say by minimising the negative log-likelihood, i.e., MLE
- Not implemented as standard in Keras –> We’re going to create a custom loss function that is equal to the negative of the log-likelihood under the assumption that our predicted wheat yields can be modelled as having a log-normal distribution.

Custom loss functions

- See "keras_build.rmd" for help writing custom loss functions
- All custom loss functions have the same broad structure

```r
custom_loss=function(y_true,y_pred){
  K <- backend()
  #Code here
}
```

Custom loss functions

Important notes:

- when writing your code, you need to remember that y_true and y_pred are tensors.
- y_true and y_pred don't necesarrily have to have the same dimension.
- the first dimension of both the input tensors is equal to the batch size.
- there is very little documentation on how to write custom loss functions.
- try to use the Keras backend functions as much as possible and avoid using R functions (they may not work).
- if a Keras function has the "axis" argument, it is asking you which dimension you want to "apply"" the function to.

Custom loss functions

```r
negLL_logNormal <- function(y_true, y_pred)
{
  K <- backend()
  # Extract the first and second columns of predictions
  mu <- (y_pred[,1])
  sigma <- K$exp(y_pred[,2])

  #Extract first column of y_true to ensure same dimension
  y <- y_true[,1]

  # Use mu and sigma as parameters
  # describing log-normal distributions
  logLike <- -1*(K$log(y) + K$log(sigma)) -
    0.5*K$log(2*pi) - (K$log(y) - mu)^2/(2*(sigma)^2)
  return( -(K$sum(logLike)))
}
```

Building a custom model

- Based on the custom loss function we just created, our model requires two outputs:
  - one for mu (location parameter of the log-normal).
  - one for sigma (scale parameter of the log-normal).
- Let's use 3 hidden layers, with 64 nodes per layer and stick with the rectified linear unit (ReLU) activation function.
- Notice, that we now have two nodes / units in the output layer of the network.

```
input.lay <- layer_input(shape = ncol(trainData_X_scaled),
      name = 'input_layer')
hidden.lay2<- input.lay %>%
  layer_dense(units = 64,   activation = "relu") %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 64, activation = "relu")
output.lay2 <- hidden.lay2 %>% layer_dense(units = 2)
```

Compiling

- We will use the same optimiser as previously
- . . . but must specify the custom loss function

```
 model2 <- keras_model(
    inputs = c(input.lay),
    outputs = c(output.lay2)
  )
model2 %>% compile(
  loss = negLL_logNormal,
  optimizer = optimizer_rmsprop(learning_rate = 0.0001)
)
```

Fitting the model

- We're also going to introduce a callback into our training procedure.
- This will reduce the learning rate when we stop seeing a reduction in the validation loss.
- A smaller learning rate means smaller changes to the parameters, so we can think of this as fine-tuning our parameters with more and more epochs.
- Other callbacks include checkpoints and early-stopping

```
callback <- list(
  callback_reduce_lr_on_plateau(
    monitor = "val_loss",
              factor = 0.25, patience = 5
    )
)
```

Fitting the model

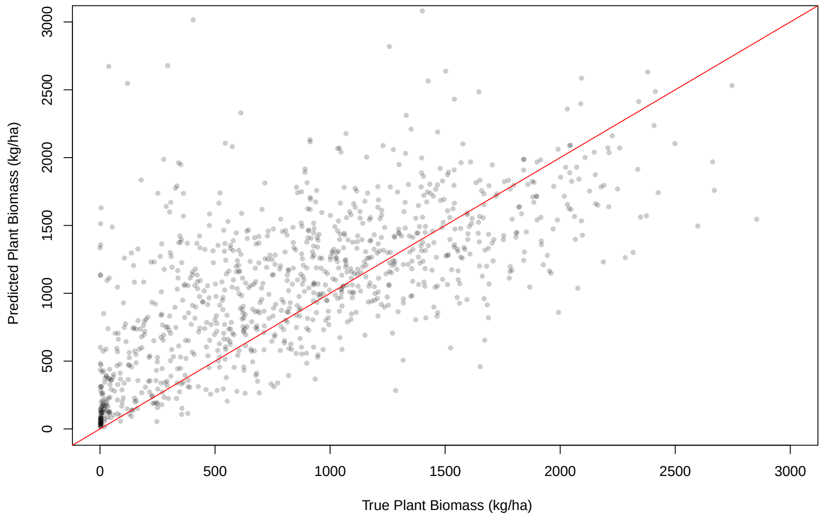
```
history <- model2 %>% fit(
  x = trainData_X_scaled, y = trainData_Y,
  epochs = 50, batch_size = 32,
  validation_data = list(testData_X_scaled, testData_Y),
  callbacks = callback
)
```

Predictions

- The model outputs two quantities for each input vector
- These are the mu and log(sigma) parameters for a log-normal predictive density.
- We can compare the mean of the log-normal to the true yield.

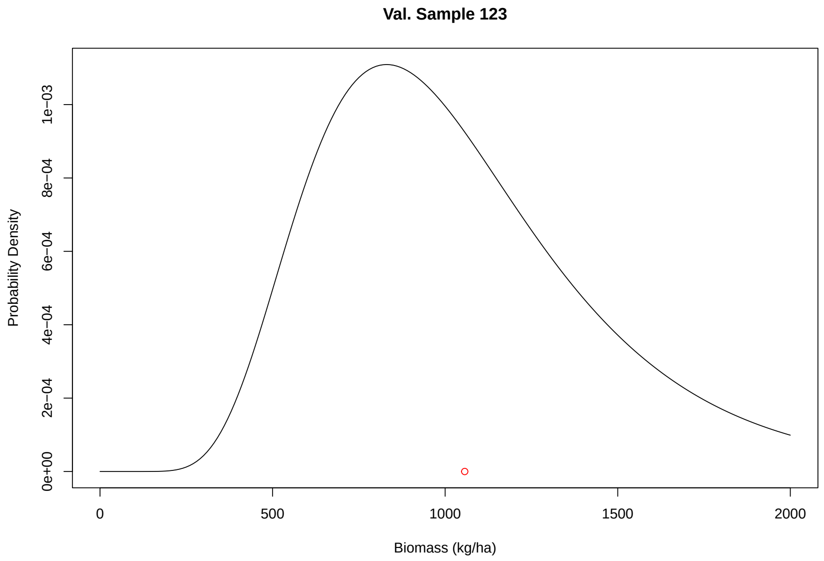
```
yhat <- predict(model2, testData_X_scaled)
mu <- yhat[, 1]
sigma <- exp(yhat[, 2])
pred_mean <- exp(mu + 0.5*sigma^2)
```

Predictions



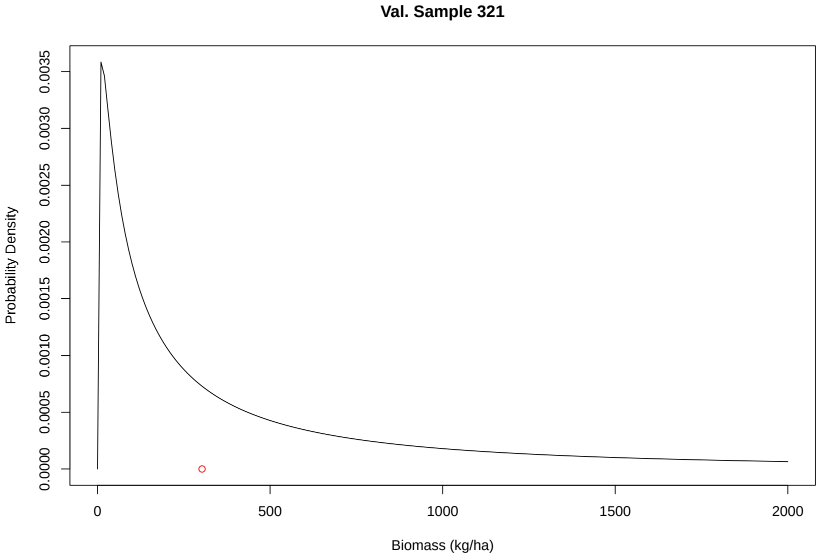
Predicted densities

- We can also look at the predicted density for specific samples



Predicted densities

- We can also look at the predicted density for specific samples



Extensions

- How do your results change if you include more nodes in the hidden layers?
- How do your results change if you make the network deeper?
- Try including some form of regularisation (e.g. Dropout or L1 regularisation) in the hidden layers to help prevent overfitting.