

Ageing is not just ageing Data Analysis Pipeline

Xavi Marsellach

2025-11-13

Reconstruction of AMAV_DATA.xlsx file from published Supplemental_Table_1.xlsx:

Purpose

Prevalence data for **31 diseases** are provided in a single consolidated workbook, **Supplemental_Table_1.xlsx**. The Python script in this section reconstructs the **AMAV lines** directly from **Supplemental_Table_1.xlsx**.

R Code

```
# Works regardless of whether the QMD runs from repo root or inside 'analysis/'.

# --- Find repo root: directory that contains 'scripts/' and 'data/' ---
find_root <- function() {
  cand <- c(".", "..", "../..")
  for (c in cand) {
    if (file.exists(file.path(c, "scripts", "build_amav_from_supplemental_Table_1.py")) &&
        file.exists(file.path(c, "data", "Supplemental_Table_1.xlsx"))) {
      return(normalizePath(c))
    }
  }
  stop("Can't locate repository root (need 'scripts/' and 'data/').")
}
root <- find_root()

# --- Paths (absolute) ---
script <- file.path(root, "scripts", "build_amav_from_supplemental_Table_1.py")
inp    <- file.path(root, "data", "Supplemental_Table_1.xlsx")
outdir <- file.path(root, "output")
out    <- file.path(outdir, "AMAV_DATA.xlsx")
dir.create(outdir, showWarnings = FALSE, recursive = TRUE)
```

```
# --- Choose Python interpreter ---
py <- Sys.getenv("PYTHON"); if (!nzchar(py)) py <- Sys.which("python3")
stopifnot(nzchar(py), file.exists(py))

# --- Run and show logs ---
cmd_args <- c(script, inp, out)
cat("$", py, paste(cmd_args, collapse = " "), "\n")
```

```
$ /Users/francesco/miniforge3/envs/ageing-amav/bin/python3 /Users/francesco/Desktop/Github/sc
res <- system2(py, args = cmd_args, stdout = TRUE, stderr = TRUE)
cat(paste(res, collapse = "\n"), "\n")
```

OK: created /Users/francesco/Desktop/Github/output/AMAV_DATA.xlsx

```
# --- Verify output ---
stopifnot(file.exists(out))
cat(sprintf(" OK: created %s\n", out))
```

OK: created /Users/francesco/Desktop/Github/output/AMAV_DATA.xlsx

Code for build_amav_from_supplemental_Table_1.py

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 """
5 Rebuild AMAV_DATA.xlsx from a single-table workbook (Supplemental_Table_1.xlsx).
6
7 Input expectation (neutral description):
8 - One Excel workbook containing a single wide table with:
9     Phenotype | Citation | 1941 | 1942 | ... | 2023
10 - Each row is one prevalence trend (study) for a given phenotype/disease.
11 - A "Citation" column may be present and is ignored for calculations.
12
13 What the script computes:
14 - For each phenotype, piecewise-linear yearly slopes per study between observed points.
15 - Yearly MAV (mean of study slopes).
16 - AMAV: cumulative MAV restricted to the last contiguous block of defined MAV.
17 - AMAV-POS: if AMAV in that block dips below zero, a positive-limb accumulation
18     starting after the AMAV valley (MODE='prev'), optionally including negative MAV.
19 - Aggregated outputs:
20     • AMAV-P : one column per phenotype (indexed by Year) using AMAV-POS if it exists,
21       otherwise AMAV.
22     • AMAV-F-Y : AMAV-P normalised by the first positive value (by Year).
23     • AMAV-F-R : Same normalisation but re-indexed 0..n per phenotype (relative time).
```

```

24     • LOG_used_column : Phenotype name and number of trends parsed.
25
26 Default paths (relative to current working directory):
27 - Input   : data/Supplemental_Table_1.xlsx      (auto-discovery fallbacks included)
28 - Output  : output/AMAV_DATA.xlsx              (directory is created if missing)
29
30 CLI:
31     python scripts/build_amav_from_supplemental_Table_1.py [input.xlsx] [output.xlsx]
32
33 Programmatic (e.g., from R with reticulate):
34     from scripts.build_amav_from_supplemental_Table_1 import build_amav
35     out_path = build_amav() # or build_amav("path/to/input.xlsx", "output/AMAV_DATA.xlsx")
36     """
37
38 from __future__ import annotations
39
40 from pathlib import Path
41 import math
42 import re
43 import sys
44 import warnings
45 from typing import Dict, List, Tuple
46
47 import numpy as np
48 import pandas as pd
49
50 warnings.filterwarnings("ignore")
51
52 # ---- Behaviour switches ----
53 MODE = "prev"          # accumulate MAV from the previous year
54 INCLUDE_NEG = True     # include negative MAV in AMAV-POS accumulation
55 EPS = 1e-12
56
57 # ---- Default locations (relative to working directory) ----
58 DEFAULT_INPUT  = Path("data/Supplemental_Table_1.xlsx")
59 DEFAULT_OUTPUT = Path("output/AMAV_DATA.xlsx")
60
61
62 # ----- Utilities -----
63 def to_number(series: pd.Series) -> pd.Series:
64     """Robust numeric conversion: handles %, comma decimals, and thousand separators."""
65     def _one(x):
66         if pd.isna(x):
67             return np.nan
68         if isinstance(x, (int, float, np.number)):

```

```

69         return float(x)
70     s = str(x).strip().replace(" ", "")
71     is_pct = s.endswith("%")
72     s = s.replace("%", "").replace(",", ".")
73     # remove thousands separators like 1.234 or 1'234
74     s = re.sub(r"(?<=\d)[\.,'](?=\d{3}\b)", "", s)
75     try:
76         v = float(s)
77     except ValueError:
78         return np.nan
79     return v / 100.0 if is_pct else v
80 return series.apply(_one)
81
82
83 def runs_true(mask: np.ndarray) -> List[Tuple[int, int]]:
84     """Return [(start, end)] index intervals where mask is True and contiguous."""
85     idx = np.where(mask)[0]
86     if idx.size == 0:
87         return []
88     runs, start, prev = [], idx[0], idx[0]
89     for k in idx[1:]:
90         if k == prev + 1:
91             prev = k
92         else:
93             runs.append((start, prev))
94             start = prev = k
95     runs.append((start, prev))
96     return runs
97
98
99 # ----- AMAV / AMAV-POS core -----
100 def build_perstudy_linearslope(raw_df: pd.DataFrame) -> pd.DataFrame:
101     """
102     raw_df columns: Year | T1 | T2 | ... | Tk
103     For each trend, build a piecewise-constant slope between successive observed points.
104     """
105     df = raw_df.sort_values("Year").reset_index(drop=True)
106     years = df["Year"].to_numpy()
107     out = pd.DataFrame({"Year": years})
108
109     for col in df.columns:
110         if col == "Year":
111             continue
112         s = pd.to_numeric(df[col], errors="coerce")
113         mask = s.notna().to_numpy()

```

```

114         obs_y = years[mask]
115         obs_v = s[mask].to_numpy(float)
116
117         arr = np.full(len(years), np.nan, float)
118         for i in range(len(obs_y) - 1):
119             y0, y1 = obs_y[i], obs_y[i + 1]
120             v0, v1 = obs_v[i], obs_v[i + 1]
121             if y1 > y0:
122                 slope = (v1 - v0) / (y1 - y0)
123                 seg = (years > y0) & (years <= y1)
124                 arr[seg] = slope
125         out[col] = arr
126
127     return out
128
129
130 def build_summary_mav_amav(slope_df: pd.DataFrame) -> pd.DataFrame:
131     """
132     - MAV: mean across study slopes per year.
133     - AMAV: cumulative MAV within the last contiguous block where MAV is defined.
134     - AMAV-POS: if AMAV dips below zero in that block, accumulate MAV from the year
135       after the valley (MODE='prev'), including negative MAV if INCLUDE_NEG=True.
136     """
137     df = slope_df.sort_values("Year").reset_index(drop=True)
138     years = df["Year"].to_numpy()
139
140     study_cols = [c for c in df.columns if c != "Year"]
141     M = df[study_cols].to_numpy(float)
142
143     datapoints = np.sum(~np.isnan(M), axis=1)
144     with np.errstate(all="ignore"):
145         mav = np.nanmean(M, axis=1)
146
147     # AMAV within the last contiguous MAV block
148     amav = np.full(len(years), np.nan, float)
149     finite_mav = ~np.isnan(mav)
150     mav_runs = runs_true(finite_mav)
151     last_s = last_e = None
152     if mav_runs:
153         last_s, last_e = mav_runs[-1]
154         acc = 0.0
155         # i from last_s+1 .. last_e+1 inclusive (AMAV[i] integrates mav[i-1])
156         for i in range(last_s + 1, last_e + 2):
157             prev = i - 1
158             if not math.isnan(mav[prev]):

```

```

159         acc += mav[prev]
160         amav[i] = acc
161
162     summary = pd.DataFrame({
163         "Year": years,
164         "DataPoints": datapoints,
165         "MAV": mav,
166         "AMAV": amav,
167     })
168
169     # AMAV-POS only if AMAV dips below zero in the final block
170     if last_s is None:
171         return summary
172
173     lo, hi = last_s + 1, last_e + 1 # indices where AMAV is defined
174     window_am = amav[lo:hi + 1]
175     has_negative = np.isfinite(window_am).any() and (window_am[np.isfinite(window_am)] < -EP
176     if not has_negative:
177         return summary
178
179     valley_rel = int(np.nanargmin(window_am))
180     valley = lo + valley_rel
181
182     amav_pos = np.full(len(amav), np.nan, float)
183     run = 0.0
184
185     # Start at valley+1 for MODE='prev'
186     start_i = max(valley + 1, lo) if MODE == "prev" else max(valley, lo)
187     for i in range(start_i, hi + 1):
188         idx_mav = (i - 1) if MODE == "prev" else i
189         v = mav[idx_mav]
190         if np.isfinite(v):
191             contrib = v if INCLUDE_NEG else max(v, 0.0)
192             run += contrib
193             amav_pos[i] = run
194
195     insert_at = list(summary.columns).index("AMAV") + 1
196     summary.insert(insert_at, "AMAV-POS", amav_pos)
197     return summary
198
199
200 def first_positive_baseline(s: pd.Series) -> float:
201     """First non-zero, non-NA value."""
202     for v in s:
203         if pd.notna(v) and v != 0:

```

```

204         return float(v)
205     return np.nan
206
207
208 # ----- Read single table from Supplemental_Table_1 -----
209 def find_table_sheet(xlsx: Path) -> str:
210     """Find the sheet whose (0,0) cell is 'Phenotype' (fallback: first sheet)."""
211     xls = pd.ExcelFile(xlsx)
212     for sh in xls.sheet_names:
213         test = pd.read_excel(xlsx, sheet_name=sh, nrows=1, header=None)
214         if not test.empty and str(test.iat[0, 0]).strip().lower() == "phenotype":
215             return sh
216     return xls.sheet_names[0]
217
218
219 def read_single_table(input_xlsx: Path) -> tuple[pd.DataFrame, List[int]]:
220     """
221     Returns:
222     - df: columns = Phenotype | <year_int_1> | ... | <year_int_k>
223     - year_order: sorted list of integer years
224
225     Drops the 'Citation' column. Removes a header-like duplicate row if needed.
226     """
227     sheet = find_table_sheet(input_xlsx)
228     df = pd.read_excel(input_xlsx, sheet_name=sheet, header=0)
229     if df.empty:
230         raise SystemExit("Input sheet appears to be empty.")
231
232     if "Phenotype" not in df.columns:
233         raise SystemExit("Cannot find 'Phenotype' column in header.")
234
235     # Drop Citation column if present
236     drop_cit = [c for c in df.columns if str(c).strip().lower() == "citation"]
237     if drop_cit:
238         df = df.drop(columns=drop_cit)
239
240     # Identify year columns (numeric or string)
241     year_cols_orig = []
242     for c in df.columns:
243         if c == "Phenotype":
244             continue
245         if isinstance(c, (int, float)) and 1800 <= int(c) <= 2100:
246             year_cols_orig.append(c)
247         elif isinstance(c, str) and re.fullmatch(r"\s*\d{4}\s*", c or ""):
248             year_cols_orig.append(c)

```

```

249
250     if not year_cols_orig:
251         raise SystemExit("No year-like columns found in the table.")
252
253     # Normalize year columns to integer names
254     year_map = {c: int(str(c).strip()) for c in year_cols_orig}
255     df = df.rename(columns=year_map)
256     year_cols = sorted(year_map.values())
257
258     # Keep only Phenotype + years
259     keep = ["Phenotype"] + year_cols
260     df = df[keep].copy()
261     df["Phenotype"] = df["Phenotype"].astype(str).str.strip()
262
263     # Remove bogus header-like row
264     df = df[df["Phenotype"].str.strip().str.lower() != "phenotype"]
265
266     # Coerce year values to numeric
267     for c in year_cols:
268         df[c] = to_number(df[c])
269
270     return df, year_cols
271
272
273 def build_disease_series(df: pd.DataFrame, year_order: List[int]) -> Dict[str, pd.Series]:
274     """
275     For each phenotype, build a per-disease AMAV/AMAV-POS series,
276     preferring AMAV-POS when present, else AMAV.
277     """
278     series_by: Dict[str, pd.Series] = {}
279     years = list(year_order)
280
281     for disease, grp in df.groupby("Phenotype", dropna=True):
282         raw = pd.DataFrame({"Year": years})
283         # Each row in grp is one trend (T1..Tk)
284         for i, (_, row) in enumerate(grp.iterrows(), start=1):
285             raw[f"T{i}"] = row[years].values
286
287         # Skip if all trends are NA
288         if raw.drop(columns=["Year"]).isna().all().all():
289             continue
290
291         slope = build_perstudy_linearslope(raw)
292         summary = build_summary_mav_amav(slope)
293

```



```

294         chosen_col = "AMAV-POS" if ("AMAV-POS" in summary.columns and summary["AMAV-POS"].n
295         ser = summary.set_index("Year")[chosen_col].copy()
296         ser.name = disease
297         series_by[disease] = ser
298
299     return series_by
300
301
302 # ----- I/O helpers -----
303 def resolve_input_output(
304     input_xlsx: Path | None,
305     output_xlsx: Path | None
306 ) -> tuple[Path, Path]:
307     """Resolve paths with sensible defaults and fallbacks."""
308     cwd = Path.cwd()
309
310     # Input resolution
311     if input_xlsx is None:
312         candidates = [
313             cwd / "data" / "Supplemental_Table_1.xlsx",
314             cwd / "analysis" / "data" / "Supplemental_Table_1.xlsx",
315             cwd / "Supplemental_Table_1.xlsx",
316         ]
317         inp = next((p for p in candidates if p.exists()), None)
318         if inp is None:
319             raise SystemExit(
320                 "Input file not found. Expected at one of: "
321                 "data/Supplemental_Table_1.xlsx, analysis/data/Supplemental_Table_1.xlsx, ."
322             )
323         else:
324             inp = Path(input_xlsx)
325             if not inp.exists():
326                 raise SystemExit(f"Input file not found: {inp}")
327
328     # Output resolution
329     out = Path(output_xlsx) if output_xlsx is not None else (cwd / DEFAULT_OUTPUT)
330     out.parent.mkdir(parents=True, exist_ok=True)
331
332     return inp, out
333
334
335 def write_workbook(amavp_df: pd.DataFrame, folds_y: pd.DataFrame,
336                   folds_r: pd.DataFrame, log_df: pd.DataFrame, out_path: Path) -> None:
337     """Write the 4 output sheets to an Excel workbook."""
338     with pd.ExcelWriter(out_path, engine="xlsxwriter") as xw:

```

```

339         amavp_df.reset_index().rename(columns={"index": "Year"}).to_excel(
340             xw, index=False, sheet_name="AMAV-P"
341         )
342         folds_y.reset_index().rename(columns={"index": "Year"}).to_excel(
343             xw, index=False, sheet_name="AMAV-F-Y"
344         )
345         folds_r.rename_axis("RelYear").reset_index().to_excel(
346             xw, index=False, sheet_name="AMAV-F-R"
347         )
348         log_df.to_excel(xw, index=False, sheet_name="LOG_used_column")
349
350
351 # ----- Orchestrator -----
352 def build_amav(input_xlsx: str | Path | None = None,
353               output_xlsx: str | Path | None = None) -> Path:
354     """
355     High-level entry point. Returns the Path of the created Excel workbook.
356     """
357     inp, out = resolve_input_output(
358         Path(input_xlsx) if input_xlsx is not None else None,
359         Path(output_xlsx) if output_xlsx is not None else None,
360     )
361
362     df, year_order = read_single_table(inp)
363     series_by = build_disease_series(df, year_order)
364     if not series_by:
365         raise SystemExit("No diseases could be parsed into AMAV/AMAV-POS series.")
366
367     # AMAV-P: union of years across all diseases
368     all_years = sorted(set().union(*[set(s.index) for s in series_by.values()]))
369     amavp_df = pd.DataFrame(index=all_years)
370     for disease, s in series_by.items():
371         amavp_df[disease] = s.reindex(all_years)
372
373     # AMAV-F-Y: normalised by first positive baseline, indexed by Year
374     folds_y = pd.DataFrame(index=amavp_df.index)
375     for col in amavp_df.columns:
376         v = amavp_df[col]
377         b = first_positive_baseline(v.dropna())
378         folds_y[col] = v / b if pd.notna(b) else np.nan
379
380     # AMAV-F-R: same but re-indexed 0..n per disease
381     rel_dict: Dict[str, pd.Series] = {}
382     max_len = 0
383     for col in amavp_df.columns:

```

```

384         v = amavp_df[col].dropna()
385         b = first_positive_baseline(v)
386         rel = (v / b) if pd.notna(b) else pd.Series(dtype=float)
387         rel.index = range(len(rel))
388         rel_dict[col] = rel
389         max_len = max(max_len, len(rel))
390
391     folds_r = pd.DataFrame(index=range(max_len))
392     for col, s in rel_dict.items():
393         folds_r[col] = s.reindex(folds_r.index)
394
395     # LOG sheet: phenotype + number of trends parsed
396     log_df = (
397         df.groupby("Phenotype", dropna=True)["Phenotype"]
398         .count()
399         .rename("Trends")
400         .reset_index()
401         .rename(columns={"Phenotype": "Disease"})
402         .sort_values("Disease")
403     )
404
405     write_workbook(amavp_df, folds_y, folds_r, log_df, out)
406     print(f" OK: created {out}")
407     return out
408
409
410 # ----- CLI -----
411 def main() -> None:
412     # CLI overrides: [input.xlsx] [output.xlsx]
413     argv = sys.argv
414     in_arg = Path(argv[1]) if len(argv) >= 2 else None
415     out_arg = Path(argv[2]) if len(argv) >= 3 else None
416     build_amav(in_arg, out_arg)
417
418
419 if __name__ == "__main__":
420     main()

```