Data Science Methods for Finance: Exam Answers

Forecasting Electricity Prices

Aaron Kaijser

June 2021

Abstract

This document includes my answers and code as well as concise explanations of my code. The original exam questions, raw data and my R script can be found on my GitHub page. Code is ran on R version 4.0.5 on Windows 10.

Contents

Introduction

First of all, it should be noted that the main package that I use for data wrangling is **data.table** (Dowle, Srinivasan, Short, & Lianoglou, 2019), which uses an SQL-like syntax and is extremely fast compared to other packages such as **dplyr**, especially when you have to work with large datasets. The data.table syntax might seem complicated at first and might not be as easy to read as dplyr's syntax, but practice makes perfect.[1] Moreover, on StackOverflow you will often find answers to more complex questions that use data.table-based solutions, which is helpful when you run into problems.

[1] A good starting tutorial: https://www.datacamp.com/courses/data-manipulation-with-datatable-in-r

Question 1

The first questions are very easy but can be tricky nonetheless. All of the questions require you to work with and manipulate dates, which can get very confusing. One thing that you should know is that the output of some of these date-wrangling functions may depend on your operating system's language. An example: base R's function *weekdays*() will transform dates to weekdates, such that *2018-01-01* becomes *Monday* on an English operating system and *Maandag* on a Dutch operating system. When you subsequently apply filters that incorporate *"Monday"*, the code does not work when it is ran on a Dutch system. Hence, I set my locale to English at the beginning of my script to let R know that I would like to work with English outputs.

Note: operating systems other than Windows may require different solutions.

```
# Setting time equal to English
Sys.setlocale("LC_TIME","English") # Windows
Sys.setlocale("LC_TIME", "C") # Should work on MacOS
```

Next, I import the raw data using the *readRDS* function since the data is an .RDS file. I immediately transform the data into a data.frame and data.table class using the *setDT* function.

```
# Importing raw data
data <- setDT(readRDS("C:/2021_Electricity_data.RDS"))
```

Question 1.1

This question is pretty straightforward. First, I calculate the average hourly electricity price. Second, I calculate the average hourly electricity price over the weekends in the dataset.

```
# Calculating average hourly prices for all days
temp <- data
temp[, hour := hour(date)]
temp[, weekday := weekdays(date)]
temp[, dutch_avg := mean(dutch_power), by = hour]
dutch_avg <- unique(temp$dutch_avg)
```

- 2: Creates a temporary copy of the raw data.
- 3: Using the *hour* function, I extract the hour of each observation and store it in a new variable *hour*.
- 4: Using the *weekdays* function I turn a regular date into a weekday, e.g. "2018-01-01" becomes "Monday" and is stored in variable *weekday*.
- 5: I create a new variable *dutch_avg*, which is the average Dutch electricity price of each hour. I group by hour in order to do so.

- 6: Extracts all unique average Dutch hourly prices and stores it in a vector named *dutch_avg*. Note that this isn't a particularly robust method for extracting values as it will delete any duplicate values. However, as all 24 hours have a different average price, this doesn't matter. I now have the mean price for each of the 24 hours in my sample.

```
# Calculating average hourly prices in weekends
temp <- temp[weekday == "Saturday" | weekday == "Sunday"]
temp[, dutch_avg_wday := mean(dutch_power), by = hour]
dutch_avg_wday <- unique(temp$dutch_avg_wday)

# Making 1 dataframe
temp <- data.frame(hour = c(0:23), dutch_avg, dutch_avg_wday)
```

- 2: Only keeps observations of weekends.
- 3: Calculates the average hourly price during weekends.
- 4: Stores the 24 unique average hourly prices in a vector called *dutch_avg_wday*.
- 7: Creates a dataframe with 1 column representing the hours and the other two being the two vectors containing the average hourly prices.

```
# Plotting values
ggplot() +
  geom_line(data=temp, aes(x=hour, y=dutch_avg, color = "blue")) +
  geom_line(data=temp, aes(x=hour, y=dutch_avg_wday, color = "red")) +
  scale_color_manual("", labels = c("All days", "Weekends"), values = c("blue"
      , "red")) +
  labs(title="Average Hourly Prices",
       x ="Hour", y = "Price") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )
```

The code above creates a simle line graph using the *ggplot* library with a custom theme. It outputs the following graph:

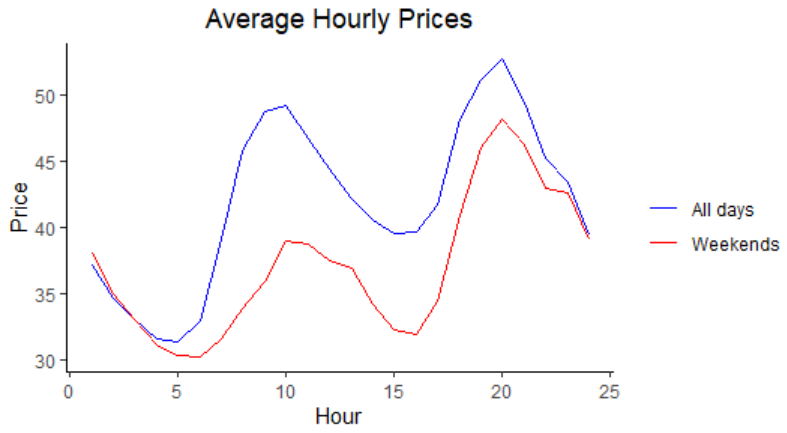

Figure 1: Average Dutch hourly electricity prices between 2018 and 2020 per MWh.

The graph shows that the average price of electricity in the Netherlands is substantially lower during the weekends. This is due to the fact that most companies are closed during the weekends and as a result there is less demand for electricity. Moreover, as there is less demand for electricity during the night, prices tend to be lower as well. During the day, the demand for electricity spikes between 09:00 and 10:00 a.m. as people get up and start working. The demand peaks between 19:00 and 20:00 as the sun has set.

Question 1.2

This question is rather easy given that we did something similar in the previous question.

```
# Calculating volatility per hour
temp <- data
temp[, hour := hour(date)]
temp[, hourly_vol := sd(dutch_power, na.rm = TRUE), by = hour]

# Plotting hourly volatility
ggplot(temp, aes(x=hour)) +
  geom_line(aes(y = hourly_vol), color = "blue") +
  labs(title="Hourly Volatility",
       x ="Hour", y = "Standard Deviation") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )
```

- 2-4: Creates a temporary copy of the data and calculates the standard deviation grouped by hour.

- 7-18: Plots the hourly volatility.

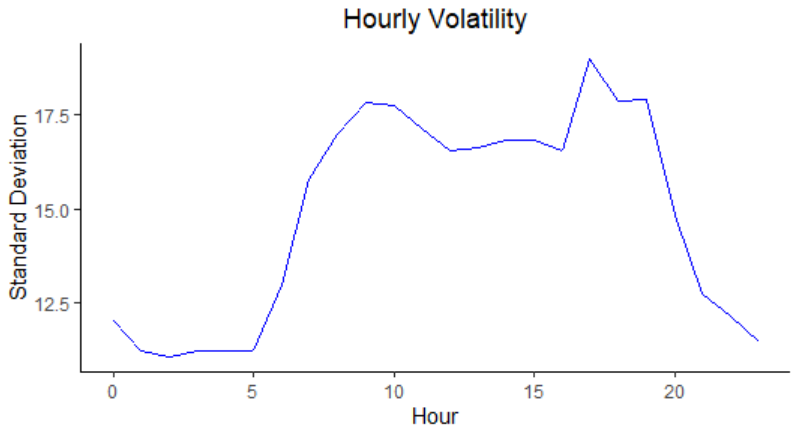

Figure 2: Hourly volatility of Dutch electricity prices.

Not surprisingly, the volatility is much higher during the day when the demand for electricity increases.

Question 1.3

What is asked is to compute the average daily electricity price for each day in the sample. Since we only have 3 years of data this implies that in order to calculate the average value day 1 of 365, we simply take the average of the electricity prices on the 1st of January in 2018, 2019 and 2020, respectively.

```
# Calculating daily mean
temp <- data
temp[, hour := hour(date)]
temp[, day := yday(date)]
temp[, daily_avg := mean(dutch_power, na.rm = TRUE), by = day]

# Plotting average daily prices
ggplot(temp, aes(x=day)) +
  geom_line(aes(y = daily_avg), color = "blue") +
  labs(title="Average Daily Prices",
       x ="Day", y = "Price") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )
```

- 4: Using the *yday* function I compute the day of the year for each observation, i.e. "2018-01-01" becomes "1".
- 5: I calculate the average electricity price for each day and ultimately obtain 366 values (366 because there is a leap year in the sample).
- 8-19: Plots the figure. The figure is somewhat choppy and you might want to consider using *geom_smooth* here (see question 1.5).

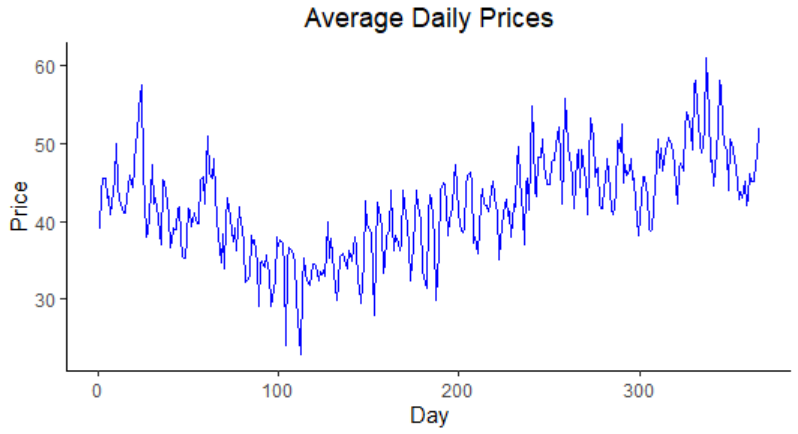

Figure 3: Average Dutch Electricity Price per Day (in MWh).

The electricity prices seem to be highest during the winter period, most likely caused by an increased demand for power as the weather gets colder and people tend to stay inside longer.

Question 1.4

This question simply requires us to apply basic filters to obtain different the different hourly solar generation numbers for two periods in a year.

```
# Calculating average hourly prices between April and September
month_vector <- c(4:9)
temp <- data
temp[, hour := hour(date)]
temp[, month := month(date)]
temp_apr_sep <- temp[month %in% month_vector]
temp_apr_sep[, avg_solar_aprsep := mean(solar, na.rm = TRUE), by = hour]

# Calculating average hourly prices between October and March
month_vector <- c(10:12, 1:3)
temp_oct_mar <- temp[month %in% month_vector]
temp_oct_mar[, avg_solar_octmar := mean(solar, na.rm = TRUE), by = hour]

```

```
# Plotting average hourly prices for Apr-Sep and Oct-Mar, respectively
ggplot() +
  geom_line(data=temp_apr_sep, aes(x=hour, y=avg_solar_aprsep, color='blue'))
      +
  geom_line(data=temp_oct_mar, aes(x=hour, y=avg_solar_octmar, color='red')) +
  scale_color_manual("", labels = c("Apr-Sep", "Oct-Mar"), values = c("blue",
      "red")) +
  labs(title="Average Hourly Solar Generation",
       x ="Hour", y = "Solar Generation") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )
```

- 5: Using the *month* function I compute the number of each month, i.e. "2018-01-01" is transformed into "1".
- 6-7: I filter out months 4 to 9 (April-September), which I defined in a vector called *monthly_vector* in (2) and calculates the mean hourly solar generation.
- 11-12: Filters out months 10 to 12 and 1 to 3 (October-March) and calculates the average solar generation.
- 15-28: Generates the plot.

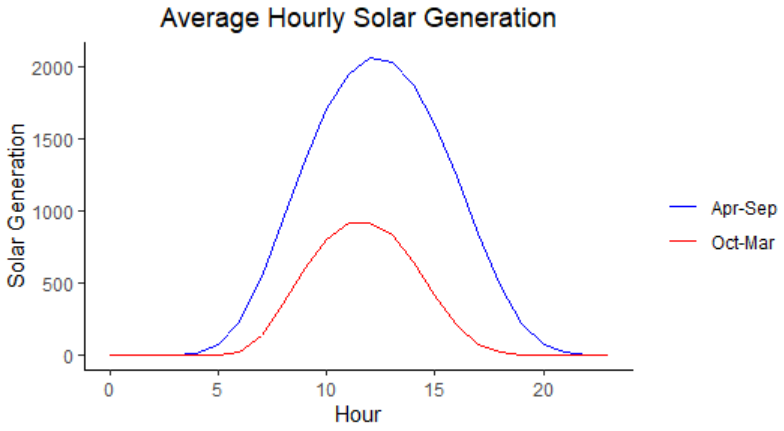

Figure 4: Average hourly solar generation forecast (in MWh)

The main takeaway from this graph is that solar generation is larger between April and September as the number of sun hours is higher in this period.

Question 1.5

This question is simply a combination of all the previous questions and is therefore pretty straightforward.

```
# Calculates daily average solar generation + off-shore wind generation
temp <- data
temp[, hour := hour(date)]
temp[, day := yday(date)]
temp[, ':=' (
  daily_avg_solar = mean(solar, na.rm = TRUE),
  daily_avg_offshore_wind = mean(wind_off_shore, na.rm = TRUE)
  ),
  by = day]

# Plotting average hourly prices for Apr-Sep and Oct-Mar, respectively
ggplot() +
  geom_smooth(data=temp, aes(x=day, y=daily_avg_solar, color='blue'), se =
      FALSE) +
  geom_smooth(data=temp, aes(x=day, y=daily_avg_offshore_wind, color='red'),
      se = FALSE) +
  scale_color_manual("", labels = c("Solar", "Off-shore Wind"), values = c("
      blue", "red")) +
  labs(title="Average Daily Solar and Off-shore Wind Generation",
       x ="Day", y = "Generation") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )
```

- 5-7: Calculates the daily average solar and off-shore wind generation at the same time.
- 12:25 Plots the corresponding figure. Note that I use the *geom_smooth* function here because the original graph was very choppy.

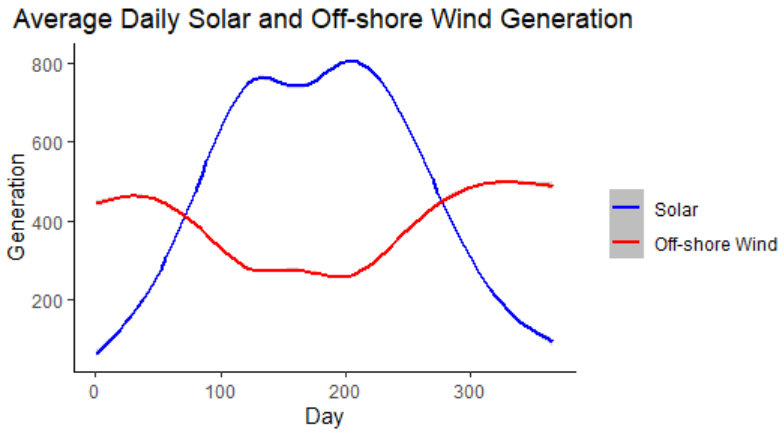

Figure 5: Average daily solar and off-shore wind generation forecasts (in MWh)

The main idea behind this plot is that it clearly shows that solar generation is much higher during the summer, whereas the off-shore wind generation is higher in the winter.

Question 2

Question 2 requires a lot more data wrangling and is more difficult than the first question. The general task is to transform the data from long to wide, such that we can use this matrix as input for our the machine learning methods that we will use in the next questions. The difficulty here is that it requires us to change the 9 columns that contain hourly information into hourly columns that contain the information for each observation. Transforming long tables to wide tables isn't something new and there are plenty of functions for that such as data.table's *dcast* and dplyr's *pivot_wider*. However, I have created a for loop that does the same.

Question 2.1

This question requires us to transform our dataset. For example, the first column called *Dutch_Power* which contains hourly variables has to be transformed into 24 separate columns, where *Dutch_Power*_1 contains the values of the first hour and *Dutch_Power*_2 of the second hour etc. In the end we want a matrix with (9 x 24) + 1 (date column) = 217 columns in total and 1096 rows (2018-01-01 to 2020-12-31).

```
1  # Importing raw data (again)
2  data <- setDT(readRDS("C:/2021_Electricity_data.RDS"))
3
4  # Computes hours which I will use in for loop
5  temp <- data
6  temp[, hour := hour(date)]
7
8  # For loop that makes a vector per hour for each of the 9 variables, which I
       ultimately turn into 1 dataframe
9  df <- setDT(data.frame(date = unique(as.Date(temp$date, format = "%Y-%m-%d %H
       :%M:%S"))))
10 empty_list <- list()
11 new_colnames <- list()
12 colnames <- names(temp)[2:(ncol(temp)-1)]
```

- 2: Starts with a fresh import of the raw data.
- 5-6: Creates a temporary dataframe with 1 column containing the hour of each observation.
- 9: Creates a dataframe which we will fill with our values in the next for loop. Note that I transform the date variable "2018-01-01 01:00:00" into "2018-01-01" by using the *as_Date* function. I then use the *unique* function to keep unique dates (i.e., "2018-01-01" occurs 9 times and since we are transforming the data from long to wide I only need this date once).
- 10: An empty list in which I will store all the vectors of each hourly variable, i.e. *Dutch_Power*_1.

- 11: An empty list in which I will store all the different 216 variable names.
- 12: A list of all the 9 column names to iterate over.

```
1 for (i in 1:length(colnames)) {
2   for (j in unique(temp$hour)) {
3     empty_list[[j+1]] <- as.vector(temp[hour == j, list(get(colnames[i]))])
4     df <- cbind(df, empty_list[[j+1]])
5   }
6   new_colnames[[i]] <- setDT(data.frame(paste0(rep(colnames[i], each=1), "_",
        1:24)))
7 }
8 new_colnames <- rbindlist(new_colnames)
9 names(df)[-1] <- new_colnames[[1]] # [[1]] changes single column data.table to
      vector (i.e. replaces V1 by dutch_power_1 etc.)
```

- 1: Loops over the 9 different features.
- 2: Within the loop, I also loop over the unique hours (24).
- 3: I fill every j+1 element of the list with a vector of each hour per feature. I use j+1 since the loop will start at 0, and element 0 does not exist. If you look closely, you'll see that I first filter the hour (hour == j) and then select the variable that I want to subset (list(get(colnames[i]))).
- 4: The vector that I just created that contains the hourly values for 1 particular feature (i.e. *Dutch_Power_1* is then combined with the dataframe we created earlier in the dataframe *df*.
- 6: Stores all 216 unique column names in a list. *setDT* makes sure that each column is in data.table format such that we can use data.table's *rbindlist* function.
- 8: Rowbinds all unique column names and creates 1 vector.
- 9. Replaces the column names of df (which at that point are simply V1 up to V217) by the vector of unique column names.

Question 2.2

This question simply requires us to cut the entire dataframe we just calculated into 2 different dataframes. One should contain *Dutch_Power_1* up to *Dutch_Power_24* (the labels/dependent variables) and the other one should contain the rest of the features (lagged by 1 day). Why should they be lagged? Because these values are not known to us. The dependent variables are the day-ahead prices, which means that we only know the features of the previous day.

```r
# Creating lagged feature dataframe
features <- df[,26:ncol(df)] # takes
features <- lapply(features, function(x) shift(x, n=1L))
features <- setDT(as.data.frame(do.call(cbind, features)))
features <- features[-1,]

# Creating corresponding labels dataframe
labels <- df[-1,1:25] # drops first row since we don't have any features for
    that date (because we lagged the features by 1 day)
```

- 2: Cuts off the date column and the 24 label columns (the first 25 columns) and extracts all the remaining columns (columns 26 until the last column).
- 3-4: Lags the entire features dataframe
- 5: Removes the first row since it is now NA after lagging the features.
- 8: Picks the first 25 columns (including the date column) to create the labels dataframe. Also drops the first row since we don't have any features here.

One thing that I would like to note here is that this particular question isn't very thought through. In the subsequent questions I will use the "labels" as features as well, hence I don't think it makes sense to make 2 different dataframes.

Question 2.3

I choose to drop any features of which more than 20% of values are equal to 0. I first calculate plot all the variables that exceed this threshold.

```r
# Counting 0 values per feature
zero_df <- vector()
for (i in 1:ncol(features)) {
  zero_df[i] <- nrow(features[get(names(features)[i]) == 0.00])
}
zero_df <- setDT(data.frame(feature = names(features), zero_values = (zero_df
    / nrow(features) * 100))) # computes % of values equal to 0 per variable
zero_df <- zero_df[zero_values > 20] # highlights the variables with more than
     20% missing values

# Plotting values that are equal to 0
ggplot(zero_df, aes(x=feature, y=zero_values)) +
  geom_bar(stat = "identity", fill = "red") +
  labs(title="Features with Values Equal to 0",
       x ="Feature", y = "Zero Values (in %)") +
  theme_bw() +
  theme(
    plot.title = element_text(hjust = 0.5),
    axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1),
```

```
    plot.background = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    axis.line = element_line(color = 'black')
  )

# Removing variables shown in graph from feature set
zero_df <- zero_df$feature
features <- features[ , !(colnames(features) %in% zero_df), with=FALSE]
```

- 2: Creates an empty vector in which I will store the amount of 0-values for each variable.
- 3-4: Loops over all the columns of the features dataset. It makes a subset for each variable - so essentially 1 column - for all the values that are equal to 0 and then stores the total number of rows (which is the number of values equal to 0) in the vector.
- 6: Generates 1 dataframe with the percentage of 0-values.
- 7: Only selects the features with more than 20% zero values.
- 10-22: Plots the figure below.
- 26: Creates vector with names of the columns we want to remove from the features.
- 27: Removes the values from the feature set based on the vector I created in (26).

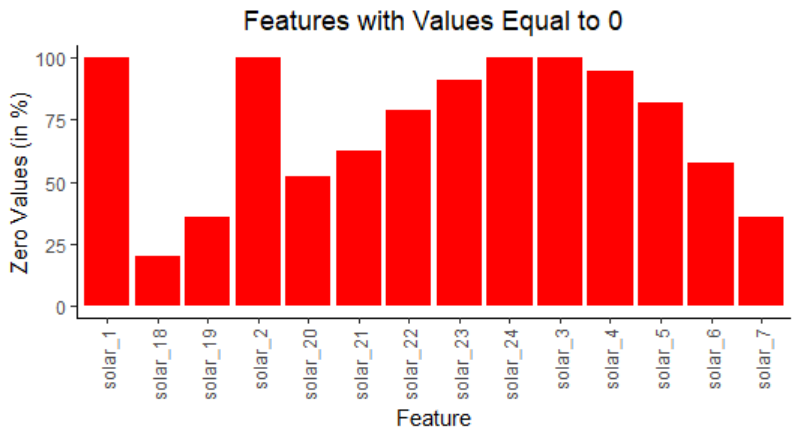

Figure 6: Features with more than 20% values equal to 0.

What you can observe is that all of these features are solar generation forecasts in the evening. They are almost all equal to 0 due to the fact that solar generation barely takes place at these times.

Question 3

For this question it is important to note that question 3.1 simply requires you to make a simple benchmark model, that is: run a linear regression on the y-variable *Dutch_Power_1* (The day-ahead Dutch electricity price between 00:00 and 01:00) with the lagged feature *Dutch_Power_24* (the previous day's day-ahead prices between 23:00 and 00:00). For question 3.2 I choose to include the lagged 'labels' *Dutch_Power_2* up to *Dutch_Power_24* as well as all the lagged features we created earlier.

Question 3.1

```
# I first create a new dataframe for this question (note that we will use 1 of
    the labels as an input for this question, which have not been lagged)
temp <- labels[, list(date, dutch_power_1, dutch_power_24)]
temp[, dutch_power_24 := shift(dutch_power_24, n=1L)]
temp <- temp[-1,]
train <- temp[date < "2020-07-01"][, date := NULL]
test <- temp[date >= "2020-07-01"][, date := NULL]
```

- 2: Creates a temporary dataframe *temp* which contains 3 columns: the dates and the day-ahead Dutch electricity prices between 00:00 and 01:00 and 23:00 and 00:00.
- 3: Lags the day-ahead prices between 23:00 and 00:00 by 1 day.
- 4: Drops the first row since we don't have any values for *Dutch_Power_24*.
- 5-6: Makes a training set that ranges from 2018-01-02 to 2020-06-30 as well as a test set that ranges from 2020-07-01 to 2020-12-31. Drops the date column since I don't want to include that in my regression.

```
# Benchmark model with OLS (I)
model <- lm(dutch_power_1 ~ dutch_power_24, data = train)
print(paste0("In-sample R-squared is: ", round(summary(model)$r.squared, 2)))

# Out-of-sample prediction with OLS model
predictions <- predict(model, newdata = test[,2], type = "response") # takes
    dutch_power_24 as input and predicts dutch_power_1

# Calculating RMSE
print(paste0("Out-of-sample RMSE is: ", round(sqrt(mean((test$dutch_power_1 -
    predictions)^2)), 2)))
```

- 2: Runs a linear regression.
- 3: Prints the R^2 of the regression.

- 6: Makes predictions using the model we just fitted and stores it in variable *predictions*.
- 9: Calculates the RMSE. It first calculates the errors between the actual values and the predictions, squares them, takes the mean and then the square root of that figure.

You should find an in-sample R^2 of 0.79 and an out-of-sample RMSE of 4.81.

Question 3.2

There are multiple things that require attention here. First, it should be noted that the dependent variable will be the day-ahead price of Dutch electricity between 00:00 and 01:00. Second, I create 1 large dataframe that includes both the "labels" and "features" that I made in question 2 because I will also use the so-called labels as features here. This is why I believe that question 2 was rather poor and shouldn't have made a distinction between the labels and the prices.

Creating the training, validation and test sets

```
# Function to calculate R-squared
rsq <- function(preds, actual) {
  rss <- sum((preds - actual) ^ 2) ## residual sum of squares
  tss <- sum((actual - mean(actual)) ^ 2) ## total sum of squares
  rsq <- 1 - rss/tss
}

# Creating 1 large features dataframe with the right lags
lagged_features <- df[,1:25] # grabs dutch_power_1 to dutch_power_24 which I
    will add to features we created before (and remove dutch_power_1)
lagged_features <- lagged_features[, -c("dutch_power_1")] # removes dutch_
    power_1 as it won't be a feature
lagged_features <- lagged_features[, (colnames(lagged_features)[-1]) := lapply
    (.SD, function(x) shift(x, n=1L)), .SDcols = colnames(lagged_features)
    [-1]][-1,] # lags all vars (except date) by 1 day and drops first row
features <- cbind(lagged_features, features) # binds features list and dutch_
    power_2 to dutch_power_24
features[, (colnames(features)[-1]) := lapply(.SD, function(x) scale(x)), .
    SDcols = colnames(features)[-1]] # scales all features (except date)
rm(lagged_features) # clean up

# Creating training, validation and test set with features
train_x <- features[date < "2020-01-01"][,-1]
val_x <- features[date >= "2020-01-01" & date < "2020-07-01"][,-1]
test_x <- features[date >= "2020-07-01"][,-1]

# Creating training, validation and test set for labels
train_y <- labels[date < "2020-01-01", list(dutch_power_1)]
val_y <- labels[date >= "2020-01-01" & date < "2020-07-01", list(dutch_power_
    1)]
test_y <- labels[date >= "2020-07-01", list(dutch_power_1)]
```

- 2-5: Creates a custom function that allows me to calculate the in-sample R^2 of lasso, PLS and random forest regressions.
- 9: Makes a dataframe with a date column and the 24 Dutch Power variables.
- 10: Removes *Dutch_Power_1* as it won't be a feature, but the dependent variable instead.
- 11: Lags the entire dataframe by 1 day except the date column.
- 12: Combines the lagged features dataframe I just created with the earlier features dataset (see question 2.2).
- 13: Scales all the features such that the mean is 0 and the standard deviation is 1.
- 17-24: Creates the training (2 years), validation (0.5 year) and test (0.5 year) sets based on the dates.

LASSO

```
# Lasso regression
# Grid + empty vector to store RMSE in
lasso_grid <- 10^seq(10, -2, length = 100)
rmse_lasso <- vector()

# Training model
for (i in 1:length(lasso_grid)) {
  lasso_model <- glmnet(train_x, train_y$dutch_power_1, alpha = 1, lambda =
      lasso_grid[i])
  predictions <- predict(lasso_model, newx = as.matrix(val_x), s = lasso_grid[
      i])
  rmse_lasso[i] <- sqrt(mean((val_y$dutch_power_1 - predictions)^2))
  message(paste0("Iteration ", i, " of ", length(lasso_grid), " complete."))
  gc() # clears memory
}

optimal_lambda_lasso <- lasso_grid[which.min(rmse_lasso)] # optimal lambda
    based on validation set

# Out-of-sample predictions
lasso_model <- glmnet(train_x, train_y$dutch_power_1, alpha = 1, lambda =
    optimal_lambda_lasso)
predictions <- predict(lasso_model, newx = as.matrix(test_x), type = "response
    ", s = optimal_lambda_lasso)
rsq_lasso <- rsq(predict(lasso_model, newx = as.matrix(train_x), type = "
    response", s = optimal_lambda_lasso), train_y$dutch_power_1) # in-sample R
    -squared
rmse_lasso <- sqrt(mean((test_y$dutch_power_1 - predictions)^2)) # out-of-
    sample RMSE
```

- 3-4: Creates a "grid", which in this case is a simple vector containing lambda values of 10^{10} to 10^{-2}. I also initiate an empty vector that I will use to store the out-of-sample RMSE values of the lasso regression.
- 7-12: A for loop which first fits the model using the values from the grid (note that $\alpha = 1$, such that it is a lasso and not a ridge regression), subsequently makes predictions using the validation set's features and finally calculates the RMSE based on the validation set's actual values of the dependent variable. I also print out the iterations to track progress. *gc* is a handy function to free up space after each iteration.
- 15: Saves the lambda value that minimizes the pseudo out-of-sample RMSE of the validation set.
- 20: Calculates the in-sample R^2 (i.e. I make predictions using the model that was trained using the training set and make predictions using the training set as well) and stores it in a vector *rsq_lasso*.
- 21: Calculates the truly out-of-sample RMSE and stores it in *rmse_lasso*.

I find an in-sample R^2 of 0.75, an out-of-sample RMSE of 5.06. The optimal λ is 0.22. The model performs worse than the benchmark we defined in the previous question as the RMSE of this model is higher.

Partial Least Squares

```
1  # creating grid for ncomp, for testing 20 ncomps on the validation set
2  ncomp_grid <- c(1:20)
3  rmse_pls <- vector()
4
5  # Training and validating model
6  for (i in 1:length(ncomp_grid)){
7    pls_model <- caret::train(
8      dutch_power_1 ~ . ,
9      data = cbind(train_y, train_x),
10     method = "pls",
11     tuneGrid = expand.grid(ncomp = ncomp_grid[i]),
12     trControl = trainControl(method = "none"))
13
14   predictions <- predict(pls_model, ncomp = ncomp_grid[i], newdata = val_x)
15   rmse_pls[i] <- sqrt(mean((val_y$dutch_power_1 - predictions)^2))
16
17   message(paste0("Iteration ", i, " of ", length(ncomp_grid), " complete."))
18   gc()
19 }
20
21 optimal_ncomp_pls <- ncomp_grid[which.min(rmse_pls)] # optimal number of
       ncomps
22
```

```r
# Out-of-sample predictions
pls_model <- caret::train(dutch_power_1 ~ . ,
                    data = cbind(train_y, train_x),
                    method = "pls",
                    tuneGrid = expand.grid(ncomp = optimal_ncomp_pls),
                    trControl = trainControl(method = "none"))
predictions <- predict(pls_model, ncomp = optimal_ncomp_pls, newdata = test_x)
rsq_pls <- rsq(predict(pls_model, ncomp = optimal_ncomp_pls, newdata = train_x
    ), train_y$dutch_power_1) # in-sample r-squared
rmse_pls <- sqrt(mean((test_y$dutch_power_1 - predictions)^2)) # out-of-sample
     RMSE
```

- 2: Generates a vector with the number of components ranging from 1-20 which I will iterate over.
- 6-18: For loop that implements the **caret** package's *train* function to train the models. Note that line 9 combines the training features and labels dataframes into 1 dataframe in order to work.

I find an in-sample R^2 of 0.77, an out-of-sample RMSE of 5.15. The optimal number of components is 8.

Random Forest

```r
# Creating grid + vector to store RMSE of models when predicting on validation
    set
random_forest_grid <- expand.grid(
  mtry = c(ncol(train_x) / 3),
  num.trees = c(1000),
  max.depth = seq(1, 20)
)
rmse_rf <- vector()

# Training and validating model
for (i in 1:nrow(random_forest_grid)) {
  rf_model <- ranger(dutch_power_1 ~ . ,
                data = cbind(train_y, train_x),
                mtry = random_forest_grid$mtry[i],
                num.trees = random_forest_grid$num.trees[i],
                max.depth = random_forest_grid$max.depth[i],
                verbose = TRUE)

  predictions <- predict(rf_model, data = val_x)
  rmse_rf[i] <- sqrt(mean((val_y$dutch_power_1 - as.vector(predictions$
     predictions))^2))
  message(paste0("Iteration ", i, " of ", nrow(random_forest_grid)))
}

optimal_mtry_rf <- random_forest_grid$mtry[which.min(rmse_rf)]
```

```
optimal_ntrees_rf <- random_forest_grid$num.trees[which.min(rmse_rf)]
optimal_depth_rf <- random_forest_grid$max.depth[which.min(rmse_rf)]

# Out of sample predictions
rf_model <- ranger(dutch_power_1 ~ . ,
                   data = cbind(train_y, train_x),
                   mtry = optimal_mtry_rf,
                   num.trees = optimal_ntrees_rf,
                   max.depth = optimal_depth_rf,
                   verbose = TRUE)

predictions <- predict(rf_model, data = test_x)
rsq_rf <- rsq(predict(rf_model, data = train_x)$predictions, train_y$dutch_
    power_1) # in-sample r-squared
rmse_rf <- sqrt(mean((test_y$dutch_power_1 - as.vector(predictions$predictions
    ))^2))
```

- 2-5: Creates a grid to iterate over. For *mtry* I use the rule of thumb of dividing the number of features by 3 (Boehmke & Greenwell, 2019).[2]

I find an in-sample R^2 of 0.94 and an out-of-sample RMSE of 5.89. The optimal *mtry* and *depth* are 67 and 9 respectively. The random forest model severely overfits to the training data and therefore most likely performs the worst out-of-sample.

[2]See: https://bradleyboehmke.github.io/HOML/random-forest.html

Question 4

This question is similar to question 3 with the difference being that you have to calculate the in-sample R^2 and out-of-sample RMSE for all 24 hours.

Question 4.1

In order to write the for loop for this question you should be aware that the dependent variable should not be scaled whatsoever. In addition, this question can be interpreted in two ways. You are asked to forecast the electricity price using the electricity prices of the last hour of the previous day. For example, in order to predict the electricity price between 14:00 and 15:00 you should use the previous day's electricy price between 13:00 and 14:00 and not the actual last hour of the previous day (i.e. 23:00 and 24:00). That is what I do here. One pitfall here is that it is somewhat difficult to code that you want to use the price of *Dutch_Power_24* in order to forecast the price of *Dutch_Power_1* unless you hardcode two different vectors with parallel prices. I chose to implement an if-else statement that simply checks whether the dependent variable is *Dutch_Power_1* and then simply picks *Dutch_Power_24* rather than n-1.

Note: Since validation is not possible for OLS, the training set ranges from 2018-01-01 to 2020-06-30 (rather than 2018-01-01 to 2019-12-31) since the validation set is not needed here.

```
# Makes training set (note that because I am performing a simple linear
    regression
# I cannot tune hyperparameters and hence I include the validation period
    (2020-01-01
# to 2020-06-30) in the training set )
labels <- df[,1:25]

# Hardcoding column names
colnames <- colnames(labels)[-1]
predictions_ols <- list()
r2_ols <- vector()
rmse_ols <- vector()

for (i in 1:length(colnames)) {
  # Creating label and features set
  train_y <- as.data.frame(labels[[i+1]]) # grabs response variable and
      removes first element
  train_y <- setDT(cbind(date = labels$date, train_y))
  train_y <- train_y[date < "2020-07-01"][-1,-1]
  names(train_y) <- colnames[i]
  # Checks whether y is dutch_power_1 so we pick the price of dutch_power_24
      as x rather than n-1 (i.e. dutch_power_2 uses dutch_power_1)
  if (colnames(train_y) == "dutch_power_1") {
    cols <- tail(colnames, 1)
    features <- labels[, ..cols]
    features <- setDT(as.data.frame(scale(features)))
```

```
23    features <- features[, (colnames(features)) := lapply(.SD, function(x)
          shift(x, n=1L)), .SDcols = colnames(features)][-1]
24    train_x <- setDT(cbind(date = labels$date[-1], features))
25    train_x <- train_x[date < "2020-07-01"][,-1]
26  } else {
27    cols <- colnames[(i-1)]
28    features <- labels[, ..cols]
29    features <- setDT(as.data.frame(scale(features)))
30    features <- features[, (colnames(features)) := lapply(.SD, function(x)
          shift(x, n=1L)), .SDcols = colnames(features)][-1]
31    train_x <- setDT(cbind(date = labels$date[-1], features))# lags feature set
          and removes first row
32    train_x <- train_x[date < "2020-07-01"][,-1]
33  }
34
35  # Performing linear regression
36  data <- cbind(train_y, train_x)
37  names(data) <- c("y", "x") # changes column names to y and x for easy
        referencing
38  model <- lm(y ~ x, data = data)
39
40  # In-sample R-squared
41  r2_ols[i] <- round(summary(model)$r.squared, 2) # stores in-sample rsquared
        in vector
42
43  # Out-of-sample RMSE
44  # Creating response variable of test set
45  test_y <- as.data.frame(labels[[i+1]]) # grabs response variable
46  test_y <- setDT(cbind(date = labels$date, test_y))
47  test_y <- test_y[date >= "2020-07-01"][,-1]
48  names(test_y) <- colnames[i]
49  # Checks whether y is dutch_power_1
50  if (colnames(train_y) == "dutch_power_1") {
51    cols <- tail(colnames, 1)
52    features <- labels[, ..cols]
53    features <- setDT(as.data.frame(scale(features)))
54    features <- features[, (colnames(features)) := lapply(.SD, function(x)
          shift(x, n=1L)), .SDcols = colnames(features)][-1]
55    test_x <- setDT(cbind(date = labels$date[-1], features))
56    test_x <- test_x[date >= "2020-07-01"][,-1]
57  } else {
58    features <- labels[, ..cols]
59    features <- setDT(as.data.frame(scale(features)))
60    features <- features[, (colnames(features)) := lapply(.SD, function(x)
          shift(x, n=1L)), .SDcols = colnames(features)][-1]
61    test_x <- setDT(cbind(date = labels$date[-1], features))# lags feature set
          and removes first row
62    test_x <- test_x[date >= "2020-07-01"][,-1]
63  }
64
65  # Out-of-sample RMSE
```

```
66   setnames(test_y, colnames(test_y), c("y"))
67   setnames(test_x, colnames(test_x), c("x"))
68   predictions <- predict(model, newdata = test_x, type = "response")
69   predictions_ols[[i]] <- predictions
70   rmse_ols[i] <- round(sqrt(mean((test_y$y - predict(model, newdata = test_x,
         type = "response"))^2)), 2)
71
72   gc() # cleans garbage
73   message(paste0("Iteration ", i, " of ", length(colnames)))
74 }
```

- 4-10: Makes 1 dataframe with 1 column being the dates and the rest being the 24 hourly electricity prices. Stores all the variable names in a vector *colnames* (note that it removes the first element; the dates). Makes several vectors to store the results as well as an empty list that stores the predictions which I will use in question 5.
- 13-17: Grabs the response variable (i+1 because the first column is the date column) and combines it with the dates column. Drops the first row (features are lagged so we cannot predict the first value) and removes the date column (column 1). Applies the right element of the *colnames* vector as the column name.
- 19-32: Creates a features set that is lagged and scaled appropriately. The if-else statement makes an exception whenever the y-variable is the electricity price of the first hour of the day such that it uses $Dutch\_Power\_24$.
- 36-41: Performs the linear regression and saves the in-sample R^2 and the model.
- 45-62: Creates a test set in the same manner that I created the training set.
- 65-73: Changes the column names to y and x for simple referencing, makes the predictions and stores them for later use and calculates the out-of-sample RMSE.

For the sake of brevity I do not include the codes to plot the figures, but these are the figures you should have:

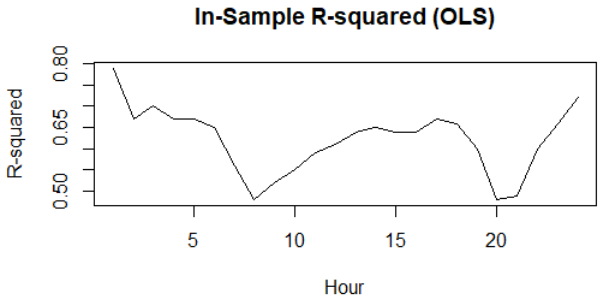

Figure 7: In-sample R^2

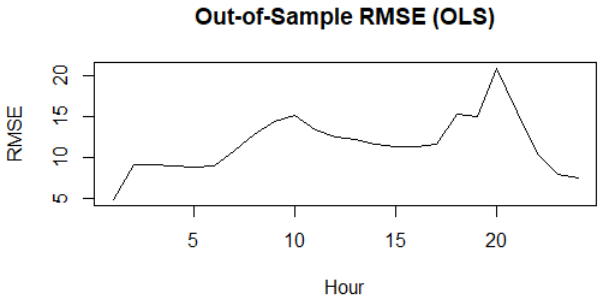

Figure 8: Out-of-sample RMSE

Question 4.2

Similar to the previous question, you should write a for loop here that matches the dependent variable with all the other features. It is a more complex combination of the previous questions.

LASSO

```
# Lasso
lasso_grid <- 10^seq(10, -2, length = 100) # hyperparameter grid
predictions_lasso <- list() # an empty list to store the LASSO predictions of each hour
colnames <- names(df)[2:25] # saves names of all 24 dependent variables to loop over (excludes the date, of course)
date <- df$date # dates that I cbind when I need to filter by date
rmse_val <- vector() # stores RMSE of validation set in order to determine optimal lambda
rmse_lasso <- vector() # stores out-of-sample RMSE
r2_lasso <- vector() # stores in-sample R2

for (i in 1:length(colnames)) {
  # Features
  features <- df[, .SD, .SDcols = !colnames[i]] # deselects y variable, keeps rest of the features
  features[, (colnames(features)[-1]) := lapply(.SD, function(x) shift(x, n=1L)), .SDcols = (colnames(features)[-1])]
  features <- features[, (colnames(features)[-1]) := lapply(.SD, function(x) scale(x)), .SDcols = (colnames(features)[-1])][-1,] # drops first row (NA after lagging variables)
  features <- features[, !(colnames(features) %in% zero_df), with=FALSE] # drops features that should be removed (see question 2)
  train_x <- features[date < "2020-01-01"][,-1]
  val_x <- features[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  test_x <- features[date >= "2020-07-01"][,-1]

  # Label
  label <- df[, .SD, .SDcols = colnames[i]]
  label <- setDT(cbind(date = date, label))[-1,]
  train_y <- label[date < "2020-01-01"][,-1]
  val_y <- label[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  test_y <- label[date >= "2020-07-01"][,-1]

  # Lasso regression
  for (j in 1:length(lasso_grid)) {
    lasso_model <- glmnet(train_x, train_y[[1]], alpha = 1, lambda = lasso_grid[j])
    predictions <- predict(lasso_model, newx = as.matrix(val_x), s = lasso_grid[j])
    rmse_val[j] <- sqrt(mean((val_y[[1]] - predictions)^2))
    gc() # clears memory
```

```
33   }
34
35   optimal_lambda_lasso <- lasso_grid[which.min(rmse_val)] # optimal lambda
36
37   lasso_model <- glmnet(train_x, train_y[[1]], alpha = 1, lambda = optimal_
         lambda_lasso) # trains model with optimal parameters
38   predictions <- predict(lasso_model, newx = as.matrix(test_x), type = "
         response", s = optimal_lambda_lasso)
39   predictions_lasso[[i]] <- predictions # stores predictions in list
40   r2_lasso[i] <- rsq(as.vector(predict(lasso_model, newx = as.matrix(train_x),
          type = "response", s = optimal_lambda_lasso)), train_y[[1]]) # in-sample
          R-squared
41   rmse_lasso[i] <- sqrt(mean((test_y[[1]] - predictions)^2)) # out-of-sample
         RMSE
42
43   message(paste0("Iteration ", i, " of ", length(colnames), " complete."))
44 }
```

- 2-8: Initiates empty vectors and a list in which I store all values.
- 12-18: Creates the right features sets for training, validation and testing.
- 21-25: Generates the correct lasso sets for training validation and testing.
- 29-32: Performs the lasso regressions iteratively using the grid I provided.
- 35-41: Calculates and stores the right values using the λ that minimized the RMSE of the validation set.

You should obtain the following values:

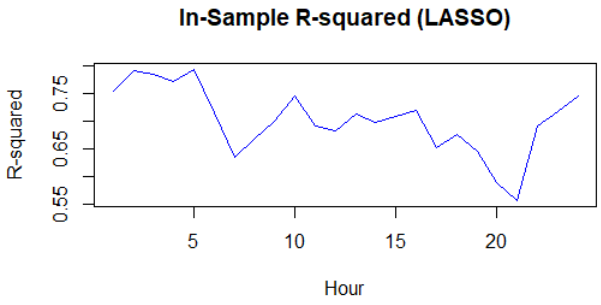

Figure 9: In-sample R^2

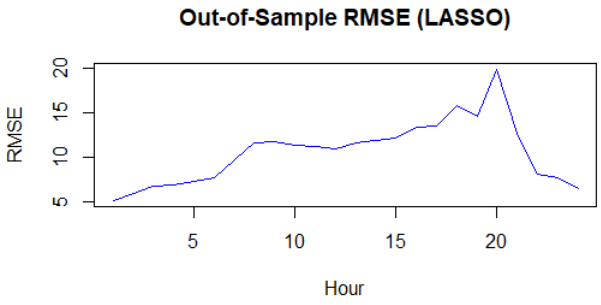

Figure 10: Out-of-sample RMSE

Partial Least Squares

```r
# Partial least squares regression
ncomp_grid <- c(1:20)
predictions_pls <- list()
colnames <- names(df)[2:25]
date <- df$date
rmse_val <- vector()
rmse_pls <- vector()
r2_pls <- vector()

for (i in 1:length(colnames)) {
  # Features
  features <- df[, .SD, .SDcols = !colnames[i]] # deselects y variable, keeps rest of the features
  features[, (colnames(features)[-1]) := lapply(.SD, function(x) shift(x, n=1L)), .SDcols = (colnames(features)[-1])]
  features <- features[, (colnames(features)[-1]) := lapply(.SD, function(x) scale(x)), .SDcols = (colnames(features)[-1])][-1,] # drops first row (NA after lagging variables)
  features <- features[, !(colnames(features) %in% zero_df), with=FALSE] # drops features that should be removed (see question 2)
  train_x <- features[date < "2020-01-01"][,-1]
  val_x <- features[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  test_x <- features[date >= "2020-07-01"][,-1]

  # Label
  label <- df[, .SD, .SDcols = colnames[i]]
  label <- setDT(cbind(date = date, label))[-1,]
  train_y <- label[date < "2020-01-01"][,-1]
  setnames(train_y, colnames(train_y), c("y"))
  val_y <- label[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  setnames(val_y, colnames(val_y), c("y"))
  test_y <- label[date >= "2020-07-01"][,-1]
  setnames(test_y, colnames(test_y), c("y"))

  # PLS regression (training + validation)
  for (j in 1:length(ncomp_grid)){
    pls_model <- caret::train(
      y ~ . ,
      data = cbind(train_y, train_x),
      method = "pls",
      tuneGrid = expand.grid(ncomp = ncomp_grid[j]),
      trControl = trainControl(method = "none"))

    predictions <- predict(pls_model, ncomp = ncomp_grid[j], newdata = val_x)
    rmse_val[j] <- sqrt(mean((val_y$y - predictions)^2))
  }

  optimal_ncomp_pls <- ncomp_grid[which.min(rmse_val)] # optimal number of ncomps
```

```
44
45   # Out-of-sample predictions
46   pls_model <- caret::train(
47     y ~ . ,
48     data = cbind(train_y, train_x),
49     method = "pls",
50     tuneGrid = expand.grid(ncomp = optimal_ncomp_pls),
51     trControl = trainControl(method = "none"))
52
53   predictions <- predict(pls_model, ncomp = optimal_ncomp_pls, newdata = test_
       x)
54   predictions_pls[[i]] <- predictions # stores predictions in list
55   rmse_pls[i] <- sqrt(mean((test_y$y - predictions)^2)) # out-of-sample RMSE
56   r2_pls[i] <- rsq(predict(pls_model, ncomp = optimal_ncomp_pls, newdata =
       train_x), train_y$y) # in-sample r-squared
57
58   message(paste0("Iteration ", i, " of ", length(colnames), " complete."))
59 }
```

- The structure of this code is similar to what I did for LASSO. Note that because I use caret's train function, some aspects are coded differently.

You should obtain the following values:

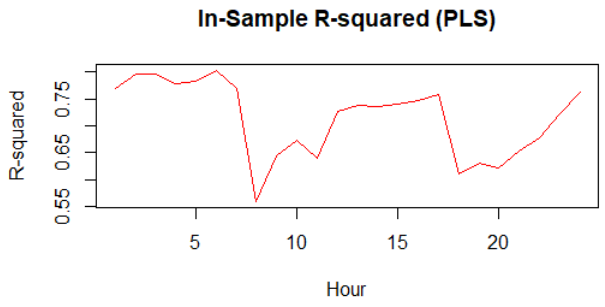

Figure 11: In-sample R^2

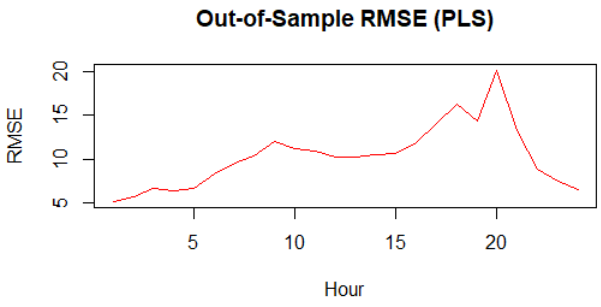

Figure 12: Out-of-sample RMSE

Random Forest

```
1 # Random Forest
2 # Creating grid + vector to store RMSE of models when predicting on validation
      set
3 random_forest_grid <- expand.grid(
4   mtry = c(201 / 3),
5   num.trees = c(1000),
6   max.depth = seq(1, 20)
7 )
8 predictions_rf <- list()
```

```r
colnames <- names(df)[2:25]
date <- df$date
rmse_val <- vector()
rmse_rf <- vector()
r2_rf <- vector()

for (i in 1:length(colnames)) {
  # Features
  features <- df[, .SD, .SDcols = !colnames[i]] # deselects y variable, keeps rest of the features
  features[, (colnames(features)[-1]) := lapply(.SD, function(x) shift(x, n=1L)), .SDcols = (colnames(features)[-1])]
  features <- features[, (colnames(features)[-1]) := lapply(.SD, function(x) scale(x)), .SDcols = (colnames(features)[-1])][-1,] # drops first row (NA after lagging variables)
  features <- features[, !(colnames(features) %in% zero_df), with=FALSE] # drops features that should be removed (see question 2)
  train_x <- features[date < "2020-01-01"][,-1]
  val_x <- features[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  test_x <- features[date >= "2020-07-01"][,-1]

  # Label
  label <- df[, .SD, .SDcols = colnames[i]]
  label <- setDT(cbind(date = date, label))[-1,]
  train_y <- label[date < "2020-01-01"][,-1]
  setnames(train_y, colnames(train_y), c("y"))
  val_y <- label[date >= "2020-01-01" & date < "2020-07-01"][,-1]
  setnames(val_y, colnames(val_y), c("y"))
  test_y <- label[date >= "2020-07-01"][,-1]
  setnames(test_y, colnames(test_y), c("y"))

  # Random forest (training + validation)
  for (j in 1:nrow(random_forest_grid)) {
    rf_model <- ranger(y ~ . ,
                  data = cbind(train_y, train_x),
                  mtry = random_forest_grid$mtry[j],
                  num.trees = random_forest_grid$num.trees[j],
                  max.depth = random_forest_grid$max.depth[j],
                  verbose = TRUE)

    predictions <- predict(rf_model, data = val_x)
    rmse_val[j] <- sqrt(mean((val_y$y - as.vector(predictions$predictions))^2))
  }

  optimal_mtry_rf <- random_forest_grid$mtry[which.min(rmse_val)]
  optimal_ntrees_rf <- random_forest_grid$num.trees[which.min(rmse_val)]
  optimal_depth_rf <- random_forest_grid$max.depth[which.min(rmse_val)]

  # Out-of-sample predictions
  rf_model <- ranger(y ~ . ,
                data = cbind(train_y, train_x),
```

```
                mtry = optimal_mtry_rf,
                num.trees = optimal_ntrees_rf,
                max.depth = optimal_depth_rf,
                verbose = TRUE)

  predictions <- predict(rf_model, data = test_x)
  predictions_rf[[i]] <- predictions # stores predictions in list
  r2_rf[i] <- rsq(predict(rf_model, data = train_x)$predictions, train_y$y) #
     in-sample r-squared
  rmse_rf[i] <- sqrt(mean((test_y$y - as.vector(predictions$predictions))^2))
     # out-of-sample RMSE

  message(paste0("Iteration ", i, " of ", length(colnames), " complete."))
}
```

You should obtain the following values:

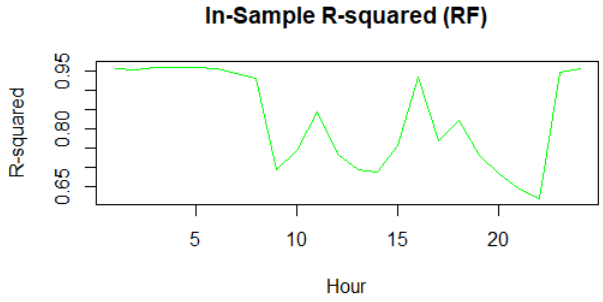

Figure 13: In-sample R^2

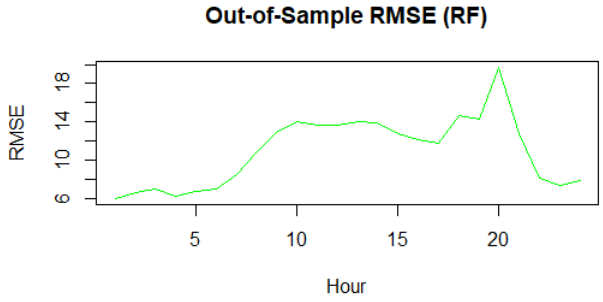

Figure 14: Out-of-sample RMSE

Plotting R^2 & RMSE

```
# Plotting all values
# Creates dataframe with r-squared values
r2_plot <- data.frame(r2_ols, r2_lasso, r2_pls, r2_rf)
# Generates plot
plot(r2_plot$r2_ols, type = "l", col = "black", ylim=c(0.45,1),
    main = "In-Sample R-squared", xlab = "Hour", ylab = "R-squared")
lines(r2_plot$r2_lasso, type = "l", col = "blue")
lines(r2_plot$r2_pls, type = "l", col = "red")
lines(r2_plot$r2_rf, type = "l", col = "green")
legend("bottomleft", legend=c("OLS", "LASSO", "PLS", "RF"),
     col=c("black", "blue", "red", "green"), lty=1:1, cex=0.5)

# Dataframe with RMSE values
rmse_plot <- data.frame(rmse_ols, rmse_lasso, rmse_pls, rmse_rf)
# Generates plot
plot(rmse_plot$rmse_ols, type = "l", col = "black",,
    main = "Out-of-Sample RMSE", xlab = "Hour", ylab = "RMSE")
```

```
lines(rmse_plot$rmse_lasso, type = "l", col = "blue")
lines(rmse_plot$rmse_pls, type = "l", col = "red")
lines(rmse_plot$rmse_rf, type = "l", col = "green")
legend("topleft", legend=c("OLS", "LASSO", "PLS", "RF"),
       col=c("black", "blue", "red", "green"), lty=1:1, cex=0.5)
```

You should obtain the following graphs:

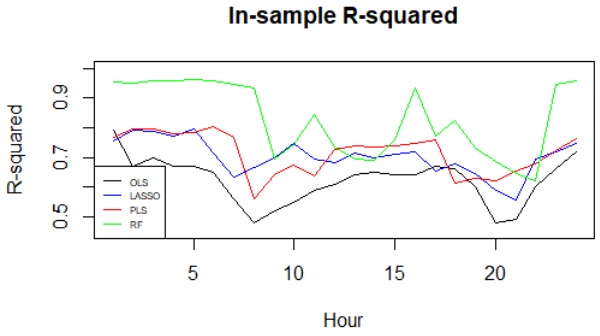

Figure 15: In-sample R^2

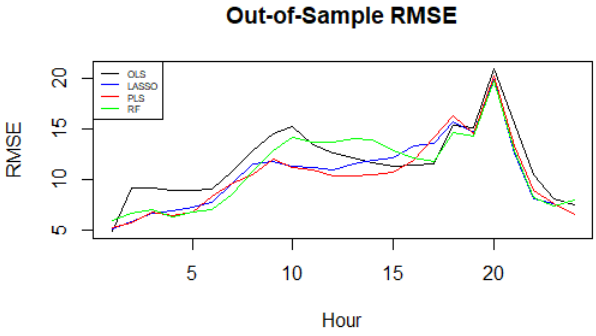

Figure 16: Out-of-sample RMSE

Question 5

Question 5.1

```
# Calculates average low and high prices and their corresponding hours
prices <- df[,1:25]
prices <- prices[date < "2020-07-01"][,-1]
prices <- t(colMeans(prices)) # transposes dataframe (long to wide)
low <- prices[,which.min(prices)] # 04:00 - 05:00 is lowest
high <- prices[,which.max(prices)] # 19:00 - 20:00 is highest
```

- 2: Creates one dataframe with all the Dutch electricity prices of each hour and a date column.
- 3-4: Creates a pseudo training set which I use to calculate the lowest and highest average price per hour. I also transpose the matrix that the *colMeans* function outputs.
- 5-6: Determines the hours of the lowest and highest average price.

You should find that prices are lowest between 04:00 and 05:00 and highest between 19:00 and 20:00. Note that this could also have been determined by looking at the graph we created in question 1.1.

```
prices <- df[,1:25]
prices <- prices[date >= "2020-07-01"][,-1]
profit <- prices[, list(dutch_power_20, dutch_power_5)]
profit[, profit := dutch_power_20 - dutch_power_5]
message(paste0("Cumulative profit: ", sum(profit$profit)))
message(paste0("Volatility: ", round(sd(profit$profit), 2)))
```

- 1-4: Calculates the profit of charging between 04:00 and 05:00 and discharging between 19:00 and 20:00.
- 5-6: Calculates the cumulative return as well as the volatility.

You should find a cumulative profit of €4,888 and a volatility of 21.29.[3]

Question 5.2

```
# LASSO
# Creating 1 large dataframe with predictions
predictions_lasso_df <- setDT(as.data.frame(do.call(cbind, predictions_lasso))
    )
```

[3]You can also translate this volatility to daily volatility by dividing the absolute volatility figure by the mean of the cumulative profit which gives you 80.16%.

```r
names(predictions_lasso_df) <- colnames(df[,2:25]) # assigns right column names
min_vector <- apply(predictions_lasso_df, 1, FUN = which.min) # determines index of column with min value
max_vector <- apply(predictions_lasso_df, 1, FUN = which.max) # determines index of column with max value

# Calculating profit of LASSO trading strategy
profit_vector_lasso <- vector()
for (i in 1:nrow(predictions_lasso_df)) {
  profit_vector_lasso[i] <- t(prices[i,])[max_vector[i]] - t(prices[i,])[min_vector[i]]
}
message(paste0("Cumulative profit: ", sum(profit_vector_lasso)))
message(paste0("Volatility: ", round(sd(profit_vector_lasso), 2)))

# Partial Least Squares
# Creating 1 large dataframe with predictions
predictions_pls_df <- setDT(as.data.frame(do.call(cbind, predictions_pls)))
names(predictions_pls_df) <- colnames(df[,2:25]) # assigns right column names
min_vector <- apply(predictions_pls_df, 1, FUN = which.min) # determines index of column with min value
max_vector <- apply(predictions_pls_df, 1, FUN = which.max) # determines index of column with max value

# Calculating profit of LASSO trading strategy
profit_vector_pls <- vector()
for (i in 1:nrow(predictions_pls_df)) {
  profit_vector_pls[i] <- t(prices[i,])[max_vector[i]] - t(prices[i,])[min_vector[i]]
}
message(paste0("Cumulative profit: ", sum(profit_vector_pls)))
message(paste0("Volatility: ", round(sd(profit_vector_pls), 2)))

# Random Forest
# Creating 1 large dataframe with predictions
predictions_rf_df <- list()
for (i in 1:length(predictions_rf)) {
  predictions_rf_df[[i]] <- predictions_rf[[i]]$predictions
}
predictions_rf_df <- setDT(as.data.frame(do.call(cbind, predictions_rf_df)))
names(predictions_rf_df) <- colnames(df[,2:25]) # assigns right column names
min_vector <- apply(predictions_rf_df, 1, FUN = which.min) # determines index of column with min value
max_vector <- apply(predictions_rf_df, 1, FUN = which.max) # determines index of column with max value

# Calculating profit of LASSO trading strategy
profit_vector_rf <- vector()
for (i in 1:nrow(predictions_rf_df)) {
```

```
45  profit_vector_rf[i] <- t(prices[i,])[max_vector[i]] - t(prices[i,])[min_
        vector[i]]
46 }
47 message(paste0("Cumulative profit: ", sum(profit_vector_rf)))
48 message(paste0("Volatility: ", round(sd(profit_vector_rf), 2)))
```

- 5-6: Determines the indices of the columns containing the lowest and highest predicted prices.
- 9-14: Subtracts the lowest actual price from the highest actual price to determine the strategy's profit of the LASSO strategy.
- 18-29: Applies the same principle to the PLS-based trading strategy.
- 33-38: Creates 1 large dataframe containing the random forest predictions. The *predict* function in combination with a *ranger*-based model doesn't actually output a vector of predictions, but rather a list containing multiple elements, 1 of those being a vector of predictions, hence I have to dedicate a for loop to it.
- 39-48: Calculates the total profit of the random forest-based strategy.

The LASSO, PLS and RF trading strategies have a cumulative profit of approximately €5,115, €4,535 and €4,915, respectively. The standard deviations are approximately 20.92, 17.54 and 22.03, respectively. It looks like the LASSO strategy is therefore somewhat superior to the other trading strategies.

Question 5.3

I somewhat deviate from what I am actually supposed to do here. Since there really isn't 1 particular model that outperforms the rest I will use the best model per hour and use that particular model's predictions for that respective hour. I will then use a simple trading strategy that exploits the average price movement (see question 1) by charging and discharging twice (note the two dips and peaks) per day.

```
1 min_rmse <- vector() # empty vector
2 for (i in 1:nrow(rmse_plot)) {
3   min_rmse[i] <- colnames(t(which.min(rmse_plot[i,])))
4 }
```

- 1: Initiates an empty vector in which I store the model name of the model with the lowest RMSE per hour.
- 2-3: Uses the *rmse_plot* dataframe that I created at the end of question 4.2 which then determines the lowest RMSE of each row and saves the corresponding model name.

In my case, this yields the following results:

Hour	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Best Model	OLS	PLS	LASSO	RF	PLS	RF	RF	PLS	LASSO	PLS	PLS	PLS
Hour	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Best Model	PLS	PLS	PLS	OLS	OLS	RF	RF	RF	LASSO	LASSO	RF	LASSO

```r
# Generates prediction dataframe using predictions of best model for each hour
# Generates OLS predictions dataframe
predictions_ols_df <- setDT(as.data.frame(do.call(cbind, predictions_ols)))
names(predictions_ols_df) <- colnames(df[,2:25]) # assigns right column names

final_prediction_df <- list()
for(i in 1:24) {
  pred_df <- as.data.frame(cbind(predictions_ols_df[[i]], predictions_lasso_df[[i]], predictions_pls_df[[i]], predictions_rf_df[[i]]))
  names(pred_df) <- c("rmse_ols", "rmse_lasso", "rmse_pls", "rmse_rf")
  final_prediction_df[[i]] <- pred_df[,min_rmse[i]]
}
final_prediction_df <- setDT(as.data.frame(do.call(cbind, final_prediction_df))) # cbinds all vectors within list
names(final_prediction_df) <- colnames(df[,2:25]) # assigns right column names

# Creates strategy
profit_strat_vec <- vector()
for (i in 1:nrow(final_prediction_df)) {
  # charge (morning)
  pred <- t(final_prediction_df[i,])
  charge_pred_morning <- which.min(pred[3:6])
  charge_pred_morning <- c(3:6)[charge_pred_morning]
  charge_real_morning <- -(t(prices[i,])[charge_pred_morning])
  # discharge (morning)
  discharge_pred_morning <- which.max(pred[8:11])
  discharge_pred_morning <- c(8:11)[discharge_pred_morning]
  discharge_real_morning <- t(prices[i,])[discharge_pred_morning]

  # charge (afternoon)
  charge_pred_after <- which.min(pred[14:17])
  charge_pred_after <- c(14:17)[charge_pred_after]
  charge_real_after <- -(t(prices[i,])[charge_pred_after])
  # discharge (afternoon)
  discharge_pred_after <- which.max(pred[18:21])
  discharge_pred_after <- c(18:21)[discharge_pred_after]
  discharge_real_after <- t(prices[i,])[discharge_pred_after]

  # Calculates total profit per day
  profit_strat_vec[i] <- (charge_real_morning + discharge_real_morning) + (charge_real_after + discharge_real_after)
}
```

```
40
41 message(paste0("Total profit is: ", sum(profit_strat_vec)))
42 message(paste0("Volatility is: ", round(sd(profit_strat_vec), 2)))
```

- 3-13: Creates a prediction dataframe for OLS' predictions (I only did this for the LASSO, PLS and Random Forest predictions earlier) and ultimately generates a prediction dataframe that consists of the predictions of the best model of the corresponding hours.
- 19: Creates a subset per day (per row).
- 20-26: Calculates the lowest predicted price among hours 3 to 6 and the highest predicted price among hours 8 to 11. It subsequently calculates the profit when charging at the lowest predicted hour and discharging during the highest predicted hour.
- 29-35: Performs the same calculations between hours 14 to 17 and 18 to 21.
- 38: Calculates the total profit.

This simple strategy yields a cumulative profit of approximately €7,331 and has a volatility of 26.06 and therefore beats the other strategies. This is of course due to the fact that it makes multiple trades per day.

References

Boehmke, B., & Greenwell, B. (2019). *Hands-on machine learning with r*. Chapman and Hall/CRC.

Dowle, M., Srinivasan, A., Short, T., & Lianoglou, S. (2019). data.table: Extension of 'data.frame'. *R package version*, *1*(8).