Code Review Example Function Headers

The code chunk below shows the header of my helper function wrapper.

```
model_handmade_folds <- function(data, horizon, dataset, lag, features,
basefeatures){

  #basefeatures <- 'Depth'

  # Make lags:
  features <- grep(features,names(data),value = T)
  basefeatures <- grep(basefeatures,names(data),value = T)
```

There are 2 standout issues I noticed immediately and that should be addressed in order to improve this function. As described in my checklist on code review, helper functions are amazing in R code. But they are only amazing if done correctly. If done correctly they improve code readability and transfer between projects. Lets focus on readability.

Readability

the first issue is no explanation of the functions inputs. Basic explanation should include an example input and short descriptions of type and shape. The second is explanation of the functions use case, explain this shorthand at the function wrapper. Thirdly it is important to explain the output of the function in the wrapper. Finally reevaluate the name of the function, does it clearly relate to the shorthand explanation of its purpose.

All these alterations also improve its transferability to other projects. There is a secondary issue in this functions first 5 lines. The parameters ‘features’ and ‘basefeatures’ used as input for the function are immediately reassigned. This is bad coding practice as there is no reason why the reassigned variables couldn’t have been used as original input for the function.

Revisions

The revised code is shown below. It now contains a precise explanation of all inputs, the functions use case and its output. The name of the function has been altered and the parameters more appropriately renamed. The reassignment of the 2 input parameters is removed and incorporated them into our function inputs/wrapper.

```
LLTO_Model <- function(data, name_of_dataset, lag, time_out_horizon,
features, predictors){
  ## Function purpose: Use Leave Location and Time Out Moddeling (LLTO)
on the
  ## DataSets of the Kaggle ACEA Smart Water Analytics Challenge.
Datasets are
  ## time series with multiple predictors and features. Lags are used
as features.
  ## A horizon gets specified to model the Time Out Cross validation.

  ## Function Output: Train Object

  ## Function input:
  # data                Dataset from Kaggle Acea Smart Water Analytics
```

```
challenge,
  #                     containing Date, numeric predictors & feature
set
  # name_of_dataset     Name of input dataset, used to save trained
model
  # lag                 Amount of lags used for features (integer
input)
  # time_out_horizon    Horizon of folds used in Time Out Cross
validation
  # features            Names of features used in dataset (numeric
variables)
  # predictors          Names of predictors used in dataset (numeric
variables)
```

Code Review Example Preprocessing Setup

The next part of the function runs through the different data preprocessing steps before the model training begins. From a user perspective I expect some control over this piece of code. As a user of a function you need these options to make it transferable. The different preprocessing steps need to be more precisely explained aswell. I also have some thoughts on code placement inside this segment.

```
  for(i in 1:length(features)){
    for(j in 1:lag){
      data$temp <- Lag(data[,features[i],+j])
      names(data)[which(names(data)=='temp')] <- paste(features[i],j,
sep = '_')
    }
  }


  data <- data[,which(colMeans(!is.na(data))>.2)]

  # Inlude seasonality:
  data$year <- as.numeric(substr(data$Date,7,10))
  data$year <- data$year - min(data$year) + 1
  data$month <- as.numeric(substr(data$Date,4,5))
  data$quarter <- ifelse(data$month <= 3,1,
                         ifelse(data$month >=4 & data$month <= 6,2,
                                ifelse(data$month >=7 & data$month <=
9,3,
                                       ifelse(data$month >9,4,NA))))

  data_long <- tidyr::pivot_longer(data, predictors,names_to =
'location', values_to = 'depth_to_groundwater')

  data_long <- data_long[complete.cases(data_long),]
  data_long <- data_long[which(data_long$depth_to_groundwater != 0),]

  #data_model <- data_long[,-grep('location|
Date|name',names(data_long))]
```

```
  temp <- data_long[,which(!names(data_long)%in%c('depth_to_
groundwater','Date','location'))]
  nzv <- nearZeroVar(temp)
# excluding variables with very low frequencies
  if(length(nzv)>0){temp <- temp[, -nzv]}
  i <- findCorrelation(cor(temp))
# excluding variables that are highly correlated with others
  if(length(i) > 0) temp <- temp[, -i]
  i <- findLinearCombos(temp)
# excluding variables that are a linear combination of others
  if(!is.null(i$remove)) temp <- temp[, -i$remove]
  data_model <- data_long[,c('depth_to_groundwater','Date','location',
names(temp))]

  data_model$Date <- as.Date(as.character(data_model$Date), format =
'%d/%m/%Y')
```

Preprocessing steps explained

The code above consists of 5 preprocessing distinct steps:

- Feature addition
- data shaping for modeling
- Removal of missing information
- Statistical preprocessing steps
- variable formatting

In the current code none of these steps are explained or identified as such. It is up to the reader of the code to figure this out. In the code below I explained these different distinct steps so they can be located easier. They can also be adjusted to a new project or dataset more easily. It is now easier to evaluate them individually and improve on the function.

Code placement

There is an argument here that the preprocessing steps need to be placed into their own function entirely. It is a distinct use case that can be seperated from training the model. Sometimes it even is a requirement in a production environment. This happens when we have to preprocess live production data as we did when training the model. The code is already greatly improved simply by reordening it into the 5 steps above.

Other considerations

One of the concerns I have reading this part of my code back is the lack of stepwise explanations. A clear example is the section on missing values, its as simple as it gets. Remove all rows that even contain a single missing value. As a new reader or user I want to know so many more things about this choice. This includes the impact of this choice, what if it removes all rows? The alternatives that were investigated and even why it removes the missing values.

Code review comes in to make projects stand up to production level quality. You could argue that this level of detail is prohibitive but I am here to argue it is a requirement. When this model is in operation any issue may arise, perhaps the missing data increases and the model fails specifically on this point. A developer may start a new project to implement imputation techniques. Maybe you already investigated this and it led to a dead end. One of the key characteristics of production level code is its ability to handle diverse outcomes and report on

them accordingly.

```
  ## Variable formatting:
  # Date variable:
  data_model$Date <- as.Date(as.character(data_model$Date), format =
'%d/%m/%Y')

  ## Feature addition:
  # Creating lags for i features in j lags
  for(i in 1:length(features)){
    for(j in 1:lag){
      data$temp <- quantmod::Lag(data[,features[i],+j])
      names(data)[which(names(data)=='temp')] <- paste(features[i],j,
sep = '_')
    }
  }
  # Inlude seasonality:
  data$year    <- as.numeric(substr(data$Date,7,10))
  data$year    <- data$year - min(data$year) + 1
  data$month   <- as.numeric(substr(data$Date,4,5))
  data$quarter <- ifelse(data$month <= 3,1,
                         ifelse(data$month >=4 & data$month <= 6,2,
                                ifelse(data$month >=7 & data$month <=
9,3,
                                       ifelse(data$month >9,4,NA))))
  # remove uncommon newly created features:
  data <- data[,which(colMeans(!is.na(data))>.2)]

  ## Data shaping for modelling
  # Long format by predictors for LLTO modelling:
  data_long <- tidyr::pivot_longer(data, predictors,names_to =
'location', values_to = 'depth_to_groundwater')

  ## Remove missing variables:
  # Complete cases
  data_long <- data_long[complete.cases(data_long),]

  ## Statistical preprocessing:
  # Remove outliers
  data_long <- data_long[which(data_long$depth_to_groundwater != 0),]
  # Preprocess features for modelling
  temp <- data_long[,which(!names(data_long)%in%c('depth_to_
groundwater','Date','location'))]
  # excluding variables with very low frequencies
  nzv <- nearZeroVar(temp)
  if(length(nzv)>0){temp <- temp[, -nzv]}
  # excluding variables that are highly correlated with others
  i <- findCorrelation(cor(temp))
  if(length(i) > 0) temp <- temp[, -i]
  # excluding variables that are a linear combination of others
  i <- findLinearCombos(temp)
  if(!is.null(i$remove)) temp <- temp[, -i$remove]
```

```
  # Selection of final feature set
  data_model <- data_long[,c('depth_to_groundwater','Date','location',
names(temp))]
```

Code Review Example Cross validation indices

One of the implementations of this function is the handmade folds for cross validation. I implemented this in the code because there were no libraries that used LLTO modelling exactly how I wanted to use it here. Looking back at my checklist, this is an important stage to bring up library optimization. Here we have a usecase where our code deliberately includes a selfprogrammed aspect, and it is our responsibility as the reviewer to verify that this was correct.

In this particular case I know that there is a CAST package in R that claims to implement LLTO modelling, but I found that it didn't create the folds for cross validation correctly. This is however not documented in the code below, which is a mistake. The code should reflect this piece of research (perhaps even so far as to show its wrong implementation). This also draws back to my previous point regarding the stepwise documentation. It is a smart move to keep initial implementations and checks of code so that later you can reuse your research to back up your final output. I have failed to do so and hence I am open to critism about why I self-programmed this set of functions.

The function of the code below is to create indices of cross validation folds for every predictor and time period defined in the dataset. Only the first time period is excluded as there is no prior data that can be used to train for it. For example a dataset with 4 predictors and 4 time periods gets 12 folds. Caret expects a names list item that includes the elements 'IndexIn' and IndexOut'.

```
# Handmade indexes:
  index_hand_design <- function(data,period, location, horizon,
location_one = NULL){
    horizon2 <- max(period)-horizon
    if(!is.null(location_one)){
      indexin <- which(data$Date >= min(period) & data$Date <=
horizon2)
      indexout <- which(data$Date > horizon2 & data$Date <=
max(period))

    } else {
      indexin <- which(data$Date >= min(period) & data$Date <= horizon2
& data$location != location)
      indexout <- which(data$Date > horizon2 & data$Date <= max(period)
& data$location == location)
    }
    output <-c(list(indexin),list(indexout))
    output
  }

  periods <- round(length(seq.Date(from = min(data_model$Date),to =
max(data_model$Date), by = 'day'))/horizon,0)
  dates   <- seq.Date(from = min(data_model$Date),to =
max(data_model$Date), by = 'day')
  indices <- 1:periods*horizon
```

```
  periods_final <- dates[indices]
  periods_final <- periods_final[!is.na(periods_final)]

  stopifnot(length(periods_final)>=4)

  for(i in 3:length(periods_final)){
    output <- list(c(periods_final[i-2], periods_final[i]))
    if(i <= 3){
      output_final <- output
    } else {
      output_final <- c(output_final, output)
    }
  }

  locations <- unique(data_model$location)

  for(i in 1:length(locations)){
    for(j in 1:length(output_final)){
      if(length(locations)==1){

        output_temp <- index_hand_design(data_model,output_final[[j]],
locations[i], horizon, location_one = 'yes')
      } else {
        output_temp <- index_hand_design(data_model,output_final[[j]],
locations[i], horizon)
      }
      if(j == 1){
        final_inner <- output_temp
      } else {
        final_inner <- c(final_inner, output_temp)
      }
    }
    if(i == 1){
      final <- final_inner
    } else {
      final <- c(final, final_inner)
    }
  }

  index_final <- list(index = final[seq(1, length.out =
length(locations)*length(output_final), by = 2)],
                      indexOut = final[seq(2, length.out
=length(locations)*length(output_final), by = 2)])
```

Code function

I find this part of the code hard to understand and read. My excuse is that it was written under time pressure after I found out it had to be done by hand. It is however a prime candidate for this reason to review here.

One of the problems is that original function, “index_hand_design”. I will refer back to section 1

about how to properly design helper functions. All problems discussed with the entire function apply to this inner helper function. There is also an argument to not define a function within a function.

To finish off these practical examples I would like to show how to change the nested for loop I used to run the index_hand_design function. Removing for loops is at the highest point of my code review checklist . Below is that code segment changed into an lapply and mapply format.

```
  if(length(locations)==1){
    final_inner <- lapply(1:length(output_final), function(x)
{index_hand_design(data_model,

output_final[[x]],

locations,

horizon,

location_one = 'yes')})
  }
  if(length(locations)>1){
    # Create dataframe with possible combinations:
    sequences <- data.frame(location = rep(locations,length(output_
final)),
                                              lengths =
rep(1:length(output_final), length(locations)))
    final <- mapply(index_hand_design, data = data_model,
sequences[,2], sequences[,1], horizon = horizon)
  }
```

The code above is endlessly more concise then the initial nested for loop. In my final pieces of code I always focus on replacing any remaining for loops into lapply or when using nested for loops mapply.

Wrapping up code review example of R Code in Caret

There were some other problems in the cross validation section, which will end up in the final code on github. I highlighted the removal of the nested for loop as a practicle example. The function was almost done anyway. It is very revealing to work through your own code from a code review perspective. I definitly recommend investing the time in optimizing your work in this way. Hopefully these practicle examples benefit your own work.