Electricity Consumption Forecasting for a Single Day (15-Minute Intervals)

Nicolas Len
Data ScienceTech Institute, 2025

Contents

```r
# Set CRAN mirror
options(repos = c(CRAN = "https://cran.rstudio.com"))

# Load necessary libraries
library(tidyverse)     # For data wrangling and visualization
library(knitr)         # For creating tables and controlling chunk output
library(kableExtra)    # For enhanced table formatting
library(ggplot2)       # For data visualization
library(dplyr)         # For data manipulation
library(tidyr)         # For data tidying
library(readr)         # For reading data
library(lubridate)     # For date manipulation
library(stringr)       # For string manipulation
library(forcats)       # For categorical data handling
library(rmarkdown)     # For document rendering
library(forecast)

# Set global chunk options
knitr::opts_chunk$set(
  echo = TRUE,         # Display code in the output
  eval = FALSE,        # Do not evaluate code chunks
  warning = FALSE,     # Suppress warnings in the output
  message = FALSE,     # Suppress messages in the output
  fig.width = 7,       # Set default figure width
  fig.height = 5,      # Set default figure height
  fig.align = 'center', # Center align all figures by default
  cache = TRUE         # Enable caching to speed up knitting
)
```

1: Preprocessing

1.1 Data upload

```r
library(readxl)
library(lubridate)

# Set up the path as a separate line
```

```r
file_path <- "C:/Users/tufma/Desktop/!!time_series/Exam Febr/Elec_30_11_train.xlsx"

# Read the Excel file
df <- read_excel(file_path)

# Define the column names
colnames(df) <- c("timestamp","temp","power")
```

1.2 Remove rows and cols

```r
# Remove the first row
df <- df[-1, ]

# Drop the 4th
if (ncol(df) >= 4) {
  df <- df[, -4]
}

# View the modified dataset
head(df)
```

1.3 Column Data Types and NAs

```r
summary(df)
```

We see that all the 3 columns are strings. We need to convert timestamps and make 'temp' and 'power' numeric. After the transformation we will double check NAs

1.4 Convert timestamps

```r
# Parse all timestamps
df$timestamp <- parse_date_time(
  df$timestamp,
  orders = c("mdY HM", "Y-m-d H:M:S"),
  tz = "UTC"
)
```

1.5 Convert to numeric

```r
df$temp <- as.numeric(df$temp)
df$power <- as.numeric(df$power)
```

1.6 Double check for NAs

```
summary(df)
```

Now, as expected we have just 96 NAs in power column which refers to the last day that we should forecast.

1.7 Check for the time series irregularities

```
# Install and load tsibble
library(tsibble)

# Convert data frame to a tsibble
df_tsibble <- as_tsibble(df, index = timestamp)

# Check if the time series is regular
is_regular <- is_regular(df_tsibble)
print(paste("Is the time series regular?", is_regular))

# Identify gaps in the time series
gaps <- scan_gaps(df_tsibble)
print("Gaps in the time series:")
print(gaps)
```

Based on the output, the dataframe is regular and there are no gaps in the timestamps.

1.8 Cutting out first lines

Despite the fact we haven't found any gaps in the data frame, by manual inspection of the data set, we can see that the observations in the first date start from 01:15 instead of 00:00. It means that the first day of the observation doesn't have the full set of observations. Taking into account that our forecast period starts at 00:00, the best way is to delete the first day from our analysis.

```
library(dplyr)
library(lubridate)
```

Step 1: Calculate the frequency of the time series in minutes

```
frequency_minutes <- as.numeric(difftime(df$timestamp[2], df$timestamp[1], units =
 ↪  "mins"))
print(paste("Detected frequency (minutes):", frequency_minutes))
```

Step 2: Calculate the expected number of observations per day

```
observations_per_day <- 24 * 60 / frequency_minutes
print(paste("Expected observations per day:", observations_per_day))
```

Step 3: Identify incomplete days

```r
incomplete_days <- df %>%
  group_by(date = as.Date(timestamp)) %>%
  summarise(observations = n(), .groups = "drop") %>%
  filter(observations < observations_per_day)

# Print days with missing observations
if (nrow(incomplete_days) > 0) {
  print("Days missing full set of observations:")
  print(incomplete_days)
} else {
  print("No missing days found.")
}
```

Step 4: Remove incomplete days from df

```r
df <- df %>%
  filter(!as.Date(timestamp) %in% incomplete_days$date)

print("Incomplete days removed. df updated.")
# Display the first few rows of the updated data frame to verify
print(head(df))
```

1.9 Divide dataframe into historical and forecast parts

```r
# Split df into historical and forecast parts based on NA in the power column
df_historical <- df %>%
  filter(!is.na(power))

df_forecast <- df %>%
  filter(is.na(power))

# Print summary information to verify the split
print("Historical Data (df_historical):")
print(head(df_historical))
print(paste("Number of historical rows:", nrow(df_historical)))

print("Forecast Data (df_forecast):")
print(head(df_forecast))
print(paste("Number of forecast rows:", nrow(df_forecast)))
```

1.10 Check for outliers

Check for power

```r
ggplot(df_historical, aes(x = timestamp, y = power)) +
geom_line() +
labs( x = "Timestamp", y = "Power Consumption")
```

Based on the graph we can obviously see at least 2 zones of outliers. Let's check for the temp

```r
ggplot(df_historical, aes(x = timestamp, y = temp)) +
geom_line() +
labs( x = "Timestamp", y = "Temp")
```

The data point reaching 100 degrees seems suspicious.

1.11 Clean the outliers

We will use tsclean function from forecast library to impute the outliers

```r
# Load the forecast package
library(forecast)

# Step 1: Convert the power and temp columns in df_historical to time series objects
# Assuming a regular frequency
power_ts <- ts(df_historical$power, frequency = 96)
temp_ts <- ts(df_historical$temp, frequency = 96)

# Step 2: tsclean to impute outliers and missing values in both columns
cleaned_power <- tsclean(power_ts)
cleaned_temp <- tsclean(temp_ts)

# Step 3: Replace the original columns in df_historical with the cleaned values
df_historical$power <- as.numeric(cleaned_power)
df_historical$temp <- as.numeric(cleaned_temp)

# Print the summary of the cleaned dataframe
summary(df_historical)
```

1.12 Double check the outliers

Check for power

```r
ggplot(df_historical, aes(x = timestamp, y = power)) +
geom_line() +
labs( x = "Timestamp", y = "Power Consumption")
```

Now the power looks better. What about temp?

```r
ggplot(df_historical, aes(x = timestamp, y = temp)) +
geom_line() +
labs( x = "Timestamp", y = "Temp")
```

Not much difference from the original series, but it still looks a bit smoother.

2: Data analysis

Taking into account that temperature influence the power consumption and power consumption doesn't influence the temperature, we are dealing with univariate time series with external regressor presented by temperature

2.1 Convert df_historical to time series object

```
# Convert power to a time series object
power_ts <- ts(df_historical$power, frequency = 96) # 96 intervals per day for 15-min
 ↪  data

# Extract temp as a time series (external regressor)
temp_ts <- ts(df_historical$temp, frequency = 96)
```

2.2 STL decomposition to analyze trend, seasonal, and residual components

```
# Decompose the time series to analyze trend and seasonality
decomposition <- stl(power_ts, s.window = "periodic")
autoplot(decomposition) +
  ggtitle("Decomposition of Power Time Series") +
  labs(x = "Time", y = "Power (kW)")
```

From the plot we can infer that there is an upward trend in the first half a year and a slight downward trend in the second half year. Taking into account that we have a lot of data points and at the same time we are required to forecast just one day in November, let's cut the dataset in such a way that we keep only downward trend. It can help our models to capture linear downward trend instead of taking into account more complicated parabolic trend. In addition, by doing so we will reduce the computation workload.

2.3 Cut the dataset

In order to optimize computation workload we will take a dataset of the last 64 days. Taking into account that we are going to train machine learning models we split the data set into 3 subsets: training, validation and test sets. This should help us to avoid overfitting. In addition I have a hypothesis (it will be checked later) that we have a strong consumption pattern through 24 hours: Night, Morning, Day, Evening. Let's add these new features to the dataset through one-hot encoding.

```
library(lubridate)

# 0) Add one-hot encoded columns based on the timestamp column
df_historical$hour <- hour(df_historical$timestamp)
df_historical$Night   <- as.integer(df_historical$hour >= 0  & df_historical$hour < 6)
df_historical$Morning <- as.integer(df_historical$hour >= 6  & df_historical$hour < 12)
df_historical$Day     <- as.integer(df_historical$hour >= 12 & df_historical$hour < 18)
df_historical$Evening <- as.integer(df_historical$hour >= 18 & df_historical$hour < 24)
df_historical$hour <- NULL  # remove temporary column

# 1) Identify earliest & latest day in df_historical
first_day <- min(df_historical$timestamp)
last_day  <- max(df_historical$timestamp)
# If df_historical doesn't have day_number yet, create it
df_historical$day_number <- as.integer(difftime(df_historical$timestamp, first_day, units
 ↪  = "days")) + 1
total_days <- max(df_historical$day_number)
```

```r
cat("First day:", format(first_day, "%Y-%m-%d"), "\n")
cat("Last  day:", format(last_day,  "%Y-%m-%d"), "\n")
cat("Total days in dataset:", total_days, "\n\n")

# 2) Desired day counts (train, val, test)
n_train <- 60
n_val   <-  2
n_test  <-  2
days_needed <- n_train + n_val + n_test

# We keep the last 'days_needed' days of the dataset
cut_start_day <- total_days - days_needed + 1
cat("We keep days", cut_start_day, "through", total_days, "...\n\n")

# 3) Build df_historical_cut
df_historical_cut <- subset(df_historical, day_number >= cut_start_day)

# 4) Define day ranges for train, val, test
train_start <- cut_start_day
train_end   <- cut_start_day + n_train - 1
val_start   <- train_end + 1
val_end     <- train_end + n_val
test_start  <- val_end + 1
test_end    <- val_end + n_test

cat("Train day_number range:", train_start, "to", train_end, "\n")
cat("Val   day_number range:", val_start, "to", val_end, "\n")
cat("Test  day_number range:", test_start, "to", test_end, "\n\n")

# 5) Keep separate dataframes:
df_historical_cut_train     <- subset(df_historical_cut, day_number >= train_start &
 ↪  day_number <= train_end)
df_historical_cut_val       <- subset(df_historical_cut, day_number >= val_start   &
 ↪  day_number <= val_end)
df_historical_cut_test      <- subset(df_historical_cut, day_number >= test_start  &
 ↪  day_number <= test_end)
df_historical_cut_train_val <- subset(df_historical_cut, day_number >= train_start &
 ↪  day_number <= val_end)

# 6) Create multivariate ts object
ts_historical_cut <- ts(df_historical_cut[ , !(names(df_historical_cut) %in%
 ↪  c("timestamp"))],
                        frequency = 96, start = c(cut_start_day, 1))

# Create the training, validation, test, and train+validation mts objects using window()
ts_historical_cut_train     <- window(ts_historical_cut, start = c(train_start, 1), end =
 ↪  c(train_end, 96))
ts_historical_cut_val       <- window(ts_historical_cut, start = c(val_start, 1),   end =
 ↪  c(val_end, 96))
ts_historical_cut_test      <- window(ts_historical_cut, start = c(test_start, 1),  end =
 ↪  c(test_end, 96))
ts_historical_cut_train_val <- window(ts_historical_cut, start = c(train_start, 1), end =
 ↪  c(val_end, 96))
```

Build a new STL plot

```
# Decompose the time series to analyze trend and seasonality
decomposition <- stl(ts_historical_cut[,"power"], s.window = "periodic")
autoplot(decomposition) +
  ggtitle("Decomposition of Power Time Series") +
  labs(x = "Time", y = "Power (kW)")
```

2.4 Cut window for further analysis

Despite the fact that we have cut our dataset, it's still too large for visual analysis purposes. Let's have look at the last month.

```
power_ts_cut_month=window(power_ts,start=c(300,1),end=c(332,96))
temp_ts_cut_month=window(temp_ts,start=c(300,1),end=c(332,96))
```

Build an STL plot

```
# Decompose the time series to analyze trend and seasonality
decomposition <- stl(power_ts_cut_month, s.window = "periodic")
autoplot(decomposition) +
  ggtitle("Decomposition of Power Time Series") +
  labs(x = "Time", y = "Power (kW)")
```

Now we can clearly see a stable seasonality pattern. Moreover, we can see that the remainder looks like it has some additional pattern (not just the white noise)

2.5 Seasonality plot

```
seasonplot(power_ts_cut_month, main = "Daily Seasonal Plot", col = rainbow(7),
 ↪  year.labels = FALSE, pch = 19)
```

From the daily seasonality plot we can clearly see a pattern that we can observe in real life. Night hours are the lowest in terms of electricity consumption, the peak hours are somewhat in the evening. Moreover, I have a hypothesis that the consumption on weekends or some weekdays can be less or instead higher than on other days. Let's check this hypothesis.

2.6 Seasonality in weekdays

Convert timestamp to Date format and aggregate to daily level

```
df_daily <- df_historical_cut %>%
  mutate(date = as.Date(timestamp)) %>%  # Extract only the date part
  group_by(date) %>%
  summarise(daily_power = mean(power, na.rm = TRUE))  # Compute daily mean power
```

Convert to a Time Series Object

```
power_ts_weekly <- ts(df_daily$daily_power, frequency = 7, start =
 ↪  c(year(df_daily$date[1]), week(df_daily$date[1])))
```

Visualize seasonality based on weekdays

```
seasonplot(power_ts_weekly, main = "Weekday Seasonal Plot",
           col = rainbow(7), year.labels = FALSE, pch = 19)
```

From the plot it's quite ambiguous whether the daily seasonality exists. Let's make ANOVA test

```
# Ensure weekday column is present
df_daily$weekday <- weekdays(df_daily$date)

# Convert weekday to a factor (ensures correct ordering)
df_daily$weekday <- factor(df_daily$weekday,
                           levels = c("Monday", "Tuesday", "Wednesday",
                                      "Thursday", "Friday", "Saturday", "Sunday"))

anova_test <- aov(daily_power ~ weekday, data = df_daily)
summary(anova_test)
```

P-value (Pr(>F)) is much greater than 0.05, meaning no significant difference in power consumption across weekdays.

F-value A very low F-value suggests that variation in power consumption is not explained by weekdays.

Sum of Squares (Sum Sq) for weekday is very small compared to the residual variance, meaning most variability in power usage comes from other factors, not the day of the week.

Thus, there is no strong evidence that power consumption varies significantly across weekdays.

2.7 Augmented Dickey-Fuller test

```
library(tseries)

# Perform the Augmented Dickey-Fuller Test
adf_result <- adf.test(ts_historical_cut[,"power"], alternative = "stationary")

# Print the result
print(adf_result)
```

The results of the Augmented Dickey-Fuller (ADF) test indicate that the time series is stationary.

2.8 Relationship Power vs Temperature

Let's have a look at the relationship between Power and Temperature

```
plot(temp_ts_cut_month, power_ts_cut_month, xlab="Temperature", ylab="Power",
 ↪  main="Scatter plot of Power vs Temperature")
```

The scatter plot shows that temperature indeed has influence on power consumption. However, the relationship is not linear or quadratic. Instead the plot shows two distinct clusters. Let's further investigate the nature of the clusters. Despite the fact that I have already checked the influence of weekdays on the power consumption, I would like to double check it on the scatter plot

```
# Extract day of the week and classify as weekday (0) or weekend (1)
df_historical_cut$day_type <- ifelse(weekdays(df_historical_cut$timestamp) %in%
 ↪  c("Saturday", "Sunday"), 1, 0)
```

Plot the points marked weekend or weekday

```
# Define colors: Blue for Weekday (0), Red for Weekend (1)
colors <- ifelse(df_historical_cut$day_type == 0, "blue", "red")

# Create scatter plot with color differentiation
plot(df_historical_cut$temp, df_historical_cut$power, col = colors, pch = 19,
     xlab = "Temperature", ylab = "Power", main = "Power vs Temperature by Day Type")

# Add legend
legend("topleft", legend = c("Weekday", "Weekend"), col = c("blue", "red"), pch = 19)
```

We have confirmed that the clusters we see are not workday vs weekend. Let's go further: we create an additional column that classifies each data point in terms of the day time: Night, Morning, Day, Evening.

```
# Extract hour from timestamp
df_historical_cut$hour <- as.numeric(format(df_historical_cut$timestamp, "%H"))

# Classify into four time clusters
df_historical_cut$daytime <- cut(df_historical_cut$hour,
                                 breaks = c(-1, 5, 11, 17, 23),
                                 labels = c("Night", "Morning", "Day", "Evening"))

# Define colors for each cluster
color_map <- c("Night" = "blue", "Morning" = "yellow", "Day" = "green", "Evening" =
 ↪  "red")
colors <- color_map[df_historical_cut$daytime]

# Create scatter plot with color differentiation
plot(df_historical_cut$temp, df_historical_cut$power, col = colors, pch = 19,
     xlab = "Temperature", ylab = "Power", main = "Power vs Temperature by Daytime
      ↪  Clusters")

# Add legend
legend("topleft", legend = names(color_map), col = color_map, pch = 19)
```

Great! Power demand follows a predictable daily cycle: Night: low, Morning: increasing, Day: higher, but variable, Evening: peak demand. For now we won't create any additional columns and let the models capture the seasonal pattern.

2.9 Results of the data analysis

The analysis reveals the following attributes of our time series:

1. The dataset is stationary;
2. There is a slight downward trend;
3. There is a strong intraday seasonality that can be devided into 4 clusters
4. Days of week follow similar energy consumption patterns
5. Temperature serves as an external regressor with a non-linear dependancy

2.10 Selection of models

Based on the task, we need to perform a forecast both with and without an external regressor. Here is the set of the models to be tested:

1. Holt-Winters (HW) - without xreg
2. ARIMA – without and with xreg
3. NNETAR – without and with xreg
4. Random Forest (RF) – without and with xreg
5. XGBoost – without and with xreg
6. Prophet (by Meta) – without and with xreg

3: Modelling

3.1 Holt-Winters

Taking into account that our dataset demonstrates seasonality and a subtle downward trend we will start our training with Holt-Winters (HW) model which is an extension of exponential smoothing used for time series forecasting, particularly when the data exhibits trend and seasonality. Based on the graphs we can conclude that seasonal variations are relatively constant. That is why we will proceed with Additive HW. As HW is deterministic we don't need validation set here that's why we will use combined train and validation set for training.

3.1.1 Default HW

Let's start with default parameters

```
library(forecast)
hw_default <- HoltWinters(ts_historical_cut_train_val[,"power"],
                          seasonal = "additive",
                          alpha = NULL,
                          beta  = NULL,
                          gamma = NULL)
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
hw_default_forecast <- forecast(hw_default, h = 2 * 96)

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
 ↪  autolayer(hw_default_forecast$mean, series="additive HW")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
hw_default_rmse <- sqrt(mean((hw_default_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("HW Default RMSE:", hw_default_rmse, "\n")
```

From the plot we can infer that the model is trying to repeat the pattern of the test set, but it considerably underperforms on the high, middle and low peaks. Let's check residuals and numeric parameters.

```
summary(hw_default)
checkresiduals(hw_default)
tsdisplay(residuals(hw_default))
```

From ACF and PCF we clearly see that unfortunately a strong autocorrelation remains in the residuals. It is also confirmed by extremely low p-value in the Ljung–Box test. Let's check the smoothing parameters the model has chosen.

```
# Extract all optimized smoothing parameters
optimized_parameters <- c(alpha = hw_default$alpha,
                          beta = hw_default$beta,
                          gamma = hw_default$gamma)

# Print the optimized parameters
cat("Optimized Default Parameters:\n")
print(optimized_parameters)
```

3.1.2 HW Finetuned

Let's use the default smoothing parameters as a starting point for the adjustment. Just to recap: - Alpha ( ) determines how much weight is given to the most recent observations (1 gives more weight to recent observations, a value close to 0 gives more weight to older observations) - Beta ( ) controls the smoothing of the trend component - Gamma ( ) determines how much weight is given to the most recent observations when estimating the seasonal pattern

As our test data are the autumn days, it makes sense to give more weight to recent seasonal patterns by increasing Gamma, as the whether is getting colder and the seasonal pattern from the latest days can be the most relevant. Also we will play around with Alpha, and at the same time will be very cautious in terms of Beta as the trend is very subtle.

```
library(forecast)
hw_ft <- HoltWinters(ts_historical_cut_train_val[,"power"],
                     seasonal = "additive",
                     alpha = 0.45,    # or just omit entirely
                     beta  = 0.00003,
                     gamma = 0.8)
```

Forecast for the next 2 days (test set)

```r
library(ggplot2)
library(forecast)
hw_ft_forecast <- forecast(hw_ft, h = 2 * 96)

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
 ↪  autolayer(hw_ft_forecast$mean, series="additive HW Finetuned")
```

RMSE calculation

```r
# Calculate the Root Mean Squared Error (RMSE)
hw_ft_rmse <- sqrt(mean((hw_ft_forecast$mean - ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("HW Finetuned RMSE:", hw_ft_rmse, "\n")
```

Through multiple iterations we have managed to find a set of parameters leading us to a visible improvement of the plot and RMSE. Let's check residuals and numeric parameters.

```r
summary(hw_ft)
checkresiduals(hw_ft)
tsdisplay(residuals(hw_ft))
```

Despite the improvement of RMSE. ACF and PCF still indicates a strong autocorrelation in the residuals. It is also confirmed by extremely low p-value in the Ljung–Box test. It means that the model struggles to catch the patterns in a proper way.

3.2 ARIMA

3.2.1 Auto-ARIMA w/o Xreg

```r
library(forecast)
arima_auto = auto.arima(
  ts_historical_cut_train_val[,"power"],
  seasonal = TRUE,
  approximation = TRUE,
  stepwise = TRUE,
  trace = TRUE,
  max.p = 15, max.q = 15,
  max.P = 2, max.Q = 2
)
```

Forecast for the next 2 days (test set)

```r
library(ggplot2)
library(forecast)
arima_auto_forecast <- forecast(arima_auto, h = 2 * 96)

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
 ↪  autolayer(arima_auto_forecast$mean, series="Auto Arima")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
arima_auto_rmse <- sqrt(mean((arima_auto_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("Auto Arima RMSE:", arima_auto_rmse, "\n")
```

The plot and RMSE immediately look better than those of HW. Let's check residuals and parameters.

```
summary(arima_auto)
checkresiduals(arima_auto)
tsdisplay(residuals(arima_auto))
```

The ACF and PACF plots indicate that the time series is relatively uncorrelated except for a set of distinct negative spikes in the PACF (and one in ACF). Notably, these spikes occur in PACF at seasonal lags—specifically at lags 96, 192, 288, etc.—with the largest spike at lag 96, and each subsequent spike decreasing in magnitude. This pattern suggests a seasonal effect with a period of our 96 observations per day. The reason could be a drop in energy consumption at night. At the next tries we will focus on seasonal AR terms (P), but first let's start with TSLM by removing the effect of covariate.

3.2.2 TSLM

```
tslm_model=tslm(power~temp+trend+season,data=ts_historical_cut_train_val)
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
tslm_model_forecast <- forecast(tslm_model, newdata = ts_historical_cut_test[,"temp"])

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
 ↪  autolayer(tslm_model_forecast$mean, series="TSLM")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
tslm_rmse <- sqrt(mean((tslm_model_forecast$mean - ts_historical_cut_test[,"power"]) ^
 ↪  2))

# Print the RMSE
cat("TSLM RMSE:", tslm_rmse, "\n")
```

The plot and RMSE is worse then Auto Arima and fine tuned HW, but interestingly it's better than default HW. It's due to the fact that we have added temperature which improves the performace. Let's check residuals and parameters.

```
summary(tslm_model)
checkresiduals(tslm_model)
tsdisplay(residuals(tslm_model))
```

This model tells us that power consumption is very well explained (about 94% of the variation) by temperature, a time trend, and seasonal effects. In particular:

• The temperature coefficient is highly significant, meaning that for every one-unit increase in temperature, power consumption increases by about 1.425 units, holding other factors constant.

• The trend coefficient is negative and highly significant, which suggests a very slight downward movement in power consumption over time. We saw it on the graph.

• The seasonal dummy variables capture strong cyclical patterns throughout the period. Some seasonal coefficients (especially those later in the year) are very large and statistically significant, indicating that power usage in certain seasons differs markedly from the baseline season. In contrast, a few seasonal dummies (e.g., season2, season3, season5, season6, season10, etc.) are not statistically significant, suggesting that not every period deviates significantly from the baseline.

ACF shows a strong repeating pattern that does not die out quickly, it may imply that we need seasonal differencing at least D=1. Also Large, slowly decaying spikes at seasonal lags can incur a seasonal MA component (>0).

Based on the plots from auto arima and tslm we need to consider non zero P,D,Q

3.2.3 Manual ARIMA w/o Xreg

let's take the parameters from the auto arima as starting point which is ARIMA(7,0,2)(0,1,0)[96] and add P=1 and Q=1

```
library(forecast)
arima_manual <- Arima(
ts_historical_cut_train_val[,"power"],
order = c(0,1,9),
seasonal = list(order = c(1,1,1))
)
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
arima_manual_forecast <- forecast(arima_manual, h = 2 * 96)

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
 ↪  autolayer(arima_manual_forecast$mean, series="Manual Arima")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
arima_manual_rmse <- sqrt(mean((arima_manual_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("Manual Arima RMSE:", arima_manual_rmse, "\n")
```

Let's check residuals and parameters.

```
summary(arima_manual)
checkresiduals(arima_manual)
tsdisplay(residuals(arima_manual))
```

After multiple iterations we have managed to achieve the best result with the following model

ARIMA(0,1,9)(1,1,1)[96]

Theoretically increase of P and Q could improve the result, but the calculation takes too long. Also the increase of q more than 9 leads to infinite calculation. Adding any non-zero parameter to p while having (1,1,1) for (P,D,Q) doesn't allow even to start training (the code breaks). Unfortunately despite the best RMSE so far we haven't managed to capture the full pattern as the residuals contain patterns. Let's test a model with this parameters with temperature as xreg.

3.2.4 Manual ARIMA w/ Xreg

let's take thi model ARIMA(0,1,9)(1,1,1)[96] and add temperature as xreg

```
library(forecast)
arima_manual_xreg <- Arima(ts_historical_cut_train_val[,"power"],
                      order = c(0,1,9),
                      seasonal = list(order = c(1,1,1)),
                      xreg = ts_historical_cut_train_val[,"temp"])
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
arima_manual_xreg_forecast <- forecast(arima_manual_xreg, h = 2 * 96,
                                  xreg = ts_historical_cut_test[,"temp"])

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
  autolayer(arima_manual_xreg_forecast$mean, series = "Manual Arima w/ Xreg")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
arima_manual_xreg_rmse <- sqrt(mean((arima_manual_xreg_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("Manual Arima w/ Xreg RMSE:", arima_manual_xreg_rmse, "\n")
```

Let's check residuals and parameters.

```
summary(arima_manual_xreg)
checkresiduals(arima_manual_xreg)
tsdisplay(residuals(arima_manual_xreg))
```

With xreg, we've managed to improve the RMSE. However, the residuals still exhibit autocorrelation. For our upcoming models, we plan to incorporate xreg right from the start of the training process.

3.3 NNAR w/ Temperature

3.3.1 NNAR w/ Temperature

Let's start with NNAR including temperature as xreg with default parameters.

```
library(forecast)
nnar_temp=nnetar(ts_historical_cut_train_val[,2],xreg=ts_historical_cut_train_val[,1])
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
nnar_temp_forecast <- forecast(nnar_temp, h = 2 * 96,
                               xreg = ts_historical_cut_test[,1])

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
  autolayer(nnar_temp_forecast$mean, series = "NNAR w/ Temp")
```

From the plot we can see that for some reasons the model performs poorly at the high peaks. Let's have a look at RMSE.

```
# Calculate the Root Mean Squared Error (RMSE)
nnar_temp_rmse <- sqrt(mean((nnar_temp_forecast$mean - ts_historical_cut_test[,"power"])
 ↪  ^ 2))

# Print the RMSE
cat("NNAR w/ Temp RMSE:", nnar_temp_rmse, "\n")
```

Interestingly RMSE is even better than the fine tuned HW, but considerably worse than all ARIMA models.

```
summary(nnar_temp)
checkresiduals(nnar_temp)
tsdisplay(residuals(nnar_temp))
```

As before we still don't see the white noise in ACF and PACF (some pattern is still here).

3.3.2 NNAR w/ Temperature & Daytime

In paragraph 2.8 devoted to the relationship between power and temperature we have identified 4 time clusters: Night, Morning, Day, Evening. Let's add these parameters through one-hot encoding into xreg along with the temperature.

```
library(forecast)
nnar_temp_dt=nnetar(ts_historical_cut_train_val[,2],xreg=ts_historical_cut_train_val[,
 ↪  c(1, 3, 4, 5, 6)])
```

Forecast for the next 2 days (test set)

```
library(ggplot2)
library(forecast)
nnar_temp_dt_forecast <- forecast(nnar_temp_dt, h = 2 * 96,
                                  xreg = ts_historical_cut_test[, c(1, 3, 4, 5, 6)])

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
  autolayer(nnar_temp_dt_forecast$mean, series = "NNAR w/ Xreg & Daytime")
```

RMSE calculation

```
# Calculate the Root Mean Squared Error (RMSE)
nnar_temp_dt_rmse <- sqrt(mean((nnar_temp_dt_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))

# Print the RMSE
cat("NNAR w/ Temp & Daytime:", nnar_temp_dt_rmse, "\n")
```

By adding daytime as an additional xreg we have substantially improved the performance of the model. The result is comparable to basic Auto Arima.

```
summary(nnar_temp_dt)
print(nnar_temp_dt)
checkresiduals(nnar_temp_dt)
tsdisplay(residuals(nnar_temp_dt))
```

The residuals also look better. Let's try to fine tune this model.

3.3.3 NNAR w/ Temp & Daytime finetuned

```
library(forecast)
# Define the tuning grid with parameters that vary
tuning_grid <- expand.grid(
  p = 30,        # Non-seasonal autoregressive lags
  P = 1,                  # Seasonal autoregressive lags
  size = 17         # Hidden layer sizes
)

best_rmse <- Inf
best_model <- NULL
best_params <- list()
total_models <- nrow(tuning_grid)
cat("Total models to train:", total_models, "\n")

for(i in 1:nrow(tuning_grid)) {
  cat("Training model", i, "of", total_models, "\n")

  # Extract parameters that vary
  p_val    <- tuning_grid$p[i]
  P_val    <- tuning_grid$P[i]
```

```
  size_val <- tuning_grid$size[i]

  # Fit NNAR model using the training set
  model <- nnetar(ts_historical_cut_train[, 2],
                  xreg = ts_historical_cut_train[, c(1, 3, 4, 5, 6)],
                  p = p_val,
                  P = P_val,
                  size = size_val,
                  repeats = 20,                # Fixed parameter
                  lambda = "auto",             # Fixed parameter
                  decay = 0,                   # Fixed parameter
                  maxit = 1000,                # Fixed parameter
                  stepmax = 1e+05,             # Fixed parameter
                  MaxNWts = 1000,              # Added parameter to increase weight limit
                  trace = FALSE)

  # Forecast over the validation horizon
  fc <- forecast(model, xreg = ts_historical_cut_val[, c(1, 3, 4, 5, 6)], h =
 ↪  length(ts_historical_cut_val[, 2]))
  rmse_val <- sqrt(mean((fc$mean - ts_historical_cut_val[, 2])^2))

  cat("RMSE for current model:", rmse_val, "\n\n")

  if(rmse_val < best_rmse) {
    best_rmse <- rmse_val
    best_model <- model
    best_params <- list(p = p_val, P = P_val, size = size_val)
  }
}

cat("Best parameters found on validation set:\n")
print(best_params)
cat("Best RMSE on validation set:", best_rmse, "\n")
```

Retrain the final model on the already combined training and validation set

```
nnar_temp_dt_ft <- nnetar(ts_historical_cut_train_val[, 2],
                      xreg = ts_historical_cut_train_val[, c(1, 3, 4, 5, 6)],
                      p = best_params$p,
                      P = best_params$P,
                      size = best_params$size,
                      repeats = 20,            # Fixed parameter
                      lambda = "auto",         # Fixed parameter
                      decay = 0,               # Fixed parameter
                      maxit = 1000,            # Fixed parameter
                      stepmax = 1e+05,         # Fixed parameter
                      trace = FALSE)
```

Finally, forecast using the final model on the test set

```
library(ggplot2)
library(forecast)
```

```
nnar_temp_dt_ft_forecast <- forecast(nnar_temp_dt_ft, xreg = ts_historical_cut_test[,
 ↪  c(1, 3, 4, 5, 6)], h = length(ts_historical_cut_test[, 2]))

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
  autolayer(nnar_temp_dt_ft_forecast$mean, series = "Finetuned NNAR w/ Temp & Daytime")
```

Let's check RMSE

```
nnar_temp_dt_ft_rmse <- sqrt(mean((nnar_temp_dt_ft_forecast$mean -
 ↪  ts_historical_cut_test[, 2])^2))
cat("RMSE Finetuned NNAR w/ Temp & Daytime:", nnar_temp_dt_ft_rmse, "\n")
```

Residuals

```
summary(nnar_temp_dt_ft)
print(nnar_temp_dt_ft)
checkresiduals(nnar_temp_dt_ft$residuals)
tsdisplay(nnar_temp_dt_ft$residuals)
```

Unfortunately, after multiple iterations we haven't managed to achieve better results than the automatic nnetar function. I was changing the number of lagged values of the time series to use as inputs to the neural network (p), number of seasonal autoregressive lags (P), and number of hidden nodes in the neural network (size).

3.4 Decision trees

3.4.1 RF with temp & daytime w/o lags

Let's test random forest with temperature and daytime (morning, night, day, evening) covariates. For now we won't add time lags into model

```
library(randomForest)
set.seed(777)
rf_dt_no_lag=randomForest(ts_historical_cut_train_val[,2],x=ts_historical_cut_train_val[,
 ↪  c(1, 3, 4, 5, 6)])
```

Let's build a graph

```
library(ggplot2)
library(randomForest)

rf_dt_no_lag_forecast=predict(rf_dt_no_lag,newdata=ts_historical_cut_test[, c(1, 3, 4, 5,
 ↪  6)])

ts_rf_dt_no_lag_forecast=ts(rf_dt_no_lag_forecast,start=c(331,1),end=c(332,96),frequency
 ↪  = 96)

autoplot(ts_historical_cut_test[,"power"], series="test
 ↪  data")+autolayer(ts_rf_dt_no_lag_forecast,series="RF temperature & daytime w/o lags")
```

Let's check RMSE

```
rf_dt_no_lag_rmse <- sqrt(mean((rf_dt_no_lag_forecast - ts_historical_cut_test[, 2])^2))
cat("RMSE RF temperature & daytime w/o lags:", rf_dt_no_lag_rmse, "\n")
```

Residuals

```
residuals_rf_dt_no_lag <- ts_historical_cut_test[, "power"] - rf_dt_no_lag_forecast


summary(rf_dt_no_lag)
print(rf_dt_no_lag)
checkresiduals(residuals_rf_dt_no_lag)
tsdisplay(residuals_rf_dt_no_lag)
```

This model demonstrates a quite poor performance. But an interesting observation is that the model is at least following the pattern of the daytime covariates. Let's test a model for which we create for each day 96 numeric labels

3.4.2 RF with temp & 96 labels w/o lags

First let's create and additional feature for train_val and test sets

```
# Pattern from 1..96
intraday_train_val <- rep(1:96, times = 62)
intraday_test <- rep(1:96, times = 2)
```

Let's double check the number of rows in ts_historical_cut_train_val

```
# Get the number of rows directly
num_rows <- nrow(ts_historical_cut_train_val)

# Print the number of rows
print(num_rows)
```

Let's add intraday column to train_val set

```
ts_historical_cut_train_val <- cbind(
  temp    = ts_historical_cut_train_val[, 1],
  power   = ts_historical_cut_train_val[, 2],
  Night  = ts_historical_cut_train_val[, 3],
  Morning  = ts_historical_cut_train_val[, 4],
  Day  = ts_historical_cut_train_val[, 5],
  Evening  = ts_historical_cut_train_val[, 6],
  day_number  = ts_historical_cut_train_val[, 7],
  intraday  = intraday_train_val[1:nrow(ts_historical_cut_train_val)]
)
```

Let's add intraday column to test set

```
ts_historical_cut_test <- cbind(
  temp    = ts_historical_cut_test[, 1],
  power   = ts_historical_cut_test[, 2],
  Night  = ts_historical_cut_test[, 3],
  Morning  = ts_historical_cut_test[, 4],
  Day  = ts_historical_cut_test[, 5],
  Evening  = ts_historical_cut_test[, 6],
  day_number  = ts_historical_cut_test[, 7],
  intraday  = intraday_test[1:nrow(ts_historical_cut_test)]
)
```

Let's fit the model

```
set.seed(777)
rf_id_no_lag=randomForest(y=ts_historical_cut_train_val[,2],x=ts_historical_cut_train_val[,
 ↪  c(1, 8)])
```

Check the graph

```
library(ggplot2)
library(randomForest)

rf_id_no_lag_forecast=predict(rf_id_no_lag,newdata=ts_historical_cut_test[, c(1, 8)])

ts_rf_id_no_lag_forecast=ts(rf_id_no_lag_forecast,start=c(331,1),end=c(332,96),frequency
 ↪  = 96)

autoplot(ts_historical_cut_test[,"power"], series="test
 ↪  data")+autolayer(ts_rf_id_no_lag_forecast,series="RF temperature & intraday w/o
 ↪  lags")
```

Let's check RMSE

```
rf_id_no_lag_rmse <- sqrt(mean((rf_id_no_lag_forecast - ts_historical_cut_test[, 2])^2))
cat("RMSE RF temperature & intraday w/o lags:", rf_id_no_lag_rmse, "\n")
```

Residuals

```
residuals_rf_id_no_lag <- ts_historical_cut_test[, "power"] - rf_id_no_lag_forecast


summary(rf_id_no_lag)
print(rf_id_no_lag)
checkresiduals(residuals_rf_id_no_lag)
tsdisplay(residuals_rf_id_no_lag)
```

Great! By adding a numeric index for each 15-minutes for each day we have managed to achieved a decent result comparable to other models. Now let's check XGBoost with the similar parameters.

3.4.3 XGBoost with temp & 96 labels w/o lags

```
set.seed(777)
library(xgboost)
xg_id_no_lag<- xgboost(data = ts_historical_cut_train_val[, c(1, 8)], label =
 ↪  ts_historical_cut_train_val[,2],
max_depth = 10, eta = .5, nrounds = 100,
nthread = 2, objective = "reg:squarederror", verbose = 0)
```

Check the graph

```
library(ggplot2)
library(xgboost)

xg_id_no_lag_forecast=predict(xg_id_no_lag,newdata=ts_historical_cut_test[, c(1, 8)])

ts_xg_id_no_lag_forecast=ts(xg_id_no_lag_forecast,start=c(331,1),end=c(332,96),frequency
 ↪  = 96)

autoplot(ts_historical_cut_test[,"power"], series="test
 ↪  data")+autolayer(ts_xg_id_no_lag_forecast,series="XGBoost temperature & intraday w/o
 ↪  lags")
```

Let's check RMSE

```
xg_id_no_lag_rmse <- sqrt(mean((xg_id_no_lag_forecast - ts_historical_cut_test[, 2])^2))
cat("RMSE XGBoost temperature & intraday w/o lags:", xg_id_no_lag_rmse, "\n")
```

XGBoost has demonstrated a worse result than RF. Let's add lags.

3.4.4 RF with temp & 96 labels w/ lags

First let's create new columns with lagged values for the train_val dataset

```
ts_lagged_train_val <- cbind(
  temp    = ts_historical_cut_train_val[, 1],
  power   = ts_historical_cut_train_val[, 2],
  Night  = ts_historical_cut_train_val[, 3],
  Morning  = ts_historical_cut_train_val[, 4],
  Day  = ts_historical_cut_train_val[, 5],
  Evening  = ts_historical_cut_train_val[, 6],
  day_number  = ts_historical_cut_train_val[, 7],
  intraday  = ts_historical_cut_train_val[, 8],
  x1 = c(rep(NA, 1), head(ts_historical_cut_train_val[, "power"], -1)),
  x2 = c(rep(NA, 2), head(ts_historical_cut_train_val[, "power"], -2)),
  x3 = c(rep(NA, 3), head(ts_historical_cut_train_val[, "power"], -3)),
  x4 = c(rep(NA, 4), head(ts_historical_cut_train_val[, "power"], -4)),
  x5 = c(rep(NA, 5), head(ts_historical_cut_train_val[, "power"], -5)),
  x6 = c(rep(NA, 6), head(ts_historical_cut_train_val[, "power"], -6)),
  x7 = c(rep(NA, 7), head(ts_historical_cut_train_val[, "power"], -7))
)
```

Remove the first 96 rows that contain NA in lag columns:

```
ts_lagged_train_val <- ts_lagged_train_val[-c(1:96), ]
```

Let's fit the model

```
library(randomForest)

rf_id_lagged <- randomForest(
  y = ts_lagged_train_val[, 2],
  x = ts_lagged_train_val[, c(1,8:15)]
)
```

Initialize the Lag Vector

```
n_train <- nrow(ts_lagged_train_val)
# The last 7 real demands from training (in descending order):
x <- c(
  ts_lagged_train_val[n_train,     "power"],  # 7th-latest
  ts_lagged_train_val[n_train - 1, "power"],  # 6th-latest
  ts_lagged_train_val[n_train - 2, "power"],
  ts_lagged_train_val[n_train - 3, "power"],
  ts_lagged_train_val[n_train - 4, "power"],
  ts_lagged_train_val[n_train - 5, "power"],
  ts_lagged_train_val[n_train - 6, "power"]   # 1st-latest
)
```

Loop Over Each Test Row

```
n_test <- nrow(ts_historical_cut_test)
prev   <- numeric(n_test)  # store predictions here

for (i in seq_len(n_test)) {
  test_row <- cbind(
    temp     = ts_historical_cut_test[i, "temp"],
    power    = ts_historical_cut_test[i, "power"],
    Night    = ts_historical_cut_test[i, "Night"],
    Morning  = ts_historical_cut_test[i, "Morning"],
    Day      = ts_historical_cut_test[i, "Day"],
    Evening  = ts_historical_cut_test[i, "Evening"],
    day_number = ts_historical_cut_test[i, "day_number"],
    intraday = ts_historical_cut_test[i, "intraday"],
    x1 = x[1],
    x2 = x[2],
    x3 = x[3],
    x4 = x[4],
    x5 = x[5],
    x6 = x[6],
    x7 = x[7]
  )

  # 2) Predict using the same columns that were in 'x=...' for training
```

```
  y_hat <- predict(
    rf_id_lagged,
    newdata = test_row[, c(1,8:15)]
  )

  # 3) Save the forecast
  prev[i] <- y_hat

  # 4) SHIFT the x vector to incorporate the new forecast as the "most recent"
  # The sample does: x=c(y, x[1:6])
  x <- c(y_hat, x[1:6])
}

# 'prev' now has 1-step-ahead forecasts for each test row
```

Wrap to ts and build a plot

```
ts_preds <- ts(
  prev,
  start = c(331, 1),
  frequency = 96
)

library(ggplot2)
autoplot(ts_historical_cut_test[,"power"], series="Test Data") +
  autolayer(ts_preds, series="RF Lagged Forecast")
```

Compute RMSE vs. actual power in test set

```
rmse_iterative <- sqrt(mean((prev - ts_historical_cut_test[, "power"])^2))

cat("RMSE (iterative lagged model):", rmse_iterative, "\n")
```

Great! By adding lags we have managed to improve the model performance

4: Forecast

4.1 Comparison of the models

Let's plot the best models

```
library(ggplot2)
library(forecast)

autoplot(ts_historical_cut_test[,"power"], series = "Test Data") +
  autolayer(hw_ft_forecast$mean, series = "Additive HW Finetuned") +
  autolayer(arima_manual_xreg_forecast$mean, series = "Manual ARIMA w/ Temp") +
  autolayer(nnar_temp_dt_forecast$mean, series = "NNAR w/ Temp & Daytime") +
  autolayer(ts_preds, series = "RF w/ Temp & Intraday Lagged") +
```

```
  xlab("Time") +
  ylab("Power") +
  ggtitle("Forecast Comparison of Multiple Models") +
  guides(colour = guide_legend(title = "Series"))
```

Let's compare RMSEs of these models

```
# Calculate each model's RMSE
hw_ft_rmse <- sqrt(mean((hw_ft_forecast$mean - ts_historical_cut_test[,"power"]) ^ 2))
arima_manual_xreg_rmse <- sqrt(mean((arima_manual_xreg_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))
nnar_temp_dt_rmse <- sqrt(mean((nnar_temp_dt_forecast$mean -
 ↪  ts_historical_cut_test[,"power"]) ^ 2))
rmse_iterative <- sqrt(mean((prev - ts_historical_cut_test[, "power"])^2))

# Combine into a data frame
rmse_values <- data.frame(
  Model = c("HW Finetuned",
            "Manual ARIMA w/ Temp",
            "NNAR w/ Temp & Daytime",
            "RF w/ Temp & Intraday Lagged"),
  RMSE = c(hw_ft_rmse,
           arima_manual_xreg_rmse,
           nnar_temp_dt_rmse,
           rmse_iterative)
)

# Order by RMSE (ascending)
rmse_values <- rmse_values[order(rmse_values$RMSE), ]

# Print the table
print(rmse_values)
```

ARIMA with temperature as covariate is the best performing model. Despite all our efforts we haven't managed to achieve similar results with all other approaches. But interestingly neural network with covariates represented by temperature and one-hot encoded daytime frames (night, morning, day, evening) has performed quite well. Let's make a forecast with our best model.

4.2 Forecast for 30/11

Let's retrain our best model on the full dataset to capture the last 2 days we were using for test.

```
library(forecast)
final_model <- Arima(ts_historical_cut[,"power"],
                     order = c(0,1,9),
                     seasonal = list(order = c(1,1,1)),
                     xreg = ts_historical_cut[,"temp"])
```

Convert df_forecast to mts

```
ts_forecast <- ts(df_forecast[,"temp"],
                          frequency = 96)
```

Make forecast

```
library(forecast)
final_forecast <- forecast(final_model, h = 1 * 96,
                                    xreg = ts_forecast[,"temp"])

autoplot(final_forecast$mean, series = "Manual Arima w/ Temp")
```

Extract the 96 forecasted values as a data frame

```
library(openxlsx)
forecast_values <- as.data.frame(final_forecast$mean)

# Save these values to Excel
write.xlsx(
  x        = forecast_values,
  file     = "NikolaiLen.xlsx",
  colNames = FALSE,
  rowNames = FALSE
)
```