EEE483 – Introduction to Energy Analytics

Tutorial: Day-Ahead Electric Load Forecasting with Multiple Linear Regression in Julia

Prepared by Assoc. Prof. Kasım ZOR

Duration: 2 lecture hours | **Language:** Julia | **Data:** real hourly national electricity load of Panama, 2015–2020

Why this tutorial?

Every hour of every day, a power system must generate exactly as much electricity as its customers consume. Because large-scale storage is still scarce, system operators, generators and retailers all need to know *tomorrow's* demand *today*. This is the **day-ahead load forecasting** problem, and it sits at the heart of day-ahead electricity markets, unit commitment and economic dispatch.

In this tutorial we solve that problem from start to finish on a real data set, using nothing more exotic than **multiple linear regression (MLR)**. Along the way we practise the Julia skills covered so far in the course (functions, arrays, `Statistics.jl`, `DataFrames.jl`, `CSV.jl` and `Plots.jl`) and, most importantly, we follow the stages of a data analytics pipeline in their proper order.

Learning outcomes

By the end of the two hours you should be able to:

1. write small Julia functions that carry out everyday energy calculations (energy, load factor, tariff periods);
2. import, inspect, summarise and visualise an hourly electricity load data set;
3. clean the data and engineer calendar, weather and lagged-load features **without leaking future information**;
4. formulate, fit and interpret MLR models, both "by hand" with linear algebra and with `GLM.jl`;
5. evaluate forecasts honestly with MAE, RMSE and MAPE against a naïve benchmark on unseen data;

6. produce a 24-hour forecast for a chosen day and discuss the limits of the model.

The pipeline we shall follow

Stage	Pipeline step	Section	Guide time
–	Julia warm-up with energy calculations	1	10 min
1	Problem definition	2	5 min
2	Data acquisition	3	5 min
3	Data understanding (exploratory data analysis)	4	20 min
4	Data preparation (cleaning and feature engineering)	5	10 min
	— end of the first lecture hour —		
5	Data partitioning	6	5 min
6	Modelling	7	20 min
7	Evaluation	8	12 min
8	Deployment	9	8 min
9	Knowledge, limitations and exercises	10	5 min

> **How to use this notebook.** Run the cells from top to bottom; later cells depend on earlier ones. Boxes marked **Checkpoint** are short exam-style questions. Try them before opening the solution.

0. Setting up

The tutorial needs five registered packages. If they are not yet installed on your computer, remove the `#` from the first two lines of the next cell and run it once (installation may take several minutes). `Dates`, `Downloads`, `Statistics` and `Printf` are part of Julia's standard library and need no installation.

Package	Role in this tutorial
`CSV.jl`	loading CSV data
`DataFrames.jl`	storing and manipulating tabular data
`Statistics.jl`, `StatsBase.jl`	descriptive statistics, autocorrelation
`GLM.jl`	fitting linear models with a formula

	interface
`Plots.jl`	creating visualisations
`Dates`	working with timestamps

In [1]:
```
# import Pkg
# Pkg.add(["CSV", "DataFrames", "GLM", "Plots", "StatsBase"])

using CSV, DataFrames, Dates, Downloads, Statistics, StatsBase, GLM

# One consistent look for every figure in the notebook
default(size = (900, 360), fmt = :png, dpi = 110, linewidth = 1.5,
        left_margin = 5Plots.mm, bottom_margin = 5Plots.mm, legend =

println("Julia version: ", VERSION)
```

```
Julia version: 1.13.3
```

FIRST LECTURE HOUR

1. Julia warm-up with energy calculations

Before touching the data, let us refresh the Julia building blocks with calculations that an energy analyst performs daily.

1.1 Energy and power

Power is the *rate* at which energy is converted, so for a constant power P sustained over a time t

$$P = \frac{E}{t} \quad \Longleftrightarrow \quad E = P\,t.$$

A power system that carries an average load of 1 200 MW for a whole day therefore delivers $1\,200 \times 24 = 28\,800$ MWh $= 28.8$ GWh. Hourly load data are usually published as *energy per hour* (MWh/h), which is numerically equal to the *average power* during that hour (MW).

In [2]:
```
# One-line function definitions
energy(P, t)     = P * t            # MW × h  →  MWh
mwh_to_toe(E)    = E / 11.63        # 1 toe (tonne of oil equivalen
mwh_to_gwh(E)    = E / 1000

E_day = energy(1200, 24)
println("Energy delivered in one day : $(E_day) MWh = $(mwh_to_gwh(
println("The same energy in toe      : $(round(mwh_to_toe(E_day), d
```

```
Energy delivered in one day : 28800 MWh = 28.8 GWh
The same energy in toe      : 2476.4 toe
```

1.2 Functions with decisions: tariff periods

Türkiye's three-rate (multi-time) retail tariff splits the day into **day** (06:00–17:00), **peak** (17:00–22:00) and **night** (22:00–06:00) periods. The function below takes an hour of the day (0–23) and returns the period, using the same `if … elseif … else … end` structure that you used for the roots of a quadratic equation.

In [3]:
```
function tariffperiod(h)
    if h < 0 || h > 23
        error("The hour must lie between 0 and 23.")
    elseif 6 <= h < 17
        return "day"
    elseif 17 <= h < 22
        return "peak"
    else
        return "night"
    end
end

for h in (3, 10, 19, 23)
    println("Hour $h belongs to the $(tariffperiod(h)) period.")
end
```

```
Hour 3 belongs to the night period.
Hour 10 belongs to the day period.
Hour 19 belongs to the peak period.
Hour 23 belongs to the night period.
```

1.3 Arrays and descriptive statistics: a miniature load profile

We now create a zero array, fill it element by element, and compute a few statistics. The six values are four-hourly average loads (MW) of an imaginary small system.

Two indicators that appear throughout the electricity industry are

$$\text{load factor} = \frac{\text{average load}}{\text{peak load}}, \qquad \text{energy} = \sum_i P_i \, \Delta t.$$

A load factor close to 1 means a flat, well-utilised system; a low load factor means that expensive capacity sits idle most of the time.

In [4]:
```
ADL = zeros(Float64, 6)          # "average daily load" array with
ADL[1] = 820                     # 00:00–04:00
ADL[2] = 790                     # 04:00–08:00
ADL[3] = 1010                    # 08:00–12:00
```

```
ADL[4] = 1090                       # 12:00–16:00
ADL[5] = 1180                       # 16:00–20:00
ADL[6] = 960                        # 20:00–24:00

avg  = mean(ADL)                    # from Statistics.jl
peak = maximum(ADL)

loadfactor(P) = mean(P) / maximum(P)

println("Average load : ", avg, " MW")
println("Peak load    : ", peak, " MW in block ", argmax(ADL))
println("Std. dev.    : ", round(std(ADL), digits = 1), " MW")
println("Load factor  : ", round(loadfactor(ADL), digits = 3))
println("Daily energy : ", sum(energy.(ADL, 4)), " MWh")      # the
```

```
Average load : 975.0 MW
Peak load    : 1180.0 MW in block 5
Std. dev.    : 151.6 MW
Load factor  : 0.826
Daily energy : 23400.0 MWh
```

> **Checkpoint 1.** (a) Which of the following are units of *energy* and which are units of *power*: GWh, MW, toe, hp, Btu, toe/year? (b) Without running any code, what does `loadfactor([500, 1000])` return? (c) What does the dot in `energy.(ADL, 4)` do?
>
> ▶ Solution

2. Pipeline stage 1 – Problem definition

A data analytics project starts with a question, not with data.

Business context. In a day-ahead electricity market (such as the one operated by EPİAŞ in Türkiye) participants submit hourly bids for the *following* day, and a market clearing price is determined for each of its 24 hours. A retailer that under-forecasts must buy the shortfall in the more expensive balancing market; a system operator that over-forecasts commits generating units that are not needed. Even a 1 % improvement in accuracy is worth a great deal of money on a national scale.

Analytical task. This is a *supervised regression* problem on *time-series* data:

> On the morning of day $D-1$, predict the average load (MW) in each of the 24 hours of day D.

What may we use? This question decides everything that follows. On the morning of day $D-1$ we know:

Information	Available on the morning of $D-1$?	Consequence for the model
Calendar of day D (hour, weekday, month, public holiday)	Yes, known years ahead	free to use
Complete load history up to the end of day $D-2$	Yes	load lagged by **48 h or more** is safe
Load of day $D-1$	Only partly (the day is not over)	load lagged by 24 h is **not** safe for all hours
Temperature of day D	Only as a *forecast*	see the note below

> **An honest note on temperature.** In real operation the model is fed with a weather *forecast*. Historical forecasts are rarely archived, so, like most of the academic literature, we use the *observed* temperature instead. Such "ex-post" forecasts are slightly optimistic. Keep this in mind when you read the final error figures.

Success criterion. A model is only useful if it beats what one could do with no model at all. Our benchmark is the **seasonal naïve** forecast: "the load in hour h of day D will equal the load in the same hour one week earlier". We shall judge success by the **mean absolute percentage error (MAPE)** on a full year of data that the model has never seen.

3. Pipeline stage 2 – Data acquisition

3.1 The data set

We use the open data set *Short-term electricity load forecasting (Panama case study)* published by Ernesto Aguilar Madrid on Mendeley Data (DOI: 10.17632/byx7sztj59.1) and described in Aguilar Madrid and António (2021). It merges three real sources on an hourly time index:

- the post-dispatch national electricity load reported by Panama's grid operator (CND);
- weather variables for three cities (Tocumen/Panama City, Santiago and David) from NASA reanalysis data;
- public-holiday and school-period calendars.

Column	Meaning	Unit
`datetime`	timestamp of the record	local time, hourly
`nat_demand`	national electricity load	MWh per hour (≡ average MW)
`T2M_xxx`	air temperature at 2 m in city `xxx`	°C

`QV2M_xxx`	specific humidity at 2 m	kg/kg
`TQL_xxx`	total column liquid water (a proxy for cloud and rain)	kg/m²
`W2M_xxx`	wind speed at 2 m	m/s
`Holiday_ID`	identifier of the public holiday (0 = none)	–
`holiday`	1 if public holiday, 0 otherwise	binary
`school`	1 if schools are in session, 0 otherwise	binary

The city suffixes are `toc` (Tocumen, Panama City), `san` (Santiago) and `dav` (David).

3.2 Downloading and importing

The file `continuous dataset.csv` can be downloaded manually from the Mendeley page above. For convenience the next cell fetches a copy from a public GitHub mirror, but only if the file is not already in your working directory. We then import the CSV file and convert it into a data frame, asking `CSV.jl` to read the timestamp as plain text so that *we* control how it is interpreted in Section 4.

In [5]:
```
DATA_URL  = "https://raw.githubusercontent.com/saul-chirinos/Electr
DATA_FILE = "continuous_dataset.csv"

if !isfile(DATA_FILE)
    Downloads.download(DATA_URL, DATA_FILE)
end

raw = DataFrame(CSV.File(DATA_FILE; types = Dict(:datetime => Strin

println("Column names      : ", names(raw))
println("Number of rows    : ", nrow(raw))
println("Number of columns : ", ncol(raw))
println("size(raw)         : ", size(raw))
```

```
Column names      : ["datetime", "nat_demand", "T2M_toc", "QV2M_to
c", "TQL_toc", "W2M_toc", "T2M_san", "QV2M_san", "TQL_san", "W2M_sa
n", "T2M_dav", "QV2M_dav", "TQL_dav", "W2M_dav", "Holiday_ID", "holi
day", "school"]
Number of rows    : 48048
Number of columns : 17
size(raw)         : (48048, 17)
```

In [6]:
```
first(raw, 5)
```

Out[6]: 5×17 DataFrame

Row	datetime	nat_demand	T2M_toc	QV2M_toc	TQL_toc	W2M_toc
	String	Float64	Float64	Float64	Float64	Float64
1	2015-01-03 01:00:00	970.345	25.8653	0.0185764	0.0161743	21.8505
2	2015-01-03 02:00:00	912.176	25.8993	0.0186533	0.0164185	22.1669
3	2015-01-03 03:00:00	900.269	25.9373	0.0187679	0.01548	22.4549
4	2015-01-03 04:00:00	889.954	25.9575	0.0188901	0.0162735	22.1105
5	2015-01-03 05:00:00	893.687	25.9738	0.0189812	0.0172806	21.1861

In [7]:
```
last(raw, 3)
```

Out[7]: 3×17 DataFrame

Row	datetime	nat_demand	T2M_toc	QV2M_toc	TQL_toc	W2M_toc
	String	Float64	Float64	Float64	Float64	Float64
1	2020-06-26 22:00:00	1081.57	26.9712	0.0204484	0.057251	10.2625
2	2020-06-26 23:00:00	1041.62	26.8675	0.0204638	0.0641785	10.3266
3	2020-06-27 00:00:00	1013.57	26.7503	0.020441	0.0639648	10.1983

4. Pipeline stage 3 – Data understanding

Never model data that you have not looked at. In this stage we check the structure and quality of the data, summarise it numerically and then visualise it. Every figure should answer a question that matters for the model.

4.1 A first statistical summary

`describe` is the quickest way to obtain the mean, minimum, median and maximum of every column, together with the number of missing values and

the element type.

In [8]:
```
describe(raw)
```

Out[8]: 17×7 DataFrame

Row	variable	mean	min	median	max	nmissin
	Symbol	Union...	Any	Union...	Any	Int64
1	datetime		2015-01-03 01:00:00		2020-06-27 00:00:00	
2	nat_demand	1182.87	85.1925	1168.43	1754.88	
3	T2M_toc	27.3991	22.9535	27.1181	35.0396	
4	QV2M_toc	0.0183126	0.0120539	0.0185897	0.0226904	
5	TQL_toc	0.0799794	0.0	0.0652008	0.52124	
6	W2M_toc	13.391	0.00897889	12.1821	39.2297	
7	T2M_san	26.921	19.7652	26.1674	39.0634	
8	QV2M_san	0.017844	0.0102468	0.0183507	0.0221648	
9	TQL_san	0.106265	8.89599e-6	0.085968	0.484985	
10	W2M_san	7.04667	0.0603937	5.99276	24.4839	
11	T2M_dav	24.7195	19.9337	24.0017	34.2162	
12	QV2M_dav	0.0168666	0.00965529	0.0171668	0.0210662	
13	TQL_dav	0.144681	3.23057e-5	0.130188	0.477783	
14	W2M_dav	3.56759	0.0154974	3.41399	10.2889	
15	Holiday_ID	0.704795	0	0.0	22	
16	holiday	0.0629371	0	0.0	1	
17	school	0.727793	0	1.0	1	

What do we learn?

- There are no missing values (`nmissing = 0` everywhere).
- The load averages roughly 1 180 MW, but the **minimum is only about 85 MW**. A national system does not lose 93 % of its demand under normal conditions, so this value is suspicious. We note it and return to it in Section 5.
- Panama is tropical: the temperature at Tocumen stays between about 23 °C and 35 °C. There is no heating season, so we expect load to *rise* with temperature (air conditioning) throughout the year.
- `datetime` is still text (`String`). We must convert it before doing

anything with time.

4.2 Parsing the timestamp and tidying the table

Two details deserve care.

1. **Type conversion.** `DateTime(text, format)` turns text into a proper timestamp.
2. **Calendar alignment.** The file starts at `2015-01-03 01:00:00` and ends at `2020-06-27 00:00:00`, so the first calendar date holds 23 records and the last holds only one. The `holiday` and `school` flags supplied with the data are aligned to the timestamps exactly as they are written, so we keep the timestamps unchanged and simply remember that those two dates are incomplete (we verify this in Section 4.3).

We also keep only the columns we need and give them shorter names. Tocumen (Panama City) is the main load centre, so its temperature becomes our principal weather variable `temp`.

In [9]:
```
fmt = dateformat"yyyy-mm-dd HH:MM:SS"

df = DataFrame(
    ts       = DateTime.(raw.datetime, fmt),              # text  →
    load     = raw.nat_demand,                            # MW
    temp     = raw.T2M_toc,                               # °C, Pan
    temp_san = raw.T2M_san,
    temp_dav = raw.T2M_dav,
    humidity = raw.QV2M_toc,
    cloud    = raw.TQL_toc,
    wind     = raw.W2M_toc,
    holiday  = raw.holiday,
    school   = raw.school,
)

println("First record : ", first(df.ts))
println("Last record  : ", last(df.ts))
println("Element type : ", eltype(df.ts))
```

```
First record : 2015-01-03T01:00:00
Last record  : 2020-06-27T00:00:00
Element type : DateTime
```

4.3 Integrity checks

A time series for forecasting should have exactly one record per hour, with no gaps and no duplicates. Let us verify this rather than assume it.

In [10]:
```
n_missing    = sum(ismissing.(Matrix(df)))
n_duplicates = nrow(df) - length(unique(df.ts))
steps        = diff(df.ts)                          # time difference
```

```
n_days       = nrow(df) / 24

println("Missing values           : ", n_missing)
println("Duplicated timestamps    : ", n_duplicates)
println("Distinct time steps      : ", unique(steps))
println("Chronologically sorted   : ", issorted(df.ts))
println("Rows divided by 24       : ", n_days)

# How many records does each calendar date hold? Show the dates tha
perday = combine(groupby(DataFrame(date = Date.(df.ts)), :date), nr
perday[perday.records .!= 24, :]
```

```
Missing values           : 0
Duplicated timestamps    : 0
Distinct time steps      : [Millisecond(3600000)]
Chronologically sorted   : true
Rows divided by 24       : 2002.0
```

Out[10]: 2×2 DataFrame

Row	date	records
	Date	Int64
1	2015-01-03	23
2	2020-06-27	1

The series is complete and perfectly regular: every step is 3 600 000 ms (one hour), and only the first and the last calendar dates are incomplete, as expected. This matters in Section 5, because it allows us to build lagged features simply by shifting rows.

4.4 Calendar variables for exploration

The `Dates` standard library extracts calendar information from a timestamp. We add these columns now because we need them for the exploratory analysis; they will also become model features later.

In [11]:
```
df.date  = Date.(df.ts)
df.year  = year.(df.ts)
df.month = month.(df.ts)
df.dow   = dayofweek.(df.ts)        # 1 = Monday, …, 7 = Sunday
df.hour  = hour.(df.ts)             # 0–23

first(df[:, [:ts, :load, :temp, :holiday, :year, :month, :dow, :hou
```

Out[11]: 5×8 DataFrame

Row	ts	load	temp	holiday	year	month	dow	hour
	DateTime	Float64	Float64	Int64	Int64	Int64	Int64	Int64
1	2015-01-03T01:00:00	970.345	25.8653	0	2015	1	6	1
2	2015-01-03T02:00:00	912.176	25.8993	0	2015	1	6	2
3	2015-01-03T03:00:00	900.269	25.9373	0	2015	1	6	3
4	2015-01-03T04:00:00	889.954	25.9575	0	2015	1	6	4
5	2015-01-03T05:00:00	893.687	25.9738	0	2015	1	6	5

4.5 Statistics of the load

We begin with the distribution of the hourly load as a whole.

In [12]:
```
L = df.load

println("Number of hours      : ", length(L))
println("Mean                 : ", round(mean(L), digits = 1), " MW"
println("Median               : ", round(median(L), digits = 1), " M
println("Standard deviation   : ", round(std(L), digits = 1), " MW"
println("Coeff. of variation  : ", round(100 * std(L) / mean(L), di
println("Minimum              : ", round(minimum(L), digits = 1), "
println("Maximum (peak)       : ", round(maximum(L), digits = 1), "
println("5 %, 25 %, 75 %, 95 % quantiles : ", round.(quantile(L, [0
println("Skewness             : ", round(skewness(L), digits = 3))
```

```
Number of hours      : 48048
Mean                 : 1182.9 MW
Median               : 1168.4 MW
Standard deviation   : 192.1 MW
Coeff. of variation  : 16.2 %
Minimum              : 85.2 MW
Maximum (peak)       : 1754.9 MW at 2020-01-17T14:00:00
5 %, 25 %, 75 %, 95 % quantiles : [908.2, 1020.1, 1327.6, 1511.4]
Skewness             : 0.248
```

The *split–apply–combine* pattern (`groupby` then `combine`) gives the same statistics for each year. From them we derive the annual energy and the load factor defined in Section 1.3.

In [13]:
```
yearly = combine(groupby(df, :year),
    nrow  => :hours,
    :load => mean    => :mean_MW,
    :load => maximum => :peak_MW,
    :load => (x -> sum(x) / 1000) => :energy_GWh,
```

```
)
yearly.load_factor = yearly.mean_MW ./ yearly.peak_MW

yearly
```

Out[13]: 6×6 DataFrame

Row	year	hours	mean_MW	peak_MW	energy_GWh	load_factor
	Int64	Int64	Float64	Float64	Float64	Float64
1	2015	8711	1131.53	1645.48	9856.78	0.687662
2	2016	8784	1157.19	1615.25	10164.7	0.716415
3	2017	8760	1182.32	1635.92	10357.1	0.722724
4	2018	8760	1205.68	1635.54	10561.7	0.737173
5	2019	8760	1243.58	1721.52	10893.8	0.722376
6	2020	4273	1170.23	1754.88	5000.4	0.666843

Average demand grew by roughly 2–3 % per year from 2015 to 2019, so the model will need a **trend** term. Note that 2015 and 2020 are incomplete years (compare the `hours` column), so their energy totals are not comparable with the others, and that the mean load of 2020 *fell*. We shall see why shortly.

As in the mid-term examination, a data frame can be sorted by any column. Which year had the highest peak? (The answer may surprise you: the record was set in January 2020, shortly before the pandemic.)

In [14]:
```
sort(yearly, :peak_MW, rev = true)
```

Out[14]: 6×6 DataFrame

Row	year	hours	mean_MW	peak_MW	energy_GWh	load_factor
	Int64	Int64	Float64	Float64	Float64	Float64
1	2020	4273	1170.23	1754.88	5000.4	0.666843
2	2019	8760	1243.58	1721.52	10893.8	0.722376
3	2015	8711	1131.53	1645.48	9856.78	0.687662
4	2017	8760	1182.32	1635.92	10357.1	0.722724
5	2018	8760	1205.68	1635.54	10561.7	0.737173
6	2016	8784	1157.19	1615.25	10164.7	0.716415

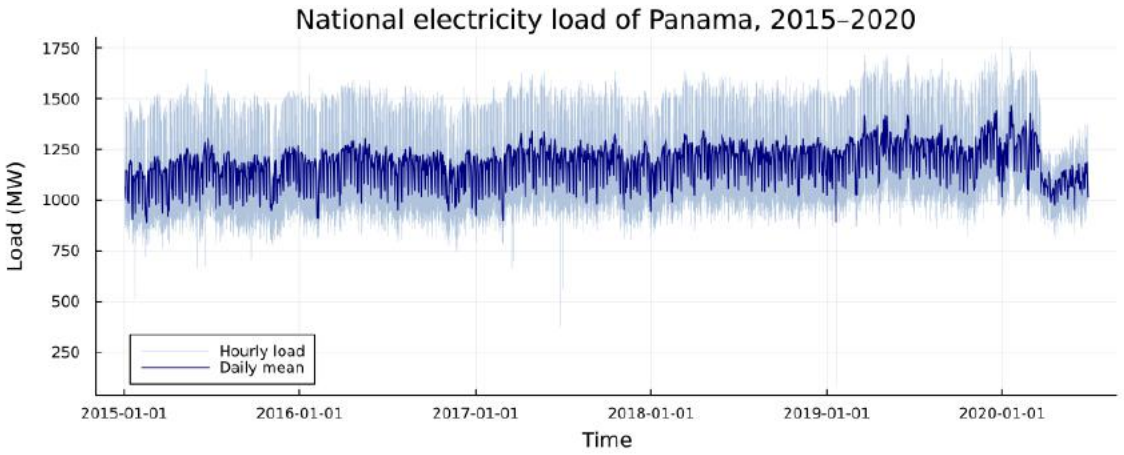
4.6 Visualisation 1 – the whole series

Question: are there a trend, seasonality or unusual episodes?

```
```

In [15]:
```
daily = combine(groupby(df, :date), :load => mean => :load)

plot(df.ts, df.load, color = :lightsteelblue, linewidth = 0.4, labe
     xlabel = "Time", ylabel = "Load (MW)", title = "National elect
     legend = :bottomleft)
plot!(DateTime.(daily.date), daily.load, color = :navy, linewidth =
```

Out[15]:



Three features stand out:

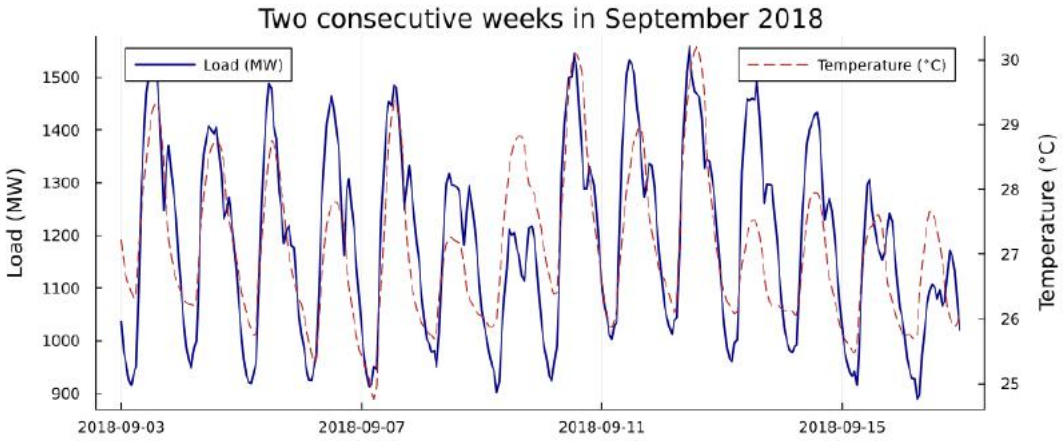
1. a gentle **upward trend** until early 2020;
2. a handful of sharp **downward spikes**, the deepest in January 2019;
3. an abrupt **drop in spring 2020**, when COVID-19 restrictions began. The process that generated the data changed, which is a textbook example of *concept drift*. We shall keep 2020 out of model building and return to it in Section 9.

4.7 Visualisation 2 – two ordinary weeks

Question: what does the load look like at the time scale at which we forecast?

In [16]:
```
week = df[(df.date .>= Date(2018, 9, 3)) .& (df.date .<= Date(2018,

plot(week.ts, week.load, color = :navy, label = "Load (MW)", ylabel
     title = "Two consecutive weeks in September 2018", legend = :t
plot!(twinx(), week.ts, week.temp, color = :firebrick, linestyle =
      label = "Temperature (°C)", ylabel = "Temperature (°C)", lege
plot!(right_margin = 18Plots.mm)
```

Out[16]:



The load repeats itself every **24 hours** (daily cycle) and every **168 hours** (weekly cycle: five similar working days, a lower Saturday and a still lower Sunday). Temperature has its own daily cycle and visibly moves with the load.

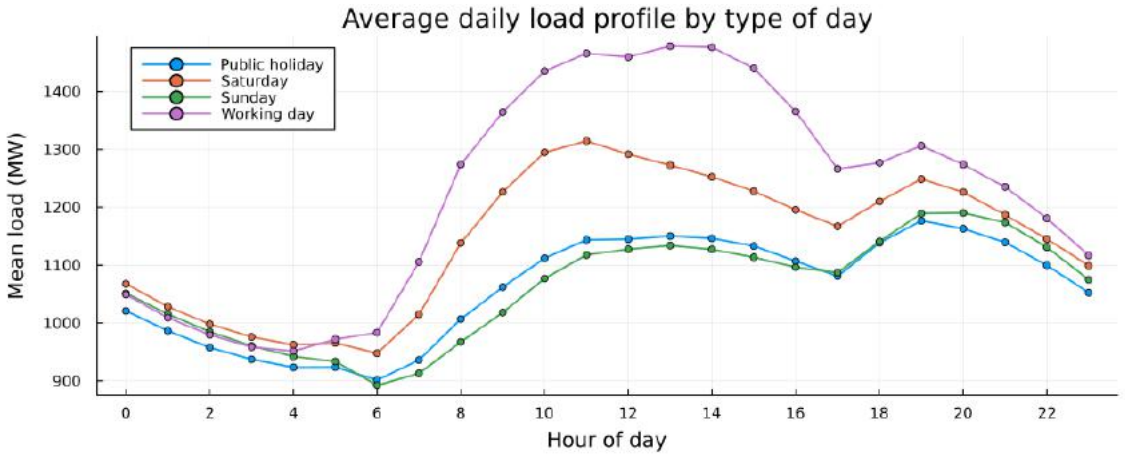
4.8 Visualisation 3 – average daily profile by type of day

Question: does the shape of the day depend on the kind of day?

In [17]:

```
function daytype(dow, holiday)
    if holiday == 1
        return "Public holiday"
    elseif dow == 6
        return "Saturday"
    elseif dow == 7
        return "Sunday"
    else
        return "Working day"
    end
end

df.daytype = daytype.(df.dow, df.holiday)

profile = combine(groupby(df, [:daytype, :hour]), :load => mean =>
sort!(profile, [:daytype, :hour])

plot(profile.hour, profile.load, group = profile.daytype, marker =
     xlabel = "Hour of day", ylabel = "Mean load (MW)", xticks = 0:
     title = "Average daily load profile by type of day", legend =
```

Out[17]:



Working days show a broad plateau from late morning to mid-afternoon (offices, commerce and air conditioning) followed by an evening shoulder. Saturdays, Sundays and public holidays lose most of the daytime plateau but converge with working days in the evening. Two modelling lessons follow:

- the effect of the **hour depends on the day of the week** (the curves are not parallel), so we shall need an hour × weekday *interaction*;
- **public holidays** behave like a different kind of day and deserve their own variable.

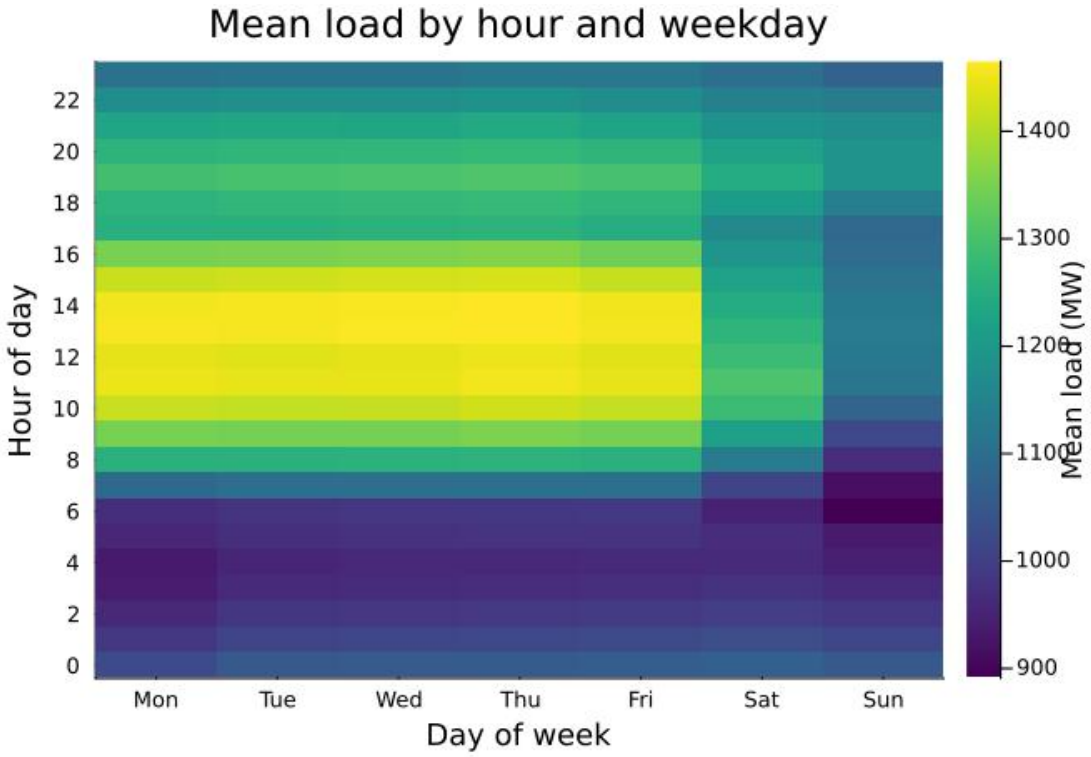
4.9 Visualisation 4 – hour × weekday heat map

The same information for all seven days at once. `dayabbr` returns the abbreviated name of a weekday.

In [18]:

```
H = [mean(df.load[(df.hour .== h) .& (df.dow .== d)]) for h in 0:23

heatmap(dayabbr.(1:7), 0:23, H, color = :viridis, size = (620, 430)
        xlabel = "Day of week", ylabel = "Hour of day", yticks = 0:
        colorbar_title = "Mean load (MW)", title = "Mean load by ho
```

Out[18]:



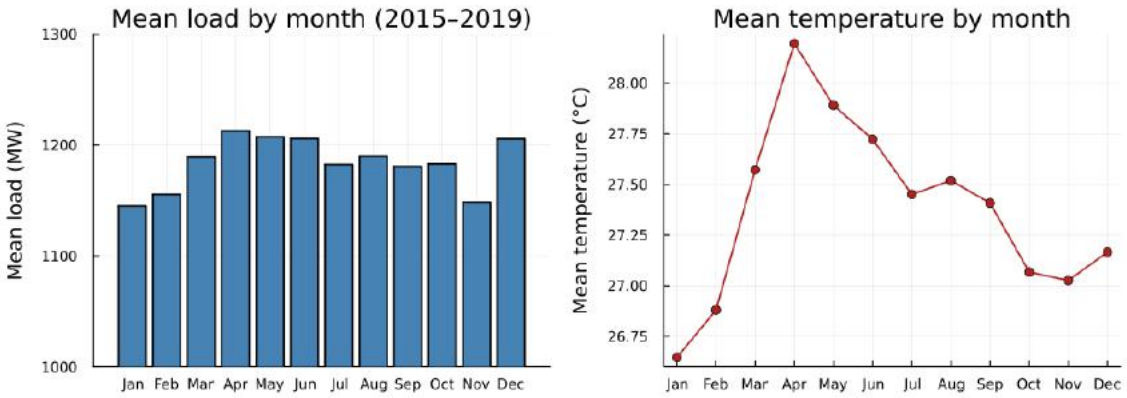
4.10 Visualisation 5 – seasonality within the year

Question: does the month matter?

In [19]:

```
monthly = combine(groupby(df[df.year .<= 2019, :], :month),
                  :load => mean => :load, :temp => mean => :temp)
sort!(monthly, :month)

p1 = bar(monthabbr.(monthly.month), monthly.load, color = :steelblu
         ylabel = "Mean load (MW)", title = "Mean load by month (20
p2 = plot(monthabbr.(monthly.month), monthly.temp, color = :firebri
          ylabel = "Mean temperature (°C)", title = "Mean temperatu
plot(p1, p2, layout = (1, 2), size = (950, 340))
```

Out[19]:

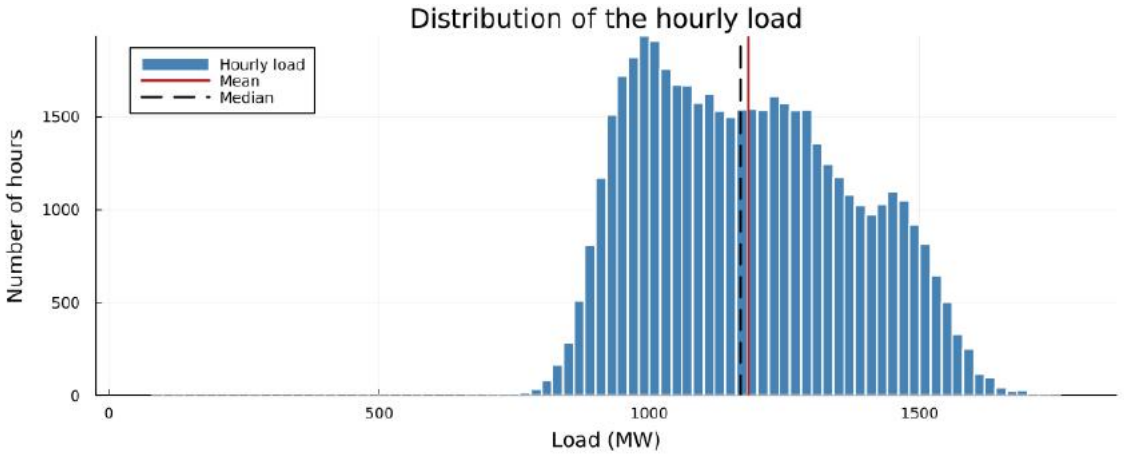


The annual cycle is mild compared with countries that have cold winters, but it exists: mean load is highest from April to June, just after the hottest weeks of the year, and lowest in January, February and November. A `month` variable will capture it.

4.11 Visualisation 6 – distribution of the load

In [20]:
```
histogram(df.load, bins = 80, color = :steelblue, linecolor = :whit
          label = "Hourly load", xlabel = "Load (MW)", ylabel = "Nu
          title = "Distribution of the hourly load")
vline!([mean(df.load)], color = :firebrick, linewidth = 2, label = '
vline!([median(df.load)], color = :black, linestyle = :dash, linewi
```

Out[20]:



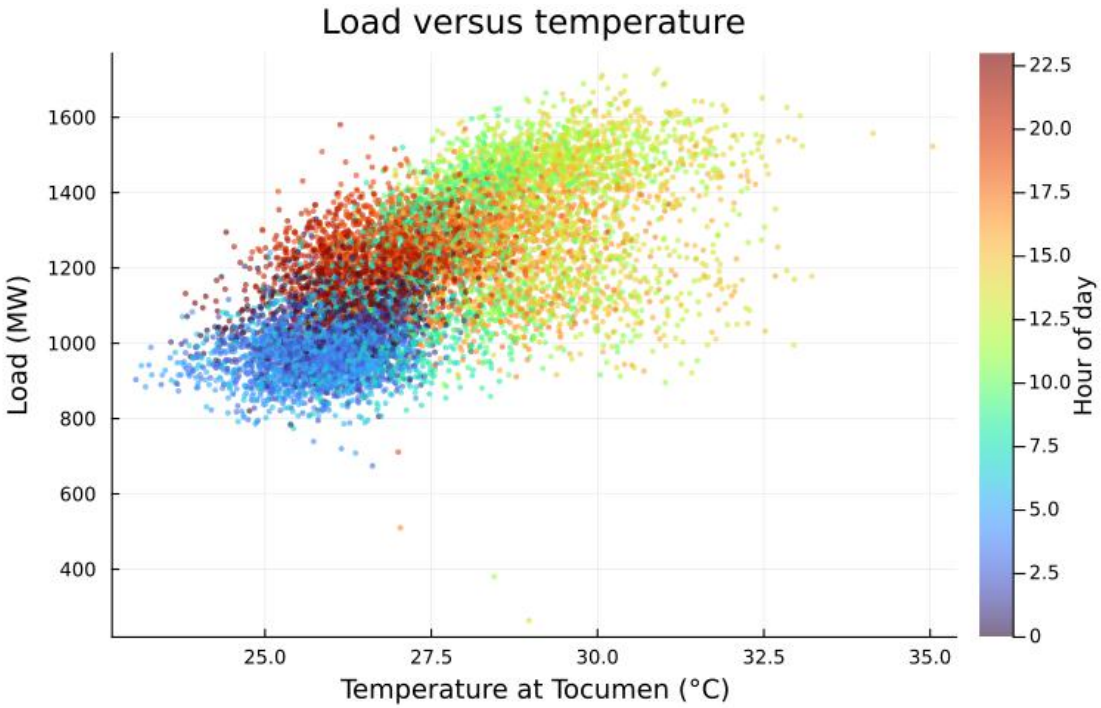
The distribution is not a single bell. It has a main hump near 1 000 MW (nights and weekends) and a broad shoulder between about 1 300 and 1 450 MW (working daytime hours), which makes it slightly right-skewed. A single mean and standard deviation describe such a mixture poorly, which is another reminder that the hour and the type of day must enter the model. Note also how far the horizontal axis stretches to the left: those barely visible bars are the suspicious low values again.

4.12 Visualisation 7 – load against temperature

Question: how strong is the weather effect, and is it linear? To avoid over-plotting we draw every fifth hour and colour the points by hour of day.

In [21]:
```
idx = 1:5:nrow(df)

scatter(df.temp[idx], df.load[idx], zcolor = df.hour[idx], color =
        markerstrokewidth = 0, alpha = 0.6, legend = false, colorba
        colorbar_title = "Hour of day", size = (700, 450),
        xlabel = "Temperature at Tocumen (°C)", ylabel = "Load (MW)'
        title = "Load versus temperature")
```

Out[21]:



Load clearly increases with temperature, but the cloud of points is wide and *banded by hour*: at a given temperature the load at 14:00 is very different from the load at 04:00. Temperature alone will not be enough, and its effect probably differs from hour to hour. The relationship is also gently curved, which motivates polynomial terms.

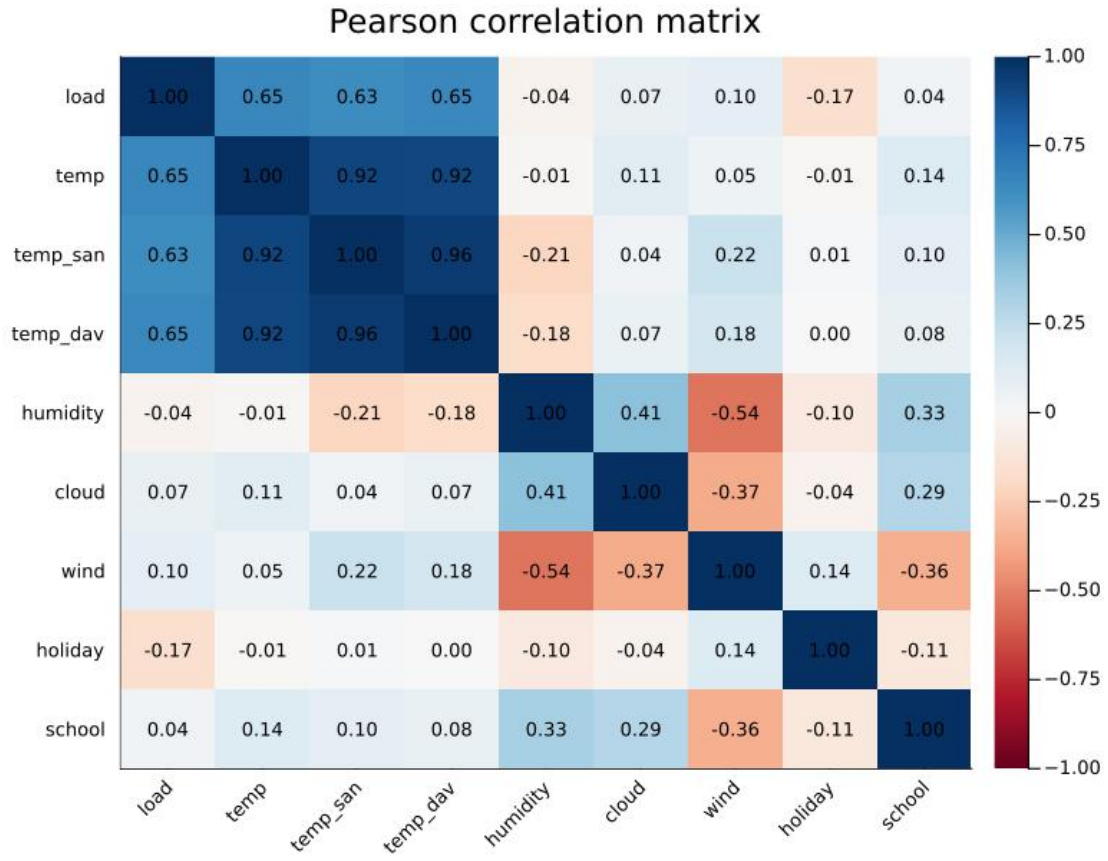
4.13 Correlation analysis

Pearson's correlation coefficient r measures the strength of the *linear* association between two variables ($-1 \le r \le 1$).

In [22]:

```
vars = [:load, :temp, :temp_san, :temp_dav, :humidity, :cloud, :win
R = cor(Matrix(df[:, vars]))

labels = string.(vars)
heatmap(labels, labels, R, color = :RdBu, clims = (-1, 1), yflip =
        xrotation = 45, title = "Pearson correlation matrix")
annotate!([(j - 0.5, i - 0.5, text(@sprintf("%.2f", R[i, j]), 8)) f
```

Out[22]:



In [23]:
```
# The first row of the matrix as a sorted table: correlation of eve
cortable = DataFrame(variable = labels[2:end], r_with_load = round.
sort(cortable, :r_with_load, rev = true)
```

Out[23]: 8×2 DataFrame

Row	variable	r_with_load
	String	Float64
1	temp	0.653
2	temp_dav	0.648
3	temp_san	0.627
4	wind	0.098
5	cloud	0.073
6	school	0.04
7	humidity	-0.037
8	holiday	-0.166

- The three temperatures correlate with the load at about $r \approx 0.65$ and very strongly **with each other**. Putting all three into a regression would create *multicollinearity* and add little new information, so we keep only `temp` (Panama City). Trying the others is left as an exercise.
- `holiday` has a small negative correlation because holidays are rare, not

because they are unimportant: Section 4.8 showed a large effect on the days concerned. **A low overall correlation does not mean that a variable is useless.**

4.14 Autocorrelation: how much does the load remember?

The autocorrelation function (ACF) is the correlation of the series with a copy of itself shifted by k hours. It tells us which *past* loads carry information about the present one.

In [24]:
```
lags = 0:336                                    # two weeks
acf  = autocor(df.load, lags)

plot(lags, acf, color = :navy, legend = :bottomleft, label = "ACF o
     xlabel = "Lag (hours)", ylabel = "Autocorrelation", xticks = 0
     title = "Autocorrelation function of the load", ylims = (0, 1)
vline!([48], color = :darkorange, linestyle = :dash, label = "48 h"
vline!([168, 336], color = :firebrick, linestyle = :dash, label = "

println("ACF at lag  24 h : ", round(acf[25], digits = 3))
println("ACF at lag  48 h : ", round(acf[49], digits = 3))
println("ACF at lag 168 h : ", round(acf[169], digits = 3))
```

```
ACF at lag  24 h : 0.809
ACF at lag  48 h : 0.694
ACF at lag 168 h : 0.89
```

The ACF peaks at every multiple of 24 hours, and among those peaks the one at **168 hours** (the same hour of the same weekday last week) is the highest that our information set allows us to use. Recall from Section 2 that a lag of 24 hours is *not* fully available on the morning of $D-1$. We shall therefore use the loads lagged by **48 h** and **168 h**.

> **Checkpoint 2.** (a) Which Julia function produced the summary table in Section 4.1? (b) Write one line that sorts `df` in descending order of `temp` and shows the first three rows. (c) Why is the ACF at lag 168 h higher than at lag 48 h?
>
> ▶ Solution

5. Pipeline stage 4 – Data preparation

Data preparation usually consumes most of the effort in a real project. Ours has two parts: **cleaning** (repairing what is wrong) and **feature engineering** (expressing what we learnt in Section 4 as columns that a linear model can use).

5.1 Detecting anomalous records

The summary statistics and the first plot revealed implausibly low loads. Let us look at the ten lowest.

In [25]:
```
first(sort(df[:, [:ts, :load, :holiday]], :load), 10)
```

Out[25]: 10×3 DataFrame

Row	ts	load	holiday
	DateTime	Float64	Int64
1	2019-01-20T12:00:00	85.1925	0
2	2019-01-20T13:00:00	135.764	0
3	2019-01-20T14:00:00	263.618	0
4	2017-06-27T12:00:00	380.594	0
5	2019-01-20T15:00:00	417.449	0
6	2019-01-20T16:00:00	475.802	0
7	2015-01-24T17:00:00	509.836	0
8	2017-07-01T14:00:00	560.701	0
9	2015-06-01T03:00:00	666.753	0
10	2017-03-17T21:00:00	668.998	0

Loads of 85–480 MW around midday on 20 January 2019, in a system that normally carries about 1 100 MW at that time of day, are consistent with a large-scale supply interruption rather than with consumers' behaviour. A few milder dips appear in 2015 and 2017. Such hours measure what the grid could *deliver*, not what customers *demanded*, and they would teach the model something false.

We need a rule that is objective and reproducible. A simple and transparent one is to compare each hour with the **same hour of the same weekday in the neighbouring weeks**:

$$\text{ratio}_t = \frac{y_t}{\text{median}\{\, y_{t-336},\; y_{t-168},\; y_{t+168},\; y_{t+336} \,\}}$$

and to flag an hour as anomalous when the ratio falls below 0.70 **and the day is not a public holiday** (holidays are legitimately low and must not be "repaired").

In [26]:
```
n = nrow(df)

# Typical load for each hour: median of the same hour one and two w
```

```
typical = [median([df.load[j] for j in (i - 336, i - 168, i + 168,

df.ratio   = df.load ./ typical
df.anomaly = (df.ratio .< 0.70) .& (df.holiday .== 0)

println("Number of flagged hours: ", sum(df.anomaly))
df[df.anomaly, [:ts, :load, :ratio, :holiday]]
```

```
Number of flagged hours: 15
```

Out[26]: 15×4 DataFrame

Row	ts	load	ratio	holiday
	DateTime	Float64	Float64	Int64
1	2015-01-24T17:00:00	509.836	0.484359	0
2	2017-03-17T21:00:00	668.998	0.524662	0
3	2017-03-17T22:00:00	673.223	0.549273	0
4	2017-03-17T23:00:00	806.993	0.696275	0
5	2017-03-21T19:00:00	711.097	0.531653	0
6	2017-03-21T20:00:00	702.881	0.532757	0
7	2017-06-27T12:00:00	380.594	0.25906	0
8	2017-06-27T13:00:00	861.799	0.58226	0
9	2017-07-01T14:00:00	560.701	0.448543	0
10	2019-01-20T11:00:00	704.641	0.631151	0
11	2019-01-20T12:00:00	85.1925	0.0742147	0
12	2019-01-20T13:00:00	135.764	0.117091	0
13	2019-01-20T14:00:00	263.618	0.22699	0
14	2019-01-20T15:00:00	417.449	0.363624	0
15	2019-01-20T16:00:00	475.802	0.417765	0

5.2 Repairing the anomalies

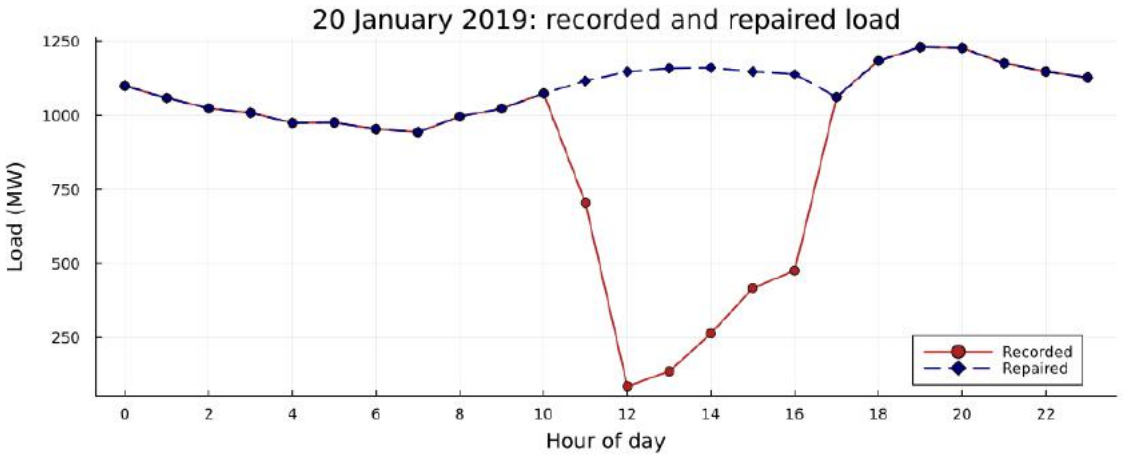
Deleting the flagged rows would break the regular hourly spacing on which our lagged features rely. Instead we **impute** each flagged value with the typical load computed above, keep the original in `load_raw`, and keep the `anomaly` flag so that the change remains traceable.

> Cleaning is a modelling decision and should always be documented. Here we clean *demand that was not served*, which no demand model can or should predict.

In [27]:
```
df.load_raw = copy(df.load)
```

```
df.load[df.anomaly] .= typical[df.anomaly]

day = df[df.date .== Date(2019, 1, 20), :]
plot(day.hour, day.load_raw, color = :firebrick, marker = :circle,
     xlabel = "Hour of day", ylabel = "Load (MW)", xticks = 0:2:23,
     title = "20 January 2019: recorded and repaired load", legend
plot!(day.hour, day.load, color = :navy, linestyle = :dash, marker
```

Out[27]:



5.3 Feature engineering

Each insight from the exploratory analysis now becomes one or more model inputs (*features* or *predictors*).

Insight from Section 4	Feature(s)	Type
Demand grows over the years	`trend` = years elapsed since the first record	numeric
Daily cycle	`hour` (0–23)	categorical
Weekly cycle	`dow` (1–7)	categorical
Annual cycle	`month` (1–12)	categorical
Holidays differ from ordinary days	`holiday` (0/1)	binary
Load rises non-linearly with temperature	`T` , `T2` , `T3` (centred temperature and its powers)	numeric
The load remembers its past	`lag48` , `lag168`	numeric

Three remarks:

- **Centring.** We use `T = temp − 27` , i.e. the deviation from a round figure close to the mean temperature. Centring leaves the fitted values unchanged but keeps `T` , `T²` and `T³` on comparable scales, which makes the computation numerically more stable.
- **Lagging by shifting rows.** Because Section 4.3 proved that the series has exactly one row per hour, the value 168 rows earlier *is* the value 168 hours earlier. The first rows have no past and receive `missing` .

- **Lags are built from the cleaned load**, otherwise the blackout would reappear as a "predictor" one week later.

In [28]:
```
# Shift a vector by k positions; the first k elements have no past
lagged(x, k) = [i > k ? x[i - k] : missing for i in eachindex(x)]

df.trend  = (0:n-1) ./ 8760                # years since the first
df.T      = df.temp .- 27                  # centred temperature
df.T2     = df.T .^ 2
df.T3     = df.T .^ 3
df.lag48  = lagged(df.load, 48)
df.lag168 = lagged(df.load, 168)

df[166:171, [:ts, :load, :lag48, :lag168]]
```

Out[28]: 6×4 DataFrame

Row	ts	load	lag48	lag168
	DateTime	Float64	Float64?	Float64?
1	2015-01-09T22:00:00	1033.86	1095.45	*missing*
2	2015-01-09T23:00:00	989.851	1044.17	*missing*
3	2015-01-10T00:00:00	949.503	985.891	*missing*
4	2015-01-10T01:00:00	906.958	962.407	970.345
5	2015-01-10T02:00:00	863.514	928.104	912.176
6	2015-01-10T03:00:00	848.445	918.0	900.269

The table confirms the mechanics: `lag168` is missing for the first 168 rows (one week), after which it holds the load observed exactly one week earlier.

5.4 The modelling table

Finally we keep the columns required for modelling and drop the rows that contain `missing` values (the first week).

In [29]:
```
model_cols = [:ts, :date, :year, :load, :trend, :hour, :dow, :month

data = dropmissing(df[:, model_cols])

println("Rows before : ", nrow(df))
println("Rows after  : ", nrow(data), "   (", nrow(df) - nrow(data)
println("Period      : ", first(data.ts), "  to  ", last(data.ts))
first(data, 3)
```

```
Rows before : 48048
Rows after  : 47880   (168 rows dropped)
Period      : 2015-01-10T01:00:00  to  2020-06-27T00:00:00
```

Out[29]: 3×14 DataFrame

Row	ts	date	year	load	trend	hour	dow	month
	DateTime	Date	Int64	Float64	Float64	Int64	Int64	Int64
1	2015-01-10T01:00:00	2015-01-10	2015	906.958	0.0191781	1	6	1
2	2015-01-10T02:00:00	2015-01-10	2015	863.514	0.0192922	2	6	1
3	2015-01-10T03:00:00	2015-01-10	2015	848.445	0.0194064	3	6	1

> **Checkpoint 3.** (a) Why would `lagged(df.load, 24)` be a questionable feature for *our* problem even though the ACF at 24 h is very high? (b) Suppose that the file had contained a gap of three missing hours. Would `lagged` still be correct? (c) Why did we not repair the low loads observed on public holidays?
>
> ▶ Solution

End of the first lecture hour.

SECOND LECTURE HOUR

6. Pipeline stage 5 – Data partitioning

A model must be judged on data that played no part in building it. For time series there is one additional rule: **never shuffle**. The split must be chronological, because in real life we always train on the past and forecast the future. A random split would let the model "see" Wednesday when predicting Tuesday of the same week, and the reported accuracy would be an illusion.

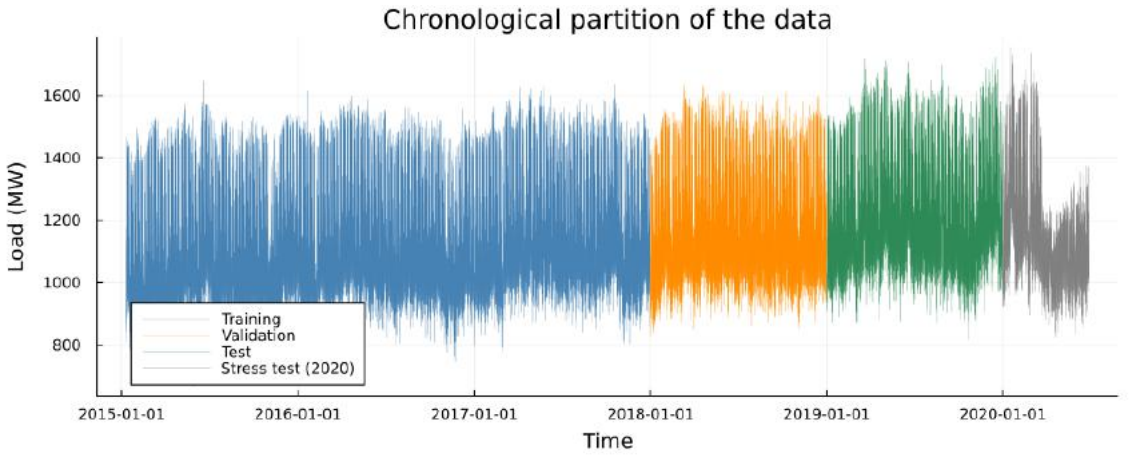
Subset	Period	Purpose
Training	January 2015 – December 2017	estimate the coefficients
Validation	2018	compare candidate models and choose one
Test	2019	report the accuracy of the chosen model, **once**
Stress test	January – June 2020	observe behaviour under COVID-19 (Section 9)

In [30]:
```
train = data[data.year .<= 2017, :]
valid = data[data.year .== 2018, :]
test  = data[data.year .== 2019, :]
covid = data[data.year .== 2020, :]

for (name, part) in (("Training", train), ("Validation", valid), ("
    println(rpad(name, 12), ": ", first(part.date), " to ", last(pa
            " | ", lpad(nrow(part), 6), " hours | mean load ", roun
end
```

```
Training    : 2015-01-10 to 2017-12-31 |  26087 hours | mean load 11
57.7 MW
Validation  : 2018-01-01 to 2018-12-31 |   8760 hours | mean load 12
05.7 MW
Test        : 2019-01-01 to 2019-12-31 |   8760 hours | mean load 12
44.1 MW
Stress test : 2020-01-01 to 2020-06-27 |   4273 hours | mean load 11
70.2 MW
```

In [31]:
```
plot(train.ts, train.load, linewidth = 0.3, color = :steelblue, lab
     xlabel = "Time", ylabel = "Load (MW)", title = "Chronological
plot!(valid.ts, valid.load, linewidth = 0.3, color = :darkorange, l
plot!(test.ts,  test.load,  linewidth = 0.3, color = :seagreen,   l
plot!(covid.ts, covid.load, linewidth = 0.3, color = :gray,       l
```

Out[31]:



7. Pipeline stage 6 – Modelling

7.1 Multiple linear regression in one paragraph

MLR assumes that the response y (the load) is a linear combination of p predictors plus a random error:

$$y_t = \beta_0 + \beta_1 x_{t,1} + \beta_2 x_{t,2} + \cdots + \beta_p x_{t,p} + \varepsilon_t \qquad \text{or, in matrix form,} \qquad \mathbf{y} =$$

Each row of the **design matrix** $\mathbf{X}$ is one hour; its first column is a column of ones (the intercept). **Ordinary least squares (OLS)** chooses the coefficients that minimise the sum of squared errors, and the solution has a closed form known as the *normal equations*:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

"Linear" refers to the *coefficients*, not to the variables. A model containing T^2, T^3 or the product of two predictors is still linear in $\boldsymbol{\beta}$, and this is what makes MLR far more flexible than it first appears.

7.2 Accuracy measures

Before building any model we define how forecasts will be scored. With actual values y_t, forecasts $\hat{y}_t$ and n hours:

Measure	Formula	Unit	Remark
Mean error (bias)	$\frac{1}{n}\sum(y_t - \hat{y}_t)$	MW	positive means under-forecasting
Mean absolute error	$\mathrm{MAE} = \frac{1}{n}\sum\lvert y_t - \hat{y}_t\rvert$	MW	easy to interpret
Root mean squared error	$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum(y_t - \hat{y}_t)^2}$	MW	penalises large errors
Mean absolute percentage error	$\mathrm{MAPE} = \frac{100}{n}\sum\left\lvert\frac{y_t - \hat{y}_t}{y_t}\right\rvert$	%	scale-free; the industry standard for load

MAPE is safe here because a national load is never close to zero. It would be a poor choice for, say, solar generation.

In [32]:
```
bias(actual, forecast) = mean(actual .- forecast)
mae(actual, forecast)  = mean(abs.(actual .- forecast))
rmse(actual, forecast) = sqrt(mean((actual .- forecast) .^ 2))
mape(actual, forecast) = 100 * mean(abs.((actual .- forecast) ./ ac

# A quick sanity check on a toy example: errors of +10, -10 and 0 o
println("MAE  = ", mae([100, 100, 100], [90, 110, 100]))
println("RMSE = ", round(rmse([100, 100, 100], [90, 110, 100]), dig
println("MAPE = ", round(mape([100, 100, 100], [90, 110, 100]), dig
```

```
MAE  = 6.666666666666667
RMSE = 8.165
MAPE = 6.667 %
```

7.3 Model M0 – the seasonal naïve benchmark

"Next Tuesday at 14:00 will look like last Tuesday at 14:00." The forecast is simply the column `lag168`; nothing is estimated. Every model that follows must beat this.

We also create a table in which the results of all models will be collected.

In [33]:
```
results = DataFrame(model = String[], parameters = Int[], train_MAP
                    valid_MAE = Float64[], valid_RMSE = Float64[],

push!(results, ("M0 Seasonal naive", 0,
                mape(train.load, train.lag168),
                mae(valid.load, valid.lag168),
                rmse(valid.load, valid.lag168),
                mape(valid.load, valid.lag168)))
results
```

Out[33]: 1×6 DataFrame

Row	model	parameters	train_MAPE	valid_MAE	valid_RMSE	valid_M
	String	Int64	Float64	Float64	Float64	Float64
1	M0 Seasonal naive	0	4.93485	53.3136	78.1843	4.4

7.4 A first regression, solved by hand

To demystify the method we fit a tiny model with two predictors using nothing but linear algebra:

$$\text{load}_t = \beta_0 + \beta_1 T_t + \beta_2 \,\text{lag168}_t + \varepsilon_t.$$

We build $\mathbf{X}$ ourselves, apply the normal equations, and then repeat the calculation with Julia's backslash operator `\`, which solves the same least-squares problem by a numerically more robust method (QR factorisation).

In [34]:
```
y = train.load
X = hcat(ones(nrow(train)), train.T, train.lag168)       # design m

beta_normal    = inv(X' * X) * X' * y                    # textbook
beta_backslash = X \ y                                   # preferre

println("Size of X            : ", size(X))
println("Normal equations     : ", round.(beta_normal, digits = 4)
println("Backslash operator   : ", round.(beta_backslash, digits =
```

```
Size of X            : (26087, 3)
Normal equations     : [248.2644, 22.0968, 0.7769]
Backslash operator   : [248.2644, 22.0968, 0.7769]
```

In practice we let `GLM.jl` do this work. Its `lm` function takes a **formula** (`response ~ predictors`) and a data frame, builds the design matrix, and

reports standard errors, *t*-statistics and confidence intervals as well.

In [35]:
```
m_hand = lm(@formula(load ~ T + lag168), train)
```

Out[35]:
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}}, GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

load ~ 1 + T + lag168

Coefficients:
──────────────────────────────────────────────────────────────────────────────
                  Coef.  Std. Error       t  Pr(>|t|)   Lower 95%   Upper 95%
──────────────────────────────────────────────────────────────────────────────
(Intercept)  248.264     3.78139      65.65    <1e-99  240.853     255.676
T             22.0968    0.388076     56.94    <1e-99   21.3361     22.8574
lag168         0.776904  0.00334203  232.46    <1e-99    0.770354    0.783455
──────────────────────────────────────────────────────────────────────────────
```

The three coefficients are identical to those we computed by hand. **Interpretation:** holding last week's load fixed, each additional degree Celsius is associated with the stated increase in load (MW); holding temperature fixed, each additional MW observed one week ago carries over as roughly three-quarters of a MW today.

7.5 Categorical predictors and dummy coding

`hour`, `dow` and `month` are stored as integers, but they are **categories**, not quantities: hour 20 is not "twice" hour 10. If we entered `hour` as a number, the model would be forced to draw a straight line through the daily profile, which Section 4.8 showed to be absurd.

The remedy is **dummy (one-hot) coding**: a categorical variable with k levels is replaced by $k-1$ binary columns, each equal to 1 for one level and 0 otherwise. The omitted level is the *reference*, absorbed by the intercept (including all k columns would make $\mathbf{X}^\top\mathbf{X}$ singular: the "dummy-variable trap"). So `hour` contributes 23 columns, `dow` 6 and `month` 11.

In `GLM.jl` we request this treatment through the `contrasts` keyword.

An **interaction**, written `a & b` in a formula, lets the effect of one variable depend on another. `dow & hour` gives every weekday its own daily profile.

In [36]:
```
coding = Dict(:hour => DummyCoding(), :dow => DummyCoding(), :month

# A helper that scores a fitted model on the validation year and ap
function addresult!(results, name, model)
    push!(results, (name, length(coef(model)),
                    mape(train.load, predict(model, train)),
                    mae(valid.load,  predict(model, valid)),
                    rmse(valid.load, predict(model, valid)),
                    mape(valid.load, predict(model, valid))))
    return results
end
```

Out[36]:
```
addresult! (generic function with 1 method)
```

7.6 Model M1 – temperature only

A deliberately poor model, to show how far weather alone can take us.

In [37]:
```
f1 = @formula(load ~ T)
m1 = lm(f1, train)
addresult!(results, "M1 Temperature only", m1)

println("R² on the training set: ", round(r2(m1), digits = 3))
m1
```

```
R² on the training set: 0.495
```

Out[37]:
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Flo
at64}}, GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{F
loat64, Matrix{Float64}, Vector{Int64}}}}, Matrix{Float64}}

load ~ 1 + T

Coefficients:
──────────────────────────────────────────────────────────────────
──────────
                 Coef.  Std. Error        t  Pr(>|t|)  Lower 95%
Upper 95%
──────────────────────────────────────────────────────────────────
──────────
(Intercept)  1119.55      0.878029  1275.08    <1e-99  1117.83
1121.28
T              81.6941    0.510599   160.00    <1e-99    80.6933
82.6949
──────────────────────────────────────────────────────────────────
──────────
```

Temperature on its own explains about half of the variance, yet the model is far worse than the naïve benchmark. It has no idea what time or what day it is.

7.7 Model M2 – calendar only

$$\text{load} = \beta_0 + \beta_1\,\text{trend} + \text{month} + \text{dow} + \text{hour} + \text{dow} \times \text{hour} + \varepsilon$$

This model knows the calendar perfectly and the weather not at all.

In [38]:
```
f2 = @formula(load ~ trend + month + dow + hour + dow & hour)
m2 = lm(f2, train; contrasts = coding)
addresult!(results, "M2 Calendar only", m2)

println("Columns of the design matrix: ", length(coef(m2)),
        "   = 1 intercept + 1 trend + 11 month + 6 dow + 23 hour + (
println("R² on the training set      : ", round(r2(m2), digits = 3)
```

```
Columns of the design matrix: 180   = 1 intercept + 1 trend + 11 mon
th + 6 dow + 23 hour + 6 × 23 interactions
R² on the training set      : 0.893
```

7.8 Model M3 – the "vanilla benchmark" (calendar × weather)

Tao Hong's *vanilla benchmark* is probably the best-known MLR model in load forecasting; it served as the benchmark of the Global Energy Forecasting Competition 2012. It adds a cubic polynomial of temperature to model M2 and allows that polynomial to change with the **month** and with the **hour**, precisely the behaviour we suspected from the banded scatter plot of Section 4.12:

$$\begin{aligned}\text{load} = \beta_0 + \beta_1\,\text{trend} + \text{month} + \text{dow} + \text{hour} + \text{dow} \times \text{hour} \\ + T + T^2 + T^3 + \text{month} \times (T + T^2 + T^3) + \text{hour} \times (T + T^2 + T\end{aligned}$$

In [39]:
```
f3 = @formula(load ~ trend + month + dow + hour + dow & hour +
                     T + T2 + T3 +
                     month & T + month & T2 + month & T3 +
                     hour & T + hour & T2 + hour & T3)
m3 = lm(f3, train; contrasts = coding)
addresult!(results, "M3 Vanilla benchmark", m3)

println("Columns of the design matrix: ", length(coef(m3)))
println("R² on the training set      : ", round(r2(m3), digits = 3)
```

```
Columns of the design matrix: 285
R² on the training set      : 0.907
```

7.9 Model M4 – adding public holidays

The vanilla model treats a public holiday that falls on a Tuesday as an ordinary Tuesday. We add the `holiday` indicator and let it interact with the hour, so that a holiday can reshape the whole daily profile (Section 4.8).

In [40]:
```
f4 = @formula(load ~ trend + month + dow + hour + dow & hour +
                     T + T2 + T3 +
                     month & T + month & T2 + month & T3 +
                     hour & T + hour & T2 + hour & T3 +
                     holiday + holiday & hour)
m4 = lm(f4, train; contrasts = coding)
```

```
addresult!(results, "M4 Vanilla + holiday", m4);
```

7.10 Model M5 – adding the recent load

Finally we add the two lagged loads that our information set permits. They let the model correct itself when the recent level of demand departs from what the calendar and the weather alone would suggest.

In [41]:
```
f5 = @formula(load ~ trend + month + dow + hour + dow & hour +
                     T + T2 + T3 +
                     month & T + month & T2 + month & T3 +
                     hour & T + hour & T2 + hour & T3 +
                     holiday + holiday & hour +
                     lag48 + lag168)
m5 = lm(f5, train; contrasts = coding)
addresult!(results, "M5 Vanilla + holiday + lags", m5);
```

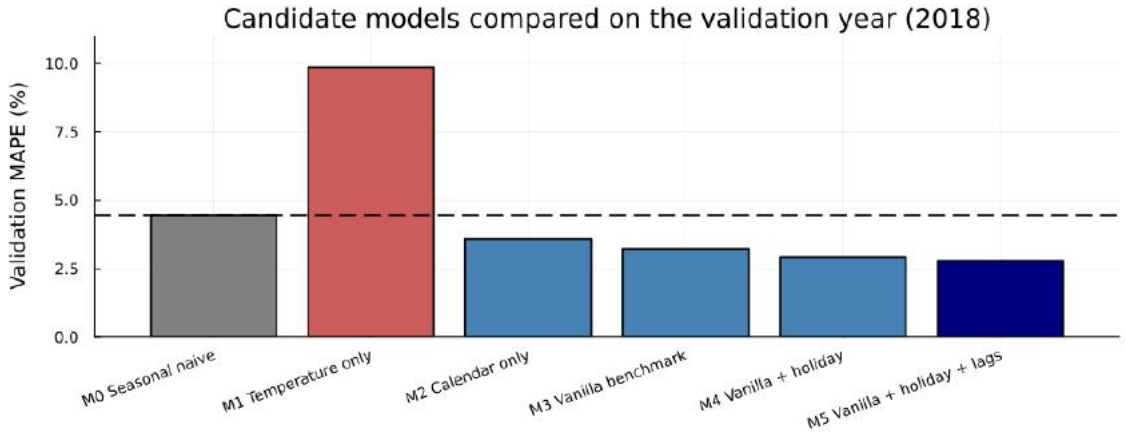
7.11 Model selection on the validation year

In [42]:
```
results[:, 3:6] = round.(results[:, 3:6], digits = 2)
show(results, allcols = true)
```

```
6×6 DataFrame
 Row │ model                          parameters  train_MAPE  valid_MAE  valid_RMSE  valid_MAPE
     │ String                         Int64       Float64     Float64    Float64     Float64
─────┼──────────────────────────────────────────────────────────────────────────────────────────
   1 │ M0 Seasonal naive                       0        4.93       53.31       78.18        4.45
   2 │ M1 Temperature only                     2        9.84      124.21      150.16        9.86
   3 │ M2 Calendar only                      180        3.77       42.16       61.56        3.58
   4 │ M3 Vanilla benchmark                  285        3.39       37.82       57.31        3.22
   5 │ M4 Vanilla + holiday                  309        2.98       34.34       45.85        2.92
   6 │ M5 Vanilla + holiday + lags           311        2.91       32.95       44.58        2.8
```

In [43]:
```
bar(results.model, results.valid_MAPE, color = [:gray, :indianred,
    legend = false, ylabel = "Validation MAPE (%)", xrotation = 20,
    title = "Candidate models compared on the validation year (2018
hline!([results.valid_MAPE[1]], color = :black, linestyle = :dash)
```

Out[43]:



Reading the table.

- **M1** (weather only) loses to the naïve benchmark, and **M2** (calendar only) already beats it. For this system *when* matters more than *how hot*.
- Each further block of features (temperature interactions, holidays, lags) reduces the validation error. That is evidence that each captures a real effect and not noise.
- The gap between `train_MAPE` and `valid_MAPE` stays small even with about 300 coefficients. With more than 25 000 training hours we are not over-fitting. (With one month of data the same model would over-fit badly; the ratio of observations to parameters matters.)

We select **M5**, the model with the lowest validation MAPE.

> **Checkpoint 4.** (a) How many columns does `hour & T` add to the design matrix when `T` itself is already in the model? (b) A classmate proposes choosing the model with the highest training R^2. What is wrong with this? (c) Why must model selection use the validation year and not the test year?
>
> ▶ Solution

8. Pipeline stage 7 – Evaluation

8.1 Refit and forecast the test year

The model *structure* is now fixed. Before the final examination we re-estimate its coefficients on **training + validation** data (2015–2018): in operation one always trains on all the history available, and the most recent year is the most relevant. The test year 2019 is then forecast once.

In [44]:

```
trainvalid = vcat(train, valid)
final      = lm(f5, trainvalid; contrasts = coding)

test.forecast = predict(final, test)
```

```
test.naive    = test.lag168
test.error    = test.load .- test.forecast

rsq(actual, forecast) = 1 - sum((actual .- forecast) .^ 2) / sum((a

scores = DataFrame(
    model = ["M0 Seasonal naive", "M5 Final MLR"],
    bias_MW = [bias(test.load, test.naive), bias(test.load, test.fo
    MAE_MW  = [mae(test.load, test.naive),  mae(test.load, test.for
    RMSE_MW = [rmse(test.load, test.naive), rmse(test.load, test.fo
    MAPE    = [mape(test.load, test.naive), mape(test.load, test.fo
    R2      = [rsq(test.load, test.naive),  rsq(test.load, test.for
)
scores[:, 2:6] = round.(scores[:, 2:6], digits = 3)
scores
```

Out[44]: 2×6 DataFrame

Row	model	bias_MW	MAE_MW	RMSE_MW	MAPE	R2
	String	Float64	Float64	Float64	Float64	Float64
1	M0 Seasonal naive	0.867	66.706	95.302	5.378	0.742
2	M5 Final MLR	-6.992	41.273	55.756	3.38	0.912

In [45]:
```
skill = 100 * (1 - scores.MAPE[2] / scores.MAPE[1])
println("The MLR model reduces the MAPE of the naive benchmark by "
```

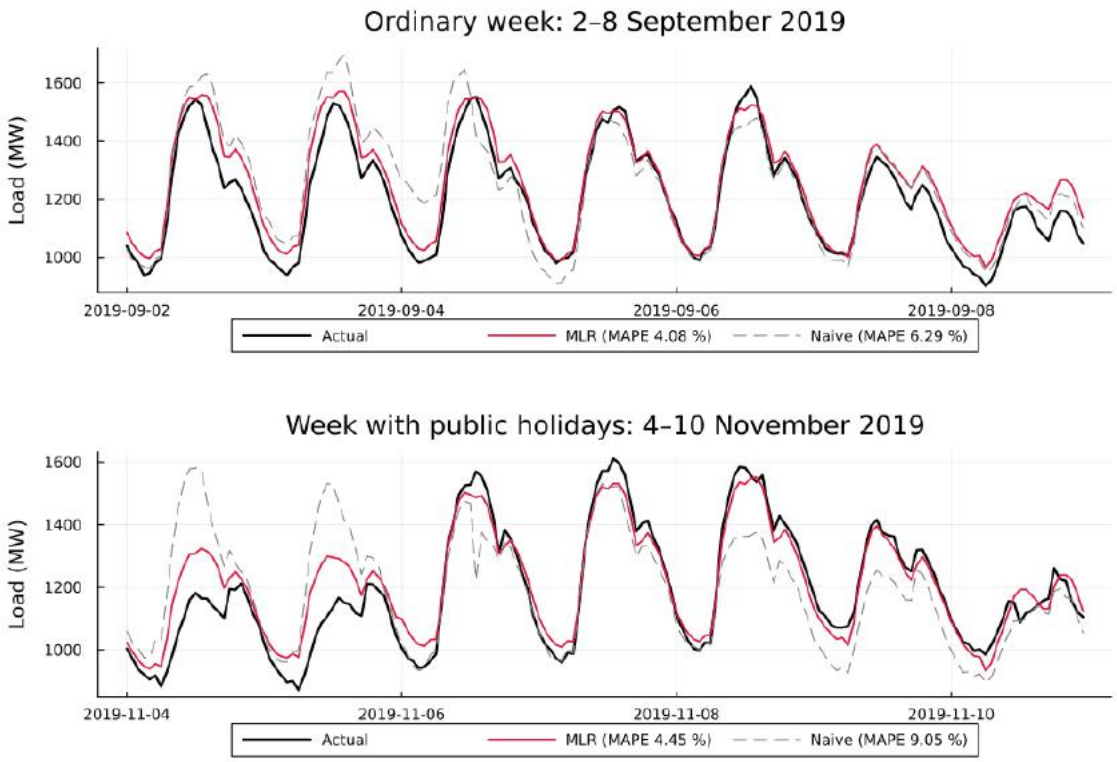
```
The MLR model reduces the MAPE of the naive benchmark by 37.2 % on t
he unseen year 2019.
```

The success criterion of Section 2 is met: on a year that played no part in fitting or selecting the model, MLR is clearly more accurate than the seasonal naïve forecast. Notice that the test error is somewhat higher than the validation error. This is normal, and it is exactly why an untouched test set is needed for an honest report.

8.2 Forecast against actual: an ordinary week and a week with holidays

In [46]:
```
function plotweek(frame, from, to, title)
    w = frame[(frame.date .>= from) .& (frame.date .<= to), :]
    p = plot(w.ts, w.load, color = :black, linewidth = 2, label = "
             title = title, legend = :outerbottom, legend_column =
    plot!(p, w.ts, w.forecast, color = :crimson, label = @sprintf("
    plot!(p, w.ts, w.naive, color = :gray, linestyle = :dash, linew
          label = @sprintf("Naive (MAPE %.2f %%)", mape(w.load, w.na
    return p
end

pA = plotweek(test, Date(2019, 9, 2),  Date(2019, 9, 8),   "Ordinar
pB = plotweek(test, Date(2019, 11, 4), Date(2019, 11, 10), "Week wi
plot(pA, pB, layout = (2, 1), size = (900, 640))
```

Out[46]:



In the ordinary week both methods reproduce the daily shape, and MLR is the closer of the two. The holiday week is more revealing: the naïve forecast copies the previous (normal) week and misses the holidays entirely, whereas MLR has been told about them through the `holiday` feature.

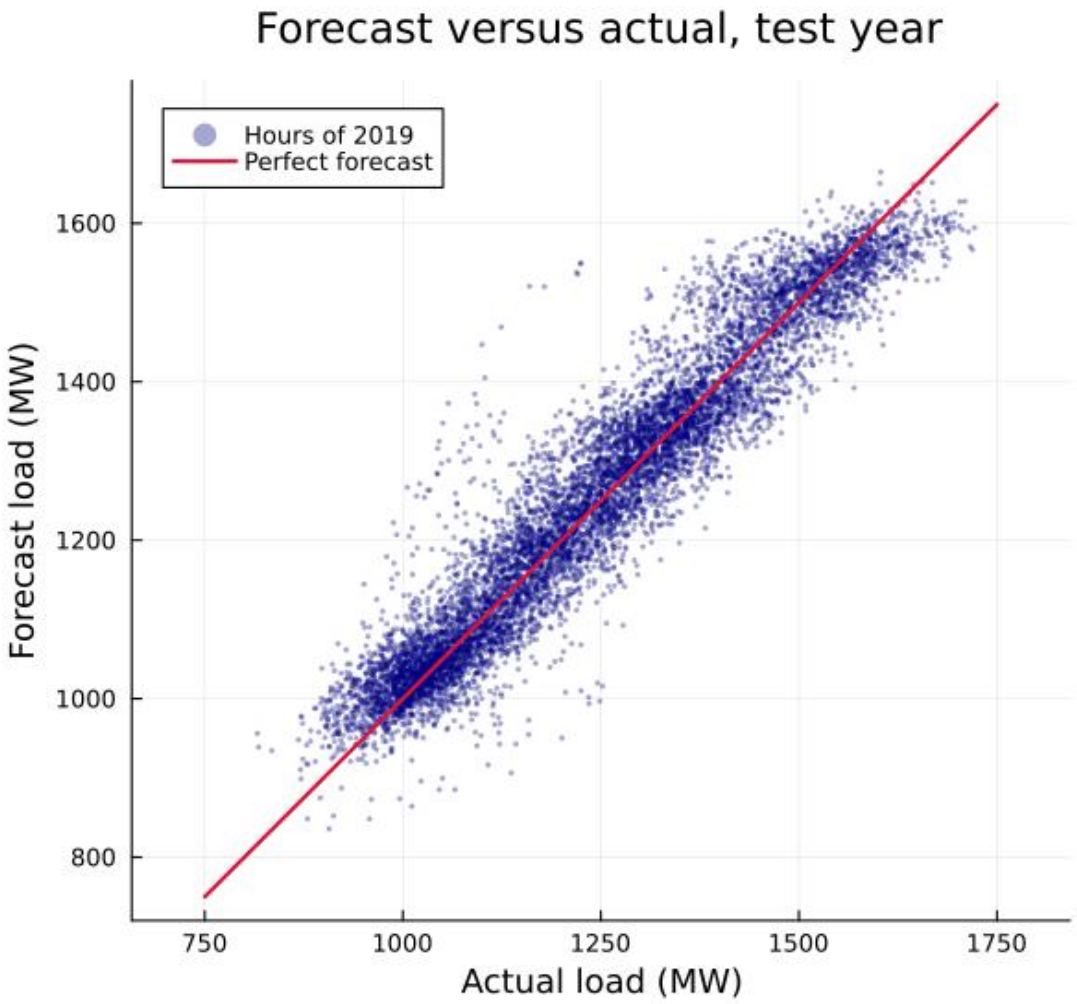
8.3 Forecast against actual: scatter plot

A perfect forecast would place every point on the 45° line.

In [47]:

```
scatter(test.load, test.forecast, markersize = 1.5, markerstrokewid
        label = "Hours of 2019", xlabel = "Actual load (MW)", ylabe
        title = "Forecast versus actual, test year", size = (560, 5
plot!([750, 1750], [750, 1750], color = :crimson, linewidth = 2, la
```

Out[47]:



8.4 Residual diagnostics

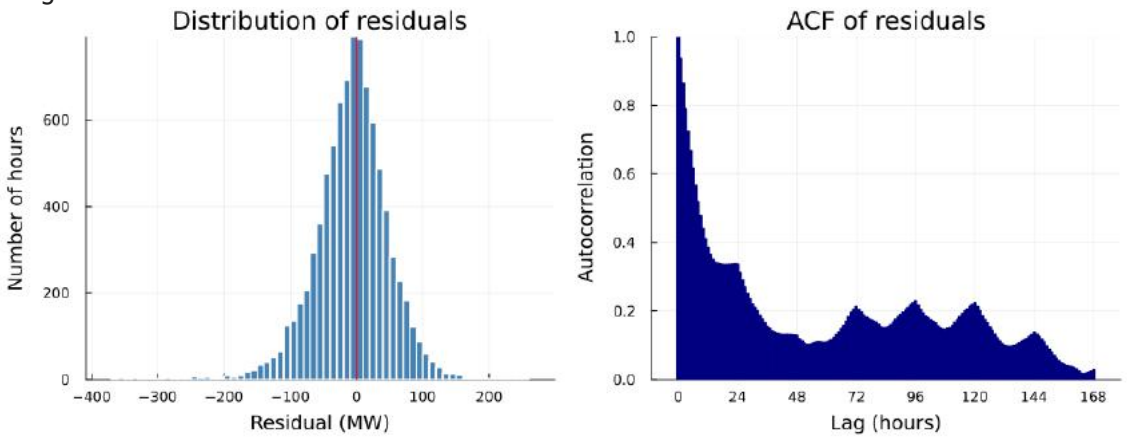
The residuals $e_t = y_t - \hat{y}_t$ are what the model failed to explain. If the model were perfect they would be pure noise: centred on zero, without structure and uncorrelated in time.

In [48]:

```
e = test.error

println("Mean of residuals (bias)      : ", round(mean(e), digits =
println("Standard deviation            : ", round(std(e), digits = 
println("5 % and 95 % quantiles        : ", round.(quantile(e, [0.0
println("Largest under-forecast        : ", round(maximum(e), digit
println("Largest over-forecast         : ", round(minimum(e), digit

h1 = histogram(e, bins = 80, color = :steelblue, linecolor = :white
               xlabel = "Residual (MW)", ylabel = "Number of hours"
vline!(h1, [0], color = :crimson, linewidth = 2)

racf = autocor(e, 0:168)
h2 = bar(0:168, racf, color = :navy, linecolor = :navy, legend = fa
         xlabel = "Lag (hours)", ylabel = "Autocorrelation", title =

plot(h1, h2, layout = (1, 2), size = (950, 360))
```

```
Mean of residuals (bias)      : -6.99 MW
Standard deviation            : 55.32 MW
5 % and 95 % quantiles        : [-98.8, 77.2] MW
Largest under-forecast        : 251.8 MW at 2019-12-08T14:00:00
Largest over-forecast         : -360.5 MW at 2019-07-01T10:00:00
```

Out[48]:



- The histogram is roughly bell-shaped and centred very close to zero. The mean residual is about −7 MW, i.e. the model over-forecast by roughly 0.6 % of the average load in 2019, which is negligible. The long, thin left tail consists of a few hours in which demand was far below anything the features could explain.
- The residuals are **strongly autocorrelated at short lags**: if the model was too low an hour ago, it is probably too low now. Information therefore remains that an MLR restricted to lags of 48 h and more cannot exploit. This is the price of the day-ahead information set, and it explains why intraday forecasts, which may use the most recent hours, are more accurate.

A practical consequence: the standard errors and *p*-values printed by `lm` assume uncorrelated errors and are therefore too optimistic here. We use the model for **prediction** and judge it by out-of-sample accuracy, not by *p*-values.

8.5 Where does the model struggle?

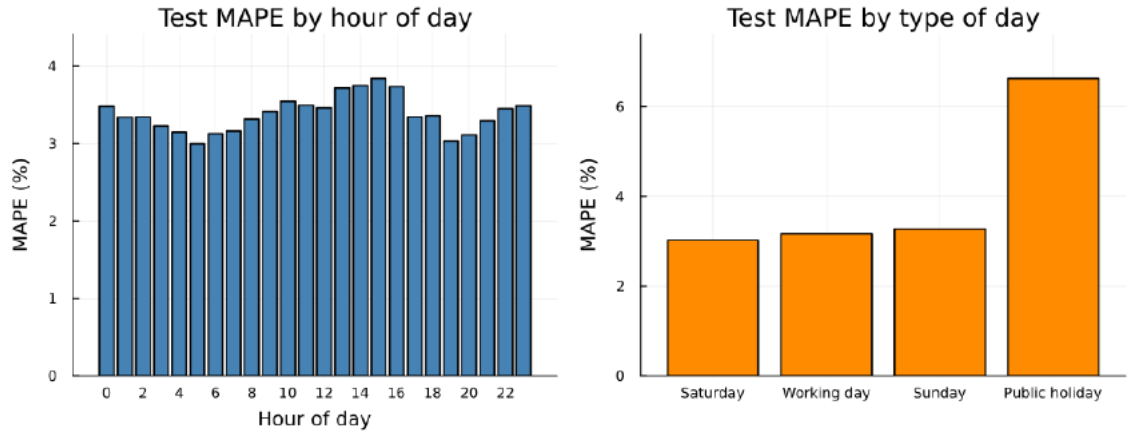
Averages hide detail. Grouping the absolute percentage error by hour and by type of day shows *when* the model is weak, which is the first step towards improving it.

In [49]:

```
test.ape     = 100 .* abs.(test.error ./ test.load)
test.daytype = daytype.(test.dow, test.holiday)

byhour = sort(combine(groupby(test, :hour), :ape => mean => :MAPE),
bytype = sort(combine(groupby(test, :daytype), :ape => mean => :MAP

b1 = bar(byhour.hour, byhour.MAPE, color = :steelblue, legend = fal
         xlabel = "Hour of day", ylabel = "MAPE (%)", title = "Test
         ylims = (0, 1.15 * maximum(byhour.MAPE)))
```

```
b2 = bar(bytype.daytype, bytype.MAPE, color = :darkorange, legend =
         ylabel = "MAPE (%)", title = "Test MAPE by type of day", y
plot(b1, b2, layout = (1, 2), size = (950, 360))
```

Out[49]:



In [50]:
```
bytype
```

Out[50]: 4×3 DataFrame

Row	daytype	MAPE	hours
	String	Float64	Int64
1	Saturday	3.01846	1200
2	Working day	3.16876	5856
3	Sunday	3.27526	1152
4	Public holiday	6.63144	552

In [51]:
```
# The five worst-forecast days of the test year
bydate = combine(groupby(test, :date), :ape => mean => :MAPE, :holi
                 :load => mean => :mean_load, :forecast => mean =>
bydate.weekday = dayname.(bydate.date)
first(sort(bydate, :MAPE, rev = true), 5)
```

Out[51]: 5×6 DataFrame

Row	date	MAPE	holiday	mean_load	mean_forecast	weekday
	Date	Float64	Int64	Float64	Float64	String
1	2019-07-01	16.3888	0	1102.1	1280.72	Monday
2	2019-04-19	15.454	1	1058.99	1222.66	Friday
3	2019-03-05	13.9495	1	1022.59	1164.77	Tuesday
4	2019-12-25	11.5051	1	1091.73	1217.73	Wednesday
5	2019-01-01	10.2937	1	1021.39	1124.5	Tuesday

Across the hours of the day the error is fairly even (roughly 3–4 %), a little higher in the early afternoon, when the load is most sensitive to weather and to human activity. The type of day matters far more: the MAPE on **public holidays is about twice** that of ordinary days. Holidays are both rare (few examples to learn from) and diverse (one indicator cannot distinguish New Year's Day from a minor holiday).

Now inspect the worst days. Four of the five are flagged holidays. The worst of all, Monday 1 July 2019, is *not* flagged, yet its load looks like that of a holiday; it was the day of a presidential inauguration in Panama. The model can only be as good as its calendar: an event that is missing from the features cannot be forecast.

8.6 Interpreting some coefficients

One advantage of MLR over "black-box" methods is that its coefficients can be read. The full table has about 300 rows; we extract three of them.

In [52]:
```
ct = DataFrame(coeftable(final))
ct[in.(ct.Name, Ref(["trend", "lag48", "lag168"])), [1, 2, 3, 6, 7]]
```

Out[52]: 3×5 DataFrame

Row	Name	Coef.	Std. Error	Lower 95%	Upper 95%
	String	Float64	Float64	Float64	Float64
1	trend	25.8252	0.271809	25.2924	26.358
2	lag48	0.0996361	0.00399273	0.0918102	0.107462
3	lag168	0.0841013	0.00396156	0.0763365	0.0918661

- `trend` : after allowing for all other effects, demand grows by roughly this many MW per year.
- `lag168` and `lag48` : both are positive and well below one. The model takes account of recent demand but does not simply copy it, as the naïve method does; the calendar and the weather carry most of the weight.

Because temperature enters through many interaction terms, its effect cannot be read from a single coefficient. Exercise 4 asks you to visualise it instead.

9. Pipeline stage 8 – Deployment

9.1 A day-ahead forecasting function

A model creates value only when it is used. The function below wraps the

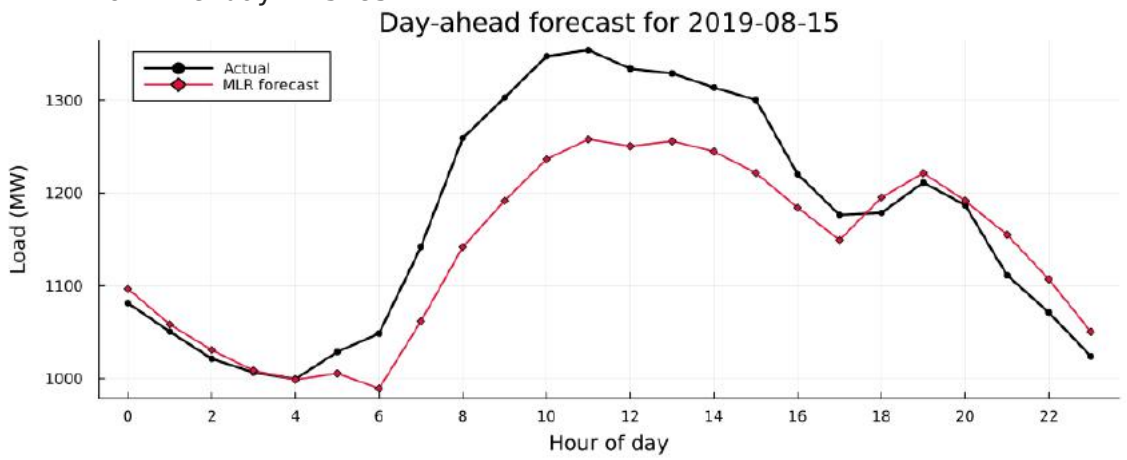
fitted model into the product that an operator needs: the 24 hourly forecasts for a chosen delivery day.

In live operation the rows passed to `predict` would be assembled on the morning of $D-1$ from (i) the calendar of day D, (ii) the weather *forecast* for day D and (iii) loads already metered up to day $D-2$. Here those rows already exist in our table.

In [53]:
```
function forecastday(model, frame, day::Date)
    rows = frame[frame.date .== day, :]
    nrow(rows) == 24 || error("No complete set of 24 hours is availa
    return DataFrame(ts = rows.ts, hour = rows.hour, forecast = pre
end

delivery_day = Date(2019, 8, 15)                    # change this
fc = forecastday(final, test, delivery_day)

println("Day-ahead forecast for ", dayname(delivery_day), " ", deli
println("  Forecast energy : ", round(sum(fc.forecast) / 1000, digi
println("  Forecast peak   : ", round(maximum(fc.forecast), digits =
        "   (actual ", round(maximum(fc.actual), digits = 1), " MW
println("  MAPE of the day : ", round(mape(fc.actual, fc.forecast),

plot(fc.hour, fc.actual, color = :black, linewidth = 2, marker = :c
     xlabel = "Hour of day", ylabel = "Load (MW)", xticks = 0:2:23,
     title = "Day-ahead forecast for $(delivery_day)")
plot!(fc.hour, fc.forecast, color = :crimson, marker = :diamond, ma
```

```
Day-ahead forecast for Thursday 2019-08-15
  Forecast energy : 27.31 GWh   (actual 28.1 GWh)
  Forecast peak   : 1258.1 MW at hour 11   (actual 1354.2 MW at hour
11)
  MAPE of the day : 3.85 %
```

Out[53]:



In [54]:
```
first(fc, 6)
```

Out[54]: 6×4 DataFrame

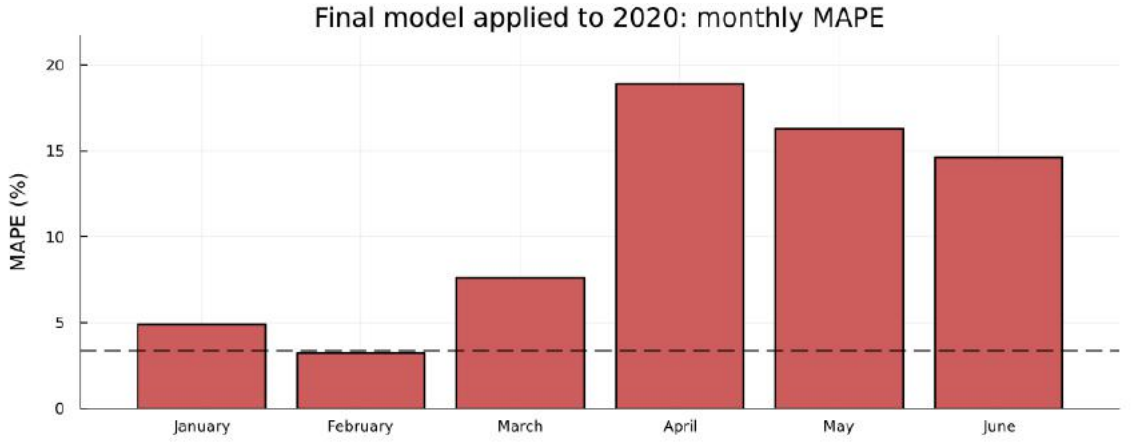
Row	ts	hour	forecast	actual
	DateTime	Int64	Float64?	Float64
1	2019-08-15T00:00:00	0	1096.74	1081.25
2	2019-08-15T01:00:00	1	1058.3	1050.47
3	2019-08-15T02:00:00	2	1030.7	1021.37
4	2019-08-15T03:00:00	3	1008.68	1006.54
5	2019-08-15T04:00:00	4	998.801	999.488
6	2019-08-15T05:00:00	5	1005.55	1028.75

9.2 Monitoring after deployment: the COVID-19 stress test

A deployed model must be monitored, because the world it learnt from can change. Let us apply the final model, unchanged, to the first half of 2020 and follow its error month by month.

In [55]:
```
covid.forecast = predict(final, covid)
covid.ape      = 100 .* abs.((covid.load .- covid.forecast) ./ covi
covid.month    = month.(covid.ts)

bymonth = sort(combine(groupby(covid, :month), :ape => mean => :MAP
                       :load => mean => :mean_load, :forecast => me
bymonth.month_name = monthname.(bymonth.month)

bar(bymonth.month_name, bymonth.MAPE, color = :indianred, legend =
    ylabel = "MAPE (%)", title = "Final model applied to 2020: mont
hline!([scores.MAPE[2]], color = :black, linestyle = :dash)
```

Out[55]:



In [56]:
```
bymonth[:, [:month_name, :mean_load, :mean_forecast, :MAPE]]
```

Out[56]: 6×4 DataFrame

Row	month_name	mean_load	mean_forecast	MAPE
	String	Float64	Float64	Float64
1	January	1276.59	1260.74	4.91257
2	February	1269.52	1256.8	3.2149
3	March	1208.99	1276.88	7.59324
4	April	1062.04	1262.66	18.9119
5	May	1088.15	1265.61	16.2668
6	June	1109.26	1272.6	14.6293

In January and February the error is of the same order as in 2019 (the dashed line). From mid-March 2020 it multiplies: actual demand collapses while the model, which knows only the calendar, the weather and loads that are at least two days old, keeps forecasting "business as usual". The lagged loads pull the forecast down a little, but not nearly enough.

No amount of historical data could have prepared the model for this. The lesson for practice is that deployment is not the end of the pipeline: errors must be **monitored**, and models **re-trained** or redesigned when the data-generating process shifts.

10. Pipeline stage 9 – Knowledge, limitations and next steps

10.1 What we did

Pipeline stage	What we did	Key Julia tools
Problem definition	day-ahead horizon, information set, benchmark, metric	–
Data acquisition	downloaded and imported an open data set	`Downloads`, `CSV.File`, `DataFrame`
Data understanding	summary statistics, eight kinds of figure, correlation, ACF	`describe`, `groupby`, `combine`, `cor`, `autocor`, `Plots`
Data preparation	timestamp convention, anomaly repair, calendar, polynomial and lag features	`Dates`, broadcasting, comprehensions
Partitioning	chronological training, validation and test sets	logical indexing

Modelling	naïve benchmark and five MLR models of growing richness	`\`, `lm`, `@formula`, `DummyCoding`
Evaluation	MAE, RMSE, MAPE, residual analysis, error breakdowns	user-defined functions
Deployment	24-hour forecasting function and monitoring under drift	`predict`

10.2 What we learnt

1. A carefully specified **linear** model beats the seasonal naïve benchmark by a clear margin on a completely unseen year. Feature engineering guided by exploratory analysis, not algorithmic sophistication, delivered that gain.
2. For this tropical system the **calendar** explains more than the **weather**, but the combination, with interactions, is better than either.
3. **Leakage** is the forecaster's cardinal sin. Every feature must pass the test: "would I really have known this value at the moment the forecast was issued?"
4. Accuracy figures mean little without a **benchmark** and without **unseen data**.

10.3 Limitations

- Observed temperatures stand in for weather forecasts, so operational errors would be somewhat higher.
- Only one weather station and one weather variable are used.
- All holidays share a single indicator; days adjacent to holidays ("bridge days") are ignored.
- The model gives **point** forecasts only. Operators increasingly require *probabilistic* forecasts (prediction intervals or quantiles).
- The coefficients are fixed after training; a rolling re-estimation (for instance, every week) would adapt more quickly to change.

10.4 Exercises

In class

1. Change `delivery_day` in Section 9.1 to a Sunday and to a public holiday of 2019. How does the daily MAPE compare with that of 15 August?
2. Using `groupby` and `combine`, compute the mean, minimum and maximum temperature for each month and sort the result in descending order of the mean.
3. Write a function `peakhour(frame, day)` that returns the hour at which the load of a given date was highest, and apply it to 20 January

2019 using `load_raw` and `load`.

Homework

4. *Temperature response.* For a working day in March at 14:00, vary the temperature from 24 °C to 34 °C while holding every other feature fixed, obtain the predictions of model M3, and plot predicted load against temperature. Repeat for 04:00 and comment.
5. *More weather.* Replace `temp` by the average of the three city temperatures, then add `humidity` to M5. Does the validation MAPE improve?
6. *Recency.* Add `lag336` (two weeks) to M5. Is the improvement on the validation year worth the extra parameter?
7. *Leakage experiment.* Add `lag24` to M5 and record the validation MAPE. Explain why the improvement, although real in the notebook, could not be obtained in day-ahead operation.
8. *Rolling origin.* Instead of fitting once, re-estimate M5 at the start of every month of 2019 using all data up to that point, and forecast only that month. Compare the resulting test MAPE with the one in Section 8.1.
9. *Türkiye.* Download hourly consumption data for Türkiye from the EPİAŞ Transparency Platform, repeat Sections 4–8, and discuss how the temperature response differs in a country with both heating and cooling seasons.

10.5 References

- Aguilar Madrid, E. (2021). *Short-term electricity load forecasting (Panama case study)*. Mendeley Data, V1. doi:10.17632/byx7sztj59.1
- Aguilar Madrid, E. and António, N. (2021). Short-term electricity load forecasting with machine learning. *Information*, 12(2), 50. doi:10.3390/info12020050
- Bezanson, J., Edelman, A., Karpinski, S. and Shah, V. B. (2017). Julia: a fresh approach to numerical computing. *SIAM Review*, 59(1), 65–98.
- Hong, T. (2010). *Short Term Electric Load Forecasting*. PhD dissertation, North Carolina State University.
- Hong, T., Pinson, P. and Fan, S. (2014). Global Energy Forecasting Competition 2012. *International Journal of Forecasting*, 30(2), 357–363.
- Hong, T. and Fan, S. (2016). Probabilistic electric load forecasting: a tutorial review. *International Journal of Forecasting*, 32(3), 914–938.
- Hyndman, R. J. and Athanasopoulos, G. (2021). *Forecasting: Principles and Practice* (3rd ed.). OTexts.

Appendix – Software environment

For reproducibility, the cell below records the versions of Julia and of the packages with which this notebook was executed.

In [57]:
```
import Pkg
println("Julia ", VERSION)
Pkg.status(["CSV", "DataFrames", "GLM", "Plots", "StatsBase"])
```

```
Julia 1.13.3
Status `~/.julia/environments/v1.13/Project.toml`
  [336ed68f] CSV v1.3.0
  [a93c6f00] DataFrames v1.8.3
  [38e38edf] GLM v1.9.5
  [91a5bcdd] Plots v1.41.7
  [2913bbd2] StatsBase v0.34.13
```
