STAT 5244 – Unsupervised Learning

Homework 1

Name: Chuyang Su UNI: cs4570

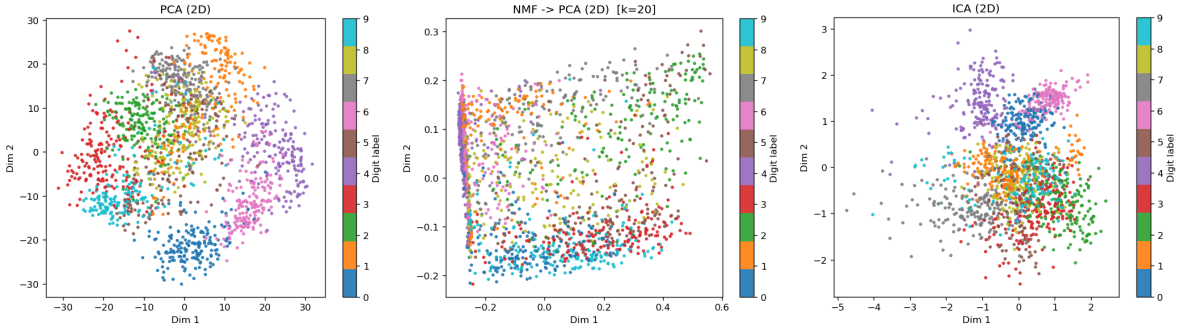
1 Dimension Reduction on Digits Data.

1.1 Apply linear dimension reduction techniques.

In this experiment, I applied three linear dimension reduction methodsand compared their performance on the `scikit-learn` *Digits* dataset ($n = 1797$, $p = 64$).

Each method projects the data into a two-dimensional latent space, on which I visualized the results and quantitatively evaluated their ability to separate the ten digit classes.

The results of this experiment are summarized below.



(a) PCA (b) NMF→PCA (c) ICA

Figure 1: 2D embeddings of the digits data using PCA, NMF→PCA, and ICA. Colors denote true digit labels.

Method	ARI	NMI	Silhouette
PCA	**0.3614**	**0.5190**	0.3993
NMF → PCA	0.1588	0.2961	**0.4080**
ICA	0.3175	0.4567	0.3673

Table 1: Quantitative comparison of linear dimension reduction methods. The best scores for each metric are bolded.

PCA. For PCA, I retained the first two principal components and projected the samples into the 2D subspace they span, which preserves the primary directions of variance. As for hyperparameters, PCA has very few tunable parameters—the main one being the number of principal components. For ease of visualization, I set the number of components to 2

(PC=2) in this experiment. Figure 1a shows that the data are well dispersed, and digits such as 0, 3, 4, 6, and 9 form clearly separated clusters. Quantitatively, PCA achieved an ARI of 0.3614, an NMI of 0.5190, and a Silhouette score of 0.3993. Except for the Silhouette score, PCA obtained the highest values among the three methods. This indicates that PCA effectively separates the digits and maintains a high level of consistency with the true labels. Although its Silhouette value (approximately 0.4) is not the highest, it still suggests reasonably compact and well-separated clusters. This minor difference can be attributed to slight overlaps between neighboring clusters in the 2D embedding, even though the overall structure aligns well with the ground-truth classes.

In addition to the 2D embedding, I plotted the PCA scree plot and a bar chart of the top ten principal components' explained variance, as shown in Figure 2. Both plots reveal that the first three components contribute substantially more variance than the rest, with the third component explaining slightly less variance than the first two but significantly more than the fourth. This supports the observation that the 2D projection loses some discriminative information, which explains why the best ARI achieved by PCA remains moderate (0.3614).

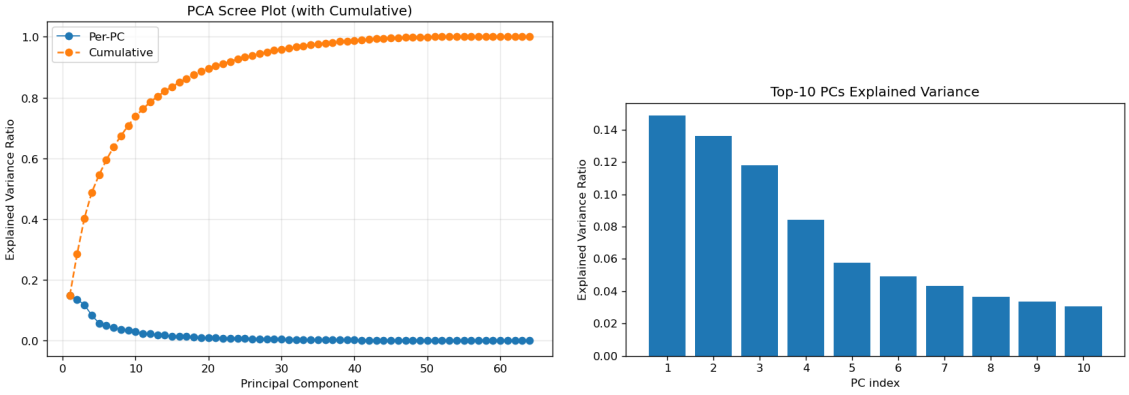


Figure 2: (Left) Scree plot showing the variance explained by each principal component; (Right) bar chart of the top-10 PCs' explained variance ratios.

Furthermore, I visualized the top ten PCA component images (Figure 3), which illustrate the principal modes of variation across digits.

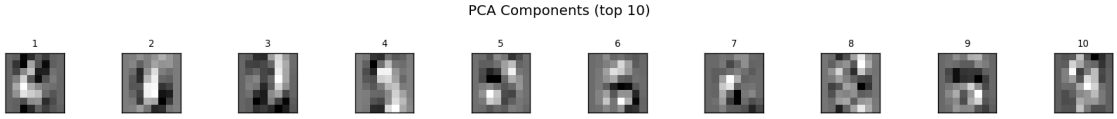


Figure 3: Top ten PCA components visualized as 8×8 basis images.

NMF→PCA. For the NMF method, I first determined the optimal number of components by minimizing the reconstruction error, which yielded $k = 20$. This means the data were decomposed into 20 non-negative basis vectors. The resulting coefficient matrix W was

then compressed into a 2D embedding space using PCA for visualization and comparison purposes, and the final result is shown in Figure 1(b).

As observed in the plot, NMF fails to clearly separate the digits in the 2D space. This is consistent with the quantitative results, where NMF achieved the lowest ARI (0.1588) and NMI (0.2961) among all methods. These findings indicate that the NMF representation captures local, part-based features rather than global discriminative structures, and that the subsequent PCA compression may distort these part-based patterns, reducing the overall interpretability.

On the other hand, NMF obtained the highest Silhouette score (0.4080), slightly higher than PCA. The embedding shows that nearly all samples are concentrated in the positive subspace, with only a small portion extending below zero (no less than -0.2). Even after PCA compression, this non-negativity-induced structure is largely preserved, resulting in an asymmetric distribution that nevertheless exhibits slightly better cluster compactness than PCA. This suggests that some of the features extracted by NMF may be better suited to local clustering in this dataset. In particular, compared to PCA, NMF tends to focus on localized regions of variation rather than global variance directions, which likely contributes to the formation of tighter, more compact clusters.

Similar to the PCA case where excluding the third principal component led to a loss of explanatory power, it is plausible that the 2D projection of NMF also omits important structural information. Specifically, the visualization reveals that the first principal component successfully separates digit "4" along the left margin, while the second principal component distinguishes digits "9" (light blue) and "0" from the rest of the samples, forming a clear boundary near the bottom region of the plot. This indicates that the first PC primarily captures the unique pattern of the digit "4", whereas the second PC isolates the shapes shared by "0" and "9". A potential 3D embedding including an additional principal component might further clarify the remaining overlapping clusters visible in the upper-right portion of the 2D space.

Finally, I visualized the NMF basis components, as shown in Figure 4. These components represent localized stroke-like patterns, providing an interpretable decomposition of the digits into additive parts.

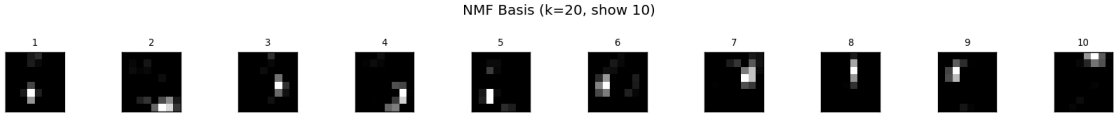


Figure 4: Top ten PCA components visualized as 8×8 basis images.

ICA. For the ICA method, I applied FastICA with two components after standardizing the data to ensure consistent feature scaling. As shown in Figure 1(c) and the summary table, ICA exhibits the most balanced overall performance among the three linear methods. Some clusters—notably digits 4, 6, and 0—are well separated and relatively compact, outperforming both PCA and NMF in local structure preservation. However, the remaining digits appear more mixed than in PCA, though less dispersed than in NMF. Visually,

the embedding forms two major groups: one consisting primarily of digits 4, 6, and 0, and another containing the other seven digits.

Quantitatively, ICA's ARI (0.3175) and NMI (0.4567) values lie between those of PCA and NMF, while its Silhouette score (0.3673) is the lowest among the three. This aligns with the visual interpretation: ICA achieves moderate global separability but weaker overall cluster compactness.

Regarding hyperparameters, ICA provides limited tuning options. Here I set `components`=2, reducing the data to a 2D space, and standardized all features prior to decomposition. Standardization is a conventional preprocessing step for ICA, as its underlying assumption relies on statistical independence between components. Consequently, the resulting embedding appears nearly spherical and centered around the origin—a distribution consistent with ICA's model assumptions but inconsistent with the inherent non-spherical structure of handwritten digits. This mismatch explains the weaker performance of ICA in this task.

Finally, the ICA component images (Figure 5) reveal contrast-like and edge-detecting patterns that emphasize stroke boundaries, differing from the smoother, global variations captured by PCA and the localized parts extracted by NMF.

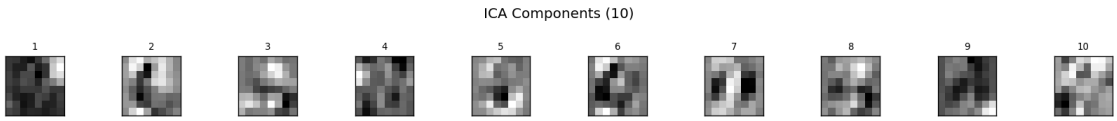


Figure 5: Top ten PCA components visualized as 8×8 basis images.

Overall Discussion. In summary, among the three linear dimension reduction techniques, PCA stands out as the most effective approach. It achieved the highest ARI (0.3614) and NMI (0.5190), and a Silhouette score (0.3993) close to the best. Visually, PCA produced the clearest and most interpretable clusters in the 2D embedding, separating several digits (such as 0, 3, 4, 6, and 9) distinctly. Given that PCA captures global variance directions, its performance would likely improve further in higher-dimensional embeddings, where additional components could better preserve discriminative variance.

1.2 Apply manifold learning approaches.

Methodology Each model was configured with typical hyperparameters and applied to the same standardized input features. The detailed settings are summarized below:

- **Kernel PCA (RBF)** — RBF kernel with $\gamma = 0.102$, chosen automatically based on the median pairwise distance.
- **Spectral Embedding** — Laplacian eigenmap–based manifold embedding with $n_{\text{neighbors}} = 10$.
- **Classical MDS** — Closed-form double-centered Euclidean distance eigendecomposition.

- **Metric MDS** — Stress minimization–based metric scaling via gradient descent.
- **t-SNE** — Perplexity of 30, PCA initialization, adaptive learning rate.
- **UMAP** — $n_{\text{neighbors}} = 15$, min_dist $= 0.1$.
- **Autoencoder (PyTorch)** — Fully connected symmetric architecture with input 64, hidden layer 32, bottleneck dimension 2, ReLU activations, sigmoid reconstruction, trained for 50 epochs with batch size 128 and learning rate 10^{-3}.

All embeddings were evaluated in 2D using k-means clustering ($k = 10$) with three quantitative metrics: Adjusted Rand Index (ARI), Normalized Mutual Information (NMI), and Silhouette coefficient.

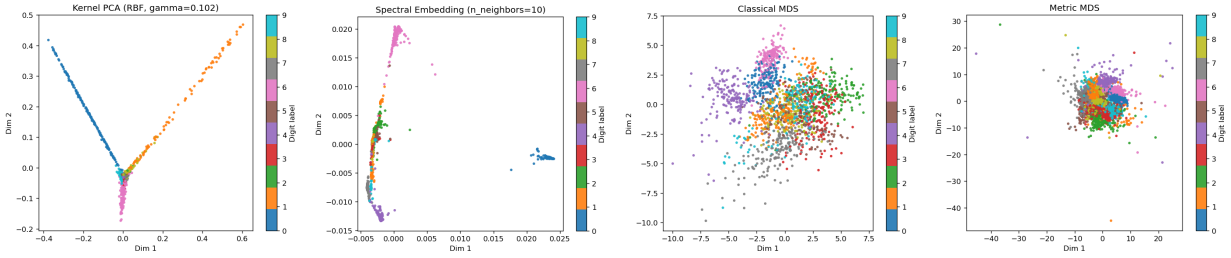


Figure 6: 2D embeddings of the digits dataset using kernel PCA, spectral embedding, classical mds and metric mds.

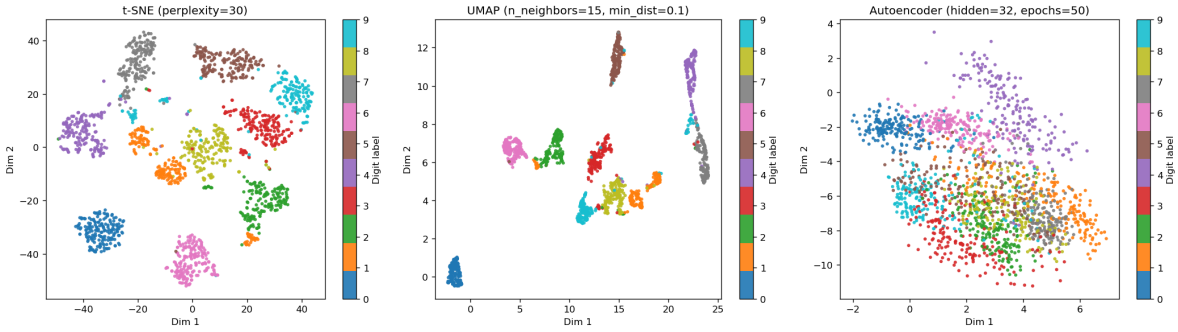


Figure 7: 2D embeddings of the digits dataset using t-SNE, UMAP and Pytorch Autoencoder.

Discussion The quantitative results and visualizations reveal clear distinctions among methods. UMAP and t-SNE perform best overall, yielding the strongest alignment with true digit labels—UMAP achieves the highest ARI (0.866) and NMI (0.891), while t-SNE produces similarly well-separated and visually coherent clusters. Spectral Embedding captures both global and local structures but exhibits partial overlap between classes. Kernel PCA attains the highest Silhouette score (0.754), reflecting locally compact but label-misaligned clusters, whereas Classical and Metric MDS show limited nonlinear capacity. The autoencoder provides a moderate nonlinear representation (ARI 0.35, NMI 0.50), illustrating

Method	ARI	NMI	Silhouette
Kernel PCA	0.0511	0.3357	**0.7541**
Spectral Embedding	0.4730	0.6681	0.6119
Classical MDS	0.3262	0.4647	0.3771
Metric MDS	0.3461	0.4722	0.3503
t-SNE	0.7695	0.8339	0.5731
UMAP	**0.8662**	**0.8912**	0.6866
Autoencoder	0.3476	0.4992	0.3958

Table 2: Quantitative comparison of seven embedding methods on the digits dataset ($k = 10$).

its potential but also the constraint of a shallow 2D bottleneck. Overall, UMAP and t-SNE best uncover the intrinsic manifold structure of handwritten digits, highlighting the strength of nonlinear embedding methods in revealing meaningful low-dimensional organization in high-dimensional data.

1.3 Discussion

Nonlinear manifold methods such as **UMAP** and **t-SNE** outperform linear approaches because they better preserve both local neighborhood relationships and global manifold geometry, thus capturing the intrinsic nonlinear structure of handwritten digits. Among all methods, **UMAP** performs best, achieving the highest ARI (0.866) and NMI (0.891) while forming compact and well-separated clusters that closely align with true digit classes. The most representative visualization is the **UMAP 2D embedding**, which clearly displays ten distinct clusters corresponding to the ten digits.

2 Open-Ended Data Analysis - Breast Cancer gene expression data.

2.1 Preprocessing

After loading the data (shape 445×359), six clinical columns were identified: `subtype`, `er_status`, `pr_status`, `her2_status`, `node`, and `metastasis`, leaving 353 gene expression features. No missing values or zero-variance genes were found, indicating high data quality. Each gene feature was standardized via Z-score normalization. Descriptive statistics of the clinical variables are summarized below:

- **Subtype:** Luminal A (200), Luminal B (106), Basal-like (79), HER2-enriched (53), Normal-like (7).
- **ER-Status:** Positive (339), Negative (100), Performed but Not Available (2), Indeterminate (2), Not Performed (2).

- **PR-Status:** Positive (291), Negative (147), Indeterminate (3), Performed but Not Available (2), Not Performed (2).
- **HER2-Status:** Negative (371), Positive (65), Equivocal (5), Not Available (4).
- **Node:** mean = 0.73, std = 0.87, range = [0, 3].
- **Metastasis:** mean = 0.025, std = 0.155 (mostly non-metastatic samples).

2.2 Methodology

Five dimension reduction methods were applied to the standardized gene expression matrix to obtain two-dimensional embeddings:

1. **Principal Component Analysis (PCA)** — linear orthogonal projection capturing maximal variance.
2. **Non-negative Matrix Factorization (NMF)** — parts-based representation using non-negative constraints (run on non-standardized data to ensure non-negativity).
3. **Spectral Embedding** — manifold learning based on graph Laplacian eigenvectors.
4. **t-SNE** — nonlinear embedding preserving local neighborhood structures.
5. **UMAP** — manifold approximation balancing local and global relationships.

Each embedding was visualized by coloring the points according to all six available clinical variables. However, for brevity and visual clarity, only the results colored by **Subtype** are shown here as representative examples. This selection illustrates overall trends while the complete set of figures (for all six variables across all five methods) is provided in the supplementary materials.

To quantitatively evaluate clustering quality with respect to molecular subtypes, k-means clustering ($k = 5$) was applied to each embedding, and three metrics were computed: Adjusted Rand Index (ARI), Normalized Mutual Information (NMI), and Silhouette coefficient.

2.3 Results

Overall, nonlinear manifold-based methods (t-SNE and UMAP) outperformed linear approaches in terms of ARI and NMI, suggesting that the underlying structure of gene expression data is highly nonlinear. UMAP achieved the highest Silhouette score (0.44), indicating that it produces well-separated clusters with clear boundaries, while t-SNE yielded the best ARI (0.21) and NMI (0.27), showing the strongest alignment with the known PAM50 subtypes. PCA achieved moderate performance, confirming its ability to preserve global variance but limited capacity to capture complex nonlinear relations. NMF performed comparably, producing slightly denser local clusters (reflected in its higher Silhouette score) but weaker subtype separation. Spectral embedding produced the most compact clusters but with limited biological interpretability due to its sensitivity to graph construction.

Method	ARI	NMI	Silhouette
PCA	0.1857	0.2399	0.3311
NMF	0.1712	0.2536	0.3842
Spectral	0.1421	0.2300	0.4020
t-SNE	**0.2074**	**0.2704**	0.3984
UMAP	0.2028	0.2780	**0.4376**

Table 3: Quantitative comparison of five dimension reduction methods on the BRCA dataset. Evaluation is based on clustering alignment with PAM50 subtypes ($k = 5$).

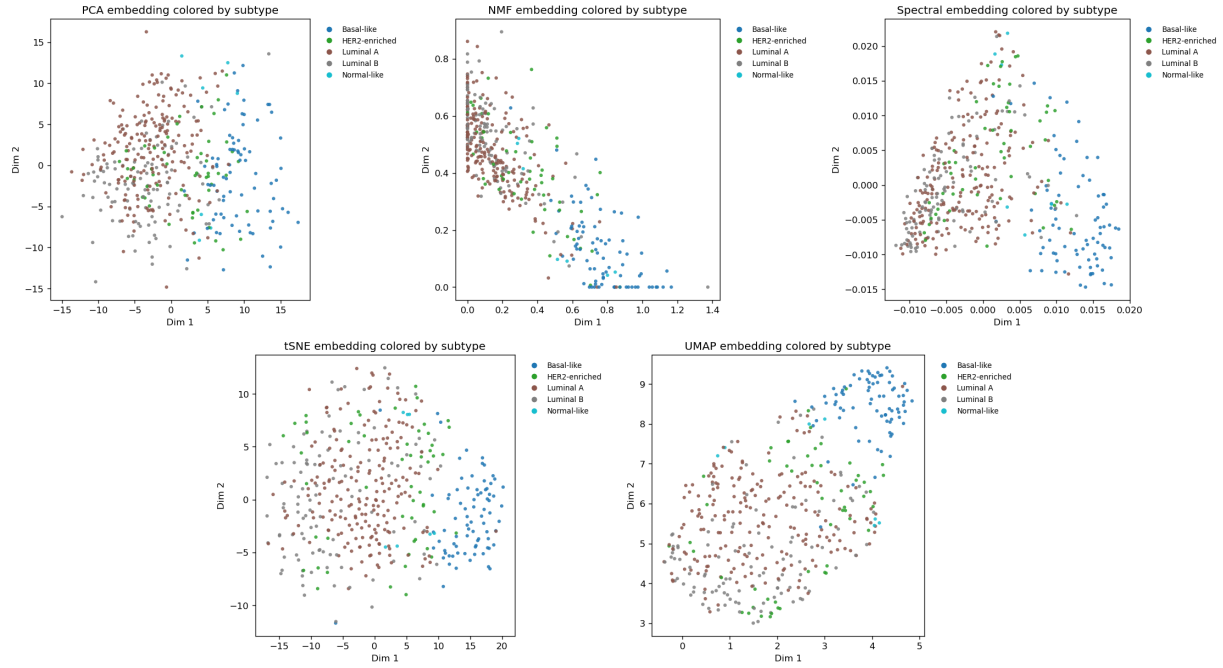


Figure 8: Two-dimensional embeddings of the BRCA gene expression data using five methods (PCA, NMF, Spectral, t-SNE, and UMAP), colored by molecular Subtype. All embeddings were also generated for the remaining five clinical variables, but only Subtype-colored results are displayed here for clarity.

2.4 Discussion

Given that no missing or low-variance genes were present, preprocessing had minimal impact on the raw data distribution. The observed performance differences mainly stem from the intrinsic characteristics of each method. The superior performance of UMAP and t-SNE suggests that manifold learning effectively captures nonlinear relationships among gene expressions that correspond to known molecular subtypes. Future work could extend this analysis by increasing latent dimensionality or incorporating autoencoders for deeper nonlinear representations.

A Appendix: Code Implementation

A.1 Problem 1a.

```python
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-10-08 17:16:54
LastEditTime: 2025-10-08 18:21:50
FilePath: /Unsupervised-Learning-Homework/Homework 1/Code/Problem_1_a.py
Description:
    This is the code part of GR5244 Unsupervised Learning Homework 1 Part 1a.
'''
import argparse
import os
from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.decomposition import PCA, NMF, FastICA
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score,
                                            silhouette_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import csv


# ----------------------------
# Utils
# ----------------------------
def ensure_dir(p):
    Path(p).mkdir(parents=True, exist_ok=True)

def plot_embedding(X2d, y, title, outpath):
    plt.figure(figsize=(6, 5), dpi=120)
    scatter = plt.scatter(X2d[:, 0], X2d[:, 1], c=y, s=12, cmap='tab10', alpha
                                            =0.9, edgecolors='none')
    plt.title(title)
    plt.xlabel("Dim 1")
    plt.ylabel("Dim 2")
    cbar = plt.colorbar(scatter, ticks=range(10))
    cbar.set_label("Digit label")
    plt.tight_layout()
    plt.savefig(outpath, bbox_inches='tight')
    plt.close()


def plot_components(components, img_shape, n_show, title, outpath):
    n = min(n_show, components.shape[0])
    cols = 10 if n >= 10 else n
    rows = int(np.ceil(n / cols))
```

```
    plt.figure(figsize=(1.4*cols, 1.4*rows), dpi=120)
    for i in range(n):
        ax = plt.subplot(rows, cols, i + 1)
        ax.imshow(components[i].reshape(img_shape), cmap='gray')
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_title(f"{i+1}", fontsize=8)
    plt.suptitle(title, y=1.02)
    plt.tight_layout()
    plt.savefig(outpath, bbox_inches='tight')
    plt.close()

def plot_pca_scree(explained_var_ratio, outpath):
    """Scree plot: per-PC variance ratio + cumulative curve"""
    import numpy as np
    import matplotlib.pyplot as plt

    r = np.array(explained_var_ratio)
    cum = np.cumsum(r)

    plt.figure(figsize=(7, 5), dpi=120)
    #
    plt.plot(np.arange(1, len(r)+1), r, marker='o', linewidth=1)
    #
    plt.plot(np.arange(1, len(r)+1), cum, marker='o', linestyle='--')
    plt.xlabel("Principal Component")
    plt.ylabel("Explained Variance Ratio")
    plt.title("PCA Scree Plot (with Cumulative)")
    plt.legend(["Per-PC", "Cumulative"])
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(str(outpath), bbox_inches='tight')
    plt.close()


def plot_pca_topk_bar(explained_var_ratio, k, outpath):
    """Bar chart for top-k PCs' explained variance ratio"""
    import numpy as np
    import matplotlib.pyplot as plt

    r = np.array(explained_var_ratio)[:k]
    idx = np.arange(1, len(r)+1)

    plt.figure(figsize=(7, 4), dpi=120)
    plt.bar(idx, r)
    plt.xlabel("PC index")
    plt.ylabel("Explained Variance Ratio")
    plt.title(f"Top-{k} PCs Explained Variance")
    plt.xticks(idx)
    plt.tight_layout()
    plt.savefig(str(outpath), bbox_inches='tight')
    plt.close()


```

```python

def kmeans_scores(X2d, y, seed):
    km = KMeans(n_clusters=10, n_init='auto', random_state=seed)
    labels = km.fit_predict(X2d)
    ari = adjusted_rand_score(y, labels)
    nmi = normalized_mutual_info_score(y, labels)
    sil = silhouette_score(X2d, labels)
    return ari, nmi, sil


# ----------------------------
# Main pipeline for 1(a)
# ----------------------------
def run(seed=0, nmf_ks=(10, 15, 20), outdir="Homework 1/Latex"):
    ensure_dir(outdir)

    # Load data
    digits = load_digits()
    X = digits.data.astype(float)
    y = digits.target
    img_shape = (8, 8)

    # ----------------------------
    # PCA (2D embedding + components)
    # ----------------------------
    pca_2 = PCA(n_components=2, random_state=seed)
    X_pca_2 = pca_2.fit_transform(X)
    plot_embedding(X_pca_2, y, "PCA (2D)", f"{outdir}/pca_2d.png")

    # For components visualization, use more PCs (e.g., 10)
    pca_10 = PCA(n_components=10, random_state=seed).fit(X)
    plot_components(pca_10.components_, img_shape, n_show=10,
                    title="PCA Components (top 10)", outpath=f"{outdir}/
                                                              pca_components.png"
                                                              )

    pca_ari, pca_nmi, pca_sil = kmeans_scores(X_pca_2, y, seed)

    pca_full = PCA(n_components=min(X.shape), random_state=seed).fit(X)
    explained = pca_full.explained_variance_ratio_

    plot_pca_scree(explained, Path(outdir) / "pca_scree.png")
    plot_pca_topk_bar(explained, k=10, outpath=Path(outdir) / "pca_top10_var.
                                             png")


    # ----------------------------
    # NMF (grid over k), then 2D via PCA-on-W for visualization
    # ----------------------------
    X_nonneg = X - X.min() if X.min() < 0 else X
    best_nmf = None
    best_rec = np.inf
    best_k = None

```

```
    for k in nmf_ks:
        nmf = NMF(n_components=k, init='nndsvda', random_state=seed, max_iter=
                                                    1000)
        W = nmf.fit_transform(X_nonneg)
        rec = nmf.reconstruction_err_
        if rec < best_rec:
            best_rec = rec
            best_nmf = nmf
            best_k = k

    # Use the best NMF
    W_best = best_nmf.transform(X_nonneg)  # (n_samples, best_k)
    # Reduce W to 2D for visualization
    nmf_to2 = PCA(n_components=2, random_state=seed)
    X_nmf_2 = nmf_to2.fit_transform(W_best)
    plot_embedding(X_nmf_2, y, f"NMF -> PCA (2D)  [k={best_k}]", f"{outdir}/
                                            nmf_2d.png")

    # Visualize NMF basis (H)
    plot_components(best_nmf.components_, img_shape, n_show=10,
                    title=f"NMF Basis (k={best_k}, show 10)", outpath=f"{
                                                            outdir}/
                                                            nmf_components.png"
                                                            )

    nmf_ari, nmf_nmi, nmf_sil = kmeans_scores(X_nmf_2, y, seed)

    # ----------------------------
    # ICA (2D embedding + components)
    # ----------------------------
    ica_pipeline = make_pipeline(StandardScaler(with_std=True), FastICA(
                                            n_components=2, random_state=seed,
                                            max_iter=1000))
    X_ica_2 = ica_pipeline.fit_transform(X)
    plot_embedding(X_ica_2, y, "ICA (2D)", f"{outdir}/ica_2d.png")

    # For components display, fit a separate ICA with more comps (e.g., 10) on
                                             standardized X
    scaler = StandardScaler(with_std=True)
    X_std = scaler.fit_transform(X)
    ica_10 = FastICA(n_components=10, random_state=seed, max_iter=1000).fit(
                                            X_std)
    plot_components(ica_10.mixing_.T, img_shape, n_show=10,  # mixing_.T
                                            component ""images
                    title="ICA Components (10)", outpath=f"{outdir}/
                                                            ica_components.png"
                                                            )

    ica_ari, ica_nmi, ica_sil = kmeans_scores(X_ica_2, y, seed)

    # ----------------------------
    # Save metrics
    # ----------------------------
    metrics_path = f"{outdir}/part1a_metrics.csv"
```

```
    with open(metrics_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Method", "Params", "ARI", "NMI", "Silhouette", "Notes"])
        writer.writerow(["PCA", "n_components=2", f"{pca_ari:.4f}", f"{pca_nmi:.4f}", f"{pca_sil:.4f}", "2D embedding"])
        writer.writerow(["NMF -> PCA", f"k={best_k}, then 2D PCA", f"{nmf_ari:.4f}", f"{nmf_nmi:.4f}", f"{nmf_sil:.4f}",
                         f"best reconstruction k among {list(nmf_ks)} (err={best_rec:.4f})"])
        writer.writerow(["ICA", "n_components=2 (with standardization)", f"{ica_ari:.4f}", f"{ica_nmi:.4f}", f"{ica_sil:.4f}", "2D embedding"])

    # Also print a neat summary
    print("\n=== Part 1(a) - Linear Methods on Digits ===")
    print(f"[PCA]       ARI={pca_ari:.4f}  NMI={pca_nmi:.4f}  Silhouette={pca_sil:.4f}")
    print(f"[NMF -> PCA]   ARI={nmf_ari:.4f}  NMI={nmf_nmi:.4f}  Silhouette={nmf_sil:.4f}  (best k={best_k}, recon_err={best_rec:.4f})")
    print(f"[ICA]       ARI={ica_ari:.4f}  NMI={ica_nmi:.4f}  Silhouette={ica_sil:.4f}")
    print(f"\nSaved figures and metrics to: {outdir}/")
    print("Figures:")
    print(" - pca_2d.png, pca_components.png")
    print(" - nmf_2d.png, nmf_components.png")
    print(" - ica_2d.png, ica_components.png")
    print("Table:")
    print(f" - {Path(metrics_path).name}")


# ----------------------------
# Entry
# ----------------------------
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed", type=int, default=25)
    parser.add_argument("--outdir", type=str, default="Homework 1/Latex/Results/Problem_1_a")
    parser.add_argument("--nmf_ks", type=int, nargs="+", default=[10, 15, 20])
    args = parser.parse_args()
    run(seed=args.seed, nmf_ks=tuple(args.nmf_ks), outdir=args.outdir)
```

A.2 Problem 1b.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
"""
Author: Chuyang Su <cs4570@columbia.edu>
Date: 2025-10-10
FilePath: /Unsupervised-Learning-Homework/Homework 1/Code/Problem_1_b.py
Description:
    GR5244 HW1 Part 1(b): Manifold learning on sklearn digits (n=1797, p=64).
        Methods: Kernel PCA, Spectral Embedding, Classical MDS, Metric MDS, t-SNE, UMAP, Autoencoder.
"""

from pathlib import Path
import argparse
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.decomposition import KernelPCA
from sklearn.manifold import SpectralEmbedding, MDS, TSNE
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score, silhouette_score, pairwise_distances
from sklearn.cluster import KMeans
import umap

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
# ----------------------------
# Utils
# ----------------------------
def ensure_dir(p):
    Path(p).mkdir(parents=True, exist_ok=True)

def set_seed(seed: int):
    import random, os
    random.seed(seed); np.random.seed(seed)
    torch.manual_seed(seed); torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

def get_device():
    return torch.device("cuda" if torch.cuda.is_available() else "cpu")


def plot_embedding(X2d, y, title, outpath):
    """Simple 2D scatter with digits colormap; no axes ticks; safe colormap API."""
    cmap = plt.colormaps.get_cmap('tab10')
    plt.figure(figsize=(6, 5), dpi=120)
    sc = plt.scatter(X2d[:, 0], X2d[:, 1], c=y, s=12, cmap=cmap, alpha=0.9, edgecolors='none')
    plt.title(title)
    plt.xlabel("Dim 1"); plt.ylabel("Dim 2")
```

```
    cbar = plt.colorbar(sc, ticks=range(10))
    cbar.set_label("Digit label")
    plt.tight_layout()
    plt.savefig(outpath, bbox_inches='tight')
    plt.close()


def kmeans_scores(X2d, y, seed, k=10):
    km = KMeans(n_clusters=k, random_state=seed, n_init=10)
    labels = km.fit_predict(X2d)
    ari = adjusted_rand_score(y, labels)
    nmi = normalized_mutual_info_score(y, labels)
    sil = silhouette_score(X2d, labels)
    return ari, nmi, sil


def classical_mds(X, n_components=2):
    """
    Classical MDS (Torgerson/Gower): eigendecomposition of double-centered
    squared-distance matrix. Returns low-d coords (can be rotated/reflected).
    """
    D = pairwise_distances(X, metric='euclidean')
    D2 = D ** 2
    n = D2.shape[0]
    J = np.eye(n) - np.ones((n, n)) / n
    B = -0.5 * J @ D2 @ J
    # Eigen-decomposition
    evals, evecs = np.linalg.eigh(B)
    # Take top components
    idx = np.argsort(evals)[::-1]
    evals = evals[idx]; evecs = evecs[:, idx]
    # Keep only positive eigenvalues
    pos = evals > 0
    evals = evals[pos]; evecs = evecs[:, pos]
    evals_k = evals[:n_components]
    evecs_k = evecs[:, :n_components]
    X_emb = evecs_k * np.sqrt(np.maximum(evals_k, 0))
    return X_emb


def auto_gamma_rbf(X):
    """
    Heuristic gamma for RBF kernel (Kernel PCA): gamma = 1 / median(pairwise
                                                    distance).
    Scale-robust; avoids manual guesswork when not provided.
    """
    d = pairwise_distances(X, metric='euclidean')
    med = np.median(d)
    if med <= 0:
        return 1.0
    return 1.0 / med

# Autoencoder

```

```python
class AE(nn.Module):
    def __init__(self, input_dim: int, hidden: int = 32, bottleneck: int = 2):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, bottleneck)  #
        )
        self.decoder = nn.Sequential(
            nn.Linear(bottleneck, hidden),
            nn.ReLU(),
            nn.Linear(hidden, input_dim),
            nn.Sigmoid()  #   [0,1]
        )

    def forward(self, x):
        z = self.encoder(x)
        xhat = self.decoder(z)
        return xhat, z


@torch.no_grad()
def encode_dataset(model: AE, loader: DataLoader, device):
    model.eval()
    zs = []
    for (xb,) in loader:
        xb = xb.to(device)
        _, z = model(xb)
        zs.append(z.cpu().numpy())
    return np.concatenate(zs, axis=0)


def train_autoencoder_pytorch(
    X_minmax: np.ndarray,
    seed: int = 0,
    hidden: int = 32,
    bottleneck: int = 2,
    epochs: int = 50,
    batch_size: int = 128,
    lr: float = 1e-3,
):
    set_seed(seed)
    device = get_device()

    X_tensor = torch.tensor(X_minmax, dtype=torch.float32)
    ds = TensorDataset(X_tensor)
    dl = DataLoader(ds, batch_size=batch_size, shuffle=True, drop_last=False)

    model = AE(input_dim=X_minmax.shape[1], hidden=hidden, bottleneck=
                                            bottleneck).to(device)
    opt = optim.Adam(model.parameters(), lr=lr)
    crit = nn.MSELoss()
    model.train()
    for _ in range(epochs):
```

```
        for (xb,) in dl:
            xb = xb.to(device)
            xhat, _ = model(xb)
            loss = crit(xhat, xb)
            opt.zero_grad()
            loss.backward()
            opt.step()

    #      2D
    #     DataLoader
    Z = encode_dataset(model, DataLoader(ds, batch_size=1024, shuffle=False),
                                          device)
    return Z


# ----------------------------
# Main pipeline for 1(b)
# ----------------------------
def run(seed=25,
        outdir="Homework 1/Latex/Results/Problem_1_b",
        kpca_gamma=None,
        spectral_n_neighbors=10,
        tsne_perplexity=30,
        umap_n_neighbors=15,
        umap_min_dist=0.1,
        ae_hidden=32,
        ae_epochs=50,
        ae_batch=128,
        ae_lr=1e-3):
    set_seed(seed)
    ensure_dir(outdir)

    # Load data
    digits = load_digits()
    X = digits.data.astype(float)         # (1797, 64)
    y = digits.target                     # (1797,)

    # Standardize (common for manifold learning)
    X_std = StandardScaler().fit_transform(X)

    # 1) Kernel PCA (RBF)
    gamma = kpca_gamma if kpca_gamma is not None else auto_gamma_rbf(X_std)
    kpca = KernelPCA(n_components=2, kernel='rbf', gamma=gamma,
                                          fit_inverse_transform=False,
                                          random_state=seed)
    X_kpca = kpca.fit_transform(X_std)
    plot_embedding(X_kpca, y, f"Kernel PCA (RBF, gamma={gamma:.3g})", f"{
                                          outdir}/kpca_2d.png")
    kpca_ari, kpca_nmi, kpca_sil = kmeans_scores(X_kpca, y, seed)

    # 2) Spectral Embedding
    se = SpectralEmbedding(n_components=2, n_neighbors=spectral_n_neighbors,
                                          random_state=seed)
    X_se = se.fit_transform(X_std)
```

```
    plot_embedding(X_se, y, f"Spectral Embedding (n_neighbors={spectral_n_neighbors})", f"{outdir}/spectral_2d.png")
    se_ari, se_nmi, se_sil = kmeans_scores(X_se, y, seed)

    # 3) Classical MDS (closed-form)
    X_cmds = classical_mds(X_std, n_components=2)
    plot_embedding(X_cmds, y, "Classical MDS", f"{outdir}/classical_mds_2d.png")
    cmds_ari, cmds_nmi, cmds_sil = kmeans_scores(X_cmds, y, seed)

    # 4) Metric MDS (sklearn MDS, metric=True)
    mds_metric = MDS(n_components=2, metric=True, normalized_stress='auto', random_state=seed)
    X_mds = mds_metric.fit_transform(X_std)
    plot_embedding(X_mds, y, "Metric MDS", f"{outdir}/metric_mds_2d.png")
    mds_ari, mds_nmi, mds_sil = kmeans_scores(X_mds, y, seed)

    # 5) t-SNE
    tsne = TSNE(n_components=2, perplexity=tsne_perplexity, learning_rate='auto', init='pca', random_state=seed)
    X_tsne = tsne.fit_transform(X_std)
    plot_embedding(X_tsne, y, f"t-SNE (perplexity={tsne_perplexity})", f"{outdir}/tsne_2d.png")
    tsne_ari, tsne_nmi, tsne_sil = kmeans_scores(X_tsne, y, seed)

    # 6) UMAP
    umap_model = umap.UMAP(n_neighbors=umap_n_neighbors, min_dist=umap_min_dist, n_components=2, random_state=seed)
    X_umap = umap_model.fit_transform(X_std)
    plot_embedding(X_umap, y, f"UMAP (n_neighbors={umap_n_neighbors}, min_dist={umap_min_dist})", f"{outdir}/umap_2d.png")
    umap_ari, umap_nmi, umap_sil = kmeans_scores(X_umap, y, seed)

    #  AutoencoderPyTorch
    X_minmax = MinMaxScaler().fit_transform(X)
    X_ae = train_autoencoder_pytorch(
        X_minmax,
        seed=seed,
        hidden=ae_hidden,
        bottleneck=2,
        epochs=ae_epochs,
        batch_size=ae_batch,
        lr=ae_lr,
    )
    plot_embedding(X_ae, y, f"Autoencoder (hidden={ae_hidden}, epochs={ae_epochs})", f"{outdir}/autoencoder_2d.png")
    ae_ari, ae_nmi, ae_sil = kmeans_scores(X_ae, y, seed)

    # Save metrics
```

```
249     import csv
250     metrics_path = f"{outdir}/part1b_metrics.csv"
251     with open(metrics_path, "w", newline="") as f:
252         writer = csv.writer(f)
253         writer.writerow(["Method", "Params", "ARI", "NMI", "Silhouette", "
                                                Notes"])
254         writer.writerow(["Kernel PCA", f"rbf, gamma={gamma:.3g}", f"{kpca_ari
                                                :.4f}", f"{kpca_nmi:.4f}", f"{
                                                kpca_sil:.4f}", "2D embedding"]
                                                )
255         writer.writerow(["Spectral", f"n_neighbors={spectral_n_neighbors}", f"
                                                {se_ari:.4f}", f"{se_nmi:.4f}",
                                                 f"{se_sil:.4f}", "2D embedding
                                                "])
256         writer.writerow(["Classical MDS", "-", f"{cmds_ari:.4f}", f"{cmds_nmi
                                                :.4f}", f"{cmds_sil:.4f}", "
                                                closed-form (eigendecomposition
                                                )"])
257         writer.writerow(["Metric MDS", "sklearn MDS(metric=True)", f"{mds_ari
                                                :.4f}", f"{mds_nmi:.4f}", f"{
                                                mds_sil:.4f}", "stress
                                                minimization"])
258         writer.writerow(["t-SNE", f"perplexity={tsne_perplexity}", f"{tsne_ari
                                                :.4f}", f"{tsne_nmi:.4f}", f"{
                                                tsne_sil:.4f}", "2D embedding"]
                                                )
259         writer.writerow(["UMAP", f"n_neighbors={umap_n_neighbors}, min_dist={
                                                umap_min_dist}", f"{umap_ari:.
                                                4f}", f"{umap_nmi:.4f}", f"{
                                                umap_sil:.4f}", "2D embedding"]
                                                )
260         writer.writerow(["Autoencoder",f"hidden={ae_hidden}, epochs={ae_epochs
                                                }, batch={ae_batch}, lr={ae_lr:
                                                g}",f"{ae_ari:.4f}", f"{ae_nmi
                                                :.4f}", f"{ae_sil:.4f}","2D
                                                bottleneck (PyTorch)"])
261
262     # Print a neat summary
263     print("\n=== Part 1(b) - Manifold Methods on Digits ===")
264     print(f"[Kernel PCA]    ARI={kpca_ari:.4f}  NMI={kpca_nmi:.4f}  Silhouette
                                        ={kpca_sil:.4f}  (gamma={gamma:.3g}
                                        )")
265     print(f"[Spectral]      ARI={se_ari:.4f}  NMI={se_nmi:.4f}  Silhouette={
                                        se_sil:.4f}  (n_neighbors={
                                        spectral_n_neighbors})")
266     print(f"[Classical MDS] ARI={cmds_ari:.4f}  NMI={cmds_nmi:.4f}  Silhouette
                                        ={cmds_sil:.4f}")
267     print(f"[Metric MDS]    ARI={mds_ari:.4f}  NMI={mds_nmi:.4f}  Silhouette={
                                        mds_sil:.4f}")
268     print(f"[t-SNE]         ARI={tsne_ari:.4f}  NMI={tsne_nmi:.4f}  Silhouette
                                        ={tsne_sil:.4f}  (perplexity={
                                        tsne_perplexity})")
269     print(f"[UMAP]          ARI={umap_ari:.4f}  NMI={umap_nmi:.4f}  Silhouette
                                        ={umap_sil:.4f}  (n_neighbors={
```

```
                                                umap_n_neighbors}, min_dist={
                                                umap_min_dist})")
270     print(f"\nSaved figures and metrics to: {outdir}/")
271     print("Figures:")
272     print(" - kpca_2d.png, spectral_2d.png, classical_mds_2d.png,
                                                metric_mds_2d.png, tsne_2d.png,
                                                umap_2d.png")
273     print("Table:")
274     print(f" - {Path(metrics_path).name}")
275     print(f"[Autoencoder]   ARI={ae_ari:.4f}  NMI={ae_nmi:.4f}  Silhouette={
                                                ae_sil:.4f}  (hidden={ae_hidden},
                                                epochs={ae_epochs})")
276
277 # ----------------------------
278 # Entry
279 # ----------------------------
280 if __name__ == "__main__":
281     parser = argparse.ArgumentParser()
282     parser.add_argument("--seed", type=int, default=0, help="random seed")
283     parser.add_argument("--outdir", type=str, default="Homework 1/Latex/
                                                Results/Problem_1_b", help="output
                                                directory")
284     parser.add_argument("--kpca_gamma", type=float, default=None, help="RBF
                                                gamma for Kernel PCA (auto if None)
                                                ")
285     parser.add_argument("--spectral_n_neighbors", type=int, default=10, help="
                                                n_neighbors for Spectral Embedding"
                                                )
286     parser.add_argument("--tsne_perplexity", type=float, default=30, help="
                                                perplexity for t-SNE")
287     parser.add_argument("--umap_n_neighbors", type=int, default=15, help="
                                                n_neighbors for UMAP")
288     parser.add_argument("--umap_min_dist", type=float, default=0.1, help="
                                                min_dist for UMAP")
289     parser.add_argument("--ae_hidden", type=int, default=32)
290     parser.add_argument("--ae_epochs", type=int, default=50)
291     parser.add_argument("--ae_batch", type=int, default=128)
292     parser.add_argument("--ae_lr", type=float, default=1e-3)
293     args = parser.parse_args()
294
295     run(seed=args.seed,
296         outdir=args.outdir,
297         kpca_gamma=args.kpca_gamma,
298         spectral_n_neighbors=args.spectral_n_neighbors,
299         tsne_perplexity=args.tsne_perplexity,
300         umap_n_neighbors=args.umap_n_neighbors,
301         umap_min_dist=args.umap_min_dist,
302         ae_hidden=args.ae_hidden,
303         ae_epochs=args.ae_epochs,
304         ae_batch=args.ae_batch,
305         ae_lr=args.ae_lr)
```

A.3 Problem 2.

```
'''
Author: Chuyang Su cs4570@columbia.edu
Date: 2025-10-09 13:41:38
LastEditors: Please set LastEditors
LastEditTime: 2025-10-09 21:33:31
FilePath: /Unsupervised-Learning-Homework/Homework 1/Code/Problem_2.py
Description:
    This data set consists of gene expression measurements for n = 445 breast
                                            cancer tumors and p = 353 genes
                                            taken from The Cancer Genome Atlas(
                                            TCGA).
    This subset of genes was selected based on whether they contain known
                                            somatic mutations in cancer.
    Additionally, this data contains clinical data on the
        (i) Subtype (denotes 5 PAM50 subtypes including Basal-like, Luminal A,
                                             Luminal B, HER2-enriched, and
                                            Normal-like),
        (ii) ER-Status(estrogen-receptor status),
        (iii) PR-Status (progesterone-receptor status),
        (iv) HER2-Status (human epidermal growth factor receptor 2 status),
        (v) Node (number of lymph nodes involved), and (vi)Metastasis (
                                            indicator for whether the
                                            cancer has metastasized).
'''
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from pathlib import Path
import os
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA, NMF
from sklearn.manifold import TSNE, SpectralEmbedding
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score,
                                        silhouette_score
import csv
import umap

# === Embedding and evaluation pipeline (UMAP included) ===
def run_embedding(X, method='pca', n_components=2, seed=0, **kwargs):
    """
    Apply a dimension reduction method and return 2D embedding.
    Assumes UMAP is available (imported at top).
    """
    method = method.lower()
    if method == 'pca':
        model = PCA(n_components=n_components, random_state=seed)
        name = "PCA"
    elif method == 'nmf':
        model = NMF(n_components=n_components, init='nndsvda', random_state=
                                            seed, max_iter=1000)
        name = "NMF"
```

```
    elif method == 'tsne':
        model = TSNE(n_components=n_components, random_state=seed, **kwargs)
        name = "tSNE"
    elif method == 'umap':
        model = umap.UMAP(n_components=n_components, random_state=seed, **
                                                    kwargs)
        name = "UMAP"
    elif method == 'spectral':
        model = SpectralEmbedding(n_components=n_components, random_state=seed
                                                    )
        name = "Spectral"
    else:
        raise ValueError(f"Unknown method: {method}")
    X_embedded = model.fit_transform(X)
    return X_embedded, name


def _encode_categories(series):
    """Encode categorical labels to integer indices and return (indices,
                                                mapping)."""
    cats = series.astype(str).fillna("NA").values
    uniq = sorted(np.unique(cats))
    mapping = {c: i for i, c in enumerate(uniq)}
    idx = np.array([mapping[c] for c in cats], dtype=int)
    return idx, mapping

def plot_embedding(X2d, meta, hue_col, title, outpath):
    """Plot and save a 2D embedding colored by a clinical variable."""
    outpath = Path(outpath)
    outpath.parent.mkdir(parents=True, exist_ok=True)

    labels = meta[hue_col].astype(str).fillna("NA")
    idx, mapping = _encode_categories(labels)
    cmap = plt.cm.get_cmap('tab10', len(mapping))

    plt.figure(figsize=(6.4, 5.2), dpi=120)
    sc = plt.scatter(X2d[:, 0], X2d[:, 1], c=idx, s=14, cmap=cmap, alpha=0.85,
                                                edgecolors='none')
    plt.title(f"{title} embedding colored by {hue_col}")
    plt.xlabel("Dim 1"); plt.ylabel("Dim 2")

    # manual legend
    handles = [plt.Line2D([0], [0], marker='o', linestyle='', markersize=6,
                          markerfacecolor=cmap(i), markeredgecolor='none')
               for i in range(len(mapping))]
    labels_sorted = list(mapping.keys())
    plt.legend(handles, labels_sorted, bbox_to_anchor=(1.02, 1), loc='upper
                                                left', fontsize=8, frameon=False)
    plt.tight_layout()
    plt.savefig(outpath, bbox_inches='tight')
    plt.close()


def evaluate_embedding(X2d, subtype_series, seed=0):
```

```
    """KMeans(k=5) on the embedding; compute ARI/NMI/Silhouette using Subtype as reference."""
    mask = subtype_series.notna()
    X_eval = X2d[mask.values]
    y_ref = subtype_series[mask].astype(str).values

    km = KMeans(n_clusters=5, random_state=seed, n_init=10)
    pred = km.fit_predict(X_eval)

    ari = adjusted_rand_score(y_ref, pred)
    nmi = normalized_mutual_info_score(y_ref, pred)
    sil = silhouette_score(X_eval, pred)
    return ari, nmi, sil


def run_problem2_pipeline(
    data_path="Homework 1/Data/BRCA_data.csv",
    outdir="Homework 1/Latex/Results/Problem_2",
    seed=0
):
    outdir = Path(outdir)
    outdir.mkdir(parents=True, exist_ok=True)

    # 1) Preprocess (writes TXT report under outdir)
    X, y_clinical, gene_cols = load_and_preprocess_brca(
        data_path=data_path,
        report_path=str(outdir / "preprocessing_report.txt")
    )

    # 2) Methods to run (UMAP included by assumption)
    methods = [
        ("pca",      dict()),
        ("nmf",      dict()),
        ("spectral", dict()),
        ("tsne",     dict(perplexity=30, learning_rate='auto', init='pca')),
        ("umap",     dict(n_neighbors=15, min_dist=0.1)),
    ]

    # 3) Clinical hues to color by (only those that exist)
    hues = [c for c in ["subtype", "er_status", "pr_status", "her2_status"] if c in y_clinical.columns]

    # 4) Run, plot, evaluate
    metrics_rows = []
    best_by_ari = (None, -1.0)

    for m, kwargs in methods:
        if m == "nmf":
            # NMF
            X_nonneg = np.maximum(X, 0)  #
            X2d, name = run_embedding(X_nonneg, method=m, seed=seed, **kwargs)
        else:
            X2d, name = run_embedding(X, method=m, seed=seed, **kwargs)

```

```
        # Evaluation (Subtype as reference, if present)
        if "subtype" in y_clinical.columns:
            ari, nmi, sil = evaluate_embedding(X2d, y_clinical["subtype"],
                                               seed=seed)
        else:
            ari = nmi = sil = np.nan

        metrics_rows.append([name, f"{ari:.4f}", f"{nmi:.4f}", f"{sil:.4f}"])
        if not np.isnan(ari) and ari > best_by_ari[1]:
            best_by_ari = (name, ari)

        # Save embedding as CSV (optional, useful for debugging/report)
        emb_path = outdir / f"{name}_embedding.csv"
        np.savetxt(emb_path, X2d, delimiter=",")

        # Plots colored by clinical variables
        for hue in hues:
            fig_path = outdir / f"{name}_{hue}.png"
            plot_embedding(X2d, y_clinical, hue_col=hue, title=name, outpath=
                                               fig_path)

    # 5) Save metrics summary
    metrics_path = outdir / "metrics_summary.csv"
    with open(metrics_path, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Method", "ARI (Subtype)", "NMI (Subtype)", "
                                           Silhouette (KMeans=5)"])
        writer.writerows(metrics_rows)

    # Console summary
    print("\n=== Problem 2: embedding metrics (Subtype as reference) ===")
    for r in metrics_rows:
        print("{:<10s}  ARI={}  NMI={}  Sil={}".format(*r))
    if best_by_ari[0] is not None:
        print(f"\nBest by ARI: {best_by_ari[0]} (ARI={best_by_ari[1]:.4f})")

    print(f"\nSaved figures & tables to: {outdir}")

def load_and_preprocess_brca(data_path: str, var_threshold: float = 1e-4,
                                    report_path: str = None):
    # 1) Load with sample IDs in index
    df = pd.read_csv(data_path, index_col=0)
    n_samples_raw, n_cols_raw = df.shape
    print(f"Raw data shape: {n_samples_raw} samples × {n_cols_raw} columns")

    # 2) Normalize column names
    df.columns = (
        df.columns.astype(str)
        .str.strip()
        .str.replace("-", "_", regex=False)
        .str.replace(" ", "_", regex=False)
        .str.lower()
    )

```

```
    # 3) Detect clinical columns robustly
    possible_clinical = ["subtype", "er_status", "pr_status", "her2_status", "
                                            node", "metastasis"]
    clinical_cols = [c for c in possible_clinical if c in df.columns]
    if not clinical_cols:
        raise ValueError("No clinical columns detected after normalization. "
                         "Please check the CSV headers.")
    print(f"Detected clinical columns: {clinical_cols}")

    # 4) Split gene vs clinical
    gene_cols = [c for c in df.columns if c not in clinical_cols]
    # Force numeric on genes; coerce errors to NaN
    df[gene_cols] = df[gene_cols].apply(pd.to_numeric, errors="coerce")

    X_df = df[gene_cols].copy()
    y_clinical = df[clinical_cols].copy()

    # 5) Missing value stats and imputation
    n_missing_total = int(X_df.isna().sum().sum())
    n_rows_with_na = int(X_df.isna().any(axis=1).sum())
    n_genes_with_na = int(X_df.isna().any(axis=0).sum())
    print(f"Missing values in gene matrix: {n_missing_total} "
          f"(rows with any NA: {n_rows_with_na}, genes with any NA: {
                                                n_genes_with_na})")

    # Impute clinical: categorical by mode, numeric by mean
    for col in y_clinical.columns:
        if y_clinical[col].dtype == "O":
            mode_vals = y_clinical[col].mode(dropna=True)
            fill_val = mode_vals.iloc[0] if not mode_vals.empty else "Unknown"
            y_clinical[col] = y_clinical[col].fillna(fill_val)
        else:
            y_clinical[col] = y_clinical[col].fillna(y_clinical[col].mean())

    # Impute genes by column median (robust)
    X_df = X_df.fillna(X_df.median())

    # 6) Standardize (Z-score)
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_df)

    # 7) Build and optionally save summary report
    summary_lines = [
        "=== BRCA Data Preprocessing Summary ===",
        f"Original shape: {n_samples_raw} samples × {n_cols_raw} columns",
        f"Detected clinical columns: {clinical_cols}",
        f"Initial gene columns: {len(gene_cols)}",
        f"Total missing values filled (genes): {n_missing_total}",
        f"Rows with any NA (genes): {n_rows_with_na}",
        f"Genes with any NA: {n_genes_with_na}",
        f"Final standardized data shape: {X_scaled.shape}",
    ]

    # --- Clinical summary by type ---
```

```
    summary_text = ["Clinical variable summary:"]
    for col in y_clinical.columns:
        if y_clinical[col].dtype == "O":
            counts = y_clinical[col].value_counts(dropna=False)
            summary_text.append(f"\n{col} (categorical):")
            summary_text.append(str(counts))
        else:
            desc = y_clinical[col].describe()
            summary_text.append(f"\n{col} (numeric):")
            summary_text.append(str(desc))

    report_text = "\n".join(summary_lines + ["\n".join(summary_text)])
    print("\n" + report_text)

    if report_path is not None:
        report_dir = Path(report_path).parent
        report_dir.mkdir(parents=True, exist_ok=True)
        with open(report_path, "w", encoding="utf-8") as f:
            f.write(report_text)
        print(f"\nPreprocessing summary saved to {report_path}")

    return X_scaled, y_clinical, gene_cols

def run_embedding(X, method='pca', n_components=2, seed=0, **kwargs):
    method = method.lower()
    if method == 'pca':
        model = PCA(n_components=n_components, random_state=seed)
        name = 'PCA'
    elif method == 'nmf':
        model = NMF(n_components=n_components, init='nndsvda', random_state=
                                                seed, max_iter=1000)
        name = 'NMF'
    elif method == 'tsne':
        model = TSNE(n_components=n_components, random_state=seed, **kwargs)
        name = 'tSNE'
    elif method == 'umap':
        model = umap.UMAP(n_components=n_components, random_state=seed, **
                                                kwargs)
        name = 'UMAP'
    elif method == 'spectral':
        model = SpectralEmbedding(n_components=n_components, random_state=seed
                                                )
        name = 'Spectral'
    else:
        raise ValueError(f"Unknown method: {method}")

    X_emb = model.fit_transform(X)
    return X_emb, name


def _encode_categories(series):
    cats = series.astype(str).fillna("NA").values
    uniq = sorted(np.unique(cats))
    mapping = {c: i for i, c in enumerate(uniq)}
```

```
    idx = np.array([mapping[c] for c in cats], dtype=int)
    return idx, mapping


def plot_embedding(X2d, meta, hue_col, title, outpath):
    outpath = Path(outpath)
    outpath.parent.mkdir(parents=True, exist_ok=True)

    labels = meta[hue_col].astype(str).fillna("NA")
    idx, mapping = _encode_categories(labels)
    cmap = plt.cm.get_cmap('tab10', len(mapping))

    plt.figure(figsize=(6.2, 5.2), dpi=120)
    sc = plt.scatter(X2d[:,0], X2d[:,1], c=idx, s=14, cmap=cmap, alpha=0.85,
                                          edgecolors='none')
    plt.title(f"{title} embedding colored by {hue_col}")
    plt.xlabel("Dim 1"); plt.ylabel("Dim 2")
    handles = [plt.Line2D([0],[0], marker='o', linestyle='',
                          markersize=6, markerfacecolor=cmap(i),
                                                        markeredgecolor
                                                        ='none')
               for i in range(len(mapping))]
    labels_sorted = list(mapping.keys())
    plt.legend(handles, labels_sorted, bbox_to_anchor=(1.02, 1), loc='upper
                                          left', fontsize=8, frameon=False)
    plt.tight_layout()
    plt.savefig(outpath, bbox_inches='tight')
    plt.close()


def evaluate_embedding(X2d, subtype_series, seed=0):
    mask = subtype_series.notna()
    X_eval = X2d[mask.values]
    y_ref = subtype_series[mask].astype(str).values

    km = KMeans(n_clusters=5, random_state=seed, n_init=10)
    pred = km.fit_predict(X_eval)

    ari = adjusted_rand_score(y_ref, pred)
    nmi = normalized_mutual_info_score(y_ref, pred)
    sil = silhouette_score(X_eval, pred)

    return ari, nmi, sil



if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--data_path", type=str, default="Homework 1/Data/
                                          BRCA_data.csv")
    parser.add_argument("--outdir", type=str, default="Homework 1/Latex/
                                          Results/Problem_2")
    parser.add_argument("--seed", type=int, default=0)
```

```
    args = parser.parse_args()

    run_problem2_pipeline(
        data_path=args.data_path,
        outdir=args.outdir,
        seed=args.seed
    )
```