```python
# src/mantra/programs/cnmf.py
from __future__ import annotations

from dataclasses import dataclass, asdict
from pathlib import Path
from typing import Optional, Tuple, Dict, Any

import numpy as np
import scanpy as sc
import scipy.sparse as sp
from sklearn.decomposition import NMF
from sklearn.cluster import KMeans

'''
python scripts/cnmf.py \
  --ad data/interim/k562_gwps_unperturbed_qc.h5ad \
  --out out/programs/k562_cnmf_hvg75 \
  --k 75 \
  --use-hvg \
  --max-cells 50000 \
  --n-restarts 20 \
  --bootstrap-fraction 0.8 \
  --seed 7 \
  --name k562_cnmf_hvg75
'''

@dataclass
class CNMFConfig:
    """
    Configuration for (consensus) NMF.

    - n_components: K (number of programs)
    - n_restarts: how many independent NMF runs to aggregate
    - bootstrap_fraction: fraction of cells to use per run (bootstrap)
    """
    n_components: int

    # data selection
    use_hvg: bool = True
    obsm_key: Optional[str] = None   # e.g. "X_pca" or None -> use .X
    layer: Optional[str] = None      # optional alt layer (e.g. "counts")
    max_cells: Optional[int] = None  # cap cells before consensus (for speed)

    # consensus parameters
    n_restarts: int = 10
    bootstrap_fraction: float = 0.8  # per-run fraction of cells (0< f â\211¤ 1)

    # NMF hyperparameters
    max_iter: int = 400
    tol: float = 1e-4
    init: str = "nndsvda"
    random_state: int = 0
    alpha_W: float = 0.0
    alpha_H: float = 0.0
    l1_ratio: float = 0.0

    # KMeans for consensus clustering
    kmeans_max_iter: int = 300
    kmeans_tol: float = 1e-4
    kmeans_n_init: int = 10

    # bookkeeping
    name: str = "k562_cnmf"


# ---------------------------------------------------------------------
# Internal helpers
# ---------------------------------------------------------------------
```

```
def _select_matrix_from_anndata(
    ad: sc.AnnData,
    cfg: CNMFConfig,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Prepare the matrix X for NMF:

        X_sel: (n_cells_sel, G)
        cell_idx: indices of cells used
        gene_idx: indices of genes used (columns)

    We apply:
      - optional HVG restriction
      - optional obsm/layer selection
      - optional global cell subsampling (max_cells)
    """
    ad_view = ad

    # --- 1) HVG restriction ---
    if cfg.use_hvg and "highly_variable" in ad.var:
        hvg_mask = ad.var["highly_variable"].to_numpy().astype(bool)
        n_hvg = int(hvg_mask.sum())
        if n_hvg == 0:
            print("[CNMF] 'highly_variable' present but no genes flagged; using all genes."
)
        else:
            print(f"[CNMF] Restricting to HVGs: n_vars = {n_hvg}", flush=True)
            ad_view = ad[:, hvg_mask].copy()

    # --- 2) choose representation ---
    if cfg.obsm_key is not None:
        if cfg.obsm_key not in ad_view.obsm:
            raise KeyError(
                f"[CNMF] obsm_key={cfg.obsm_key!r} not found. "
                f"Available: {list(ad_view.obsm.keys())}"
            )
        X = ad_view.obsm[cfg.obsm_key]
        print(f"[CNMF] Using obsm[{cfg.obsm_key!r}] with shape {X.shape}", flush=True)
    elif cfg.layer is not None:
        if cfg.layer not in ad_view.layers:
            raise KeyError(
                f"[CNMF] layer={cfg.layer!r} not found. "
                f"Available: {list(ad_view.layers.keys())}"
            )
        X = ad_view.layers[cfg.layer]
        print(f"[CNMF] Using layer[{cfg.layer!r}] with shape {X.shape}", flush=True)
    else:
        X = ad_view.X
        print(f"[CNMF] Using ad.X with shape {X.shape}", flush=True)

    # --- 3) dense + non-negativity check ---
    if sp.issparse(X):
        X = X.toarray()
    X = np.asarray(X, dtype=np.float32)

    if (X < 0).any():
        raise ValueError(
            "[CNMF] Input matrix has negative entries. "
            "NMF assumes non-negative data. Check preprocessing."
        )

    # --- 4) optional global cell subsampling ---
    n_cells = X.shape[0]
    if cfg.max_cells is not None and n_cells > cfg.max_cells:
        rng = np.random.default_rng(cfg.random_state)
        cell_idx = np.sort(rng.choice(n_cells, size=cfg.max_cells, replace=False))
        print(
            f"[CNMF] Global subsample: cells {n_cells} â\206\222 {cfg.max_cells}",
            flush=True,
        )
    else:
        cell_idx = np.arange(n_cells, dtype=int)

    X_sel = X[cell_idx, :]
    gene_idx = np.arange(X_sel.shape[1], dtype=int)

    return X_sel, cell_idx, gene_idx


def _fit_single_nmf(
    X: np.ndarray,
    cfg: CNMFConfig,
    run_seed: int,
) -> Tuple[np.ndarray, np.ndarray, float, int]:
    """
    Fit a single NMF:

        X â\211\210 W_cells @ H
        W_cells: (n_cells_run, K)
        H:       (K, G)

    Return:
        W_cells, H, frob_rmse, n_iter
    """
    n_cells_run, G = X.shape
    K = int(cfg.n_components)

    print(
        f"[CNMF]   NMF run with seed={run_seed}, n_cells={n_cells_run}, G={G}, K={K}",
        flush=True,
    )

    nmf = NMF(
        n_components=K,
        init=cfg.init,
        max_iter=cfg.max_iter,
        tol=cfg.tol,
        random_state=run_seed,
        alpha_W=cfg.alpha_W,
        alpha_H=cfg.alpha_H,
        l1_ratio=cfg.l1_ratio,
        solver="cd",
        verbose=0,
    )

    W_cells = nmf.fit_transform(X)       # (n_cells_run, K)
    H = nmf.components_                  # (K, G)

    recon = W_cells @ H
    frob_err = np.linalg.norm(X - recon, ord="fro") / np.sqrt(X.size)

    print(
        f"[CNMF]   NMF done. Frobenius RMSE per entry={frob_err:.4f}, n_iter={nmf.n_iter_}"
,
        flush=True,
    )

    return W_cells, H, float(frob_err), int(nmf.n_iter_)


# ---------------------------------------------------------------------
# Public API: run_cnmf + save_cnmf_result
# ---------------------------------------------------------------------
```

```python
def run_cnmf(
    ad: sc.AnnData,
    cfg: CNMFConfig,
) -> Dict[str, Any]:
    """
    Consensus NMF:

      1) Select X from AnnData (HVG / obsm / layer / max_cells).
      2) For r = 1..n_restarts:
           - bootstrap cells
           - run NMF with seed = random_state + r
           - collect program loadings W_genes^{(r)} = H^{(r)T}  [G, K]
      3) Stack all programs into matrix P: (R*K, G)
           - optionally L2-normalize rows
      4) KMeans on P into K clusters
      5) Consensus W_genes = cluster_centers^T  [G, K]

    Returns a dict with:
      - "W_consensus": [G, K]
      - "programs_all": [R*K, G]
      - "cluster_labels": [R*K]
      - "frob_rmse_runs": list of RMSE per run
      - "n_iter_runs": list of iterations per run
      - "cell_idx": global cell indices used
      - "gene_idx": gene indices used
      - "cfg": CNMFConfig as dict
    """
    X_base, cell_idx, gene_idx = _select_matrix_from_anndata(ad, cfg)
    n_cells_base, G = X_base.shape
    K = int(cfg.n_components)
    R = int(cfg.n_restarts)

    print(
        f"[CNMF] Consensus NMF: base matrix shape={X_base.shape}, "
        f"K={K}, n_restarts={R}, bootstrap_fraction={cfg.bootstrap_fraction}",
        flush=True,
    )

    rng = np.random.default_rng(cfg.random_state)

    all_programs = []      # will hold (R*K, G)
    frob_rmse_runs = []
    n_iter_runs = []

    for r in range(R):
        # per-run bootstrap of cells
        if cfg.bootstrap_fraction < 1.0:
            n_cells_run = int(np.ceil(cfg.bootstrap_fraction * n_cells_base))
            # sample WITHOUT replacement is typical in cNMF
            boot_idx = np.sort(
                rng.choice(n_cells_base, size=n_cells_run, replace=False)
            )
            X_run = X_base[boot_idx, :]
        else:
            X_run = X_base

        run_seed = cfg.random_state + r
        W_cells, H, rmse, n_iter = _fit_single_nmf(X_run, cfg, run_seed)

        frob_rmse_runs.append(rmse)
        n_iter_runs.append(n_iter)

        # gene-level program loadings: W_genes [G, K]
        W_genes = H.T  # (G, K)

        # L2-normalize each program so clustering is about *shape* not scale
        norms = np.linalg.norm(W_genes, axis=0, keepdims=True) + 1e-8
        W_genes_norm = W_genes / norms  # (G, K)

        # append each program as a separate row
        for j in range(K):
            all_programs.append(W_genes_norm[:, j])

    programs_all = np.stack(all_programs, axis=0)  # (R*K, G)
    print(
        f"[CNMF] Collected {programs_all.shape[0]} program vectors "
        f"for consensus clustering.",
        flush=True,
    )

    # ----- KMeans clustering in program space -----
    kmeans = KMeans(
        n_clusters=K,
        random_state=cfg.random_state,
        n_init=cfg.kmeans_n_init,
        max_iter=cfg.kmeans_max_iter,
        tol=cfg.kmeans_tol,
        verbose=0,
    )
    labels = kmeans.fit_predict(programs_all)       # (R*K,)
    centers = kmeans.cluster_centers_              # (K, G) in normalized space

    # ----- Stability metrics per consensus program -----
    # 1) how many original programs ended up in each cluster?
    K = int(cfg.n_components)
    R = int(cfg.n_restarts)
    program_counts = np.bincount(labels, minlength=K)  # [K]

    # 2) run-level coverage: for each cluster k, in how many restarts
    #    did at least one of that run’s programs land in cluster k?
    # programs are ordered as [run0: 0..K-1], [run1: K..2K-1], ...
    run_coverage = np.zeros(K, dtype=np.float32)  # [K]
    for r in range(R):
        start = r * K
        end = (r + 1) * K
        labels_r = labels[start:end]  # cluster IDs for run r’s K programs
        # which clusters got at least one program from this run?
        present = np.unique(labels_r)
        run_coverage[present] += 1.0

    # normalize to fraction of runs (0..1)
    run_coverage /= float(R)

    print(
        "[CNMF] Stability summary: "
        f"coverage min={run_coverage.min():.3f}, "
        f"median={np.median(run_coverage):.3f}, "
        f"max={run_coverage.max():.3f}",
        flush=True,
    )


    # Undo L2 norm scalingâ\200\224not really needed, but we can renormalize to unit L2
    # and then rely on downstream scaling via Î\224E anyway.
    # We’ll just ensure non-negative and L2-normalize columns of W_consensus.
    W_consensus = centers.T  # (G, K)

    # enforce non-negativity (numerical noise)
    W_consensus = np.clip(W_consensus, a_min=0.0, a_max=None)

    # optional column-wise normalization (e.g., sum to 1 per program)
    col_sums = W_consensus.sum(axis=0, keepdims=True) + 1e-8
    W_consensus = W_consensus / col_sums

    print(
        f"[CNMF] Consensus W shape: {W_consensus.shape}. "
        f"Mean RMSE across runs: {np.mean(frob_rmse_runs):.4f}",
        flush=True,
    )

    result: Dict[str, Any] = {
        "W_consensus": W_consensus.astype(np.float32),     # [G, K]
        "programs_all": programs_all.astype(np.float32),   # [R*K, G]
        "cluster_labels": labels.astype(np.int32),         # [R*K]
        "program_counts": program_counts.astype(np.int32),# [K]
        "run_coverage": run_coverage.astype(np.float32),   # [K] in [0,1]
        "frob_rmse_runs": frob_rmse_runs,
        "n_iter_runs": n_iter_runs,
        "cell_idx": cell_idx,
        "gene_idx": gene_idx,
        "cfg": asdict(cfg),
    }


    return result


def save_cnmf_result(
    out_dir: Path,
    ad: sc.AnnData,
    result: Dict[str, Any],
    prefix: str = "k562_cnmf",
) -> None:
    """
    Save consensus NMF artifacts:

      - {prefix}_W_consensus.npy     : [G, K]
      - {prefix}_programs_all.npy    : [R*K, G]
      - {prefix}_cluster_labels.npy  : [R*K]
      - {prefix}_genes.npy
      - {prefix}_cells.npy
      - {prefix}_manifest.yml
    """
    out_dir.mkdir(parents=True, exist_ok=True)

    W_consensus = result["W_consensus"]           # [G, K]
    programs_all = result["programs_all"]         # [R*K, G]
    labels = result["cluster_labels"]             # [R*K]
    cell_idx = result["cell_idx"]
    gene_idx = result["gene_idx"]
    cfg = result["cfg"]
    program_counts = result.get("program_counts", None)
    run_coverage = result.get("run_coverage", None)

    genes = np.array(ad.var_names)[gene_idx]
    cells = np.array(ad.obs_names)[cell_idx]

    # main artifacts
    np.save(out_dir / f"{prefix}_W_consensus.npy", W_consensus)
    np.save(out_dir / f"{prefix}_programs_all.npy", programs_all)
    np.save(out_dir / f"{prefix}_cluster_labels.npy", labels)
    np.save(out_dir / f"{prefix}_cluster_labels.npy", labels)
    np.save(out_dir / f"{prefix}_cluster_labels.npy", labels)
    np.save(out_dir / f"{prefix}_genes.npy", genes)
    np.save(out_dir / f"{prefix}_cells.npy", cells)
    if program_counts is not None:
        np.save(out_dir / f"{prefix}_program_counts.npy", program_counts)
    if run_coverage is not None:
        np.save(out_dir / f"{prefix}_run_coverage.npy", run_coverage)

    # lightweight manifest
    import yaml  # type: ignore
    manifest = {
        "shape": {
            "W_consensus": list(W_consensus.shape),
            "programs_all": list(programs_all.shape),
        },
        "genes_n": int(genes.size),
        "cells_n": int(cells.size),
        "genes_head": [str(g) for g in genes[:10]],
        "cells_head": [str(c) for c in cells[:10]],
        "rmse_runs": [float(x) for x in result["frob_rmse_runs"]],
        "n_iter_runs": [int(x) for x in result["n_iter_runs"]],
        "config": cfg,
    }

    if program_counts is not None:
        manifest["program_counts"] = program_counts.tolist()
    if run_coverage is not None:
        manifest["run_coverage"] = run_coverage.tolist()

    with (out_dir / f"{prefix}_manifest.yml").open("w") as f:
        yaml.safe_dump(manifest, f)

    print(f"[CNMF] Saved consensus W + manifest to {out_dir}", flush=True)
```

```python
# src/mantra/programs/config.py
from __future__ import annotations

from mantra.config import CNMFConfig, CNMFResults

__all__ = [
    "CNMFConfig",
    "CNMFResults",
]
```

```python
# src/mantra/programs/__init__.py

from .cnmf import run_cnmf    # or whatever your main entry point is
from .config import CNMFConfig, CNMFResults

__all__ = [
    "run_cnmf",
    "CNMFConfig",
    "CNMFResults",
]
```