Practical Forecasting in R with fable & modeltime

A short, example–driven book

Dario Fanucchi

Table of contents

1 Practical Forecasting in R with fable & modeltime

A short, example–driven book

This book is a compact, opinionated guide to modern time series forecasting in R, with a focus on:

- Classical, statistically principled methods using the **fable** ecosystem.
- Machine–learning–style forecasting using **modeltime**, **tidymodels**, and friends.
- Practical extensions for retail and supply–chain–flavoured data: promotions, prices, hierarchies, and global models.
- Optional **AutoML** and scalable modelling using **H2O**.

The style is example–led: every concept is tied to runnable R code.

You can read it straight through, or jump to the part that matches how you work today:

- Part I: fable
- Part II: modeltime + tidymodels
- Part III: Extensions (promos, hierarchies, feature engineering, global vs per–SKU)
- Part IV: AutoML with H2O

2 Part I — Forecasting with fable

3 1. Time series as tsibbles

Motivation

The fable ecosystem is built around the `tsibble` class: a tidy time–series structure with an explicit time index and (optionally) one or more keys. If your data is not in a tsibble, everything else becomes awkward.

```r
library(tsibble)
library(feasts)
library(fable)

data <- tsibble(
  date  = as.Date("2020-01-01") + 0:729,
  y     = sin(2*pi*(0:729)/7) + rnorm(730, 0, 0.2),
  index = date
)

data
```

You now have a single daily series with 730 observations.

---

4 2. Baseline forecasts: mean, naive, seasonal naive

Motivation

Good baselines are essential. Mean, naive, and seasonal–naive are trivial to compute but surprisingly hard to beat on some data. They also give you quick sanity checks.

```
data %>%
  model(
    mean   = MEAN(y),
    naive  = NAIVE(y),
    snaive = SNAIVE(y)
  ) %>%
  forecast(h = "30 days")
```

Each model returns a fable; you can bind, plot, and calculate accuracy with the same tools.

---

5 3. ARIMA

Motivation

ARIMA (and seasonal ARIMA) is still a workhorse model, especially when you want a univariate, interpretable, statistically principled baseline.

```
fit_arima <- data %>%
  model(arima = ARIMA(y))

report(fit_arima)
fit_arima %>% forecast(h = "30 days")
```

`ARIMA()` automatically identifies appropriate orders (including seasonal). `report()` gives you parameter estimates and diagnostics.

---

6 4. ETS (Exponential Smoothing)

Motivation

ETS models handle level, trend, and seasonality with explicit error formulations. They are robust and often perform very well on business series.

```
fit_ets <- data %>%
  model(ets = ETS(y))

fit_ets %>% report()
fit_ets %>% forecast(h = "30 days")
```

ETS models are often a strong alternative to ARIMA when seasonal patterns are stable and the noise structure is well described by exponential smoothing.

7 5. STL + ETS (decomposition–based forecasting)

Motivation

Sometimes you want to separate trend/seasonality from the short–term noise, both for interpretability and modelling flexibility. STL decomposition plus ETS on the seasonally adjusted component is a powerful pattern.

```
fit_stl <- data %>%
  model(
    stl_ets = decomposition_model(
      STL(y ~ season(window = 7)),
      ETS(season_adjust)
    )
  )

fit_stl %>%
  forecast(h = "30 days")
```

Here STL extracts a weekly seasonal component; ETS models the seasonally adjusted series.

---

8 6. ARIMAX: regression with ARIMA errors

Motivation

Most real–world forecasting problems need exogenous drivers: price, promotions, macro variables, events, etc. ARIMAX (regression with ARIMA errors) is the classical tool for this.

```
library(dplyr)

data_x <- data %>%
  mutate(promo = rbinom(n(), 1, 0.1))

fit_arimax <- data_x %>%
  model(arimax = ARIMA(y ~ promo))

report(fit_arimax)

future_x <- tsibble(
  date  = seq.Date(max(data$date) + 1, by = "day", length.out = 30),
  promo = 0,
  index = date
)

fit_arimax %>%
  forecast(new_data = future_x)
```

- `promo` captures uplift relative to the baseline.
- The ARIMA error structure accounts for remaining autocorrelation.

You can add more regressors (price, other features) in the same way.

---

9 7. Multiple series with keys

Motivation

In retail and operations, you rarely have a single series. You typically have thousands (SKU × store × region). fable's key semantics let you fit one model per series with identical code.

```
data_multi <- tsibble(
  id    = rep(letters[1:5], each = 730),
  date = rep(as.Date("2020-01-01") + 0:729, 5),
  y     = rnorm(5 * 730),
  key   = id,
  index = date
)

fit_multi <- data_multi %>%
  model(arima = ARIMA(y))

fit_multi %>%
  forecast(h = "30 days")
```

Each key (`id`) gets its own ARIMA model. You can still summarise accuracy and forecasts across all keys.

---

10 8. Hierarchies & reconciliation (including MinT)

Motivation

Hierarchical and grouped time series (e.g. SKU → brand → category → region → country) require *coherent* forecasts: children should add up to parents. Reconciliation methods adjust a set of base forecasts to satisfy aggregation constraints while staying close to the originals.

10.1 8.1 Conceptual view

Stack all series at time t into a vector $\mathbf{y}$. There exists a summing matrix $\mathbf{S}$ such that

$$\mathbf{y}_t = \mathbf{S}\,\mathbf{b}_t$$

where $\mathbf{b}$ are the bottom–level series (e.g., SKUs).

If you fit arbitrary models to each series, you obtain base forecasts $\hat{\mathbf{y}}_h$ that in general are **not coherent**.

Reconciliation finds adjusted forecasts

$$\tilde{\mathbf{y}}_h = \mathbf{S}\,\mathbf{P}\,\hat{\mathbf{y}}_h^{[b]}$$

such that:

- Coherence holds by construction (aggregation uses $\mathbf{S}$).
- The adjustments are optimal according to some criterion.

MinT (“Minimum Trace”) chooses $\mathbf{P}$ to minimise the trace of the reconciled error covariance, using an estimate of the base forecast error covariance matrix $\mathbf{W}$.

In one common parameterisation:

$$\tilde{\mathbf{y}}_h = \mathbf{S}(\mathbf{S}^\top \mathbf{W}^{-1}\mathbf{S})^{-1}\mathbf{S}^\top \mathbf{W}^{-1}\hat{\mathbf{y}}_h$$

Intuition:

- Series with **high variance** forecasts get shrunk more towards coherent aggregates.
- Series whose **errors are highly correlated** with others share information more strongly.

In practice, fable estimates $\mathbf{W}$ from in–sample residuals and performs this matrix algebra for you.

10.2 8.2 Using reconciliation in fable

```
library(fabletools)
library(tsibbledata)

tourism <- tourism  # key = (Region, Purpose)

fit <- tourism %>%
  model(ets = ETS(Trips))

rec <- fit %>%
  reconcile(
    bu   = bottom_up(ets),
    mint = min_trace(ets),
    mint_shr = min_trace(ets, method = "mint_shrink")
  )

fc <- rec %>%
  forecast(h = "3 years")
```

Guidelines:

- **bottom_up()**: use when bottom–level series are well measured and you care most about them.
- **min_trace() / mint_shrink**: use when you want balanced accuracy across levels, and when noise at the bottom is substantial.

---

11 9. Rolling–origin evaluation (tscv)

Motivation

A single train/test split can be misleading in time series. Rolling–origin (or "time series cross–validation") repeatedly trains on expanding windows and evaluates a fixed horizon ahead.

```
cv <- data %>%
  stretch_tsibble(.init = 365, .step = 30)

cv_results <- cv %>%
  model(arima = ARIMA(y)) %>%
  forecast(h = 30)

accuracy(cv_results, data)
```

Each row of the stretched tsibble represents one training window; `forecast(h = 30)` produces 30–day–ahead forecasts for that window.

---

12 10. Combining forecasts

Motivation

Forecast combinations are cheap and often yield gains over any single model. Even simple averages can be very effective.

```r
fit <- data %>%
  model(
    arima = ARIMA(y),
    ets   = ETS(y),
    mean  = MEAN(y)
  )

combo <- fit %>%
  mutate(
    combo = (arima + ets + mean) / 3
  )

combo %>%
  forecast(h = "30 days")
```

You can also use regression–based combinations or more sophisticated weighting schemes via `regress_combination()` in fabletools.

13 Part II — Forecasting with modeltime & tidymodels

14 11. Time series splits & the ML mindset

Motivation

Machine–learning–style forecasting treats time series as supervised learning with strong temporal structure. The critical decision is the *evaluation protocol*; you must respect time when you split data.

```
library(tidymodels)
library(modeltime)
library(timetk)

data <- tibble(
  date = as.Date("2020-01-01") + 0:729,
  y    = sin(2*pi*(0:729)/7) + rnorm(730, 0, 0.2)
)

splits <- time_series_split(
  data,
  assess     = 90,
  cumulative = TRUE
)
```

- `training(splits)` gives history up to the cutoff.
- `testing(splits)` gives the final 90 days for evaluation.

---

15 12. A basic XGBoost forecast

Motivation

Gradient–boosted trees (XGBoost, LightGBM, CatBoost) are state of the art in many retail forecasting benchmarks when combined with good feature engineering.

```
rec <- recipe(y ~ date, data = training(splits)) %>%
  step_timeseries_signature(date) %>%
  step_rm(contains("hour"), contains("minute"), contains("second"), contains("am.pm")) %>%
  step_normalize(all_numeric_predictors())

xgb_spec <- boost_tree(
  trees      = 2000,
  learn_rate = 0.03,
  tree_depth = 8
) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

wf_xgb <- workflow() %>%
  add_recipe(rec) %>%
  add_model(xgb_spec)

fit_xgb <- fit(wf_xgb, training(splits))
```

Forecast and evaluate:

```
xgb_tbl <- modeltime_table(fit_xgb)

xgb_tbl %>%
  modeltime_calibrate(testing(splits)) %>%
  modeltime_forecast(
    new_data    = testing(splits),
    actual_data = data
  )
```

16 13. Adding a classical ARIMA baseline

Motivation

Even when you believe ML will win, you should verify it beats a decent statistical model.

```
fit_arima <- workflow() %>%
  add_recipe(rec) %>%
  add_model(
    arima_reg() %>%
      set_engine("auto_arima")
  ) %>%
  fit(training(splits))

model_tbl <- modeltime_table(
  fit_xgb,
  fit_arima
)

model_tbl %>%
  modeltime_calibrate(testing(splits)) %>%
  modeltime_accuracy()
```

You now have side–by–side accuracy for ML and classical methods.

---

17 14. Exogenous regressors: promos, prices, features

Motivation

Machine–learning models handle many regressors naturally. This is key for retail and supply–chain: promotions, prices, macro variables, calendar features, etc.

```
set.seed(1)

data_x <- data %>%
  mutate(
    promo = rbinom(n(), 1, 0.1),
    price = runif(n(), 9, 12)
  )

splits_x <- time_series_split(
  data_x,
  assess     = 90,
  cumulative = TRUE
)

rec_x <- recipe(y ~ ., data = training(splits_x)) %>%
  step_timeseries_signature(date) %>%
  step_rm(contains("hour"), contains("minute"), contains("second"), contains("am.pm")) %>%
  step_normalize(all_numeric_predictors())

wf_xgb_x <- workflow() %>%
  add_recipe(rec_x) %>%
  add_model(xgb_spec)

fit_xgb_x <- fit(wf_xgb_x, training(splits_x))
```

You can interpret feature importance with packages like `vip`, `DALEX`, or `iml`.

---

18 15. Time–aware hyperparameter tuning

Motivation

If you tune XGBoost with random cross–validation, you leak future information. Instead, use time–series CV (rolling origin) for tuning.

```
cv_folds <- time_series_cv(
  data_x,
  assess     = 90,
  skip       = 30,
  cumulative = TRUE
)

tuned <- tune_grid(
  wf_xgb_x,
  resamples = cv_folds,
  grid      = 20,
  control   = control_grid(verbose = TRUE)
)

best <- select_best(tuned, "rmse")

final_wf <- finalize_workflow(wf_xgb_x, best)
fit_final <- fit(final_wf, data_x)
```

The tuned workflow can then be used in `modeltime_table()` and productionised.

---

19 16. Ensembles in modeltime

Motivation

Ensembling multiple models (ML + classical) often yields robustness and extra accuracy at very low marginal cost.

```
model_tbl <- modeltime_table(
  fit_xgb_x,
  fit_arima
)

ensemble <- model_tbl %>%
  ensemble_average(type = "mean")

ensemble %>%
  modeltime_calibrate(testing(splits_x)) %>%
  modeltime_accuracy()
```

You can also use weighted ensembles or meta–learners via `stacks`.

---

20 17. Global models across many series

Motivation

Global models train one model across many series (e.g. SKUs, depots, stores), often outperforming per–series models when data is sparse.

```
data_panel <- tibble(
  sku   = rep(letters[1:20], each = 730),
  date  = rep(as.Date("2020-01-01") + 0:729, 20),
  y     = rnorm(20 * 730) + as.numeric(as.factor(sku)) * 0.2,
  promo = rbinom(20 * 730, 1, 0.1),
  price = runif(20 * 730, 9, 12)
)

splits_panel <- time_series_split(
  data_panel,
  assess     = 90,
  cumulative = TRUE
)

rec_panel <- recipe(y ~ sku + date + promo + price,
                    data = training(splits_panel)) %>%
  step_timeseries_signature(date) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_numeric_predictors())

wf_global <- workflow() %>%
  add_recipe(rec_panel) %>%
  add_model(xgb_spec)

fit_global <- fit(wf_global, training(splits_panel))
```

This XGBoost model learns patterns shared across SKUs and depots.

---

21 18. Prophet in modeltime

Motivation

Prophet is a popular model for business time series with strong seasonal patterns and holiday effects. `modeltime` wraps it in the same interface as your other models.

```r
library(prophet)

prop_spec <- prophet_reg(
  seasonality_yearly = TRUE,
  seasonality_weekly = TRUE,
  seasonality_daily  = FALSE,
  changepoint_num    = 25,
  changepoint_range  = 0.9
) %>%
  set_engine("prophet")

rec_prophet <- recipe(y ~ date, data = training(splits))  # Prophet needs date and y

wf_prophet <- workflow() %>%
  add_recipe(rec_prophet) %>%
  add_model(prop_spec)

fit_prophet <- fit(wf_prophet, training(splits))
```

Prophet with extra regressors:

```r
rec_prophet_x <- recipe(y ~ date + promo + price,
                        data = training(splits_x)) %>%
  step_mutate(
    promo = as.numeric(promo),
    price = price
  )

fit_prophet_x <- workflow() %>%
  add_recipe(rec_prophet_x) %>%
```

```
  add_model(prop_spec) %>%
  fit(training(splits_x))
```

Boosted Prophet (`prophet_boost()`) adds a gradient–boosting component on Prophet's residuals; the interface is analogous, with `set_engine("prophet_xgboost")`.

22 Part III — Extensions, retail patterns & feature engineering

23 19. Advanced timetk usage

Motivation

`timetk` is a powerful toolkit for time–based feature engineering, visualisation, and cross–validation. It plays very nicely with both tidymodels and direct data–frame workflows.

23.1 19.1 Time series signatures

You can generate time–based features directly, without recipes:

```
library(timetk)

data_sig <- data %>%
  tk_augment_timeseries_signature(date) %>%
  select(-contains("hour"), -contains("minute"),
         -contains("second"), -contains("am.pm"))
```

This yields fields like year, month, week, day of week, quarter, etc.

23.2 19.2 Lags and sliding windows

```
data_lags <- data_sig %>%
  tk_augment_lags(y, .lags = c(1, 7, 14))

data_roll <- data_lags %>%
  tk_augment_slidify(
    .value   = y,
    .f       = ~ mean(.x, na.rm = TRUE),
    .period = 7,
    .align   = "right",
    .partial = FALSE,
    .names   = "roll_mean_7"
  )
```

This produces lagged versions of y and a rolling 7–day mean.

23.3 19.3 Visualising time–series CV plans

```
cv_plan <- time_series_cv(
  data,
  assess     = 90,
  skip       = 30,
  cumulative = TRUE
)

cv_plan %>%
  tk_time_series_cv_plan() %>%
  plot_time_series_cv_plan(
    .date_var = date,
    .value    = y
  )
```

This allows you to inspect your rolling window splits visually and confirm they make sense.

---

24 20. Promotions, price & cannibalisation

Motivation

In retail, static seasonality is the easy part. The interesting dynamics come from promotions, pricing, and interactions across products (cannibalisation). We look at classical (fable) and ML (modeltime) approaches.

24.1 20.1 ARIMAX in fable for promo & price

```
data_retail <- tsibble(
  date   = as.Date("2020-01-01") + 0:729,
  sales  = rpois(730, 100),
  promo  = rbinom(730, 1, 0.1),
  price  = runif(730, 9, 12),
  index  = date
)

fit_arimax <- data_retail %>%
  model(
    arimax = ARIMA(sales ~ promo + price)
  )

fit_arimax %>% report()
```

Interpretation:

- Coefficient on `promo` average incremental units during promo.
- Coefficient on `price` price elasticity (units per price unit); often negative.

You can add lags and interactions, e.g. `promo + lag(promo) + price + promo:holiday_flag`.

24.2 20.2 ML global model for promos & price

```
data_panel <- tibble(
  sku   = rep(letters[1:20], each = 730),
  date  = rep(as.Date("2020-01-01") + 0:729, 20),
  sales = rpois(20 * 730, 100),
  promo = rbinom(20 * 730, 1, 0.1),
  price = runif(20 * 730, 9, 12)
)

splits_p <- time_series_split(
  data_panel,
  assess     = 90,
  cumulative = TRUE
)

rec_promos <- recipe(sales ~ sku + date + promo + price,
                     data = training(splits_p)) %>%
  step_timeseries_signature(date) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_numeric_predictors())

wf_promos <- workflow() %>%
  add_recipe(rec_promos) %>%
  add_model(xgb_spec)

fit_promos <- fit(wf_promos, training(splits_p))
```

For cannibalisation, you precompute features such as:

- `category_sales_ex_sku`
- `brand_sales`
- `competitor_price`

via group–by / lag operations, then feed into the recipe.

---

25 21. Feature engineering cookbook for ML time series

Motivation

Global ML models live or die on features. This chapter summarises common patterns you can compose.

25.1 21.1 Time–based features

- `step_timeseries_signature(date)` or `tk_augment_timeseries_signature()`.
- Fourier terms for long seasonalities (e.g. yearly with daily data).

25.2 21.2 Lags & rolling statistics

In recipes:

```
rec_lags <- rec %>%
  step_lag(y, lag = c(1, 7, 14)) %>%
  step_roll_mean(y, lag = 7, window = 7,
                 align = "right", id = "roll_mean_7") %>%
  step_roll_sd(y, lag = 7, window = 7,
               align = "right", id = "roll_sd_7")
```

These capture local dynamics that trees exploit well.

25.3 21.3 Calendar & holiday features

Use `step_holiday()` or precomputed holiday tables to add dummies for important days, long weekends, etc.

25.4 21.4 Price & promo–derived features

Examples:

- `discount_pct = pmax(0, (list_price - price) / list_price)`
- `promo_flag = as.integer(discount_pct > 0.1)`
- `price_index` vs category average.

25.5 21.5 Cross–series interaction features

Use group–by / summarise to compute, for each date and category:

- `category_sales`
- `category_sales_ex_sku`
- `top_brand_share`

Then join back to the main table and feed into your recipe.

---

26 22. Per–SKU vs global models

Motivation

A key design choice: do you fit a separate model per SKU (e.g., ARIMA in fable) or a global model (e.g., XGBoost in modeltime)? It's worth comparing them quantitatively.

26.1 22.1 Per–SKU ARIMA (fable)

```
ts_panel <- as_tsibble(
  data_panel,
  key   = sku,
  index = date
)

split_date <- as.Date("2021-12-31")

train_ts <- ts_panel %>%
  filter(date <= split_date)

test_ts <- ts_panel %>%
  filter(date > split_date)

fit_per <- train_ts %>%
  model(arima = ARIMA(sales))

fc_per <- fit_per %>%
  forecast(h = n_distinct(test_ts$date))

acc_per <- fc_per %>%
  accuracy(test_ts, by = "sku") %>%
  select(sku, rmse_per = RMSE)
```

26.2 22.2 Global XGBoost (modeltime)

```
splits_panel <- time_series_split(
  data_panel,
  assess     = sum(data_panel$date > split_date),
  cumulative = TRUE
)

rec_global <- recipe(sales ~ sku + date + promo + price,
                     data = training(splits_panel)) %>%
  step_timeseries_signature(date) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_numeric_predictors())

wf_global <- workflow() %>%
  add_recipe(rec_global) %>%
  add_model(xgb_spec)

fit_global <- fit(wf_global, training(splits_panel))

fc_global <- modeltime_table(fit_global) %>%
  modeltime_calibrate(testing(splits_panel)) %>%
  modeltime_forecast(
    new_data    = testing(splits_panel),
    actual_data = data_panel
  )
```

Map to per–SKU RMSE:

```
library(dplyr)
library(yardstick)

acc_global <- fc_global %>%
  group_by(sku) %>%
  summarise(
    rmse_global = rmse_vec(truth = .value, estimate = .pred)
  )

comparison <- acc_per %>%
  left_join(acc_global, by = "sku") %>%
  mutate(diff = rmse_per - rmse_global)
```

You can now see how many SKUs prefer global vs per–SKU models and by how much.

27 23. End–to–end engine pattern

Motivation

In production you want a repeatable pipeline: ingest data, build features, train models, backtest, and store configuration for deployment.

A minimal pattern:

1. **Process tables** (facts + dimensions).
2. **Feature builder** function.
3. **Modelling** function(s) (fable and/or modeltime).
4. **Backtest harness** (loop over cutoff dates).
5. **Model registry** (chosen config per segment).

Pseudocode for modelling step:

```
fit_forecast_models <- function(features, horizon, engine = c("fable", "modeltime")) {
  engine <- match.arg(engine)
  if (engine == "fable") {
    ts <- features %>%
      as_tsibble(key = c(sku, depot), index = date)

    ts %>% model(ARIMA(qty))
  } else {
    splits <- time_series_split(features, assess = horizon, cumulative = TRUE)

    rec <- recipe(qty ~ ., data = training(splits)) %>%
      step_timeseries_signature(date) %>%
      step_dummy(all_nominal_predictors()) %>%
      step_normalize(all_numeric_predictors())

    wf <- workflow() %>%
      add_recipe(rec) %>%
      add_model(xgb_spec)

    fit(wf, training(splits))
```

```
  }
}
```

You then wrap this in backtesting and orchestration appropriate to your environment (e.g., cron, Airflow, Hudson–style process orchestrator).

28 Part IV — AutoML & H2O via modeltime.h2o

29 24. H2O & AutoML for forecasting

Motivation

Sometimes you want to search over many model classes automatically and get a strong baseline without hand–tuning. **H2O AutoML** is a mature AutoML system for tabular data. `modeltime.h2o` plugs it into the modeltime workflow, giving you AutoML–style model search while keeping time–series semantics in your resampling and features.

---

30 25. Initialising H2O

```
library(h2o)
library(modeltime.h2o)

h2o.init()
```

You can control memory and cluster size via arguments to `h2o.init()` if needed.

---

31 26. A basic AutoML forecasting workflow

We reuse the `data_x` example with `y`, `date`, `promo`, and `price`.

```
splits_x <- time_series_split(
  data_x,
  assess     = 90,
  cumulative = TRUE
)

rec_h2o <- recipe(y ~ ., data = training(splits_x)) %>%
  step_timeseries_signature(date) %>%
  step_rm(contains("hour"), contains("minute"), contains("second"), contains("am.pm")) %>%
  step_normalize(all_numeric_predictors())
```

Specify an AutoML regression model:

```
h2o_spec <- automl_reg(
  max_runtime_mins = 10,
  max_models       = 20,
  seed             = 123
) %>%
  set_engine("h2o")
```

Build workflow and fit:

```
wf_h2o <- workflow() %>%
  add_recipe(rec_h2o) %>%
  add_model(h2o_spec)

fit_h2o <- wf_h2o %>%
  fit(training(splits_x))
```

Evaluate:

```r
modeltime_table(fit_h2o) %>%
  modeltime_calibrate(testing(splits_x)) %>%
  modeltime_accuracy()
```

Forecast:

```r
modeltime_table(fit_h2o) %>%
  modeltime_calibrate(testing(splits_x)) %>%
  modeltime_forecast(
    new_data    = testing(splits_x),
    actual_data = data_x
  )
```

---

32 27. What H2O AutoML is doing under the hood

- Splits the training data internally into training/validation.
- Tries many model classes: GBM, Random Forest, GLM, deep learning, stacked ensembles, etc.
- Ranks them on a validation metric (e.g. RMSE or MAE).
- Returns a "leader" model that is exposed through the modeltime interface.

For deeper inspection, you can access the underlying H2O objects:

```
leaderboard <- h2o.get_leaderboard()
leaderboard
```

---

33 28. Time–series semantics & caveats

Important

H2O AutoML itself does **not** understand time. It sees a supervised regression problem. The time–series semantics come from:

- How you create features (lags, rolling stats, time signatures, promo/price variables).
- How you split and resample data (`time_series_split()`, `time_series_cv()`).

Guidelines:

- Always use **time–based splits** for evaluation.
- Prefer to handle leakage–sensitive operations (like scaling and lagging) via recipes or timetk, not inside AutoML.
- Be cautious with automatic random CV inside H2O; keep your main validation loop in your R/rsample layer.

---

34 29. When to use H2O AutoML vs manual models

Use AutoML when:

- You want a strong baseline quickly.
- You're exploring a new dataset and want to know what is achievable.
- You're happy with a "black–box-ish" model bundle and care more about accuracy than fine control.

Use manual models (fable/modeltime) when:

- You want explicit control over model class and structure (e.g. ARIMAX vs XGBoost vs DeepAR).
- Interpretability and diagnostics matter (especially in trade/revenue discussions).
- You have custom constraints or loss functions that AutoML does not support.

In practice, a good pattern is:

1. Start with **fable** models and naive baselines for sanity.
2. Add **modeltime** global ML models (e.g. XGBoost / LightGBM) with well–engineered features.
3. Use **H2O AutoML** as an additional candidate in your modeltime ensembles.
4. Select the combination that works best across your backtests and business constraints.