pbmc3k_dataset_exploration

November 23, 2025

0.1 Step 1: Make sure we only look at T-Cells

[ ]:
```
import scanpy as sc
import numpy as np

adata = sc.datasets.pbmc3k()
print(adata)  # ~2700 cells × 32738 genes

# Remove genes that are expressed in less than 3 cells
sc.pp.filter_genes(adata, min_cells=3)
# Make sure that sums of counts per cell are equal
sc.pp.normalize_total(adata, target_sum=1e4)
# Some genes have very high counts, so log-transform the data
sc.pp.log1p(adata)

# keep all genes
X_full = adata.X.toarray() if hasattr(adata.X, "toarray") else adata.X
print(X_full.shape)  # should be ~2700 cells × 32738 genes

# inspect one cell's expression vector
one_cell = X_full[0]
print(one_cell[:100])  # first 100 gene expression values
```

```
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/_utils/__init__.py:33: FutureWarning:
`__version__` is deprecated, use `importlib.metadata.version('anndata')`
instead.
  from anndata import __version__ as anndata_version
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/__init__.py:24: FutureWarning: `__version__` is
deprecated, use `importlib.metadata.version('anndata')` instead.
  if Version(anndata.__version__) >= Version("0.11.0rc2"):
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/readwrite.py:16: FutureWarning: `__version__`
is deprecated, use `importlib.metadata.version('anndata')` instead.
  if Version(anndata.__version__) >= Version("0.11.0rc2"):
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/datasets/_utils.py:35: FutureWarning:
`__version__` is deprecated, use `importlib.metadata.version('anndata')`
```

```
instead.
  if Version(ad.__version__).release >= (0, 8):
```

```
AnnData object with n_obs × n_vars = 2700 × 32738
    var: 'gene_ids'
(2700, 13714)
[0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        1.6358733 0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        1.6358733 0.        0.        0.
 0.        0.        0.        2.2265546 0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.        0.        0.        0.        0.        0.
 0.        0.       ]
```

[ ]:
```python
import scanpy as sc

# Compute PCA (dimensionality reduction)
sc.pp.pca(adata, n_comps=50)

# Compute neighborhood graph
sc.pp.neighbors(adata, n_neighbors=15, n_pcs=30)

# Cluster with the Leiden algorithm
sc.tl.leiden(adata, resolution=0.5)

# Optional: visualize clusters
sc.tl.umap(adata)
sc.pl.umap(adata, color='leiden')

# Common PBMC marker genes
markers = ["CD3D", "CD3E", "MS4A1", "LYZ", "NKG7", "GNLY", "PPBP"]

# Visualize their expression on UMAP
sc.pl.umap(adata, color=markers)

markers = ["CD3D", "CD3E", "MS4A1", "LYZ", "NKG7", "GNLY", "PPBP"]
sc.pl.matrixplot(adata, markers, groupby="leiden")
```

```
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/preprocessing/_pca/__init__.py:245:
FutureWarning: `__version__` is deprecated, use
```

```
`importlib.metadata.version('anndata')` instead.
  Version(ad.__version__) < Version("0.9")
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/neighbors/__init__.py:430: FutureWarning: Use
obsm (e.g. `k in adata.obsm` or `adata.obsm.keys() | {'u'}`) instead of
AnnData.obsm_keys, AnnData.obsm_keys is deprecated and will be removed in the
future.
  if "X_diffmap" in adata.obsm_keys():
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found.
Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
/var/folders/rr/cq_rkbv541d24_j5xzk1s3sc0000gn/T/ipykernel_16362/595385248.py:10
: FutureWarning: In the future, the default backend for leiden will be igraph
instead of leidenalg.

 To achieve the future defaults please pass: flavor="igraph" and n_iterations=2.
directed must also be False to work with igraph's implementation.
  sc.tl.leiden(adata, resolution=0.5)
```

[ ]:
```
adata_t = adata[adata.obs["leiden"] == "0"].copy()  # replace "0" with your␣
↪T-cell cluster ID
len(adata_t)  # number of T cells
sc.pp.pca(adata_t, n_comps=50)
sc.pp.neighbors(adata_t, n_neighbors=15, n_pcs=30)
sc.tl.umap(adata_t)
sc.tl.diffmap(adata_t)
sc.pl.umap(adata_t, color='leiden')
sc.pl.diffmap(adata_t, components=['1,2'])
```

[ ]:
```
662
```

0.2 Step 2: Now that we have selected just the T-Cells. Lets extract the manifold these T-Cells live on

[ ]:
```
import numpy as np, scipy.sparse as sp, torch

adata_t = adata[adata.obs["leiden"] == "0"].copy()

# 1) kNN graph on your T cells (already have PCA)
sc.pp.neighbors(adata_t, n_neighbors=15, n_pcs=30, metric="cosine")

# 2) affinity W, Markov P, symmetric Laplacian L
W = adata_t.obsp["connectivities"].tocsr()
d = np.asarray(W.sum(axis=1)).ravel()
P = sp.diags(1.0/(d+1e-12)) @ W
S = 0.5*(P + P.T)
```

```
L = sp.eye(W.shape[0], format="csr") - S

# 1) X: SciPy sparse -> dense NumPy -> torch
X_np = adata_t.X.toarray() if sp.issparse(adata_t.X) else np.asarray(adata_t.X)
X_torch = torch.from_numpy(X_np.astype(np.float32))  # (n_cells, n_genes)

# 2) L: SciPy sparse -> COO -> torch sparse
L_coo = L.tocoo()
idx_np = np.vstack([L_coo.row, L_coo.col]).astype(np.int64)   # shape (2, nnz)
val_np = L_coo.data.astype(np.float32)                        # shape (nnz,)
indices = torch.from_numpy(idx_np)
values  = torch.from_numpy(val_np)
L_torch = torch.sparse_coo_tensor(indices, values, size=L_coo.shape).coalesce()

print("X_torch:", X_torch.shape, "  L_torch:", L_torch.shape, "nnz:", L_torch._nnz())
```

```
X_torch: torch.Size([662, 13714])   L_torch: torch.Size([662, 662]) nnz: 14960
```

```
/Users/nicholaskhorasani/Documents/CodingProjects/diff_manifold_testing/venv/lib
/python3.12/site-packages/scanpy/neighbors/__init__.py:430: FutureWarning: Use
obsm (e.g. `k in adata.obsm` or `adata.obsm.keys() | {'u'}`) instead of
AnnData.obsm_keys, AnnData.obsm_keys is deprecated and will be removed in the
future.
  if "X_diffmap" in adata.obsm_keys():
```

[ ]:

1 Train Diffusion model (Guassian Noise and Manifold Noise)

[ ]:
```
# ===== modular denoiser training with pluggable noise =====
import math, torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader

# ---- noise processes ----
class NoiseProcess:
    """Interface: given clean X (n_cells x d), return noised Xt and (optional) conditioning info."""
    def __init__(self): pass
    def sample(self, X: torch.Tensor, **kwargs) -> torch.Tensor:
        raise NotImplementedError

class GaussianNoise(NoiseProcess):
    def __init__(self, sigma: float = 0.1):
        super().__init__()
        self.sigma = sigma
```

```
    def sample(self, X: torch.Tensor, **kwargs) -> torch.Tensor:
        return X + self.sigma * torch.randn_like(X)

class ManifoldHeat(NoiseProcess):
    """Applies k steps of Xt <- S @ Xt ; pass S (sparse torch), steps (int) or␣
 ↪t→steps mapping."""
    def __init__(self, S_sparse: torch.Tensor, steps: int = 5):
        super().__init__()
        assert S_sparse.is_sparse
        self.S = S_sparse.coalesce()
        self.steps = steps
    def sample(self, X: torch.Tensor, steps: int = None, **kwargs) -> torch.
 ↪Tensor:
        k = self.steps if steps is None else steps
        Xt = X
        for _ in range(k):
            Xt = torch.sparse.mm(self.S, Xt)
        return Xt

# ---- tiny model factory ----
def make_mlp(d: int, hidden: int = 512) -> nn.Module:
    return nn.Sequential(nn.Linear(d, hidden), nn.GELU(), nn.Linear(hidden, d))

# ---- trainer ----
def train_denoiser(
    X: torch.Tensor,                                    # (n_cells, n_genes)␣
 ↪float32
    topk_idx: torch.Tensor,                             # 1D indices of selected␣
 ↪genes/features
    noise: NoiseProcess,                                # e.g., GaussianNoise(...)␣
 ↪or ManifoldHeat(S, ...)
    model: nn.Module = None,
    epochs: int = 5,
    batch_size: int = 64,
    lr: float = 1e-3,
    weight_decay: float = 1e-4,
    val_split: float = 0.1,
    noise_kwargs: dict = None,                          # extra args to noise.
 ↪sample (e.g., steps, sigma)
    device: str = "cpu",
):
    noise_kwargs = noise_kwargs or {}
    X = X.to(device)
    # select features
    X0 = X[:, topk_idx].contiguous()
```

```
    # make a noised copy using the chosen process
    Xt = noise.sample(X, **noise_kwargs)[:, topk_idx].contiguous()

    # split
    n = X0.shape[0]
    perm = torch.randperm(n, device=device)
    split = int((1 - val_split) * n)
    idx_tr, idx_va = perm[:split], perm[split:]

    train_dl = DataLoader(TensorDataset(Xt[idx_tr], X0[idx_tr]),␣
↪batch_size=batch_size, shuffle=True)
    val_dl   = DataLoader(TensorDataset(Xt[idx_va], X0[idx_va]),␣
↪batch_size=2*batch_size, shuffle=False)

    # model
    d = X0.shape[1]
    model = model or make_mlp(d)
    model = model.to(device)
    opt = torch.optim.AdamW(model.parameters(), lr=lr,␣
↪weight_decay=weight_decay)
    loss_fn = nn.MSELoss()

    for epoch in range(epochs):
        # (optional) resample noise each epoch for robustness:
        Xt_full = noise.sample(X, **noise_kwargs)[:, topk_idx].contiguous()
        Xt_tr_epoch, Xt_va_epoch = Xt_full[idx_tr], Xt_full[idx_va]

        # swap datasets' inputs
        train_dl = DataLoader(TensorDataset(Xt_tr_epoch, X0[idx_tr]),␣
↪batch_size=batch_size, shuffle=True)
        val_dl   = DataLoader(TensorDataset(Xt_va_epoch, X0[idx_va]),␣
↪batch_size=2*batch_size, shuffle=False)

        model.train()
        tr_loss = 0.0
        for x_in, x_out in train_dl:
            opt.zero_grad(set_to_none=True)
            pred = model(x_in)
            loss = loss_fn(pred, x_out)
            loss.backward()
            opt.step()
            tr_loss += loss.item() * x_in.size(0)
        tr_loss /= len(train_dl.dataset)

        model.eval()
        va_loss = 0.0
        with torch.no_grad():
```

```
            for x_in, x_out in val_dl:
                pred = model(x_in)
                va_loss += loss_fn(pred, x_out).item() * x_in.size(0)
        va_loss /= len(val_dl.dataset)
        print(f"[{noise.__class__.__name__}] epoch {epoch+1}: train␣
 ↪MSE={tr_loss:.4f}  val MSE={va_loss:.4f}")

    # final eval on full data
    model.eval()
    with torch.no_grad():
        Xt_eval = noise.sample(X, **noise_kwargs)[:, topk_idx]
        X_hat   = model(Xt_eval)
        mse_all = torch.mean((X_hat - X0)**2).item()

    return model, mse_all
```

[ ]:
```
# You already have: X_torch (n_cells x n_genes), S_torch (sparse), both on CPU
device = "cpu"  # or "cuda" if you move tensors to GPU

# pick features: top-2000 variable genes
with torch.no_grad():
    var = X_torch.var(dim=0, unbiased=False)
topk = torch.topk(var, k=min(2000, X_torch.shape[1])).indices

# 1) Train with MANIFOLD heat
manifold_noise = ManifoldHeat(S_sparse=S_torch, steps=2)
model_m, mse_m = train_denoiser(
    X=X_torch, topk_idx=topk, noise=manifold_noise,
    epochs=5, device=device, noise_kwargs={"steps": 2}
)
print("Manifold denoiser MSE:", mse_m)

# 2) Train with GAUSSIAN baseline
gauss_noise = GaussianNoise(sigma=0.1)
model_g, mse_g = train_denoiser(
    X=X_torch, topk_idx=topk, noise=gauss_noise,
    epochs=5, device=device, noise_kwargs={"sigma": 0.1}
)
print("Gaussian denoiser MSE:", mse_g)
```

```
[ManifoldHeat] epoch 1: train MSE=0.8450  val MSE=0.6057
[ManifoldHeat] epoch 2: train MSE=0.5798  val MSE=0.5671
[ManifoldHeat] epoch 3: train MSE=0.5594  val MSE=0.5562
[ManifoldHeat] epoch 4: train MSE=0.5521  val MSE=0.5501
[ManifoldHeat] epoch 5: train MSE=0.5440  val MSE=0.5483
Manifold denoiser MSE: 0.5439212322235107
[GaussianNoise] epoch 1: train MSE=0.8412  val MSE=0.5806
[GaussianNoise] epoch 2: train MSE=0.5503  val MSE=0.5320
```

```
[GaussianNoise] epoch 3: train MSE=0.5176  val MSE=0.5237
[GaussianNoise] epoch 4: train MSE=0.5024  val MSE=0.5201
[GaussianNoise] epoch 5: train MSE=0.4894  val MSE=0.5169
Gaussian denoiser MSE: 0.4818757176399231
```

[ ]:
```python
import torch

# Use the same topk as before
with torch.no_grad():
    var = X_torch.var(dim=0, unbiased=False)
topk = torch.topk(var, k=min(2000, X_torch.shape[1])).indices
X0 = X_torch[:, topk]

# Manifold corruption already defined:
X_man = ManifoldHeat(S_sparse=S_torch, steps=5).sample(X_torch)[:, topk]
delta_man = torch.norm(X_man - X0, dim=1).mean().item()

# Solve for sigma so Gaussian has same mean L2 deviation
def find_sigma(target, X, idx, tol=1e-4):
    lo, hi = 1e-6, 2.0
    for _ in range(30):
        mid = 0.5*(lo+hi)
        Xg = X[:, idx] + mid*torch.randn_like(X[:, idx])
        val = torch.norm(Xg - X[:, idx], dim=1).mean().item()
        if val < target: lo = mid
        else: hi = mid
    return 0.5*(lo+hi)

sigma_matched = find_sigma(delta_man, X_torch, topk)
print("Matched sigma:", sigma_matched)
```

```
Matched sigma: 0.7717827636696748
```

[ ]:
```python
import numpy as np

# Build a dense-by-vector multiply helper for L (keep L sparse)
def lap_energy(L_sp, X):
    # X: (n_cells, d)
    # Return per-cell energy using x^T L x = sum_j L_ij x_i·x_j (do via (L X)␣
 ↪then row-wise dot)
    LX = torch.from_numpy((L_sp @ X.numpy()).astype(np.float32)) if␣
 ↪isinstance(X, torch.Tensor) else L_sp @ X
    if isinstance(X, torch.Tensor):
        return (X * LX).sum(dim=1).cpu().numpy()
    else:
        return (X * LX).sum(axis=1)

# Get outputs from your trained models:
```

```
# model_m (manifold denoiser), model_g (gaussian denoiser with matched sigma)
with torch.no_grad():
    X_man_noised = ManifoldHeat(S_sparse=S_torch, steps=5).sample(X_torch)[:,␣
 ↪topk]
    Xg_noised   = X_torch[:, topk] + sigma_matched * torch.randn_like(X0)

    X_hat_m = model_m(X_man_noised)
    X_hat_g = model_g(Xg_noised)

# Energies: real vs denoised
E_real = lap_energy(L, X0)
E_m    = lap_energy(L, X_hat_m)
E_g    = lap_energy(L, X_hat_g)

print("Mean Laplacian energy:")
print("  Real:      ", float(np.mean(E_real)))
print("  Manifold̂ : ", float(np.mean(E_m)))
print("  Gaussiañ : ", float(np.mean(E_g)))
```

```
Mean Laplacian energy:
  Real:       1005.293701171875
  Manifold̂ :  -114.62344360351562
  Gaussiañ :  96.97181701660156
```

[ ]:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[79], line 5
      3 adata_t.obsm["X_denoised_manifold"] = model_m(X_man_noised).detach().
 ↪numpy()
      4 adata_t.obsm["X_denoised_gauss"] = model_g(Xg_noised).detach().numpy()
----> 5 sc.pl.umap(adata_t, color=None, basis=          )  # overlay next step

File ~/Documents/CodingProjects/diff_manifold_testing/venv/lib/python3.12/
 ↪site-packages/scanpy/plotting/_tools/scatterplots.py:686, in umap(adata,␣
 ↪**kwargs)
    627 @_wraps_plot_scatter
    628 @_doc_params(
    629     adata_color_etc=doc_adata_color_etc,
   (…)    633 )
    634 def umap(adata: AnnData, **kwargs) -> Figure | Axes | list[Axes] | None
    635     """Scatter plot in UMAP basis.
    636
    637     Parameters
   (…)    684
    685     """
--> 686     return embedding(adata,        , **kwargs)
```

```
TypeError: embedding() got multiple values for argument 'basis'
```

[ ]:

[ ]: