S1_stats_functions.py

Source listing · 280 lines · generated 2026-07-28

```
"""
S1_stats_functions.py
--------------------------------------------------------------------------
Author-written statistical routines used in:
"Metabolic syndrome-like phenotype, BMI-based metabolic burden, and
 size-threshold thyroid nodules: a cross-sectional study of 50,023
 health-examination attendees."

All estimators are implemented from first principles in NumPy (no statsmodels /
sklearn), so that every reported model is fully specified and reproducible.

Contents
    logit(y, X)              logistic regression (Newton-Raphson / IRLS) -> OR
    poisson_robust(y, X)     modified Poisson w/ robust (sandwich) variance -> PR/RR
    poisson_offset(y, X, off) Poisson rate model with log person-time offset -> IRR
    vif(X)                   variance inflation factors
    rcs_basis(x, knots)      restricted cubic spline basis (Harrell) + Wald test helper
    cochran_armitage(...)    trend test in ordered proportions
    ttest / prop_test        group comparison with standardized mean difference (SMD)
    evalue(estimate)         VanderWeele & Ding E-value
    mice(df, impute, fixed)  multiple imputation by chained equations (+ Rubin pooling)

Python 3.10, NumPy >= 1.24.
--------------------------------------------------------------------------
"""
import numpy as np
import math


# ---------------------------------------------------------------------- helpers
def _phi(z):
    """Standard normal CDF via the error function."""
    return 0.5 * (1.0 + math.erf(z / math.sqrt(2.0)))


def pval_z(z):
    """Two-sided p-value for a z statistic."""
    return 2.0 * (1.0 - _phi(abs(z)))


def _design(X):
    X = np.asarray(X, float)
    if X.ndim == 1:
        X = X[:, None]
    return np.column_stack([np.ones(len(X)), X])          # add intercept


def _pack(beta, se, names):
```

```
    """Assemble a coefficient dict with ratio (exp), 95% CI and p-value."""
    out = {"intercept": {"coef": beta[0], "se": se[0]}}
    for i, nm in enumerate(names):
        c, s = beta[i + 1], se[i + 1]
        z = c / s
        out[nm] = {"coef": c, "se": s,
                   "ratio": math.exp(c),
                   "lo": math.exp(c - 1.96 * s),
                   "hi": math.exp(c + 1.96 * s),
                   "z": z, "p": pval_z(z)}
    return out


# ---------------------------------------------------------------------------- logistic
def logit(y, X, names=None, maxit=100, tol=1e-9):
    """Binary logistic regression by Newton-Raphson (Fisher scoring / IRLS).

    Returns a dict keyed by predictor name; each entry has the odds ratio
    ('ratio'), its 95% CI ('lo','hi') and Wald p-value ('p').
    """
    y = np.asarray(y, float)
    Xd = _design(X)
    b = np.zeros(Xd.shape[1])
    for _ in range(maxit):
        eta = np.clip(Xd @ b, -30, 30)
        p = 1.0 / (1.0 + np.exp(-eta))
        W = np.clip(p * (1 - p), 1e-9, None)
        H = Xd.T @ (Xd * W[:, None])                    # observed information
        g = Xd.T @ (y - p)                              # score
        step = np.linalg.solve(H, g)
        b += step
        if np.max(np.abs(step)) < tol:
            break
    cov = np.linalg.inv(H)
    se = np.sqrt(np.diag(cov))
    names = names or [f"x{i}" for i in range(Xd.shape[1] - 1)]
    res = _pack(b, se, names)
    res["_cov"] = cov                                   # full covariance (for joint tests)
    return res


# ---------------------------------------------------------------- modified Poisson (PR)
def poisson_robust(y, X, names=None, maxit=100, tol=1e-8):
    """Poisson regression with a log link and robust (sandwich) variance.

    For a binary outcome this is Zou's 'modified Poisson' estimator of the
    prevalence / risk ratio (Am J Epidemiol 2004;159:702-6).
    """
    y = np.asarray(y, float)
    Xd = _design(X)
    b = np.zeros(Xd.shape[1])
    for _ in range(maxit):
        mu = np.exp(np.clip(Xd @ b, -30, 30))
```

```
        H = Xd.T @ (Xd * mu[:, None])
        g = Xd.T @ (y - mu)
        step = np.linalg.solve(H, g)
        b += step
        if np.max(np.abs(step)) < tol:
            break
    mu = np.exp(np.clip(Xd @ b, -30, 30))
    bread = np.linalg.inv(Xd.T @ (Xd * mu[:, None]))
    meat = Xd.T @ (Xd * ((y - mu) ** 2)[:, None])          # robust (HC0) meat
    cov = bread @ meat @ bread
    se = np.sqrt(np.diag(cov))
    return _pack(b, se, names or [f"x{i}" for i in range(Xd.shape[1] - 1)])


def poisson_offset(y, X, log_time, names=None, maxit=100, tol=1e-9):
    """Poisson rate model with a log person-time offset and robust variance
    (incidence-rate ratios). Used in the companion cohort analysis."""
    y = np.asarray(y, float)
    off = np.asarray(log_time, float)
    Xd = _design(X)
    b = np.zeros(Xd.shape[1])
    for _ in range(maxit):
        mu = np.exp(np.clip(off + Xd @ b, -30, 30))
        H = Xd.T @ (Xd * mu[:, None])
        g = Xd.T @ (y - mu)
        step = np.linalg.solve(H, g)
        b += step
        if np.max(np.abs(step)) < tol:
            break
    mu = np.exp(np.clip(off + Xd @ b, -30, 30))
    bread = np.linalg.inv(Xd.T @ (Xd * mu[:, None]))
    meat = Xd.T @ (Xd * ((y - mu) ** 2)[:, None])
    cov = bread @ meat @ bread
    se = np.sqrt(np.diag(cov))
    return _pack(b, se, names or [f"x{i}" for i in range(Xd.shape[1] - 1)])


# ---------------------------------------------------------------------------- collinearity
def vif(X):
    """Variance inflation factor for each column of X (design without intercept)."""
    X = np.asarray(X, float)
    out = {}
    for j in range(X.shape[1]):
        y = X[:, j]
        Z = np.column_stack([np.ones(len(X)), np.delete(X, j, axis=1)])
        beta, *_ = np.linalg.lstsq(Z, y, rcond=None)
        r2 = 1 - ((y - Z @ beta) ** 2).sum() / ((y - y.mean()) ** 2).sum()
        out[j] = np.inf if r2 >= 1 else 1.0 / (1.0 - r2)
    return out


# ---------------------------------------------------------------- restricted cubic spline
def rcs_basis(x, knots):
```

```
    """Harrell's restricted cubic spline basis. k knots -> (k-1) columns
    (a linear term plus k-2 nonlinear terms)."""
    x = np.asarray(x, float)
    t = np.asarray(knots, float)
    k = len(t)
    cols = [x]
    for j in range(k - 2):
        d1 = np.clip(x - t[j], 0, None) ** 3
        d2 = np.clip(x - t[k - 2], 0, None) ** 3 * (t[k - 1] - t[j]) / (t[k - 1] - t[k - 2])
        d3 = np.clip(x - t[k - 1], 0, None) ** 3 * (t[k - 2] - t[j]) / (t[k - 1] - t[k - 2])
        cols.append((d1 - d2 + d3) / (t[k - 1] - t[0]) ** 2)
    return np.column_stack(cols)


def joint_wald_p(beta, cov, df):
    """Joint Wald test p-value for `df` coefficients (used to test spline
    non-linearity). For df=2 the chi-square survival is exp(-W/2)."""
    W = beta @ np.linalg.inv(cov) @ beta
    if df == 2:
        return math.exp(-W / 2.0)
    # general df: Wilson-Hilferty approximation to the chi-square tail
    z = ((W / df) ** (1 / 3) - (1 - 2 / (9 * df))) / math.sqrt(2 / (9 * df))
    return 1 - _phi(z)


# ------------------------------------------------------------------------- trend test
def cochran_armitage(counts, totals, scores=None):
    """Cochran-Armitage test for a linear trend in ordered proportions."""
    counts = np.asarray(counts, float)
    totals = np.asarray(totals, float)
    if scores is None:
        scores = np.arange(len(counts), dtype=float)
    N = totals.sum()
    p = counts.sum() / N
    sbar = (totals * scores).sum() / N
    num = (counts * (scores - sbar)).sum()
    var = p * (1 - p) * (totals * (scores - sbar) ** 2).sum()
    z = num / math.sqrt(var)
    return z, pval_z(z)


# ------------------------------------------------------------------ descriptive comparisons
def ttest(a, b):
    """Two-sample t-test plus the standardized mean difference (SMD)."""
    a = np.asarray(a, float); a = a[~np.isnan(a)]
    b = np.asarray(b, float); b = b[~np.isnan(b)]
    ma, mb = a.mean(), b.mean()
    va, vb = a.var(ddof=1), b.var(ddof=1)
    na, nb = len(a), len(b)
    t = (ma - mb) / math.sqrt(va / na + vb / nb)
    sp = math.sqrt(((na - 1) * va + (nb - 1) * vb) / (na + nb - 2))
    smd = (ma - mb) / sp if sp > 0 else float("nan")
    return dict(m1=ma, s1=math.sqrt(va), n1=na, m2=mb, s2=math.sqrt(vb), n2=nb,
```

```
                t=t, p=pval_z(t), smd=smd)


def prop_test(x1, n1, x2, n2):
    """Two-proportion z-test plus the standardized mean difference."""
    p1, p2 = x1 / n1, x2 / n2
    p = (x1 + x2) / (n1 + n2)
    z = (p1 - p2) / math.sqrt(p * (1 - p) * (1 / n1 + 1 / n2))
    smd = (p1 - p2) / math.sqrt(p * (1 - p)) if 0 < p < 1 else float("nan")
    return dict(p1=p1, p2=p2, z=z, p=pval_z(z), smd=smd)


# ------------------------------------------------------------------------- E-value
def evalue(rr):
    """VanderWeele & Ding E-value for a risk/prevalence ratio (Ann Intern Med
    2017). For an odds ratio with a common outcome, first convert to an
    approximate risk ratio with `or_to_rr` below."""
    rr = rr if rr >= 1 else 1.0 / rr
    return rr + math.sqrt(rr * (rr - 1))


def or_to_rr(odds_ratio, p0):
    """Convert an odds ratio to an approximate risk ratio given the baseline
    (unexposed) risk p0."""
    return odds_ratio / ((1 - p0) + p0 * odds_ratio)


# ---------------------------------------------------------- multiple imputation (MICE)
def mice(df, impute, fixed, m=20, iters=10, seed=2024):
    """Multiple imputation by chained equations with predictive-mean-style
    linear models, returning a list of `m` completed copies of `df`.

    impute : columns with missing values to be imputed
    fixed  : fully observed predictors always included (e.g., age, sex, outcome)

    Each incomplete variable is regressed on all other imputed variables plus
    the fixed predictors; missing values are drawn from the fitted model with
    added residual noise. Pool downstream estimates with Rubin's rules.
    """
    import pandas as pd
    rng = np.random.default_rng(seed)
    miss = {v: df[v].isna().values for v in impute}
    base = df.copy()
    for v in impute:                                    # mean start values
        base[v] = base[v].fillna(df[v].mean())
    completed = []
    for _ in range(m):
        cur = base.copy()
        for _ in range(iters):
            for v in impute:
                if not miss[v].any():
                    continue
                pred = [p for p in impute if p != v] + list(fixed)
```

```
                obs = ~miss[v]
                Xo = np.column_stack([np.ones(obs.sum())] + [cur.loc[obs, p].values for p in pred])
                beta, *_ = np.linalg.lstsq(Xo, cur.loc[obs, v].values, rcond=None)
                sd = (cur.loc[obs, v].values - Xo @ beta).std()
                Xm = np.column_stack([np.ones(miss[v].sum())] + [cur.loc[miss[v], p].values for p in pred])
                cur.loc[miss[v], v] = Xm @ beta + rng.normal(0, sd, miss[v].sum())
        completed.append(cur)
    return completed


def rubin_pool(log_estimates, log_ses):
    """Combine log-scale estimates and standard errors across imputations
    (Rubin's rules); returns (ratio, lo, hi)."""
    q = np.asarray(log_estimates, float)
    u = np.asarray(log_ses, float) ** 2
    m = len(q)
    qbar = q.mean()
    T = u.mean() + (1 + 1 / m) * q.var(ddof=1)
    se = math.sqrt(T)
    return math.exp(qbar), math.exp(qbar - 1.96 * se), math.exp(qbar + 1.96 * se)
```