```python
#!/usr/bin/env python3
from __future__ import annotations

import argparse
import json
import os
import time
from pathlib import Path
from typing import Any, Dict, TypedDict, Literal

import numpy as np
import pandas as pd
import scanpy as sc  # type: ignore
import yaml
from numpy.typing import NDArray
from sklearn.decomposition import NMF

# ---- Typed config ---------------------------------------------------------------

InitType = Literal["random", "nndsvd", "nndsvda", "nndsvdar", "custom"]


class CNMFParams(TypedDict):
    K: int
    max_iter: int
    init: InitType
    random_state: int


class Params(TypedDict):
    hvg_n_top_genes: int
    pct_mito_max: float
    min_genes: int
    cnmf: CNMFParams


def _coerce_init(x: str) -> InitType:
    allowed: tuple[InitType, ...] = (
        "random",
        "nndsvd",
        "nndsvda",
        "nndsvdar",
        "custom",
    )
    if x not in allowed:
        raise ValueError(f"cnmf.init must be one of {allowed}, got {x!r}")
    return x


def load_params(path: Path) -> Params:
    raw: Dict[str, Any] = yaml.safe_load(path.read_text())
    cnmf_raw: Dict[str, Any] = raw["cnmf"]
    params: Params = {
        "hvg_n_top_genes": int(raw["hvg_n_top_genes"]),
        "pct_mito_max": float(raw["pct_mito_max"]),
        "min_genes": int(raw["min_genes"]),
        "cnmf": {
            "K": int(cnmf_raw["K"]),
            "max_iter": int(cnmf_raw["max_iter"]),
            "init": _coerce_init(str(cnmf_raw["init"])),
            "random_state": int(cnmf_raw["random_state"]),
        },
    }
    return params


# ---- CLI ------------------------------------------------------------------------
```

```python
def build_argparser() -> argparse.ArgumentParser:
    ap = argparse.ArgumentParser(description="Fit cNMF programs on unperturbed HVGs.")
    ap.add_argument("--params", required=True, help="configs/params.yml")
    ap.add_argument("--in", dest="inp", required=True, help="out/interim")
    ap.add_argument("--out", required=True, help="out/interim")
    return ap


# ---- Utils -------------------------------------------------------------------


def as_dense(X: Any) -> NDArray[np.float64]:
    """Convert AnnData matrix to a dense numpy array (float64)."""
    if hasattr(X, "toarray"):
        return X.toarray().astype(np.float64, copy=False)  # type: ignore[no-any-return]
    return np.asarray(X, dtype=np.float64)


# ---- Main --------------------------------------------------------------------


def main() -> None:
    args = build_argparser().parse_args()
    P = load_params(Path(args.params))

    ad = sc.read_h5ad("data/interim/unperturbed_qc.h5ad")

    mask: NDArray[np.bool_] = ad.var["highly_variable"].to_numpy()  # type: ignore[no-any-r
eturn]
    X: NDArray[np.float64] = as_dense(ad[:, mask].X) # type: ignore

    # Pylance is now satisfied because "init" is InitType (a Literal union)
    nmf = NMF(
        n_components=P["cnmf"]["K"],
        init=P["cnmf"]["init"],
        random_state=P["cnmf"]["random_state"],
        max_iter=P["cnmf"]["max_iter"],
    )
    H: NDArray[np.float64] = nmf.fit_transform(np.maximum(X, 0.0))
    W: NDArray[np.float64] = nmf.components_.T  # genesÃ\227K

    ad.obsm["cnmf_H"] = H
    ad.varm["cnmf_W"] = W

    out_dir = Path(args.out)
    out_dir.mkdir(exist_ok=True, parents=True)
    gene_names = ad.var_names[mask]
    cols = [f"P{k}" for k in range(P["cnmf"]["K"])]

    pd.DataFrame(W, index=gene_names, columns=cols).to_csv(
        out_dir / "program_loadings_W.csv"
    )

    manifest: dict[str, Any] = {
        "time": time.time(),
        "git": os.popen("git rev-parse --short HEAD").read().strip(),
        "outputs": ["out/interim/program_loadings_W.csv"],
        "params": P["cnmf"],
    }
    (out_dir / "manifest_cnmf.json").write_text(json.dumps(manifest, indent=2))


if __name__ == "__main__":
    main()
```

```python
import numpy as np
from scipy.sparse.linalg import eigsh

# Optional plotting / clustering helpers (for plot_result)
# Comment these out if you don't need them.
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
import matplotlib.pyplot as plt
from typing import Any
from pathlib import Path

def _standardize(X: np.ndarray, axis: int = 0) -> np.ndarray:
    """
    Center and scale X along the given axis (like R's scale()).
    axis=0 => column-wise standardization.
    """
    X = np.asarray(X, float)
    mean = X.mean(axis=axis, keepdims=True)
    std = X.std(axis=axis, ddof=1, keepdims=True)
    std[std == 0] = 1.0
    return (X - mean) / std


############################################################
##### Utilities used in both DCOL-PCA & DCOL-CCA ##########
############################################################

def scol_matrix_order(a: np.ndarray, x: np.ndarray) -> np.ndarray | float:
    """
    Python version of scol.matrix.order(a, x).

    x: 1D array used to order samples.
    a: either a vector of length n_samples or a matrix with shape (n_rows, n_samples).
       Returns:
         - scalar if a is a vector / single-row
         - 1D array of length n_rows if a is 2D (row-wise DCOL distances).
    """
    a = np.asarray(a)
    x = np.asarray(x)
    order = np.argsort(x)

    # a is effectively a vector (R's "is.null(nrow(a)) | nrow(a) == 1")
    if a.ndim == 1 or a.shape[0] == 1:
        a_vec = a.ravel()[order]
        d = np.diff(a_vec)
        dd = np.sum(d ** 2)
        return float(dd)

    # otherwise: matrix case, rows = features, cols = samples
    a_sorted = a[:, order]
    d = np.diff(a_sorted, axis=1)
    dd = np.sum(d ** 2, axis=1)  # rowSums
    return dd


def find_dcol(a: np.ndarray, b: np.ndarray, n_nodes: int = 1) -> np.ndarray:
    """
    Python version of findDCOL(a, b, nNodes).

    a, b: 2D arrays with shape (n_rows, n_samples).
          Rows are features, columns are samples.
    n_nodes: kept for API parity; current implementation is sequential.

    Returns
    -------
    dcol : np.ndarray, shape (nrow(a), nrow(b))
        Symmetric DCOL distance matrix when a and b refer to the same set.
    """
    a = np.asarray(a)
    b = np.asarray(b)

    # vector vs vector case
    if a.ndim == 1 or a.shape[0] == 1:
        # a, b are treated as vectors
        return np.array(scol_matrix_order(a, b), ndmin=1)

    n_a = a.shape[0]
    n_b = b.shape[0]

    # NOTE: for simplicity, this is sequential. You can parallelize these loops
    # with multiprocessing / joblib if needed.
    dcolab = np.zeros((n_a, n_b), dtype=float)
    dcolba = np.zeros((n_a, n_b), dtype=float)

    # dcolab[i_column] = scol_matrix_order(a, b[i, ])
    for i in range(n_b):
        dcolab[:, i] = scol_matrix_order(a, b[i, :])

    # dcolba[j_row] = scol_matrix_order(b, a[j, ])
    for j in range(n_a):
        dcolba[j, :] = scol_matrix_order(b, a[j, :])

    # retain the smaller entry to enforce symmetry
    dcol = np.minimum(dcolab, dcolba)
    return dcol


def get_cov(dcol_matrix: np.ndarray, X: np.ndarray, Y: np.ndarray) -> np.ndarray | float:
    """
    Python version of getCov(DCOLMatrix, X, Y).

    X, Y: data matrices with shape (n_samples, n_features), rows = samples.
    dcol_matrix:
      - scalar / length-1 => vector case (single pair).
      - 2D matrix (p x p) => DCOL distances between features (columns of Y).

    Returns
    -------
    - scalar in the vector case
    - CovMatrix (DCOL-correlation matrix), same shape as dcol_matrix in matrix case.
    """
    dcol = np.asarray(dcol_matrix, float)
    X = np.asarray(X, float)
    Y = np.asarray(Y, float)

    # Vector / scalar case
    if dcol.ndim == 0 or (dcol.ndim == 1 and dcol.shape[0] == 1):
        X = X.ravel()
        Y = Y.ravel()
        n = X.shape[0]
        var_Y = np.var(Y, ddof=1)
        if var_Y <= 0:
            # Degenerate case: no variance in Y, return 0 correlation
            return 0.0
        value = np.sqrt(max(0.0, 1.0 - dcol.item() / (2.0 * (n - 2.0) * var_Y)))
        return float(value)

    # Matrix case
    n = X.shape[0]
    var_list = np.var(Y, axis=0, ddof=1)  # sample variance over samples

    eps = 1e-12
    zero_var = var_list <= eps
    if np.any(zero_var):
        print(
            f"[get_cov] {zero_var.sum()} zero-variance features; stabilizing",
            flush=True,
        )

    scale = np.zeros_like(var_list)
    ok = ~zero_var
    scale[ok] = 1.0 / (2.0 * (n - 2.0) * var_list[ok])

    # eigenMapMatMult(DCOLMatrix, diag(scale)) == column-wise scaling by 'scale'
    cov_matrix = 1.0 - dcol * scale  # broadcast scale across rows
    cov_matrix[cov_matrix < 0] = 0.0  # clamp negs to 0 for num stability

    # For zero-var features, zero out row/cl and set diag to 1
    if np.any(zero_var):
        cov_matrix[:, zero_var] = 0.0
        cov_matrix[zero_var, :] = 0.0
        idx = np.where(zero_var)[0]
        cov_matrix[idx, idx] = 1.0

    cov_matrix = np.sqrt(cov_matrix)
    return cov_matrix


############################################################
##### DCOL-PCA (feature-based and cell-based versions) ####
############################################################

def dcol_pca0(
    X: np.ndarray,
    image: int = 0,
    k: int = 4,
    labels=None,
    Scale: bool = True,
    nNodes: int = 1,
    nPC_max: int = 100,
) -> dict[str, Any]:
    """
    Python version of Dcol_PCA0(X, ...).
    PCA with n = cells and k = principal k features
    Parameters
    ----------
    X : array-like, shape (n_samples, n_features)
        Columns are features (genes), rows are samples (cells).
    image : int
        If 1, you can add plotting code here (not implemented by default).
    k : int
        Number of dimensions to keep for 'data.r' (visualization).
    labels : array-like, optional
        Group labels for plotting (unused unless you add plotting).
    Scale : bool
        Whether to standardize features before computing DCOL.
    nNodes : int
        Kept for API compatibility; current implementation is sequential.
    nPC_max : int
        Maximum number of principal components to compute.

    Returns
    -------
    dict with keys:
      - 'cov_D'      : DCOL-based correlation matrix (p x p)
      - 'vecs'    : eigenvectors of cov_D (p x nPC)
      - 'vals'     : eigenvalues (nPC,)
      - 'data_r'     : embedding (n_samples x min(k, nPC))
      - 'X_proj'     : full projection (n_samples x nPC)
    """
    X = np.asarray(X, float)
    X_o = X.copy()  # used for final projection

    if Scale:
        X = _standardize(X, axis=0)  # column-wise (features)

    # DCOL matrix over features: findDCOL(t(X), t(X))
    DcolMatrix = find_dcol(X.T, X.T, n_nodes=nNodes)
    cov_D = get_cov(DcolMatrix, X, X)  # DCOL-correlation matrix

    # Suppose cov_D is the matrix passed to eigsh
    print("[dcol_pca] cov_D finite?", np.isfinite(cov_D).all(), flush=True)
    print("[dcol_pca] cov_D min/max:", np.nanmin(cov_D), np.nanmax(cov_D), flush=True)

    # Eigen-decomposition (like RSpectra::eigs_sym on symmetric cov_D)
    p = cov_D.shape[0]
    nPC = min(nPC_max, p)

    # Enforce symmetry and remove NaN/Inf just in case
    cov_D = 0.5 * (cov_D + cov_D.T)
    cov_D = np.nan_to_num(cov_D, nan=0.0, posinf=0.0, neginf=0.0)

    # Add tiny diagonal jitter for numerical stablility
    cov_D.flat[:: p + 1] += 1e-8

    if p <= nPC + 10:
        # small matrix: use dense eigh
        vals, vecs = np.linalg.eigh(cov_D)
        idx = np.argsort(vals)[::-1][:nPC]
        vals = vals[idx]
        vecs = vecs[:, idx]
    else:
        # large matrix: sparse eigensolver
        vals, vecs = eigsh(cov_D, k=nPC, which="LM")
        idx = np.argsort(vals)[::-1]
        vals = vals[idx]
        vecs = vecs[:, idx]

    # Project original (unscaled) X onto eigenvectors
    X_proj = X_o @ vecs  # (n_samples x nPC)
    data_r = X_proj[:, : min(k, X_proj.shape[1])]

    # You can add plotting here if image == 1 (using matplotlib).

    return {
        "cov_D": cov_D,
        "vecs": vecs,
        "vals": vals,
        "data_r": data_r,
        "X_proj": X_proj,
    }


def dcol_pca(
    X: np.ndarray,
    image: int = 0,
    k: int = 4,
    labels=None,
    Scale: bool = True,
    nNodes: int = 1,
    nPC_max: int = 100,
):
    """
    Python version of the alternative Dcol_PCA(X, ...).

    This version operates more on cell-cell similarity. In the R code,
    it transposes X, scales across samples, and builds a DCOL matrix
    that ultimately yields a cell-level embedding.

    Parameters
    ----------
    X : array-like, shape (n_samples, n_features)
```

```
        Rows are samples/cells, columns are features.
    image, k, labels, Scale, nNodes, nPC_max : as above.

    Returns
    -------
    dict with keys:
      - 'cov_D'    : DCOL-based correlation / similarity between cells (n_samples x n_sampl
es)
      - 'vecs' : eigenvectors (n_samples x nPC)
      - 'vals'   : eigenvalues (nPC,)
      - 'data_r'    : embedding (n_samples x min(k, nPC))
    """
    X = np.asarray(X, float)
    X_o = X.copy()
    X_t = X.T  # features x samples

    if Scale:
        # In the R version, scale() is applied to X after transposing,
        # so this standardizes each sample (column) across features.
        X_t = _standardize(X_t, axis=0)

    # DCOL over cells: findDCOL(t(X), t(X)) with the modified X_t
    DcolMatrix = find_dcol(X_t.T, X_t.T, n_nodes=nNodes)
    cov_D = get_cov(DcolMatrix, X_t, X_t)  # n_samples x n_samples

    n_cells = cov_D.shape[0]
    nPC = min(nPC_max, n_cells)

    if n_cells <= nPC + 10:
        vals, vecs = np.linalg.eigh(cov_D)
        idx = np.argsort(vals)[::-1][:nPC]
        vals = vals[idx]
        vecs = vecs[:, idx]
    else:
        vals, vecs = eigsh(cov_D, k=nPC, which="LM")
        idx = np.argsort(vals)[::-1]
        vals = vals[idx]
        vecs = vecs[:, idx]

    PCs = vecs
    data_r = PCs[:, : min(k, PCs.shape[1])]

    # Again, you can add plotting if image == 1.

    return {
        "cov_D": cov_D,
        "vecs": vecs,
        "vals": vals,
        "data_r": data_r,
    }


######### Visualize results (optional) ############

def plot_result(reduced_data: np.ndarray, group_info, k: int = 2):
    """
    Rough Python version of plot.result().

    reduced_data : array-like, shape (n_samples, d)
        Low-dimensional embedding (e.g. output of dcol_pca['data_r'] or X_proj).
    group_info : array-like
        True group labels.
    k : int
        Number of dimensions from reduced_data to use.

    Returns
    -------
    ARI (float). Also produces a scatter plot when k <= 2.
    """
    reduced_data = np.asarray(reduced_data, float)
    group_info = np.asarray(group_info)

    n_clusters = len(np.unique(group_info))
    km = KMeans(n_clusters=n_clusters, n_init=10, random_state=0)
    km.fit(reduced_data[:, :k])
    ari = round(adjusted_rand_score(group_info, km.labels_), 3)

    if k <= 2:
        plt.figure()
        plt.scatter(
            reduced_data[:, 0],
            reduced_data[:, 1],
            c=group_info,
            s=20,
            cmap="tab10",
        )
        plt.title(f"ARI = {ari}")
        plt.xlabel("Factor1")
        plt.ylabel("Factor2")
        plt.tight_layout()
    else:
        # For k > 2, you could expand this to pairplots if needed.
        plt.figure()
        plt.scatter(
            reduced_data[:, 0],
            reduced_data[:, 1],
            c=group_info,
            s=20,
            cmap="tab10",
        )
        plt.title(f"ARI (first 2 dims) = {ari}")
        plt.xlabel("Factor1")
        plt.ylabel("Factor2")
        plt.tight_layout()

    return ari


def plot_spectral(
    vals: np.ndarray, out_dir: Path, title_prefix: str = "DCOL-PCA"
) -> Path:
    """
    Make scree + cumulative variance plots for spectral decomp e.g., PCA, DCOL-PCA, etc eig
envalues.

    Parameters
    ----------
    vals : array-like
        1D array of spectra (eigenvalues) (typically res["Evalues"]),
        sorted in descending order.
    title_prefix : str
        Prefix for subplot titles (e.g. "DCOL-PCA", "PCA", etc.)
    """
    ev = np.asarray(vals, float)

    # Guard against weird inputs
    ev = ev[ev > 0]  # ignore non-positive eigenvalues if any
    if ev.size == 0:
        print("[plot_dcol_scree] No positive eigenvalues to plot.")
        return

    var_ratio = ev / ev.sum()
    cum_ratio = np.cumsum(var_ratio)
    k = np.arange(1, ev.size + 1)

    fig, axes = plt.subplots(2, 1, figsize=(6, 6), constrained_layout=True)
    # Scree: raw eigenvalues
    axes[0].plot(k, ev, marker="o")
    axes[0].set_title(f"{title_prefix}: Eigenvalues (Scree)")
    axes[0].set_xlabel("Component index")
    axes[0].set_ylabel("Eigenvalue")

    # Cumulative variance
    axes[1].plot(k, cum_ratio, marker="o")
    axes[1].set_title(f"{title_prefix}: Cumulative 'variance' explained")
    axes[1].set_xlabel("Number of components")
    axes[1].set_ylabel("Cumulative fraction")
    axes[1].set_ylim(0, 1.05)

    k_png = out_dir / f"{title_prefix}.png"
    plt.savefig(k_png, bbox_inches="tight", dpi=160)
    plt.close(fig)
    return k_png
```

```
#!/usr/bin/env python
from __future__ import annotations

import argparse
from pathlib import Path
from typing import Optional

import numpy as np
import scanpy as sc
import scipy.sparse as sp

from sklearn.decomposition import PCA
from sklearn.manifold import Isomap, SpectralEmbedding

# Optional PHATE
try:
    import phate  # type: ignore
except ImportError:
    phate = None  # type: ignore


# ---------------------------------------------------------------------
# CLI
# ---------------------------------------------------------------------


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description=(
            "Compute multiple dimensionality reductions on a QCâ\200\231d AnnData with HVGs
 "
            "and store them as embeddings in .obsm."
        )
    )
    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Input AnnData file (QCâ\200\231d, with HVGs annotated).",
    )
    p.add_argument(
        "--out",
        type=str,
        default=None,
        help="Output AnnData file (default: overwrite --ad).",
    )
    p.add_argument(
        "--n-components",
        type=int,
        default=20,
        help="Target embedding dimensionality for most methods (default: 20).",
    )
    p.add_argument(
        "--n-neighbors",
        type=int,
        default=30,
        help="k for kNN-based methods (diffmap, UMAP, PHATE, Isomap, spectral).",
    )
    p.add_argument(
        "--seed",
        type=int,
        default=0,
        help="Random seed for reproducibility.",
    )
    p.add_argument(
        "--hvg-trunc",
        type=int,
        default=None,
        help=(
            "Number of HVGs to keep in 'X_hvg_trunc'. "
            "Default: use n-components if not set."
        ),
    )
    p.add_argument(
        "--only-hvg-phate",
        action="store_true",
        help=(
            "Run ONLY HVG-truncated embedding and PHATE (if installed), "
            "skipping PCA/diffmap/UMAP/Isomap/spectral."
        ),
    )
    return p


# ---------------------------------------------------------------------
# Helpers
# ---------------------------------------------------------------------


def to_dense(x):
    """Convert sparse -> dense; ensure numpy array."""
    if sp.issparse(x):
        return x.toarray()
    return np.asarray(x)


def ensure_hvg_view(ad: sc.AnnData) -> sc.AnnData:
    """
    Return an AnnData subset to HVGs (if annotated), otherwise a full copy.

    All embeddings are computed on this HVG view, but stored in the
    original AnnData (ad.obsm[...] on the full object).
    """
    if "highly_variable" in ad.var:
        mask = ad.var["highly_variable"].to_numpy().astype(bool)
        n_hvg = int(mask.sum())
        if n_hvg > 0:
            print(f"[HVG] Using HVGs only: n_vars = {n_hvg}", flush=True)
            return ad[:, mask].copy()
        else:
            print(
                "[HVG] 'highly_variable' present but no genes flagged; "
                "using all genes."
            )
            return ad.copy()
    else:
        print("[HVG] No 'highly_variable' flag; using all genes.", flush=True)
        return ad.copy()


# ---------------------------------------------------------------------
# Embeddings
# ---------------------------------------------------------------------


def add_hvg_truncated(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_genes: int,
    key: str = "X_hvg_trunc",
) -> None:
    """
    'HVG truncated' embedding: keep top genes by normalized dispersion
    (or first n) and store raw expression for those genes in .obsm[key].

    Shape: (n_cells, n_genes)
    """
    print(f"[HVG-trunc] Computing HVG-truncated embedding (n_genes={n_genes})", flush=True)

    if "dispersions_norm" in ad_hvg.var:
        disp = ad_hvg.var["dispersions_norm"].to_numpy()
        order = np.argsort(disp)[::-1]  # descending
        keep_idx = order[: min(n_genes, ad_hvg.n_vars)]
    else:
        print(
            "  No 'dispersions_norm' in ad.var; taking first "
            f"{min(n_genes, ad_hvg.n_vars)} HVGs."
        )
        keep_idx = np.arange(min(n_genes, ad_hvg.n_vars))

    X_hvg = ad_hvg.X[:, keep_idx]
    X_hvg = to_dense(X_hvg).astype(np.float32)

    ad.obsm[key] = X_hvg
    ad.uns[f"{key}_genes"] = ad_hvg.var_names[keep_idx].tolist()
    print(f"  Stored '{key}' with shape {X_hvg.shape}.", flush=True)


def add_pca_scanpy(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    seed: int,
) -> None:
    """
    PCA via Scanpy (ARPACK-based on covariance). Stores ad.obsm['X_pca'].
    """
    print(f"[PCA] Computing Scanpy PCA with n_comps={n_components}", flush=True)
    sc.tl.pca(
        ad_hvg,
        n_comps=n_components,
        svd_solver="arpack",
        random_state=seed,
    )
    X_pca = ad_hvg.obsm["X_pca"][:, :n_components].astype(np.float32)
    ad.obsm["X_pca"] = X_pca
    print("  Stored 'X_pca' with shape", X_pca.shape, flush=True)


def add_neighbors_diffmap_umap(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    n_neighbors: int,
    seed: int,
) -> None:
    """
    Build kNN graph on PCA space (from ad_hvg), then:
      - Diffusion Map â\206\222 ad.obsm['X_diffmap']
      - UMAP         â\206\222 ad.obsm['X_umap']
    """
    print(
        f"[neighbors+DM+UMAP] neighbors: n_neighbors={n_neighbors}, "
        f"based on X_pca; n_comps={n_components}",
        flush=True,
    )

    sc.pp.neighbors(
        ad_hvg,
        n_neighbors=n_neighbors,
        use_rep="X_pca",
        random_state=seed,
    )
    print("  Computing Diffusion Map...", flush=True)
    sc.tl.diffmap(ad_hvg, n_comps=n_components)
    X_dm = ad_hvg.obsm["X_diffmap"][:, :n_components].astype(np.float32)
    ad.obsm["X_diffmap"] = X_dm
    print("  Stored 'X_diffmap' with shape", X_dm.shape, flush=True)

    print("  Computing UMAP...", flush=True)
    sc.tl.umap(
        ad_hvg,
        n_components=n_components,
        random_state=seed,
    )
    X_umap = ad_hvg.obsm["X_umap"].astype(np.float32)
    ad.obsm["X_umap"] = X_umap
    print("  Stored 'X_umap' with shape", X_umap.shape, flush=True)


def add_phate(
    ad: sc.AnnData,
    ad_hvg: sc.AnnData,
    n_components: int,
    n_neighbors: int,
    seed: int,
) -> None:
    """
    PHATE embedding stored as ad.obsm['X_phate'], if phate is installed.
    """
    if phate is None:
        print("[PHATE] phate not installed; skipping.", flush=True)
        return

    print(
        f"[PHATE] Computing PHATE (n_components={n_components}, knn={n_neighbors})",
        flush=True,
    )
    X = to_dense(ad_hvg.X)
    ph = phate.PHATE(
        n_components=n_components,
        knn=n_neighbors,
        n_jobs=-1,
        random_state=seed,
    )
    X_phate = ph.fit_transform(X).astype(np.float32)

    ad.obsm["X_phate"] = X_phate
    print("  Stored 'X_phate' with shape", X_phate.shape, flush=True)


# ---------------------------------------------------------------------
# Main
# ---------------------------------------------------------------------


def main() -> None:
    args = build_argparser().parse_args()

    np.random.seed(args.seed)

    in_path = Path(args.ad)
    out_path = Path(args.out) if args.out is not None else in_path

    print(f"[load] Reading AnnData from {in_path} ...", flush=True)
    ad = sc.read_h5ad(in_path)

    # Work on HVG subset for all manifold methods
    ad_hvg = ensure_hvg_view(ad)

    # 1) HVG truncated (top genes by dispersion)
    hvg_trunc_n = args.hvg_trunc or args.n_components
    add_hvg_truncated(ad, ad_hvg, n_genes=hvg_trunc_n)

    if args.only_hvg_phate:
        # Just HVG-trunc + PHATE, skip everything else
        add_phate(
            ad,
            ad_hvg,
            n_components=args.n_components,
            n_neighbors=args.n_neighbors,
            seed=args.seed,
        )
    else:
        # 2) PCA (Scanpy)
        add_pca_scanpy(ad, ad_hvg, n_components=args.n_components, seed=args.seed)

        # 3) Diffusion Map + UMAP (kNN graph on PCA)
        add_neighbors_diffmap_umap(
            ad,
            ad_hvg,
            n_components=args.n_components,
            n_neighbors=args.n_neighbors,
            seed=args.seed,
        )

        # 4) PHATE (optional if installed)
        add_phate(
            ad,
            ad_hvg,
            n_components=args.n_components,
            n_neighbors=args.n_neighbors,
            seed=args.seed,
        )



    print(f"[save] Writing updated AnnData with embeddings to {out_path} ...", flush=True)
    ad.write(out_path)
    print("[done]", flush=True)


if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python
"""
Inspect an .h5ad file to understand its structure for QC ingestion.

Usage:
  conda run -n venv python scripts/inspect_h5ad.py \
      --h5ad data/raw/weinreb/stateFate_inVitro/stateFate_inVitro_normed_counts.h5ad
"""

import argparse
import numpy as np
import scanpy as sc
import pandas as pd
from pandas.api.types import is_numeric_dtype, is_categorical_dtype


def inspect_h5ad(path: str) -> None:
    print(f"Loading {path}")
    ad = sc.read_h5ad(path, backed="r")

    print("\n=== AnnData overview ===")
    print(ad)
    print("shape (n_cells, n_genes):", ad.shape)
    print("X class:", type(ad.X))

    # ---------- OBS (cell metadata) ----------
    print("\n=== OBS (cell metadata) ===")
    print("obs columns:", list(ad.obs.columns))
    print("\nobs.head():")
    print(ad.obs.head())

    print("\n[obs summary by column]")
    for col in ad.obs.columns:
        s = ad.obs[col]
        nunique = s.nunique()
        print(f"\n---- {col} ----")
        print("dtype:", s.dtype)
        print("n_unique:", nunique)

        if nunique <= 20:
            # categorical-ish: show value counts
            print("value_counts():")
            print(s.value_counts().head(20))
        else:
            # high-cardinality: branch by dtype
            if is_numeric_dtype(s):
                print(
                    "min/mean/max:",
                    float(s.min()),
                    float(s.mean()),
                    float(s.max()),
                )
            elif is_categorical_dtype(s):
                cats = s.cat.categories
                print(f"categorical with {len(cats)} categories")
                print("categories (first 20):", list(cats[:20]))
            else:
                print("example values:", s.iloc[:10].tolist())

    # Highlight likely-important columns for QC / modeling if present
    interesting_obs = [
        "timepoint",
        "day",
        "treatment",
        "condition",
        "sample",
        "batch",
        "clone",
        "clone_id",
        "lineage",
        "cell_type",
        "state",
    ]
    print("\n=== Selected interesting obs columns (if present) ===")
    for col in interesting_obs:
        if col in ad.obs:
            print(f"\n---- {col} ----")
            s = ad.obs[col]
            print("dtype:", s.dtype)
            print("n_unique:", s.nunique())
            print(s.value_counts().head(20))

    # ---------- VAR (gene metadata) ----------
    print("\n=== VAR (gene metadata) ===")
    print("var columns:", list(ad.var.columns))
    print("\nvar.head():")
    print(ad.var.head())
    print("\nvar_names (first 10):")
    print(ad.var_names[:10].tolist())


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--h5ad",
        type=str,
        default="data/raw/stateFate_inVitro_normed_counts.h5ad",
        help="Path to .h5ad file to inspect",
    )
    args = parser.parse_args()
    inspect_h5ad(args.h5ad)


if __name__ == "__main__":
    main()
```

```python
#!/usr/bin/env python3
from __future__ import annotations

import argparse
import json
import os
from pathlib import Path
from typing import Any, Dict, List

import numpy as np
import pandas as pd
import scanpy as sc  # type: ignore
from scipy import sparse
import subprocess
import yaml
import matplotlib.pyplot as plt
from dcol_pca import dcol_pca0, plot_spectral

# Example use
# conda run -n venv python scripts/01_qc_eda.py   --params configs/params.yml   --out out/i
nterim   --ad data/raw/K562_gwps/k562_replogie.h5ad   --report   --report-to-gcs gs://mantr
a-mlfg-prod-uscentral1-8e7a/out/interim   --plot-max-cells 10000

def build_argparser() -> argparse.ArgumentParser:
    ap = argparse.ArgumentParser(description="QC + EDA for unperturbed cells.")
    ap.add_argument("--params", required=True, help="configs/params.yml")
    ap.add_argument("--out", required=True, help="out/interim")
    ap.add_argument("--ad", required=True, help="path to unperturbed .h5ad")
    return ap

def prep(ad: sc.AnnData, params: Dict[str, Any]):
    n_cells = ad.n_obs

    # Remove genes that are not statistically relevant (< 0.1% of cells)
    min_cells = max(3, int(0.001 * n_cells))
    sc.pp.filter_genes(ad, min_cells=min_cells)

    # Remove empty droplets (cells with no detected genes)
    sc.pp.filter_cells(ad, min_genes=int(params["qc"]["min_genes"]))

    # Drop zero-count cells
    totals = np.ravel(ad.X.sum(axis=1))
    ad = ad[totals > 0, :].copy()

    # Cells with high percent of mitochondrial DNA are dying or damaged
    ad = ad[ad.obs["mitopercent"] < float(params["qc"]["max_pct_mt"])].copy()

    print("AnnData layers:", list(ad.layers.keys()), flush=True)
    print("AnnData obs columns:", list(ad.obs.columns), flush=True)
    print("AnnData var columns:", list(ad.var.columns), flush=True)

    # How many genes/cells remain just before HVG?
    print("n_obs, n_vars:", ad.n_obs, ad.n_vars, flush=True)

    # Check for inf/nan in means explicitly:
    X = ad.X
    if sparse.issparse(X):
        means = np.asarray(X.mean(axis=0)).ravel()
    else:
        means = np.nanmean(X, axis=0)

    print("Means finite?", np.all(np.isfinite(means)), flush=True)
    print("Means min/max:", np.nanmin(means), np.nanmax(means), flush=True)
    print("# non-finite means:", np.sum(~np.isfinite(means)), flush=True)

    # No raw counts object so we must use ad.X
    sc.pp.highly_variable_genes(
        ad,
        n_top_genes=int(params["hvg_n_top_genes"]),
        flavor="seurat_v3",
        subset=False,
    )

    ad = ad[:, ad.var["highly_variable"]].copy()

    # now normalize/log on X (leave counts in layer untouched)
    sc.pp.normalize_total(ad, target_sum=1e4)
    sc.pp.log1p(ad)

    return ad


def main() -> None:
    args = build_argparser().parse_args()
    params: Dict[str, Any] = yaml.safe_load(Path(args.params).read_text())

    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    # --- Load full AnnData in backed mode (no 61 GiB dense allocation) ---
    ad_full = sc.read_h5ad(args.ad, backed="r")
    print(
        f"[load] full AnnData: n_obs={ad_full.n_obs}, n_vars={ad_full.n_vars}",
        flush=True,
    )

    # --- Define unperturbed / control cells: gene == 'non-targeting' ---
    if "gene" not in ad_full.obs:
        raise ValueError(
            "'gene' column not found in ad.obs. "
            f"Available columns: {list(ad_full.obs.columns)}"
        )

    is_ctrl = np.asarray(ad_full.obs["gene"] == "non-targeting")
    n_ctrl = int(is_ctrl.sum())
    n_pert = int((~is_ctrl).sum())
    print(f"[split] control/non-targeting cells: {n_ctrl}", flush=True)
    print(f"[split] perturbed cells: {n_pert}", flush=True)

    if n_ctrl == 0:
        raise ValueError("No control cells with gene == 'non-targeting' found.")

    # --- Materialize ONLY the non-targeting cells in memory ---
    ad = ad_full[is_ctrl, :].to_memory()
    print(
        f"[load] using {ad.n_obs} non-targeting cells for QC + dim reduction",
        flush=True,
    )

    if not sparse.issparse(ad.X):
        ad.X = sparse.csr_matrix(ad.X)

    for col in ad.obs.columns:
        print(f"self.{col}: {ad.obs[col].dtype}", flush=True)

    print()
    for col in ad.var.columns:
        print(f"self.{col}: {ad.var[col].dtype}", flush=True)

    # QC processing
    qc_ad = prep(ad.copy(), params)
    print(f"[write] writing QC AnnData to {out_dir}", flush=True)

    qc_ad.write_h5ad(out_dir)
    print("[done]", flush=True)
```

```python
if __name__ == "__main__":
    main()
```

```
# scripts/train_energy.py

from __future__ import annotations

from pathlib import Path
from typing import Dict, Any

import argparse
import yaml
import torch
import scanpy as sc
import numpy as np

from mantra.config import EnergyModelConfig, EnergyTrainConfig
from mantra.eggfm.trainer import train_energy_model


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(description="Train EGGFM energy model on K562")
    p.add_argument("--params", type=str, required=True,
                   help="YAML params file (must contain eggfm_model and eggfm_train)")
    p.add_argument("--ad", type=str, required=True,
                   help="Preprocessed K562 AnnData (e.g. data/interim/k562_replogle_prep.h5
ad)")
    p.add_argument("--out", type=str, required=True,
                   help="Output directory for checkpoints (e.g. out/models/eggfm)")
    return p


def subset_anndata(ad: sc.AnnData, n_cells_sample: int, random_state: int) -> sc.AnnData:
    """
    Randomly subset AnnData rows (cells).
    If n_cells_sample >= n_obs or n_cells_sample <= 0, returns ad.copy().
    """
    n = ad.n_obs
    m = min(int(n_cells_sample), n)
    if m <= 0 or m == n:
        return ad.copy()

    rng = np.random.default_rng(random_state)
    idx = rng.choice(n, size=m, replace=False)
    return ad[idx].copy()

def main() -> None:
    args = build_argparser().parse_args()

    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    params: Dict[str, Any] = yaml.safe_load(Path(args.params).read_text())

    model_cfg = EnergyModelConfig(**params.get("eggfm_model", {}))
    train_cfg = EnergyTrainConfig(**params.get("eggfm_train", {}))

    # 1) Load prepped K562 AnnData
    ad = sc.read_h5ad(args.ad)

    # 2) optional subsample for this experiment
    train_n_cells = params["eggfm_train"].get("n_cells_sample", None)
    if train_n_cells is not None:
        ad_prep = subset_anndata(ad, train_n_cells, random_state=params.get("seed", 0))
    else:
        ad_prep = ad

    # 3) restrict to HVGs if present
    if "highly_variable" in ad_prep.var:
        ad_prep = ad_prep[:, ad_prep.var["highly_variable"]].copy()
        print(f"Using HVGs only: n_vars = {ad_prep.n_vars}")
        # ---- NEW: further clamp HVGs to top N by dispersions_norm ----
        max_hvg = params["eggfm_train"].get("max_hvg", None)
        if max_hvg is not None and ad_prep.n_vars > max_hvg:
            if "dispersions_norm" in ad_prep.var:
                disp = ad_prep.var["dispersions_norm"].to_numpy()
                order = np.argsort(disp)[::-1]  # descending
            else:
                # fallback: arbitrary but deterministic
                order = np.arange(ad_prep.n_vars)

            keep_idx = order[:max_hvg]
            ad_prep = ad_prep[:, keep_idx].copy()
            print(
                f"Subsetting HVGs from {len(order)} â\206\222 {ad_prep.n_vars} "
                f"(top {max_hvg} by dispersions_norm)"
            )
    else:
        print("No 'highly_variable' flag in ad.var; using all genes as-is.")

    # 4) Train energy model
    bundle = train_energy_model(
        ad_prep=ad_prep,
        model_cfg=model_cfg,
        train_cfg=train_cfg,
    )

    energy_model = bundle.model
    mean = bundle.mean
    std = bundle.std
    var_names = bundle.feature_names

    # 5) Save checkpoint
    ckpt = {
        "state_dict": energy_model.state_dict(),
        "model_cfg": {
            "hidden_dims": list(model_cfg.hidden_dims),
        },
        "n_genes": energy_model.n_genes,
        "var_names": var_names,
        "mean": mean,
        "std": std,
        "space": bundle.space,
    }

    # Derive a short tag for the representation space, e.g. "expr", "pca", "phate"
    space_tag = str(bundle.space).replace("X_", "")  # if you ever use "X_pca", etc.
    n_genes = int(energy_model.n_genes)

    ckpt_name = f"eggfm_energy_k562_{space_tag}_hvg{n_genes}.pt"
    ckpt_path = out_dir / ckpt_name

    torch.save(ckpt, ckpt_path)
    print(f"Saved EGGFM energy checkpoint to {ckpt_path}", flush=True)


if __name__ == "__main__":
    main()
```

```python
# scripts/grn_train.py

from __future__ import annotations

from pathlib import Path
from typing import Dict, Any, Optional

import argparse
import yaml
import numpy as np
import torch
import scanpy as sc
from scipy import sparse as sp_sparse
from torch.utils.data import DataLoader

from mantra.config import (
    GRNModelConfig,
    GRNTrainConfig,
    GRNLossConfig,
)
from mantra.grn.dataset import K562RegDeltaDataset
from mantra.grn.models import GRNGNN, TraitHead
from mantra.grn.priors import build_energy_prior_from_ckpt
from mantra.grn.trainer import GRNTrainer


def build_argparser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(
        description="Train GRN GNN on K562 with pre-trained EGGFM energy prior"
    )

    p.add_argument(
        "--params",
        type=str,
        required=True,
        help="YAML params file (must contain grn_model, grn_train, grn_loss)",
    )
    p.add_argument(
        "--out",
        type=str,
        required=True,
        help="Output directory for GRN checkpoints / logs",
    )
    p.add_argument(
        "--ad",
        type=str,
        required=True,
        help="Preprocessed K562 AnnData (same gene order as Î\224E)",
    )
    p.add_argument(
        "--train-npz",
        type=str,
        required=True,
        help="NPZ with aggregated (reg_idx, deltaE, deltaP_obs, deltaY_obs, dose)",
    )
    p.add_argument(
        "--val-npz",
        type=str,
        default=None,
        help="Optional NPZ for validation set",
    )
    p.add_argument(
        "--adj",
        type=str,
        default=None,
        help="Optional .npy adjacency [G,G]. If not provided, uses identity.",
    )
    p.add_argument(
        "--cnmf-W",
        type=str,
        default=None,
        help="Optional .npy cNMF loadings W [G,K]. If missing, uses identity [G,G].",
    )
    p.add_argument(
        "--energy-ckpt",
        type=str,
        required=True,
        help="Path to pre-trained EGGFM energy checkpoint (.pt)",
    )
    return p


def main() -> None:
    args = build_argparser().parse_args()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    # ---- load params ----
    params: Dict[str, Any] = yaml.safe_load(Path(args.params).read_text())

    grn_model_cfg = GRNModelConfig(**params.get("grn_model", {}))
    grn_train_cfg = GRNTrainConfig(**params.get("grn_train", {}))
    grn_loss_cfg = GRNLossConfig(**params.get("grn_loss", {}))

    # ---- load AnnData ----
    qc_ad = sc.read_h5ad(args.ad)

    # ---- datasets ----
    train_ds = K562RegDeltaDataset(Path(args.train_npz))
    val_ds: Optional[K562RegDeltaDataset] = (
        K562RegDeltaDataset(Path(args.val_npz))
        if args.val_npz is not None
        else None
    )

    G = train_ds.n_genes
    n_regulators = train_ds.n_regulators

    # ---- adjacency ----
    if args.adj is not None:
        A_np = np.load(args.adj).astype(np.float32)
    else:
        A_np = np.eye(G, dtype=np.float32)
    A = torch.from_numpy(A_np).to(device)

    # ---- cNMF W ----
    if args.cnmf_W is not None:
        W_np = np.load(args.cnmf_W).astype(np.float32)  # [G,K]
    else:
        # identity: effectively disables program loss when lambda_prog > 0
        W_np = np.eye(G, dtype=np.float32)
    W = torch.from_numpy(W_np).to(device)

    # ---- reference state x_ref ----
    X = qc_ad.X
    if sp_sparse.issparse(X):
        X = X.toarray()
    X = np.asarray(X, dtype=np.float32)

    x_ref_np = X.mean(axis=0)  # [G], default: mean over all cells
    if x_ref_np.shape[0] != G:
        raise ValueError(
            f"Gene dimension mismatch: x_ref has {x_ref_np.shape[0]} genes, "
            f"but Î\224E has {G}."
        )
    x_ref = torch.from_numpy(x_ref_np).to(device)

    # ---- dataloaders ----
    train_loader = DataLoader(
        train_ds,
        batch_size=grn_train_cfg.batch_size,
        shuffle=True,
    )
    val_loader = None
    if val_ds is not None:
        val_loader = DataLoader(
            val_ds,
            batch_size=grn_train_cfg.batch_size,
            shuffle=False,
        )

    # ---- energy prior (pretrained EGGFM) ----
    energy_prior = build_energy_prior_from_ckpt(
        ckpt_path=args.energy_ckpt,
        gene_names=qc_ad.var_names,
        device=device,
    )

    # ---- GRN model ----
    model = GRNGNN(
        n_regulators=n_regulators,
        n_genes=G,
        n_layers=grn_model_cfg.n_layers,
        gene_emb_dim=grn_model_cfg.gene_emb_dim,
        hidden_dim=grn_model_cfg.hidden_dim,
        dropout=grn_model_cfg.dropout,
        use_dose=grn_model_cfg.use_dose,
    ).to(device)

    # ---- optional trait head ----
    trait_head: Optional[TraitHead] = None
    if grn_model_cfg.n_traits > 0:
        K = W_np.shape[1]
        trait_head = TraitHead(
            n_programs=K,
            n_traits=grn_model_cfg.n_traits,
            hidden_dim=grn_model_cfg.trait_hidden_dim,
        ).to(device)

    # ---- trainer ----
    trainer = GRNTrainer(
        grn_model=model,
        trait_head=trait_head,
        A=A,
        x_ref=x_ref,
        W=W,
        energy_fn=energy_prior,
        loss_cfg=grn_loss_cfg,
        train_cfg=grn_train_cfg,
        device=str(device),
    )

    trainer.fit(train_loader, val_loader)

    # ---- save best checkpoint ----
    ckpt = {
        "model_state_dict": trainer.best_model_state,
        "trait_head_state_dict": (
            trainer.best_trait_state if trait_head is not None else None
        ),
        "grn_model_cfg": grn_model_cfg.__dict__,
        "grn_train_cfg": grn_train_cfg.__dict__,
        "grn_loss_cfg": grn_loss_cfg.__dict__,
        "n_regulators": n_regulators,
        "n_genes": G,
        "W": W_np,
        "A": A_np,
        "x_ref": x_ref_np,
        "prior_type": "energy_ckpt",  # we used a pre-trained EGGFM checkpoint
        "energy_ckpt_path": str(Path(args.energy_ckpt).resolve()),
    }

    ckpt_path = out_dir / "grn_k562_energy_prior.pt"
    torch.save(ckpt, ckpt_path)
    print(f"Saved GRN checkpoint to {ckpt_path}", flush=True)


if __name__ == "__main__":
    main()
```

```python
# EGGFM/utils.py

from typing import Optional
import numpy as np
import scanpy as sc


def subsample_adata(
    ad: sc.AnnData,
    max_cells: Optional[int] = None,
    seed: int = 0,
) -> sc.AnnData:
    """
    Return a (possibly) subsampled AnnData for quick experiments.

    - If max_cells is None or >= n_obs, returns ad unchanged (no copy).
    - Otherwise, randomly sample max_cells cells without replacement and copy().
    """
    n = ad.n_obs
    if max_cells is None or max_cells >= n:
        print(f"[subsample_adata] Using all {n} cells (max_cells={max_cells})", flush=True)
        return ad

    rng = np.random.RandomState(seed)
    idx = rng.choice(n, size=max_cells, replace=False)
    idx.sort()
    print(
        f"[subsample_adata] Subsampling {max_cells} / {n} cells "
        f"(seed={seed})",
        flush=True,
    )
    return ad[idx].copy()
```