AttnKNN_Experiment

November 25, 2025

1 Attn-KNN — Experiment Notebook

This notebook collects everything needed to run, reproduce, and extend experiments for **Attention-Weighted Learnable k-NN (Attn-KNN)**.

It is designed for a local GPU setup (macOS M-series or Linux with CUDA). It includes:

- Dataset download & preprocessing for **tabular, image, and text** datasets
- Training pipeline (embedder + attention head + FAISS indexing)
- Evaluation metrics (accuracy, F1, ECE, NLL, AUROC for OOD)
- Visualization utilities (UMAP/TSNE, attention heatmaps, reliability diagrams)
- Experiment plan & checklists for full study (CIFAR-10/100, MNIST, UCI Adult, Wine, Iris, ImageNet subset, AG News, SST-2, miniImageNet)

> **Note:** The notebook contains runnable code cells and placeholders. Heavy training steps are marked and should be run on GPU machines. Replace placeholder paths and adjust hyperparameters as needed.

2 check if FAISS is installed in macos arm64 using homebrew

3 brew install faiss

3.1 0) Environment Setup

Install required packages (run once). On macOS M1/M2/M4, use MPS-enabled PyTorch wheel or the CPU wheel if needed.

```
# Create virtualenv (optional)
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
```

requirements.txt should include: `torch`, `torchvision`, `faiss-cpu` (or `faiss-gpu` on CUDA), `scikit-learn`, `umap-learn`, `matplotlib`, `seaborn`, `pandas`, `tqdm`.

[54]:
```
%%bash
set -e
# Use uv to create an isolated environment
if ! command -v uv >/dev/null 2>&1; then
```

```
  python3 -m pip install --user uv
fi
uv venv .venv --python 3.14
source .venv/bin/activate

# Install required deps (Mac ARM-optimized)
uv pip install \
  torch torchvision torchaudio \
  faiss-cpu \
  numpy pandas scikit-learn matplotlib seaborn \
  tqdm \
  jupyter ipykernel \
  pillow requests \
  plotly \
  scipy \
  jupyter-cjk-xelatex

# Register kernel
python -m ipykernel install --user --name=attnknn --display-name "AttnKNN (Mac␣
↪M4)"
```

```
Using CPython 3.14.0 interpreter at:
/opt/homebrew/opt/python@3.14/bin/python3.14
Creating virtual environment at: .venv
warning: A virtual environment already exists at

`.venv`. In the future, uv will require `--clear` to replace it
Activate with: source .venv/bin/activate
Resolved 127 packages in 2.52s
   Building jupyter-cjk-xelatex==0.2
      Built jupyter-cjk-xelatex==0.2
Prepared 1 package in 134ms
Installed 1 package in 1ms
 + jupyter-cjk-xelatex==0.2

Installed kernelspec attnknn in /Users/taher/Library/Jupyter/kernels/attnknn
```

[ ]:
```python
# CRITICAL FIX: Disable OpenMP BEFORE any imports to prevent segfault crashes!
# PyTorch, FAISS, and scikit-learn each load their own libomp.dylib on macOS
# which causes fatal crashes when they conflict. Force single-threaded mode.
import os

# MUST be set BEFORE importing torch, faiss, sklearn, numpy, etc.
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
os.environ['OMP_NUM_THREADS'] = '1'  # CRITICAL: Force single-threaded OpenMP
os.environ['MKL_NUM_THREADS'] = '1'
os.environ['OPENBLAS_NUM_THREADS'] = '1'
os.environ['VECLIB_MAXIMUM_THREADS'] = '1'
os.environ['NUMEXPR_NUM_THREADS'] = '1'

import platform, multiprocessing
import torch

is_macos_arm = (platform.system() == 'Darwin' and platform.machine() in␣
 ↪('arm64', 'aarch64'))

# PyTorch can use multiple threads safely (it uses its own threadpool, not␣
 ↪OpenMP)
torch_threads = min(16, multiprocessing.cpu_count())
try:
    torch.set_num_threads(torch_threads)
    torch.set_num_interop_threads(max(1, torch_threads // 2))
    if hasattr(torch, 'set_float32_matmul_precision'):
        torch.set_float32_matmul_precision('high')
except Exception:
    pass

print(f'PyTorch threads: {torch_threads}')
print(f'OpenMP threads: {os.environ.get("OMP_NUM_THREADS", "not set")}␣
 ↪(CRITICAL for macOS stability)')
if is_macos_arm:
    print('Running on macOS Apple Silicon (ARM64).')
    if torch.backends.mps.is_available():
        print('MPS backend available - GPU acceleration enabled!')

# Import FAISS after torch, with OMP disabled
try:
    import faiss
    print('FAISS loaded (single-threaded mode to prevent crashes).')
except ImportError as e:
    print('FAISS not available:', repr(e))
```

[ ]:
```
# Data root and run configuration (ordered)
from pathlib import Path
import os, json
print('[CONFIG] Initializing data root and run parameters...')
env_root = os.environ.get('ATTNKNN_DATA_ROOT') or os.environ.get('DATA_ROOT')
default_candidates = [
    Path('/Users/taher/Projects/attn_knn_repo/data'),
    Path.cwd().parent / 'data',
    Path.cwd() / 'data'
]
if env_root:
    DATA_ROOT = Path(env_root).expanduser().resolve()
else:
    DATA_ROOT = next((p.resolve() for p in default_candidates if p.exists()),␣
  ↪default_candidates[-1].resolve())
RESULTS_DIR = Path('results').resolve()
RESULTS_DIR.mkdir(parents=True, exist_ok=True)
# Compute sensible defaults for parallelism
_cpu_count = os.cpu_count() or 8
_default_workers = min(16, max(4, _cpu_count))
_default_faiss_threads = min(16, max(4, _cpu_count))
RUN_CONFIG = {
    'k': 10,
    'batch_size_image': 128,
    'batch_size_tabular': 256,
    'index_type': 'Flat',            # Flat, HNSW, FlatIP
    'use_faiss_gpu': False,
    'epochs_attention': 0,
    'seed': 42,
    'num_workers': _default_workers,
    'prefetch_factor': 4,
    'persistent_workers': True,
    'use_amp_mps': True,
    'faiss_threads': _default_faiss_threads,
    'image_size': 128,
    'vectorize_probs': True
}
print(f"[CONFIG] DATA_ROOT: {DATA_ROOT} (exists={DATA_ROOT.exists()})")
print(f"[CONFIG] RESULTS_DIR: {RESULTS_DIR}")
print(f"[CONFIG] k={RUN_CONFIG['k']} index={RUN_CONFIG['index_type']}␣
  ↪bs_img={RUN_CONFIG['batch_size_image']}␣
  ↪bs_tab={RUN_CONFIG['batch_size_tabular']}␣
  ↪workers={RUN_CONFIG['num_workers']}␣
  ↪faiss_threads={RUN_CONFIG['faiss_threads']}␣
  ↪img_sz={RUN_CONFIG['image_size']}")
```

3.2 Notebook outline

1. Quick imports & helper functions
2. Data loading per domain (image, tabular, text)
3. Model components (embedder, attention head)
4. Memory bank & FAISS integration
5. Training loop (offline / episodic modes)
6. Evaluation metrics & calibration
7. Visualizations (UMAP, attention maps, reliability diagrams)
8. Experiment execution checklist and templates
9. Saving results for paper and LaTeX tables

[ ]:
```
# 1) Quick imports & helper functions (self-contained, no src dependency)
import os, sys, json, math, random, time
```

```
from pathlib import Path

# CRITICAL: Disable OpenMP to prevent crashes on macOS (multiple libomp␣
↪conflict)
# This MUST be set before importing numpy, torch, sklearn, etc.
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
os.environ['OMP_NUM_THREADS'] = '1'
os.environ['MKL_NUM_THREADS'] = '1'
os.environ['OPENBLAS_NUM_THREADS'] = '1'

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, TensorDataset
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, f1_score, log_loss, roc_auc_score,␣
↪confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tqdm.auto import tqdm
import pandas as pd

# UMAP optional import with TSNE/PCA fallback
try:
    import umap
    HAS_UMAP = True
except Exception:
    umap = None
    HAS_UMAP = False
    from sklearn.manifold import TSNE
    from sklearn.decomposition import PCA


def reduce_to_2d(embeddings: np.ndarray, random_state: int = 42, max_tsne: int␣
↪= 5000) -> np.ndarray:
    n = embeddings.shape[0]
    if HAS_UMAP:
        reducer = umap.UMAP(n_components=2, random_state=random_state)
        return reducer.fit_transform(embeddings)
    # Fallback: TSNE for small sets, PCA for large
    if n <= max_tsne:
        return TSNE(n_components=2, random_state=random_state, init='pca').
↪fit_transform(embeddings)
    return PCA(n_components=2).fit_transform(embeddings)
```

```
def ece_score(probs_np: np.ndarray, labels_np: np.ndarray, n_bins: int = 10) ->␣
↪float:
    """Expected Calibration Error."""
    confidences = probs_np.max(axis=1)
    predictions = probs_np.argmax(axis=1)
    bins = np.linspace(0, 1, n_bins + 1)
    ece = 0.0
    for i in range(n_bins):
        mask = (confidences > bins[i]) & (confidences <= bins[i + 1])
        if mask.sum() > 0:
            acc_bin = (predictions[mask] == labels_np[mask]).mean()
            conf_bin = confidences[mask].mean()
            ece += (mask.sum() / len(labels_np)) * abs(acc_bin - conf_bin)
    return float(ece)


# Helper: set device
device = torch.device('cuda' if torch.cuda.is_available() else ('mps' if torch.
↪backends.mps.is_available() else 'cpu'))
print('Device:', device)
if not HAS_UMAP:
    print('UMAP not available; using TSNE/PCA fallback for projections')
```

[ ]:
```
# 2) Data loading recipes (image, tabular, text)

# -- Image datasets: CIFAR-10, CIFAR-100, MNIST, ImageNet-subset
from torchvision import transforms, datasets

def get_image_loaders(name='CIFAR10', batch_size=128, data_dir='./data'):
    if name == 'CIFAR10':
        transform = transforms.Compose([
            transforms.RandomCrop(32, padding=4),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize((0.4914,0.4822,0.4465),(0.247,0.243,0.261))
        ])
        train = datasets.CIFAR10(root=data_dir, train=True, download=True,␣
↪transform=transform)
        test = datasets.CIFAR10(root=data_dir, train=False, download=True,␣
↪transform=transforms.Compose([
            transforms.ToTensor(), transforms.Normalize((0.4914,0.4822,0.
↪4465),(0.247,0.243,0.261))]))
    elif name == 'CIFAR100':
        train = datasets.CIFAR100(root=data_dir, train=True, download=True,␣
↪transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5071,0.4867,0.4408),(0.2675,0.2565,0.2761))]))
```

```
        test = datasets.CIFAR100(root=data_dir, train=False, download=True,␣
↪transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5071,0.4867,0.4408),(0.2675,0.2565,0.2761))]))
    elif name == 'MNIST':
        train = datasets.MNIST(root=data_dir, train=True, download=True,␣
↪transform=transforms.Compose([transforms.Resize((32,32)), transforms.
↪ToTensor(), transforms.Normalize((0.1307,),(0.3081,))]))
        test = datasets.MNIST(root=data_dir, train=False, download=True,␣
↪transform=transforms.Compose([transforms.Resize((32,32)), transforms.
↪ToTensor(), transforms.Normalize((0.1307,),(0.3081,))]))
    else:
        raise ValueError('Dataset not implemented yet')
    train_loader = DataLoader(train, batch_size=batch_size, shuffle=True,␣
↪num_workers=4)
    test_loader = DataLoader(test, batch_size=batch_size, shuffle=False,␣
↪num_workers=4)
    return train_loader, test_loader

# -- Tabular datasets: UCI Adult, Wine, Iris
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def load_uci_tabular(url_or_path, target_col, test_size=0.2, random_state=42):
    df = pd.read_csv(url_or_path)
    df = df.dropna()
    X = df.drop(columns=[target_col])
    y = df[target_col]
    # simple preprocessing: numerical scaling; categorical one-hot if any
    X = pd.get_dummies(X)
    scaler = StandardScaler()
    Xs = scaler.fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(Xs, y,␣
↪test_size=test_size, random_state=random_state)
    return X_train, X_test, y_train.values, y_test.values

# -- Text datasets: optional (AG News, SST-2 via HuggingFace)
# The HuggingFace stack is optional; skip if not installed
try:
    from datasets import load_dataset
    from transformers import AutoTokenizer
    HAS_HF = True
except Exception:
    HAS_HF = False
```

```
def load_text_dataset(name='ag_news', split_ratio=0.2, max_samples=None,␣
↪tokenizer_name='bert-base-uncased'):
    if not HAS_HF:
        raise ImportError('HuggingFace datasets/transformers not installed.␣
↪Install to enable text experiments.')
    ds = load_dataset(name)
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    def tokenize(batch):
        return tokenizer(batch['text'], truncation=True, padding='max_length',␣
↪max_length=128)
    ds = ds.map(lambda x: tokenize(x), batched=True)
    if max_samples:
        ds = ds.select(range(max_samples))
    train_ds = ds['train']
    test_ds = ds['test']
    return train_ds, test_ds, tokenizer

print('Data loader utilities ready')
```

[ ]:
```
# 3) Model components: Embedder and Attention Head (INLINE - OPTIMIZED for␣
↪speed)
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import random
import os


def set_seed(seed: int = 42) -> None:
    """Set all random seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    try:
        torch.use_deterministic_algorithms(False)
    except Exception:
        pass


class SimpleEmbedder(nn.Module):
    """ResNet18 backbone for 224x224 images (use SmallImageEmbedder for 32x32).
↪"""

    def __init__(
        self,
        embed_dim: int = 128,
        base: str = 'resnet18',
        pretrained: bool = False
    ) -> None:
        super().__init__()
        weights = models.ResNet18_Weights.IMAGENET1K_V1 if pretrained else None
        m = models.resnet18(weights=weights)
        in_dim = m.fc.in_features
        m.fc = nn.Identity()
        self.backbone = m
        self.proj = nn.Linear(in_dim, embed_dim)
        self.embed_dim = embed_dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        z = self.backbone(x)
        z = self.proj(z)
        z = F.normalize(z, dim=1)
        return z


class SmallImageEmbedder(nn.Module):
    """FAST ResNet18 modified for 32x32 images (CIFAR/MNIST) - NO resize needed.
↪"""

    def __init__(self, embed_dim: int = 128, in_channels: int = 3, pretrained:␣
↪bool = False) -> None:
        super().__init__()
        weights = models.ResNet18_Weights.IMAGENET1K_V1 if pretrained else None
        m = models.resnet18(weights=weights)
        # Modify conv1 for 32x32: smaller kernel, no stride, no maxpool
        m.conv1 = nn.Conv2d(in_channels, 64, kernel_size=3, stride=1,␣
↪padding=1, bias=False)
        m.maxpool = nn.Identity()  # Remove maxpool - keeps spatial dims
        in_dim = m.fc.in_features
        m.fc = nn.Identity()
        self.backbone = m
        self.proj = nn.Linear(in_dim, embed_dim)
        self.embed_dim = embed_dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        z = self.backbone(x)
        z = self.proj(z)
        z = F.normalize(z, dim=1)
        return z
```

```python
class AttnKNNHead(nn.Module):
    """Attention over k neighbors - VECTORIZED (no Python loops)."""

    def __init__(
        self,
        embed_dim: int = 128,
        proj_dim: int = 64,
        temperature_init: float = 1.0,
        learnable_temp: bool = True
    ) -> None:
        super().__init__()
        self.query_proj = nn.Linear(embed_dim, proj_dim)
        self.key_proj = nn.Linear(embed_dim, proj_dim)
        self.learnable_temp = learnable_temp
        if learnable_temp:
            self.log_tau = nn.Parameter(torch.tensor(float(np.
 ↪log(temperature_init))))
        else:
            self.register_buffer('log_tau', torch.tensor(0.0))

    def forward(self, q: torch.Tensor, neigh: torch.Tensor) -> torch.Tensor:
        qp = F.normalize(self.query_proj(q), dim=1)
        kp = F.normalize(self.key_proj(neigh), dim=2)
        logits = torch.einsum('bp,bkp->bk', qp, kp)
        tau = torch.exp(self.log_tau) if self.learnable_temp else torch.
 ↪tensor(1.0, device=logits.device)
        weights = F.softmax(logits / tau.clamp(min=1e-3), dim=1)
        return weights


class MLPEmbedder(nn.Module):
    """Simple MLP embedder for tabular data."""

    def __init__(self, input_dim: int, embed_dim: int = 64) -> None:
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, embed_dim)
        )
        self.embed_dim = embed_dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        z = self.net(x)
        z = F.normalize(z, dim=1)
        return z
```

```python
# ============================================================================
# VECTORIZED probability aggregation (FAST - MPS compatible)
# ============================================================================

def aggregate_neighbor_probs_uniform_np(
    neigh_labels: np.ndarray,
    num_classes: int,
    k: int
) -> np.ndarray:
    """NumPy-based uniform kNN - guaranteed fast on all platforms.

    Args:
        neigh_labels: (B, k) numpy array of neighbor labels
        num_classes: number of classes
        k: number of neighbors
    Returns:
        (B, num_classes) probability numpy array
    """
    B = neigh_labels.shape[0]
    probs = np.zeros((B, num_classes), dtype=np.float32)
    # Vectorized using np.add.at
    batch_idx = np.arange(B)[:, None]  # (B, 1)
    np.add.at(probs, (batch_idx, neigh_labels), 1.0 / k)
    return probs


def aggregate_neighbor_probs_weighted_np(
    neigh_labels: np.ndarray,
    weights: np.ndarray,
    num_classes: int
) -> np.ndarray:
    """NumPy-based weighted kNN - guaranteed fast on all platforms.

    Args:
        neigh_labels: (B, k) numpy array of neighbor labels
        weights: (B, k) numpy array of attention weights
        num_classes: number of classes
    Returns:
        (B, num_classes) probability numpy array
    """
    B = neigh_labels.shape[0]
    probs = np.zeros((B, num_classes), dtype=np.float32)
    batch_idx = np.arange(B)[:, None]
    np.add.at(probs, (batch_idx, neigh_labels), weights)
    return probs
```

```python
# PyTorch versions (for GPU acceleration when available)
def aggregate_neighbor_probs_uniform(
    neigh_labels: torch.Tensor,
    num_classes: int,
    k: int
) -> torch.Tensor:
    """Vectorized uniform kNN using scatter_add (MPS compatible)."""
    B = neigh_labels.size(0)
    device = neigh_labels.device
    probs = torch.zeros(B, num_classes, device=device)
    weights = torch.full_like(neigh_labels, 1.0 / k, dtype=torch.float32)
    probs.scatter_add_(1, neigh_labels, weights)
    return probs


def aggregate_neighbor_probs_weighted(
    neigh_labels: torch.Tensor,
    weights: torch.Tensor,
    num_classes: int
) -> torch.Tensor:
    """Vectorized weighted kNN using scatter_add (MPS compatible)."""
    B = neigh_labels.shape[0]
    device = neigh_labels.device
    probs = torch.zeros(B, num_classes, device=device)
    probs.scatter_add_(1, neigh_labels, weights)
    return probs


# Backwards-compatible aliases
EmbedderResNet18 = SimpleEmbedder
AttnHead = AttnKNNHead

print('Model components ready (OPTIMIZED with vectorized aggregation)')
```

[ ]:
```python
# Helper loader function for tabular data (MLPEmbedder is defined in Model␣
↪components cell)


def get_tabular_loaders_from_arrays(
    X_train: np.ndarray,
    y_train: np.ndarray,
    X_test: np.ndarray,
    y_test: np.ndarray,
    batch_size: int = 128
) -> tuple[DataLoader, DataLoader]:
    """Create DataLoaders from numpy arrays for tabular data."""
    X_train_t = torch.tensor(X_train, dtype=torch.float32)
    y_train_t = torch.tensor(y_train, dtype=torch.long)
    X_test_t = torch.tensor(X_test, dtype=torch.float32)
    y_test_t = torch.tensor(y_test, dtype=torch.long)
    train_ds = TensorDataset(X_train_t, y_train_t)
    test_ds = TensorDataset(X_test_t, y_test_t)
    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=0)
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=0)
    return train_loader, test_loader


print('Tabular loader helper ready (MLPEmbedder in Model components cell)')
```

[ ]:

```
# 4) Memory bank & FAISS integration (INLINE - self-contained)
import faiss


def build_faiss_index(
    embeddings: np.ndarray,
    index_type: str = 'Flat',
    metric: str = 'L2',
    use_gpu: bool = False
) -> faiss.Index:
    """Build FAISS index from embeddings.

    Args:
        embeddings: (N, D) float32 array
        index_type: 'Flat' or 'HNSW'
        metric: 'L2' (Euclidean) or 'IP' (Inner Product / cosine for normalized)
        use_gpu: Whether to use GPU resources

    Returns:
        FAISS index with embeddings added
    """
    d = embeddings.shape[1]
    if index_type == 'HNSW':
        index = faiss.IndexHNSWFlat(d, 32)
    elif metric.upper() == 'IP':
        index = faiss.IndexFlatIP(d)
    else:
        index = faiss.IndexFlatL2(d)

    if use_gpu:
        try:
            res = faiss.StandardGpuResources()
            index = faiss.index_cpu_to_gpu(res, 0, index)
        except Exception as e:
            print(f'[FAISS] GPU not available, using CPU: {e}')

    index.add(embeddings.astype('float32'))
    return index


class MemoryBank:
    """Simple memory bank for kNN operations using FAISS."""

    def __init__(self, dim: int, index_type: str = 'Flat', metric: str = 'L2')␣
 ↪-> None:
        self.dim = dim
        self.index_type = index_type
        self.metric = metric
        self.index: faiss.Index = self._make_index()
        self.labels: np.ndarray = np.array([], dtype=np.int64)
        self._embeddings: np.ndarray | None = None

    def _make_index(self) -> faiss.Index:
        if self.index_type == 'HNSW':
            return faiss.IndexHNSWFlat(self.dim, 32)
        elif self.metric.upper() == 'IP':
            return faiss.IndexFlatIP(self.dim)
        return faiss.IndexFlatL2(self.dim)

    def build(self, embeddings: np.ndarray, labels: np.ndarray) -> None:
        embeddings = embeddings.astype('float32')
        self._embeddings = embeddings.copy()
        self.index = self._make_index()
        self.index.add(embeddings)
        self.labels = labels.astype(np.int64)

    def search(self, queries: np.ndarray, k: int) -> tuple[np.ndarray, np.
 ↪ndarray, np.ndarray]:
        queries = queries.astype('float32')
        dists, idxs = self.index.search(queries, k)
        labels = np.full_like(idxs, -1)
        valid = idxs >= 0
        labels[valid] = self.labels[idxs[valid]]
        return dists, idxs, labels

    def get_embeddings_by_index(self, idxs: np.ndarray) -> np.ndarray:
        """Get embeddings by indices. idxs: (B, k) -> returns (B, k, D)"""
        if self._embeddings is None:
            raise RuntimeError('Memory bank not built')
        return self._embeddings[idxs]


def build_memorybank_from_embeddings(
    embeddings: np.ndarray,
    labels: np.ndarray,
    index_type: str = 'Flat',
    metric: str = 'L2'
) -> MemoryBank:
    """Build MemoryBank from embeddings and labels."""
    mem = MemoryBank(dim=int(embeddings.shape[1]), index_type=index_type,␣
  ↪metric=metric)
    mem.build(embeddings, labels)
    return mem

print('FAISS utilities ready (inline MemoryBank)')
```

```
[ ]: # 7) Visualizations: UMAP, attention heatmap, reliability diagram
import matplotlib.pyplot as plt

def plot_umap(embeddings, labels, title='Projection (UMAP/TSNE/PCA)'):
    proj = reduce_to_2d(embeddings)
    plt.figure(figsize=(8,6))
    plt.scatter(proj[:,0], proj[:,1], c=labels, s=5, cmap='tab10')
    plt.title(title)
    plt.show()

def plot_attention_weights(neighbor_imgs, attn_weights, labels=None):
    # neighbor_imgs: list of images (PIL/array), attn_weights: (k,)
    k = len(neighbor_imgs)
    plt.figure(figsize=(k*2,2.5))
    for i in range(k):
        plt.subplot(1,k,i+1)
        plt.imshow(np.transpose(neighbor_imgs[i], (1,2,0)))
        plt.title(f' ={attn_weights[i]:.2f}')
        plt.axis('off')
    plt.show()

def plot_reliability(probs, labels, n_bins=10):
    confidences = probs.max(axis=1)
    predictions = probs.argmax(axis=1)
    bins = np.linspace(0,1,n_bins+1)
    bin_centers = 0.5*(bins[:-1]+bins[1:])
    accuracies = []
    confidences_mean = []
    for i in range(n_bins):
        mask = (confidences > bins[i]) & (confidences <= bins[i+1])
        if mask.sum() > 0:
            accuracies.append((predictions[mask]==labels[mask]).mean())
            confidences_mean.append(confidences[mask].mean())
        else:
            accuracies.append(np.nan); confidences_mean.append(np.nan)
    plt.figure()
    plt.plot(bin_centers, accuracies, marker='o', label='accuracy')
    plt.plot(bin_centers, confidences_mean, marker='x', label='confidence')
    plt.plot([0,1],[0,1], linestyle='--', color='gray')
    plt.xlabel('Confidence'); plt.ylabel('Accuracy'); plt.legend(); plt.
  ↪title('Reliability diagram')
    plt.show()

print('Visualization utilities ready')
```

4 8) Experiment plan & checklist

The full study should run the following experiments (for each dataset/domain):

- Datasets:
  - Images: MNIST, CIFAR-10, CIFAR-100, ImageNet subset (Imagenet100)
  - Tabular: UCI Adult, Wine Quality, Iris
  - Text: AG News, SST-2 (use BERT tokenizer + encoder as embedder)
  - Few-shot: miniImageNet / tieredImageNet for few-shot evaluation
  - Robustness: CIFAR-10-C, adversarial attacks (FGSM/PGD)
- Baselines:
  - Classic k-NN (Euclidean) on raw features
  - k-NN on PCA-reduced features
  - k-NN on pretrained embeddings (e.g., ResNet features)
  - Learned-embedding + k-NN (metric loss) without attention
  - Attn-KNN (ours)
- Metrics:
  - Accuracy, Macro F1
  - Calibration: ECE, NLL
  - OOD detection: AUROC using entropy or attention entropy
  - Retrieval: neighbor precision@k
  - Efficiency: embedding time, ANN search time, memory footprint
- Ablations:
  - Attention vs uniform weighting
  - Metric loss vs none
  - Temperature learnable vs fixed
  - ANN approximation (Flat vs HNSW) effect
- Repeats: 5 seeds per experiment, report mean ± std; statistical tests (paired t-test)

Save results in JSON/CSV for automatic figure generation.

```
[ ]: # 9) Save results & export functions
     import json

     def save_results(results, path='results/attnknn_results.json'):
         os.makedirs(os.path.dirname(path), exist_ok=True)
         with open(path, 'w') as f:
             json.dump(results, f, indent=2)
         print('Saved results to', path)

     print('Notebook utilities ready. Follow the plan to run experiments and␣
      ↪populate results.')
```

```
[ ]: # FAISS index persistence helpers (save/load) and memory arrays
     import os
     import numpy as np
     try:
         import faiss
     except ImportError:
         faiss = None


     def _ensure_dir(path: str):
         os.makedirs(path, exist_ok=True)


     def index_to_cpu(index):
         if faiss is None:
             raise RuntimeError('FAISS not available')
         try:
             return faiss.index_gpu_to_cpu(index)
         except Exception:
             return index


     def save_faiss_index(index, filepath='results/faiss_indices/index.flat'):
         if faiss is None:
             print('FAISS not available; cannot save index')
             return None
         _ensure_dir(os.path.dirname(filepath))
         idx_cpu = index_to_cpu(index)
         faiss.write_index(idx_cpu, filepath)
         print(f'Saved FAISS index -> {filepath}')
         return filepath
```

```
def load_faiss_index(filepath='results/faiss_indices/index.flat'):
    if faiss is None:
        raise RuntimeError('FAISS not available')
    if not os.path.exists(filepath):
        raise FileNotFoundError(filepath)
    idx = faiss.read_index(filepath)
    print(f'Loaded FAISS index <- {filepath}')
    return idx


def save_memory_arrays(embeddings: np.ndarray, labels: np.ndarray,␣
↪dirpath='results/faiss_indices', name='memory'):
    _ensure_dir(dirpath)
    path = os.path.join(dirpath, f'{name}.npz')
    np.savez_compressed(path, embeddings=embeddings.astype('float32'),␣
↪labels=labels.astype('int64'))
    print(f'Saved memory arrays -> {path} (emb={embeddings.shape},␣
↪labels={labels.shape})')
    return path


def load_memory_arrays(path='results/faiss_indices/memory.npz'):
    d = np.load(path)
    emb = d['embeddings'].astype('float32')
    lab = d['labels'].astype('int64')
    print(f'Loaded memory arrays <- {path} (emb={emb.shape}, labels={lab.
↪shape})')
    return emb, lab
```

[ ]:
```
# Metrics visualization & LaTeX export helpers

def results_to_latex_table(results_dict, caption='Quick Benchmarks', label='tab:
↪quick_bench'):
    # results_dict: {name: {accuracy, f1_macro, nll, ece, recall@k}}
    headers = ['Dataset', 'Acc', 'F1', 'NLL', 'ECE', 'Recall@k']
    lines = []
    lines.append('\\begin{table}[h]')
    lines.append('  \\centering')
    lines.append('  \\begin{tabular}{lrrrrr}')
    lines.append('    ' + ' & '.join(headers) + ' \\\n    ')
    lines.append('    \\hline')
    for name, m in results_dict.items():
        row = [name,
               f"{m.get('accuracy', 0):.3f}",
               f"{m.get('f1_macro', 0):.3f}",
               f"{m.get('nll', 0):.3f}",
               f"{m.get('ece', 0):.3f}",
              f"{m.get('recall@k', 0):.3f}"]
        lines.append('    ' + ' & '.join(row) + ' \\')
    lines.append('  \\end{tabular}')
    lines.append(f'  \\caption{{{caption}}}')
    lines.append(f'  \\label{{{label}}}')
    lines.append('\\end{table}')
    tex = '\n'.join(lines)
    os.makedirs('results', exist_ok=True)
    out_path = 'results/attnknn_table.tex'
    with open(out_path, 'w') as f:
        f.write(tex)
    print('Wrote LaTeX table ->', out_path)
    return tex

# Example of aggregating previously saved quick results
# agg = {}
# for name in ['cifar10_quick', 'cifar100_quick', 'wine_quick']:
#     path = f'results/{name}.json'
#     if os.path.exists(path):
#         with open(path, 'r') as f:
#             agg[name] = json.load(f)
# if agg:
#     _ = results_to_latex_table(agg, caption='Attn-KNN Quick Benchmarks',␣
↪label='tab:attnknn_quick')
```

[ ]:
```
# Dataset discovery under DATA_ROOT
from pathlib import Path
from typing import List, Dict, Tuple
import os

print('[DISCOVERY] Scanning for datasets under', DATA_ROOT)

_IMAGE_EXTS = {'.jpg', '.jpeg', '.png', '.bmp', '.gif', '.webp'}


def is_image_file(p: Path) -> bool:
    return p.suffix.lower() in _IMAGE_EXTS


def has_class_subdirs(root: Path, min_classes: int = 2) -> bool:
    if not root.is_dir():
        return False
    class_dirs = [d for d in root.iterdir() if d.is_dir()]
    class_dirs = [d for d in class_dirs if any(is_image_file(f) for f in d.
↪rglob('*') if f.is_file())]
    return len(class_dirs) >= min_classes
```

```python
def discover_image_roots(base: Path) -> List[Dict[str, Path]]:
    candidates: List[Path] = []
    for parent in [base, base / 'images']:
        if not parent.exists():
            continue
        # Heuristic 1: folders that contain 'train' and 'test'
        for d in parent.iterdir():
            if not d.is_dir():
                continue
            if (d / 'train').exists() and (d / 'test').exists():
                candidates.append(d)
            elif has_class_subdirs(d):
                candidates.append(d)
    # Deduplicate
    seen = set()
    out: List[Dict[str, Path]] = []
    for c in candidates:
        key = str(c.resolve())
        if key in seen:
            continue
        seen.add(key)
        out.append({'name': c.name, 'path': c.resolve()})
    return out


def discover_tabular_csvs(base: Path) -> List[Dict[str, Path]]:
    out: List[Dict[str, Path]] = []
    for parent in [base, base / 'tabular']:
        if not parent.exists():
            continue
        for p in parent.rglob('*.csv'):
            # limit depth a bit to avoid scanning huge trees
            if len(p.relative_to(parent).parts) > 3:
                continue
            out.append({'name': p.stem, 'path': p.resolve()})
    # Deduplicate by absolute path
    seen = set()
    uniq: List[Dict[str, Path]] = []
    for d in out:
        key = str(d['path'])
        if key in seen:
            continue
        seen.add(key)
        uniq.append(d)
    return uniq
```

```
DISCOVERED = {
    'images': discover_image_roots(DATA_ROOT),
    'tabular': discover_tabular_csvs(DATA_ROOT)
}
print(f"[DISCOVERY] Found {len(DISCOVERED['images'])} image dataset roots and␣
↪{len(DISCOVERED['tabular'])} CSV files.")
for d in DISCOVERED['images']:
    print('  [image]', d['name'], '->', d['path'])
for d in DISCOVERED['tabular'][:10]:
    print('  [csv]', d['name'], '->', d['path'])
if len(DISCOVERED['tabular']) > 10:
    print('  ... (more CSVs truncated)')
```

[ ]:
```
# ImageFolder loaders using discovered roots
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

print('[LOADERS] Preparing ImageFolder loader helpers...')


def make_imagefolder_loaders(root_dir: Path, batch_size: int = 128):
    # Accept structures: either with train/test subdirs, or a single folder␣
↪with class subdirs
    root_dir = Path(root_dir)
    if (root_dir / 'train').exists() and (root_dir / 'test').exists():
        train_root = root_dir / 'train'
        test_root = root_dir / 'test'
    else:
        # if no explicit split, perform a random split using a subset
        train_root = root_dir
        test_root = root_dir

    normalize = transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.
↪225))
    common_t = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        normalize,
    ])

    train_ds = datasets.ImageFolder(train_root, transform=common_t)
    test_ds = datasets.ImageFolder(test_root, transform=common_t)

    # If single-folder case, create a deterministic split
    if train_root == test_root:
        n = len(train_ds)
        idx = np.arange(n)
        rng = np.random.default_rng(RUN_CONFIG['seed'])
        rng.shuffle(idx)
        split = int(0.8 * n)
        train_idx, test_idx = idx[:split], idx[split:]
        train_subset = torch.utils.data.Subset(train_ds, train_idx.tolist())
        test_subset = torch.utils.data.Subset(test_ds, test_idx.tolist())
        train_loader = DataLoader(train_subset, batch_size=batch_size,␣
↪shuffle=True, num_workers=4)
        test_loader = DataLoader(test_subset, batch_size=batch_size,␣
↪shuffle=False, num_workers=4)
    else:
        train_loader = DataLoader(train_ds, batch_size=batch_size,␣
↪shuffle=True, num_workers=4)
        test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
↪num_workers=4)

    print(f"[LOADERS] ImageFolder: classes={len(train_ds.classes)}␣
↪train={len(train_loader.dataset)} test={len(test_loader.dataset)} from␣
↪{root_dir}")
    return train_loader, test_loader, len(train_ds.classes)
```

[ ]:
```
# Evaluation helpers for the pipeline - VECTORIZED (no slow Python loops!)

print('[EVAL] Defining OPTIMIZED evaluation functions...')


def evaluate_attn_knn_verbose(
    embedder: nn.Module,
    attn_head: nn.Module,
    index: faiss.Index,
    mem_embs: np.ndarray,
    mem_labels: np.ndarray,
    test_loader: DataLoader,
    k: int = 10,
    desc: str = 'Eval'
) -> tuple[dict, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Evaluate attention-weighted kNN - VECTORIZED."""
    embedder.eval()
    attn_head.eval()
    all_preds, all_targets, all_probs = [], [], []
    num_classes = int(mem_labels.max()) + 1
    with torch.no_grad():
        for xb, yb in tqdm(test_loader, desc=desc):
            xb = xb.to(device)
            zq = embedder(xb)
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index.search(q_np, k)
            neighbor_embs = mem_embs[I]
            neighbor_embs_t = torch.from_numpy(neighbor_embs).to(device)
            neigh_labels = torch.from_numpy(mem_labels[I]).long().to(device)
            attn_w = attn_head(zq, neighbor_embs_t)
            # VECTORIZED aggregation
            probs = aggregate_neighbor_probs_weighted(neigh_labels, attn_w,␣
↪num_classes)
            preds = probs.argmax(dim=1).cpu().numpy()
            all_preds.append(preds)
            all_targets.append(yb.cpu().numpy())
            all_probs.append(probs.cpu().numpy())
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_targets)
    probs = np.vstack(all_probs)
    probs_clipped = np.clip(probs, 1e-9, 1.0)  # Clip for log_loss
    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, average='macro')
    nll = log_loss(y_true, probs_clipped)
    ece_val = ece_score(probs, y_true)
    cm = confusion_matrix(y_true, y_pred)
    return {'accuracy': acc, 'f1_macro': f1, 'nll': nll, 'ece': ece_val},␣
↪y_true, y_pred, probs, cm


def evaluate_uniform_knn_verbose(
    embedder: nn.Module,
    index: faiss.Index,
    mem_embs: np.ndarray,
    mem_labels: np.ndarray,
    test_loader: DataLoader,
    k: int = 10,
    desc: str = 'EvalUniform'
) -> tuple[dict, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """Evaluate uniform (majority voting) kNN - VECTORIZED."""
    embedder.eval()
    all_preds, all_targets, all_probs = [], [], []
    num_classes = int(mem_labels.max()) + 1
    with torch.no_grad():
        for xb, yb in tqdm(test_loader, desc=desc):
            xb = xb.to(device)
            zq = embedder(xb)
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index.search(q_np, k)
            neigh_labels = torch.from_numpy(mem_labels[I]).long().to(device)
            # VECTORIZED aggregation
            probs = aggregate_neighbor_probs_uniform(neigh_labels, num_classes,␣
↪k)
            preds = probs.argmax(dim=1).cpu().numpy()
            all_preds.append(preds)
            all_targets.append(yb.cpu().numpy())
            all_probs.append(probs.cpu().numpy())
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_targets)
    probs = np.vstack(all_probs)
    probs_clipped = np.clip(probs, 1e-9, 1.0)  # Clip for log_loss
    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, average='macro')
    nll = log_loss(y_true, probs_clipped)
    ece_val = ece_score(probs, y_true)
    cm = confusion_matrix(y_true, y_pred)
    return {'accuracy': acc, 'f1_macro': f1, 'nll': nll, 'ece': ece_val},␣
↪y_true, y_pred, probs, cm
```

[ ]:
```
# Confusion matrix plotting utility
import matplotlib.pyplot as plt
import seaborn as sns

print('[PLOTS] Confusion matrix plotter ready')


def plot_confusion(cm, class_names=None, title='Confusion Matrix',␣
↪savepath=None):
    plt.figure(figsize=(6, 5))
    ax = sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')
    ax.set_xlabel('Predicted')
    ax.set_ylabel('True')
    ax.set_title(title)
    if class_names is not None:
        ax.set_xticklabels(class_names, rotation=45, ha='right')
        ax.set_yticklabels(class_names, rotation=0)
    plt.tight_layout()
    if savepath:
        os.makedirs(os.path.dirname(savepath), exist_ok=True)
        plt.savefig(savepath, dpi=150)
        print('[PLOTS] Saved', savepath)
        plt.close()
    else:
        plt.show()
```

[ ]:
```
# Run-all pipeline over discovered datasets, saving metrics

print('[PIPELINE] Preparing run-all execution...')
```

```python
def build_memory_from_loader_verbose(embedder, loader, desc='BuildMem'):
    embedder.eval()
    embs, labels = [], []
    total = 0
    with torch.no_grad():
        for i, (xb, yb) in enumerate(loader, 1):
            if i % 20 == 0:
                print(f"[PIPELINE] {desc}: processed {i} batches...")
            xb = xb.to(device)
            z = embedder(xb)
            z = F.normalize(z, dim=1)
            embs.append(z.cpu().numpy())
            labels.append(yb.numpy())
            total += xb.size(0)
    embs = np.vstack(embs).astype('float32')
    labels = np.concatenate(labels).astype('int64')
    print(f"[PIPELINE] {desc}: done. embs={embs.shape} labels={labels.shape}")
    return embs, labels


def run_image_dataset(root: Path, name: str, k: int):
    print(f"[RUN] Image dataset: {name} -> {root}")
    train_loader, test_loader, num_classes = make_imagefolder_loaders(root,␣
 ↪batch_size=RUN_CONFIG['batch_size_image'])
    embedder = EmbedderResNet18(embed_dim=128, pretrained=False).to(device)
    # Build memory
    mem_embs, mem_labels = build_memory_from_loader_verbose(embedder,␣
 ↪train_loader, desc=f'BuildMem:{name}')
    mem_bank = build_memorybank_from_embeddings(mem_embs, mem_labels,␣
 ↪index_type=RUN_CONFIG['index_type'])
    index = mem_bank.index
    # Attention head (optional training skipped by default)
    attn = AttnHead(embed_dim=128, proj_dim=64).to(device)
    metrics, y_true, y_pred, probs, cm = evaluate_attn_knn_verbose(embedder,␣
 ↪attn, index, mem_embs, mem_labels, test_loader, k=k, desc=f'Eval:{name}')
    # Persist
    out_prefix = RESULTS_DIR / f'image_{name}'
    os.makedirs(out_prefix, exist_ok=True)
    save_memory_arrays(mem_embs, mem_labels, dirpath=str(out_prefix),␣
 ↪name='memory')
    save_faiss_index(index, filepath=str(out_prefix / 'index.faiss'))
    save_results(metrics, path=str(out_prefix / 'metrics.json'))
    np.save(str(out_prefix / 'y_true.npy'), y_true)
    np.save(str(out_prefix / 'y_pred.npy'), y_pred)
    np.save(str(out_prefix / 'probs.npy'), probs)
    plot_confusion(cm, title=f'Confusion: {name}', savepath=str(out_prefix /␣
↪'confusion.png'))
    print(f"[RUN] {name} metrics: {metrics}")
    return name, metrics


def infer_csv_target_column(csv_path: Path) -> str:
    import pandas as pd
    df = pd.read_csv(csv_path, nrows=5)
    # Heuristic: label/target/class/quality or last column fallback
    candidates = [c for c in df.columns if c.lower() in ('label', 'target',␣
↪'class', 'quality', 'y')]
    return candidates[0] if candidates else df.columns[-1]


def run_tabular_csv(csv_path: Path, k: int):
    print(f"[RUN] Tabular CSV: {csv_path.name} -> {csv_path}")
    target_col = infer_csv_target_column(csv_path)
    print(f"[RUN] Using target column: {target_col}")
    X_train, X_test, y_train, y_test = load_uci_tabular(str(csv_path),␣
↪target_col=target_col)
    # Label encoding if needed
    from sklearn.preprocessing import LabelEncoder
    le = LabelEncoder()
    y_train_enc = le.fit_transform(y_train)
    y_test_enc = le.transform(y_test)
    train_loader, test_loader = get_tabular_loaders_from_arrays(X_train,␣
↪y_train_enc, X_test, y_test_enc, batch_size=RUN_CONFIG['batch_size_tabular'])
    embedder = MLPEmbedder(input_dim=X_train.shape[1], embed_dim=128).to(device)
    mem_embs, mem_labels = build_memory_from_loader_verbose(embedder,␣
↪train_loader, desc=f'BuildMem:{csv_path.stem}')
    mem_bank = build_memorybank_from_embeddings(mem_embs, mem_labels,␣
↪index_type=RUN_CONFIG['index_type'])
    index = mem_bank.index
    attn = AttnHead(embed_dim=128, proj_dim=64).to(device)
    metrics, y_true, y_pred, probs, cm = evaluate_attn_knn_verbose(embedder,␣
↪attn, index, mem_embs, mem_labels, test_loader, k=k, desc=f'Eval:{csv_path.
↪stem}')
    out_prefix = RESULTS_DIR / f'tabular_{csv_path.stem}'
    os.makedirs(out_prefix, exist_ok=True)
    save_memory_arrays(mem_embs, mem_labels, dirpath=str(out_prefix),␣
↪name='memory')
    save_faiss_index(index, filepath=str(out_prefix / 'index.faiss'))
    save_results(metrics, path=str(out_prefix / 'metrics.json'))
    np.save(str(out_prefix / 'y_true.npy'), y_true)
    np.save(str(out_prefix / 'y_pred.npy'), y_pred)
    np.save(str(out_prefix / 'probs.npy'), probs)
    plot_confusion(cm, title=f'Confusion: {csv_path.stem}',␣
  ↪savepath=str(out_prefix / 'confusion.png'))
    print(f"[RUN] {csv_path.stem} metrics: {metrics}")
    return csv_path.stem, metrics


def run_all_discovered(k=None, limit_images=None, limit_tabular=None):
    k = k or RUN_CONFIG['k']
    results = {}
    # Images
    count = 0
    for d in DISCOVERED['images']:
        if limit_images is not None and count >= limit_images:
            break
        name, m = run_image_dataset(d['path'], d['name'], k)
        results[f'image:{name}'] = m
        count += 1
    # Tabular
    count = 0
    for d in DISCOVERED['tabular']:
        if limit_tabular is not None and count >= limit_tabular:
            break
        name, m = run_tabular_csv(d['path'], k)
        results[f'csv:{name}'] = m
        count += 1
    # Save aggregate
    agg_path = RESULTS_DIR / 'aggregate_metrics.json'
    with open(agg_path, 'w') as f:
        json.dump(results, f, indent=2)
    print('[PIPELINE] Wrote aggregate metrics ->', agg_path)
    return results

print('[PIPELINE] Ready. Use run_all_discovered(limit_images=...,␣
  ↪limit_tabular=...) to execute.')
```

```
[ ]: # Aggregate summary and LaTeX export
print('[AGG] Aggregate summary cell ready')


def aggregate_and_export_table(aggregate_path=None, caption='Attn-KNN Results',␣
  ↪label='tab:attnknn_results'):
    aggregate_path = aggregate_path or (RESULTS_DIR / 'aggregate_metrics.json')
    if not os.path.exists(aggregate_path):
        print('[AGG] No aggregate metrics found at', aggregate_path)
        return None
    with open(aggregate_path, 'r') as f:
        results = json.load(f)
    # Write LaTeX
    tex = results_to_latex_table(results, caption=caption, label=label)
    print('[AGG] Exported LaTeX table. Entries:', len(results))
    return results, tex

# Example usage (uncomment after running):
# agg_results, tex = aggregate_and_export_table()
```

[ ]:
```
# DIAGNOSTIC: Quick test to verify everything works before running full␣
  ↪experiment
# Run this cell first to catch any issues early!

print('=' * 60)
print('DIAGNOSTIC: Testing components...')
print('=' * 60)

# 1. Test device
print(f'1. Device: {device}')

# 2. Test SmallImageEmbedder
test_embedder = SmallImageEmbedder(embed_dim=128, in_channels=3).to(device)
test_input = torch.randn(4, 3, 32, 32).to(device)
with torch.no_grad():
    test_out = test_embedder(test_input)
    if device.type == 'mps':
        torch.mps.synchronize()
print(f'2. SmallImageEmbedder: input {test_input.shape} -> output {test_out.
  ↪shape}')

# 3. Test FAISS
test_embs = np.random.randn(100, 128).astype('float32')
test_index = faiss.IndexFlatL2(128)
test_index.add(test_embs)
D, I = test_index.search(test_embs[:5], k=10)
print(f'3. FAISS: indexed {test_index.ntotal} vectors, search returned shape {I.
  ↪shape}')

# 4. Test NumPy aggregation
test_labels = np.random.randint(0, 10, size=(4, 10))
test_probs = aggregate_neighbor_probs_uniform_np(test_labels, num_classes=10,␣
  ↪k=10)
print(f'4. NumPy uniform aggregation: input {test_labels.shape} -> probs␣
  ↪{test_probs.shape}')

test_weights = np.random.rand(4, 10).astype('float32')
test_weights = test_weights / test_weights.sum(axis=1, keepdims=True)
```

```
test_probs_w = aggregate_neighbor_probs_weighted_np(test_labels, test_weights,␣
  ↪num_classes=10)
print(f'5. NumPy weighted aggregation: input {test_labels.shape}, weights␣
  ↪{test_weights.shape} -> probs {test_probs_w.shape}')

# 5. Test AttnKNNHead
test_attn = AttnKNNHead(embed_dim=128, proj_dim=64).to(device)
test_q = torch.randn(4, 128).to(device)
test_neigh = torch.randn(4, 10, 128).to(device)
with torch.no_grad():
    test_attn_out = test_attn(test_q, test_neigh)
    if device.type == 'mps':
        torch.mps.synchronize()
print(f'6. AttnKNNHead: q {test_q.shape}, neigh {test_neigh.shape} -> weights␣
  ↪{test_attn_out.shape}')

# Clean up
del test_embedder, test_input, test_out, test_embs, test_index
del test_labels, test_probs, test_weights, test_probs_w
del test_attn, test_q, test_neigh, test_attn_out

print('=' * 60)
print('All diagnostics PASSED! Ready to run experiments.')
print('=' * 60)
```

4.1 Quick Experiments (CIFAR-10 and Iris)

Run these cells to generate credible results for your pre-proposal. They compare uniform kNN vs Attn-KNN on image (CIFAR-10) and tabular (Iris) data, reporting accuracy, F1, NLL, and ECE.

[48]:
```
# CIFAR-10 Quick Experiment: Uniform kNN vs Attn-KNN (OPTIMIZED v2)
# Uses native 32x32 images with SmallImageEmbedder + NumPy aggregation

from torchvision import transforms, datasets
import time

print('[CIFAR10] Starting OPTIMIZED quick experiment (v2)...')
print(f'[CIFAR10] Using device: {device}')
set_seed(42)

# Data transforms - NATIVE 32x32, NO RESIZE
normalize = transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.
  ↪261))
t_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    normalize
])
t_test = transforms.Compose([
    transforms.ToTensor(),
    normalize
])

# Load CIFAR-10
data_root = str(DATA_ROOT) if DATA_ROOT.exists() else './data'
print(f'[CIFAR10] Loading from {data_root}')
train_ds = datasets.CIFAR10(root=data_root, train=True, download=True,␣
 ↪transform=t_train)
test_ds = datasets.CIFAR10(root=data_root, train=False, download=True,␣
 ↪transform=t_test)

# DataLoader settings optimized for MPS (no pin_memory, num_workers=2)
train_loader = DataLoader(train_ds, batch_size=256, shuffle=True, num_workers=2)
test_loader = DataLoader(test_ds, batch_size=256, shuffle=False, num_workers=2)
print(f'[CIFAR10] Train: {len(train_ds)}, Test: {len(test_ds)}')

# Use SmallImageEmbedder for 32x32 images (much faster than resizing to 224x224)
embedder = SmallImageEmbedder(embed_dim=128, in_channels=3, pretrained=False).
 ↪to(device)
embedder.eval()
print(f'[CIFAR10] SmallImageEmbedder ready on {device}')

# Build memory bank
print('[CIFAR10] Building memory bank...')
mem_embs_list, mem_labels_list = [], []
with torch.no_grad():
    for xb, yb in tqdm(train_loader, desc='Building memory'):
        xb = xb.to(device)
        z = embedder(xb)
        mem_embs_list.append(z.cpu().numpy().astype('float32'))
        mem_labels_list.append(yb.numpy().astype('int64'))

mem_embs = np.vstack(mem_embs_list)
mem_labels = np.concatenate(mem_labels_list)
print(f'[CIFAR10] Memory: embs={mem_embs.shape}, labels={mem_labels.shape}')

# FAISS index
index = build_faiss_index(mem_embs, index_type='Flat', metric='L2',␣
 ↪use_gpu=False)
print(f'[CIFAR10] FAISS index built: {index.ntotal} vectors')

# Evaluation functions - NumPy-based aggregation (guaranteed fast!)
K = 10
```

```
NUM_CLASSES = 10


def eval_uniform_cifar(k: int = K) -> dict:
    """Evaluate uniform kNN using NumPy aggregation (fast on all platforms)."""
    embedder.eval()
    all_preds, all_labels, all_probs = [], [], []
    start_time = time.time()
    with torch.no_grad():
        for batch_idx, (xb, yb) in enumerate(tqdm(test_loader, desc='Eval␣
 ↪Uniform')):
            xb = xb.to(device)
            zq = embedder(xb)
            # Force MPS sync before CPU transfer
            if device.type == 'mps':
                torch.mps.synchronize()
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index.search(q_np, k)
            neigh_labels = mem_labels[I]  # (B, k) numpy array
            # NumPy aggregation - guaranteed fast!
            probs = aggregate_neighbor_probs_uniform_np(neigh_labels,␣
 ↪NUM_CLASSES, k)
            preds = probs.argmax(axis=1)
            all_preds.append(preds)
            all_labels.append(yb.numpy())
            all_probs.append(probs)
    elapsed = time.time() - start_time
    print(f'[Uniform] Evaluation completed in {elapsed:.1f}s')
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_labels)
    probs = np.vstack(all_probs)
    # Clip probabilities to [eps, 1] to avoid log(0) and satisfy sklearn's␣
 ↪validation
    probs_clipped = np.clip(probs, 1e-9, 1.0)
    return {
        'accuracy': float(accuracy_score(y_true, y_pred)),
        'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
        'nll': float(log_loss(y_true, probs_clipped)),
        'ece': float(ece_score(probs, y_true))
    }


def eval_attn_cifar(k: int = K) -> dict:
    """Evaluate attention-weighted kNN using NumPy aggregation."""
    embedder.eval()
    attn = AttnKNNHead(embed_dim=128, proj_dim=64, temperature_init=1.0,␣
 ↪learnable_temp=True).to(device)
```

```
    attn.eval()
    all_preds, all_labels, all_probs = [], [], []
    start_time = time.time()
    with torch.no_grad():
        for batch_idx, (xb, yb) in enumerate(tqdm(test_loader, desc='Eval␣
↪Attn-KNN')):
            xb = xb.to(device)
            zq = embedder(xb)
            # Force MPS sync before CPU transfer
            if device.type == 'mps':
                torch.mps.synchronize()
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index.search(q_np, k)
            neighbor_embs = mem_embs[I]
            neighbor_embs_t = torch.from_numpy(neighbor_embs).to(device)
            neigh_labels = mem_labels[I]  # Keep as numpy
            weights = attn(zq, neighbor_embs_t)
            if device.type == 'mps':
                torch.mps.synchronize()
            weights_np = weights.cpu().numpy()
            # NumPy aggregation - guaranteed fast!
            probs = aggregate_neighbor_probs_weighted_np(neigh_labels,␣
↪weights_np, NUM_CLASSES)
            preds = probs.argmax(axis=1)
            all_preds.append(preds)
            all_labels.append(yb.numpy())
            all_probs.append(probs)
    elapsed = time.time() - start_time
    print(f'[Attn-KNN] Evaluation completed in {elapsed:.1f}s')
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_labels)
    probs = np.vstack(all_probs)
    # Clip probabilities to [eps, 1] to avoid log(0) and satisfy sklearn's␣
↪validation
    probs_clipped = np.clip(probs, 1e-9, 1.0)
    return {
        'accuracy': float(accuracy_score(y_true, y_pred)),
        'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
        'nll': float(log_loss(y_true, probs_clipped)),
        'ece': float(ece_score(probs, y_true))
    }


# Run evaluations
print('[CIFAR10] Evaluating Uniform kNN...')
uniform_metrics = eval_uniform_cifar(k=K)
print('[CIFAR10] Evaluating Attn-KNN...')
```

```python
attn_metrics = eval_attn_cifar(k=K)

# Save results
out_dir = RESULTS_DIR / 'cifar10_quick'
os.makedirs(out_dir, exist_ok=True)
with open(out_dir / 'metrics_uniform.json', 'w') as f:
    json.dump(uniform_metrics, f, indent=2)
with open(out_dir / 'metrics_attn.json', 'w') as f:
    json.dump(attn_metrics, f, indent=2)

# Print comparison
print('\n' + '='*60)
print('CIFAR-10 Results (k={})'.format(K))
print('='*60)
print(f"{'Metric':<12} {'Uniform kNN':>15} {'Attn-KNN':>15}")
print('-'*60)
for key in ['accuracy', 'f1_macro', 'nll', 'ece']:
    print(f"{key:<12} {uniform_metrics[key]:>15.4f} {attn_metrics[key]:>15.4f}")
print('='*60)
print(f'Results saved to {out_dir}')
```

```
Building memory: 100%|      | 196/196 [00:18<00:00, 10.62it/s]

[CIFAR10] Memory: embs=(50000, 128), labels=(50000,)
[CIFAR10] FAISS index built: 50000 vectors
[CIFAR10] Evaluating Uniform kNN…

Eval Uniform: 100%|      | 40/40 [00:14<00:00,  2.74it/s]

[Uniform] Evaluation completed in 16.9s
[CIFAR10] Evaluating Attn-KNN…

Eval Attn-KNN: 100%|      | 40/40 [00:14<00:00,  2.72it/s]

[Attn-KNN] Evaluation completed in 16.9s


============================================================
CIFAR-10 Results (k=10)
============================================================
Metric           Uniform kNN        Attn-KNN
------------------------------------------------------------
accuracy              0.2865          0.2908
f1_macro              0.2788          0.2819
nll                   4.4639          4.4640
ece                   0.0996          0.0968
============================================================
Results saved to
/Users/taher/Projects/attn_knn_repo/notebook/results/cifar10_quick
```

```
[49]: # Iris Quick Experiment: Tabular Attn-KNN
# Demonstrates domain generality on a classic tabular dataset

print('[IRIS] Starting quick experiment...')
set_seed(42)

# Load Iris dataset (sklearn has it built-in)
from sklearn.datasets import load_iris
iris = load_iris()
X_iris = iris.data.astype(np.float32)
y_iris = iris.target.astype(np.int64)

# Train/test split
X_train_iris, X_test_iris, y_train_iris, y_test_iris = train_test_split(
    X_iris, y_iris, test_size=0.2, random_state=42, stratify=y_iris
)

# Standardize
scaler = StandardScaler()
X_train_iris = scaler.fit_transform(X_train_iris).astype(np.float32)
X_test_iris = scaler.transform(X_test_iris).astype(np.float32)

print(f'[IRIS] Train: {X_train_iris.shape}, Test: {X_test_iris.shape}')

# DataLoaders
train_loader_iris = DataLoader(
    TensorDataset(torch.tensor(X_train_iris), torch.tensor(y_train_iris)),
    batch_size=256, shuffle=False
)
test_loader_iris = DataLoader(
    TensorDataset(torch.tensor(X_test_iris), torch.tensor(y_test_iris)),
    batch_size=256, shuffle=False
)

# MLP Embedder for tabular
embedder_iris = MLPEmbedder(input_dim=X_train_iris.shape[1], embed_dim=64).
  ↪to(device)
embedder_iris.eval()

# Build memory
print('[IRIS] Building memory bank...')
mem_embs_iris_list, mem_labels_iris_list = [], []
with torch.no_grad():
    for xb, yb in train_loader_iris:
        xb = xb.to(device)
        z = embedder_iris(xb)
        mem_embs_iris_list.append(z.cpu().numpy().astype('float32'))
        mem_labels_iris_list.append(yb.numpy().astype('int64'))

mem_embs_iris = np.vstack(mem_embs_iris_list)
mem_labels_iris = np.concatenate(mem_labels_iris_list)
print(f'[IRIS] Memory: embs={mem_embs_iris.shape}, labels={mem_labels_iris.
  ↪shape}')

# FAISS index
index_iris = build_faiss_index(mem_embs_iris, index_type='Flat', metric='L2',␣
  ↪use_gpu=False)

# Evaluation
K_IRIS = 5
NUM_CLASSES_IRIS = 3


def eval_uniform_iris(k: int = K_IRIS) -> dict:
    embedder_iris.eval()
    all_preds, all_labels, all_probs = [], [], []
    with torch.no_grad():
        for xb, yb in test_loader_iris:
            xb = xb.to(device)
            zq = embedder_iris(xb)
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index_iris.search(q_np, k)
            neigh_labels_t = torch.from_numpy(mem_labels_iris[I]).long().
  ↪to(device)
            # VECTORIZED aggregation
            probs = aggregate_neighbor_probs_uniform(neigh_labels_t,␣
  ↪NUM_CLASSES_IRIS, k)
            preds = probs.argmax(dim=1).cpu().numpy()
            all_preds.append(preds)
            all_labels.append(yb.numpy())
            all_probs.append(probs.cpu().numpy())
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_labels)
    probs = np.vstack(all_probs)
    return {
        'accuracy': float(accuracy_score(y_true, y_pred)),
        'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
        'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
        'ece': float(ece_score(probs, y_true))
    }


def eval_attn_iris(k: int = K_IRIS) -> dict:
    embedder_iris.eval()
    attn_iris = AttnKNNHead(embed_dim=64, proj_dim=32, temperature_init=1.0,␣
↪learnable_temp=True).to(device)
    attn_iris.eval()
    all_preds, all_labels, all_probs = [], [], []
    with torch.no_grad():
        for xb, yb in test_loader_iris:
            xb = xb.to(device)
            zq = embedder_iris(xb)
            q_np = zq.cpu().numpy().astype('float32')
            D, I = index_iris.search(q_np, k)
            neighbor_embs = mem_embs_iris[I]
            neighbor_embs_t = torch.from_numpy(neighbor_embs).to(device)
            neigh_labels_t = torch.from_numpy(mem_labels_iris[I]).long().
↪to(device)
            weights = attn_iris(zq, neighbor_embs_t)
            # VECTORIZED aggregation
            probs = aggregate_neighbor_probs_weighted(neigh_labels_t, weights,␣
↪NUM_CLASSES_IRIS)
            preds = probs.argmax(dim=1).cpu().numpy()
            all_preds.append(preds)
            all_labels.append(yb.cpu().numpy())
            all_probs.append(probs.cpu().numpy())
    y_pred = np.concatenate(all_preds)
    y_true = np.concatenate(all_labels)
    probs = np.vstack(all_probs)
    return {
        'accuracy': float(accuracy_score(y_true, y_pred)),
        'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
        'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
        'ece': float(ece_score(probs, y_true))
    }


# Run evaluations
print('[IRIS] Evaluating Uniform kNN...')
uniform_metrics_iris = eval_uniform_iris(k=K_IRIS)
print('[IRIS] Evaluating Attn-KNN...')
attn_metrics_iris = eval_attn_iris(k=K_IRIS)

# Save results
out_dir_iris = RESULTS_DIR / 'iris_quick'
os.makedirs(out_dir_iris, exist_ok=True)
with open(out_dir_iris / 'metrics_uniform.json', 'w') as f:
    json.dump(uniform_metrics_iris, f, indent=2)
with open(out_dir_iris / 'metrics_attn.json', 'w') as f:
    json.dump(attn_metrics_iris, f, indent=2)
```

```python
# Print comparison
print('\n' + '='*60)
print('Iris Results (k={})'.format(K_IRIS))
print('='*60)
print(f"{'Metric':<12} {'Uniform kNN':>15} {'Attn-KNN':>15}")
print('-'*60)
for key in ['accuracy', 'f1_macro', 'nll', 'ece']:
    print(f"{key:<12} {uniform_metrics_iris[key]:>15.4f}␣
 ↪{attn_metrics_iris[key]:>15.4f}")
print('='*60)
print(f'Results saved to {out_dir_iris}')
```

```
[IRIS] Starting quick experiment…
[IRIS] Train: (120, 4), Test: (30, 4)
[IRIS] Building memory bank…
[IRIS] Memory: embs=(120, 64), labels=(120,)
[IRIS] Evaluating Uniform kNN…
[IRIS] Evaluating Attn-KNN…

============================================================
Iris Results (k=5)
============================================================
Metric           Uniform kNN        Attn-KNN
------------------------------------------------------------
accuracy              0.8667          0.8667
f1_macro              0.8667          0.8667
nll                   0.1519          0.1490
ece                   0.1067          0.1052
============================================================
Results saved to /Users/taher/Projects/attn_knn_repo/notebook/results/iris_quick
```

[50]:
```python
# Aggregate Results and Export LaTeX Table
# Run this after both CIFAR-10 and Iris experiments complete

print('[AGGREGATE] Combining results...')

# Load saved results
results_summary = {}

cifar_uni_path = RESULTS_DIR / 'cifar10_quick' / 'metrics_uniform.json'
cifar_attn_path = RESULTS_DIR / 'cifar10_quick' / 'metrics_attn.json'
iris_uni_path = RESULTS_DIR / 'iris_quick' / 'metrics_uniform.json'
iris_attn_path = RESULTS_DIR / 'iris_quick' / 'metrics_attn.json'

if cifar_uni_path.exists():
    with open(cifar_uni_path) as f:
        results_summary['CIFAR-10 Uniform'] = json.load(f)
```

```
if cifar_attn_path.exists():
    with open(cifar_attn_path) as f:
        results_summary['CIFAR-10 Attn-KNN'] = json.load(f)
if iris_uni_path.exists():
    with open(iris_uni_path) as f:
        results_summary['Iris Uniform'] = json.load(f)
if iris_attn_path.exists():
    with open(iris_attn_path) as f:
        results_summary['Iris Attn-KNN'] = json.load(f)

# Print summary table
print('\n' + '='*80)
print('AGGREGATE RESULTS SUMMARY')
print('='*80)
print(f"{'Dataset/Method':<25} {'Accuracy':>10} {'F1-Macro':>10} {'NLL':>10}␣
↪{'ECE':>10}")
print('-'*80)
for name, m in results_summary.items():
    print(f"{name:<25} {m['accuracy']:>10.4f} {m['f1_macro']:>10.4f} {m['nll']:
↪>10.4f} {m['ece']:>10.4f}")
print('='*80)

# Export LaTeX table
def results_to_latex(results: dict, caption: str = 'Attn-KNN Quick Benchmarks',␣
↪label: str = 'tab:attnknn_quick') -> str:
    lines = [
        '\\begin{table}[h]',
        '  \\centering',
        '  \\begin{tabular}{lrrrr}',
        '    \\toprule',
        '    Dataset/Method & Accuracy & F1-Macro & NLL & ECE \\\\',
        '    \\midrule'
    ]
    for name, m in results.items():
        row = f"    {name} & {m['accuracy']:.4f} & {m['f1_macro']:.4f} &␣
↪{m['nll']:.4f} & {m['ece']:.4f} \\\\"
        lines.append(row)
    lines.extend([
        '    \\bottomrule',
        '  \\end{tabular}',
        f'  \\caption{{{caption}}}',
        f'  \\label{{{label}}}',
        '\\end{table}'
    ])
    return '\n'.join(lines)

latex_table = results_to_latex(results_summary)
```

```python
latex_path = RESULTS_DIR / 'attnknn_quick_table.tex'
with open(latex_path, 'w') as f:
    f.write(latex_table)
print(f'\n[AGGREGATE] LaTeX table saved to {latex_path}')
print('\nLaTeX output:')
print(latex_table)

# Save aggregate JSON
agg_path = RESULTS_DIR / 'aggregate_quick.json'
with open(agg_path, 'w') as f:
    json.dump(results_summary, f, indent=2)
print(f'\n[AGGREGATE] Aggregate JSON saved to {agg_path}')
```

```
[AGGREGATE] Combining results…

================================================================================
AGGREGATE RESULTS SUMMARY
================================================================================
Dataset/Method              Accuracy   F1-Macro        NLL        ECE
--------------------------------------------------------------------------------
CIFAR-10 Uniform              0.2865     0.2788     4.4639     0.0996
CIFAR-10 Attn-KNN             0.2908     0.2819     4.4640     0.0968
Iris Uniform                  0.8667     0.8667     0.1519     0.1067
Iris Attn-KNN                 0.8667     0.8667     0.1490     0.1052
================================================================================

[AGGREGATE] LaTeX table saved to
/Users/taher/Projects/attn_knn_repo/notebook/results/attnknn_quick_table.tex

LaTeX output:
\begin{table}[h]
  \centering
  \begin{tabular}{lrrrr}
    \toprule
    Dataset/Method & Accuracy & F1-Macro & NLL & ECE \\
    \midrule
    CIFAR-10 Uniform & 0.2865 & 0.2788 & 4.4639 & 0.0996 \\
    CIFAR-10 Attn-KNN & 0.2908 & 0.2819 & 4.4640 & 0.0968 \\
    Iris Uniform & 0.8667 & 0.8667 & 0.1519 & 0.1067 \\
    Iris Attn-KNN & 0.8667 & 0.8667 & 0.1490 & 0.1052 \\
    \bottomrule
  \end{tabular}
  \caption{Attn-KNN Quick Benchmarks}
  \label{tab:attnknn_quick}
\end{table}

[AGGREGATE] Aggregate JSON saved to
/Users/taher/Projects/attn_knn_repo/notebook/results/aggregate_quick.json
```

4.2 Data Loaders for Your Local Datasets

These loaders handle the specific datasets in your `/data` folder:

- **Image**: ImageNet, MNIST, CIFAR-10/100 (use `SimpleEmbedder`)
- **Tabular**: Adult, Iris, Wine-Quality (use `MLPEmbedder` with preprocessing)

[51]:
```
# Data loaders for YOUR local datasets in /data folder

DATA_PATH = Path('/Users/taher/Projects/attn_knn_repo/data')

# ============================================================================
# TABULAR DATA LOADERS (use MLPEmbedder)
# ============================================================================

def load_adult_dataset(
    data_path: Path = DATA_PATH / 'adult'
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, int]:
    """Load UCI Adult dataset with proper preprocessing for mixed features.

    Returns:
        X_train, X_test, y_train, y_test, input_dim
    """
    # Column names for Adult dataset
    columns = [
        'age', 'workclass', 'fnlwgt', 'education', 'education-num',
        'marital-status', 'occupation', 'relationship', 'race', 'sex',
        'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',␣
↪'income'
    ]

    # Load train and test
    df_train = pd.read_csv(data_path / 'adult.data', names=columns,␣
↪skipinitialspace=True)
    df_test = pd.read_csv(data_path / 'adult.test', names=columns,␣
↪skipinitialspace=True, skiprows=1)

    # Clean target (remove trailing period in test set)
    df_train['income'] = df_train['income'].str.strip()
    df_test['income'] = df_test['income'].str.strip().str.rstrip('.')

    # Drop rows with missing values (marked as '?')
    df_train = df_train.replace('?', np.nan).dropna()
    df_test = df_test.replace('?', np.nan).dropna()

    # Separate features and target
    X_train = df_train.drop(columns=['income'])
    y_train = (df_train['income'] == '>50K').astype(np.int64)
    X_test = df_test.drop(columns=['income'])
    y_test = (df_test['income'] == '>50K').astype(np.int64)

    # One-hot encode categorical columns
    categorical_cols = ['workclass', 'education', 'marital-status',␣
  ↪'occupation',
                        'relationship', 'race', 'sex', 'native-country']
    X_train = pd.get_dummies(X_train, columns=categorical_cols)
    X_test = pd.get_dummies(X_test, columns=categorical_cols)

    # Align columns (test may have different categories)
    X_train, X_test = X_train.align(X_test, join='left', axis=1, fill_value=0)

    # Standardize numerical features
    scaler = StandardScaler()
    X_train_np = scaler.fit_transform(X_train.values).astype(np.float32)
    X_test_np = scaler.transform(X_test.values).astype(np.float32)

    print(f'[ADULT] Train: {X_train_np.shape}, Test: {X_test_np.shape}, Classes:
  ↪ 2')
    return X_train_np, X_test_np, y_train.values, y_test.values, X_train_np.
  ↪shape[1]


def load_iris_dataset(
    data_path: Path = DATA_PATH / 'iris'
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, int]:
    """Load Iris dataset from local files.

    Returns:
        X_train, X_test, y_train, y_test, input_dim
    """
    # Load iris.data (no header)
    columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',␣
  ↪'class']
    df = pd.read_csv(data_path / 'iris.data', names=columns)
    df = df.dropna()

    X = df.drop(columns=['class']).values.astype(np.float32)
    y_str = df['class'].values

    # Encode labels
    le = LabelEncoder()
    y = le.fit_transform(y_str).astype(np.int64)

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    # Standardize
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train).astype(np.float32)
    X_test = scaler.transform(X_test).astype(np.float32)

    print(f'[IRIS] Train: {X_train.shape}, Test: {X_test.shape}, Classes:␣
  ↪{len(le.classes_)}')
    return X_train, X_test, y_train, y_test, X_train.shape[1]


def load_wine_dataset(
    data_path: Path = DATA_PATH / 'wine-quality',
    wine_type: str = 'red'  # 'red' or 'white'
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, int]:
    """Load Wine Quality dataset.

    Returns:
        X_train, X_test, y_train, y_test, input_dim
    """
    filename = f'winequality-{wine_type}.csv'
    df = pd.read_csv(data_path / filename, sep=';')

    X = df.drop(columns=['quality']).values.astype(np.float32)
    y = df['quality'].values.astype(np.int64)

    # Wine quality ranges 3-9, remap to 0-indexed
    y = y - y.min()

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Standardize
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train).astype(np.float32)
    X_test = scaler.transform(X_test).astype(np.float32)

    num_classes = len(np.unique(y))
    print(f'[WINE-{wine_type.upper()}] Train: {X_train.shape}, Test: {X_test.
  ↪shape}, Classes: {num_classes}')
    return X_train, X_test, y_train, y_test, X_train.shape[1]
```

```
# ============================================================================
# IMAGE DATA LOADERS (use SimpleEmbedder / ResNet18)
# ============================================================================

def get_mnist_loaders(
    data_path: Path = DATA_PATH / 'MNIST',
    batch_size: int = 256
) -> tuple[DataLoader, DataLoader, int]:
    """Load MNIST dataset - NATIVE 32x32 (padded from 28x28), NO resize!"""
    transform = transforms.Compose([
        transforms.Pad(2),  # 28x28 -> 32x32
        transforms.Grayscale(num_output_channels=3),  # SmallImageEmbedder␣
↪expects 3 channels
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    train_ds = datasets.MNIST(root=str(data_path.parent), train=True,␣
↪download=True, transform=transform)
    test_ds = datasets.MNIST(root=str(data_path.parent), train=False,␣
↪download=True, transform=transform)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,␣
↪num_workers=4, pin_memory=True)
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
↪num_workers=4, pin_memory=True)

    print(f'[MNIST] Train: {len(train_ds)}, Test: {len(test_ds)}, Classes: 10␣
↪(32x32 native)')
    return train_loader, test_loader, 10


def get_cifar10_loaders(
    data_path: Path = DATA_PATH,
    batch_size: int = 256
) -> tuple[DataLoader, DataLoader, int]:
    """Load CIFAR-10 dataset - NATIVE 32x32, NO resize!"""
    normalize = transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.
↪261))

    train_transform = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize
    ])
    test_transform = transforms.Compose([
        transforms.ToTensor(),
        normalize
    ])

    train_ds = datasets.CIFAR10(root=str(data_path), train=True, download=True,␣
  ↪transform=train_transform)
    test_ds = datasets.CIFAR10(root=str(data_path), train=False, download=True,␣
  ↪transform=test_transform)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,␣
  ↪num_workers=4, pin_memory=True)
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
  ↪num_workers=4, pin_memory=True)

    print(f'[CIFAR10] Train: {len(train_ds)}, Test: {len(test_ds)}, Classes: 10␣
  ↪(32x32 native)')
    return train_loader, test_loader, 10


def get_cifar100_loaders(
    data_path: Path = DATA_PATH,
    batch_size: int = 256
) -> tuple[DataLoader, DataLoader, int]:
    """Load CIFAR-100 dataset - NATIVE 32x32, NO resize!"""
    normalize = transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565,␣
  ↪0.2761))

    train_transform = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize
    ])
    test_transform = transforms.Compose([
        transforms.ToTensor(),
        normalize
    ])

    train_ds = datasets.CIFAR100(root=str(data_path), train=True,␣
  ↪download=True, transform=train_transform)
    test_ds = datasets.CIFAR100(root=str(data_path), train=False,␣
  ↪download=True, transform=test_transform)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,␣
  ↪num_workers=4, pin_memory=True)
```

```
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
↪num_workers=4, pin_memory=True)

    print(f'[CIFAR100] Train: {len(train_ds)}, Test: {len(test_ds)}, Classes:␣
↪100 (32x32 native)')
    return train_loader, test_loader, 100


print('Local data loaders ready!')
print(f'DATA_PATH: {DATA_PATH}')
```

```
Local data loaders ready!
DATA_PATH: /Users/taher/Projects/attn_knn_repo/data
```

[52]:
```
# Run experiments on ALL your local datasets
# This generates comprehensive results for your pre-proposal

set_seed(42)
ALL_RESULTS = {}

# =============================================================================
# TABULAR EXPERIMENTS (Adult, Iris, Wine)
# =============================================================================

def run_tabular_experiment(
    name: str,
    X_train: np.ndarray,
    X_test: np.ndarray,
    y_train: np.ndarray,
    y_test: np.ndarray,
    input_dim: int,
    embed_dim: int = 64,
    k: int = 5
) -> dict:
    """Run uniform kNN and Attn-KNN on tabular data - VECTORIZED."""
    print(f'\n{"="*60}')
    print(f'Running {name} experiment (k={k}) on {device}')
    print(f'{"="*60}')

    # Create loaders
    train_loader = DataLoader(
        TensorDataset(torch.tensor(X_train), torch.tensor(y_train)),
        batch_size=256, shuffle=False
    )
    test_loader = DataLoader(
        TensorDataset(torch.tensor(X_test), torch.tensor(y_test)),
        batch_size=256, shuffle=False
    )

    # Embedder
    embedder = MLPEmbedder(input_dim=input_dim, embed_dim=embed_dim).to(device)
    embedder.eval()

    # Build memory
    mem_embs_list, mem_labels_list = [], []
    with torch.no_grad():
        for xb, yb in train_loader:
            xb = xb.to(device)
            z = embedder(xb)
            mem_embs_list.append(z.cpu().numpy().astype('float32'))
            mem_labels_list.append(yb.numpy().astype('int64'))
    mem_embs = np.vstack(mem_embs_list)
    mem_labels = np.concatenate(mem_labels_list)

    # FAISS index
    index = build_faiss_index(mem_embs, index_type='Flat', metric='L2')
    num_classes = int(mem_labels.max()) + 1

    # Evaluate uniform kNN - VECTORIZED
    def eval_uniform():
        all_preds, all_labels, all_probs = [], [], []
        with torch.no_grad():
            for xb, yb in test_loader:
                xb = xb.to(device)
                zq = embedder(xb)
                q_np = zq.cpu().numpy().astype('float32')
                D, I = index.search(q_np, k)
                neigh_labels_t = torch.from_numpy(mem_labels[I]).long().
↪to(device)
                # VECTORIZED aggregation
                probs = aggregate_neighbor_probs_uniform(neigh_labels_t,␣
↪num_classes, k)
                all_preds.append(probs.argmax(dim=1).cpu().numpy())
                all_labels.append(yb.numpy())
                all_probs.append(probs.cpu().numpy())
        y_pred = np.concatenate(all_preds)
        y_true = np.concatenate(all_labels)
        probs = np.vstack(all_probs)
        return {
            'accuracy': float(accuracy_score(y_true, y_pred)),
            'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
            'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
            'ece': float(ece_score(probs, y_true))
        }
    # Evaluate Attn-KNN - VECTORIZED
    def eval_attn():
        attn = AttnKNNHead(embed_dim=embed_dim, proj_dim=32).to(device)
        attn.eval()
        all_preds, all_labels, all_probs = [], [], []
        with torch.no_grad():
            for xb, yb in test_loader:
                xb = xb.to(device)
                zq = embedder(xb)
                q_np = zq.cpu().numpy().astype('float32')
                D, I = index.search(q_np, k)
                neighbor_embs = mem_embs[I]
                neighbor_embs_t = torch.from_numpy(neighbor_embs).to(device)
                neigh_labels_t = torch.from_numpy(mem_labels[I]).long().
 ↪to(device)
                weights = attn(zq, neighbor_embs_t)
                # VECTORIZED aggregation
                probs = aggregate_neighbor_probs_weighted(neigh_labels_t,␣
 ↪weights, num_classes)
                all_preds.append(probs.argmax(dim=1).cpu().numpy())
                all_labels.append(yb.cpu().numpy())
                all_probs.append(probs.cpu().numpy())
        y_pred = np.concatenate(all_preds)
        y_true = np.concatenate(all_labels)
        probs = np.vstack(all_probs)
        return {
            'accuracy': float(accuracy_score(y_true, y_pred)),
            'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
            'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
            'ece': float(ece_score(probs, y_true))
        }

    uniform_metrics = eval_uniform()
    attn_metrics = eval_attn()

    print(f'{name} Uniform kNN: {uniform_metrics}')
    print(f'{name} Attn-KNN:    {attn_metrics}')

    return {'uniform': uniform_metrics, 'attn': attn_metrics}


# Run tabular experiments
print('\n' + '='*80)
print('TABULAR EXPERIMENTS')
print('='*80)
```

```
# Iris
X_train, X_test, y_train, y_test, input_dim = load_iris_dataset()
ALL_RESULTS['Iris'] = run_tabular_experiment('Iris', X_train, X_test, y_train,␣
 ↪y_test, input_dim, k=5)

# Wine (Red)
X_train, X_test, y_train, y_test, input_dim = load_wine_dataset(wine_type='red')
ALL_RESULTS['Wine-Red'] = run_tabular_experiment('Wine-Red', X_train, X_test,␣
 ↪y_train, y_test, input_dim, k=5)

# Adult
X_train, X_test, y_train, y_test, input_dim = load_adult_dataset()
ALL_RESULTS['Adult'] = run_tabular_experiment('Adult', X_train, X_test,␣
 ↪y_train, y_test, input_dim, k=10)

print('\nTabular experiments complete!')
```

```
================================================================================
TABULAR EXPERIMENTS
================================================================================
[IRIS] Train: (120, 4), Test: (30, 4), Classes: 3

============================================================
Running Iris experiment (k=5) on mps
============================================================
Iris Uniform kNN: {'accuracy': 0.8666666666666667, 'f1_macro':
0.8666666666666667, 'nll': 0.1519246545382899, 'ece': 0.10666666825612385}
Iris Attn-KNN:    {'accuracy': 0.8666666666666667, 'f1_macro':
0.8666666666666667, 'nll': 0.14902813665203815, 'ece': 0.105183074871699}
[WINE-RED] Train: (1279, 11), Test: (320, 11), Classes: 6

============================================================
Running Wine-Red experiment (k=5) on mps
============================================================
Wine-Red Uniform kNN: {'accuracy': 0.55, 'f1_macro': 0.2916268514094601, 'nll':
2.6157889991325973, 'ece': 0.13375008124858143}
Wine-Red Attn-KNN:    {'accuracy': 0.525, 'f1_macro': 0.28073049821644736,
'nll': 2.6172374517591455, 'ece': 0.1537281825207174}
[ADULT] Train: (30162, 104), Test: (15060, 104), Classes: 2

============================================================
Running Adult experiment (k=10) on mps
============================================================
Adult Uniform kNN: {'accuracy': 0.8240371845949536, 'f1_macro':
0.7391832853534446, 'nll': 0.5502130199484698, 'ece': 0.016394416463485918}
Adult Attn-KNN:    {'accuracy': 0.8198539176626826, 'f1_macro':
```

```
0.7428007863522104, 'nll': 0.5503866006597492, 'ece': 0.024738147185934832}

Tabular experiments complete!
```

[53]:
```
# IMAGE EXPERIMENTS (MNIST, CIFAR-10, CIFAR-100) - OPTIMIZED
# Uses SmallImageEmbedder for 32x32 images (NO resize!) + VECTORIZED aggregation

def run_image_experiment(
    name: str,
    train_loader: DataLoader,
    test_loader: DataLoader,
    num_classes: int,
    embed_dim: int = 128,
    k: int = 10
) -> dict:
    """Run uniform kNN and Attn-KNN on image data - OPTIMIZED."""
    print(f'\n{"="*60}')
    print(f'Running {name} experiment (k={k}) on {device}')
    print(f'{"="*60}')

    # Use SmallImageEmbedder for 32x32 images (MUCH faster than resizing to␣
 ↪224x224)
    embedder = SmallImageEmbedder(embed_dim=embed_dim, in_channels=3,␣
 ↪pretrained=False).to(device)
    embedder.eval()

    # Build memory
    print(f'[{name}] Building memory bank...')
    mem_embs_list, mem_labels_list = [], []
    with torch.no_grad():
        for xb, yb in tqdm(train_loader, desc='Building memory'):
            xb = xb.to(device)
            z = embedder(xb)
            mem_embs_list.append(z.cpu().numpy().astype('float32'))
            mem_labels_list.append(yb.numpy().astype('int64'))
    mem_embs = np.vstack(mem_embs_list)
    mem_labels = np.concatenate(mem_labels_list)
    print(f'[{name}] Memory: {mem_embs.shape}')

    # FAISS index
    index = build_faiss_index(mem_embs, index_type='Flat', metric='L2')

    # Evaluate uniform kNN - VECTORIZED
    def eval_uniform():
        all_preds, all_labels, all_probs = [], [], []
        with torch.no_grad():
            for xb, yb in tqdm(test_loader, desc='Eval Uniform'):
                xb = xb.to(device)
                zq = embedder(xb)
                q_np = zq.cpu().numpy().astype('float32')
                D, I = index.search(q_np, k)
                neigh_labels_t = torch.from_numpy(mem_labels[I]).long().
↪to(device)
                # VECTORIZED aggregation
                probs = aggregate_neighbor_probs_uniform(neigh_labels_t,␣
↪num_classes, k)
                all_preds.append(probs.argmax(dim=1).cpu().numpy())
                all_labels.append(yb.numpy())
                all_probs.append(probs.cpu().numpy())
        y_pred = np.concatenate(all_preds)
        y_true = np.concatenate(all_labels)
        probs = np.vstack(all_probs)
        return {
            'accuracy': float(accuracy_score(y_true, y_pred)),
            'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
            'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
            'ece': float(ece_score(probs, y_true))
        }

    # Evaluate Attn-KNN - VECTORIZED
    def eval_attn():
        attn = AttnKNNHead(embed_dim=embed_dim, proj_dim=64).to(device)
        attn.eval()
        all_preds, all_labels, all_probs = [], [], []
        with torch.no_grad():
            for xb, yb in tqdm(test_loader, desc='Eval Attn-KNN'):
                xb = xb.to(device)
                zq = embedder(xb)
                q_np = zq.cpu().numpy().astype('float32')
                D, I = index.search(q_np, k)
                neighbor_embs = mem_embs[I]
                neighbor_embs_t = torch.from_numpy(neighbor_embs).to(device)
                neigh_labels_t = torch.from_numpy(mem_labels[I]).long().
↪to(device)
                weights = attn(zq, neighbor_embs_t)
                # VECTORIZED aggregation
                probs = aggregate_neighbor_probs_weighted(neigh_labels_t,␣
↪weights, num_classes)
                all_preds.append(probs.argmax(dim=1).cpu().numpy())
                all_labels.append(yb.cpu().numpy())
                all_probs.append(probs.cpu().numpy())
        y_pred = np.concatenate(all_preds)
        y_true = np.concatenate(all_labels)
        probs = np.vstack(all_probs)
        return {
            'accuracy': float(accuracy_score(y_true, y_pred)),
            'f1_macro': float(f1_score(y_true, y_pred, average='macro')),
            'nll': float(log_loss(y_true, np.clip(probs, 1e-9, 1.0))),
            'ece': float(ece_score(probs, y_true))
        }

    uniform_metrics = eval_uniform()
    attn_metrics = eval_attn()

    print(f'{name} Uniform kNN: {uniform_metrics}')
    print(f'{name} Attn-KNN:    {attn_metrics}')

    return {'uniform': uniform_metrics, 'attn': attn_metrics}


# Run image experiments
print('\n' + '='*80)
print('IMAGE EXPERIMENTS (this may take a while)')
print('='*80)

# MNIST (smaller, runs faster)
train_loader, test_loader, num_classes = get_mnist_loaders(batch_size=128)
ALL_RESULTS['MNIST'] = run_image_experiment('MNIST', train_loader, test_loader,␣
 ↪num_classes, k=10)

# CIFAR-10
train_loader, test_loader, num_classes = get_cifar10_loaders(batch_size=64)
ALL_RESULTS['CIFAR-10'] = run_image_experiment('CIFAR-10', train_loader,␣
 ↪test_loader, num_classes, k=10)

# CIFAR-100 (100 classes, harder)
train_loader, test_loader, num_classes = get_cifar100_loaders(batch_size=64)
ALL_RESULTS['CIFAR-100'] = run_image_experiment('CIFAR-100', train_loader,␣
 ↪test_loader, num_classes, k=10)

print('\nImage experiments complete!')
```

```
================================================================================
IMAGE EXPERIMENTS (this may take a while)
================================================================================

100%|      | 9.91M/9.91M [00:13<00:00, 713kB/s]
100%|      | 28.9k/28.9k [00:02<00:00, 14.1kB/s]
100%|      | 1.65M/1.65M [00:12<00:00, 130kB/s]
100%|      | 4.54k/4.54k [00:00<00:00, 3.21MB/s]
```

```
[MNIST] Train: 60000, Test: 10000, Classes: 10 (32x32 native)

============================================================
Running MNIST experiment (k=10) on mps
============================================================
[MNIST] Building memory bank…

Building memory:   0%|          | 0/469 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Building memory: 100%|     | 469/469 [00:07<00:00, 62.19it/s]

[MNIST] Memory: (60000, 128)

Eval Uniform: 100%|     | 79/79 [00:02<00:00, 27.94it/s]
Eval Attn-KNN:   0%|          | 0/79 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Eval Attn-KNN: 100%|     | 79/79 [00:02<00:00, 27.23it/s]

MNIST Uniform kNN: {'accuracy': 0.9537, 'f1_macro': 0.9531739473165077, 'nll':
0.23207003786015107, 'ece': 0.024270000825822362}
MNIST Attn-KNN:    {'accuracy': 0.9526, 'f1_macro': 0.9520236987140829, 'nll':
0.23206299508480305, 'ece': 0.020811673849821116}
[CIFAR10] Train: 50000, Test: 10000, Classes: 10 (32x32 native)

============================================================
Running CIFAR-10 experiment (k=10) on mps
============================================================
[CIFAR-10] Building memory bank…

Building memory:   0%|          | 0/782 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Building memory: 100%|     | 782/782 [00:32<00:00, 24.37it/s]

[CIFAR-10] Memory: (50000, 128)

Eval Uniform: 100%|     | 157/157 [00:28<00:00,  5.59it/s]
Eval Attn-KNN:   0%|          | 0/157 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Eval Attn-KNN: 100%|     | 157/157 [00:28<00:00,  5.59it/s]
```

```
CIFAR-10 Uniform kNN: {'accuracy': 0.3076, 'f1_macro': 0.30111670984454736,
'nll': 4.3305873108086965, 'ece': 0.09031000546216962}
CIFAR-10 Attn-KNN:    {'accuracy': 0.3081, 'f1_macro': 0.3000668579428926,
'nll': 4.3305915156868995, 'ece': 0.09027209089398384}
[CIFAR100] Train: 50000, Test: 10000, Classes: 100 (32x32 native)

============================================================
Running CIFAR-100 experiment (k=10) on mps
============================================================
[CIFAR-100] Building memory bank…

Building memory:   0%|          | 0/782 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Building memory: 100%|     | 782/782 [00:32<00:00, 24.34it/s]

[CIFAR-100] Memory: (50000, 128)

Eval Uniform: 100%|     | 157/157 [00:27<00:00,  5.62it/s]
Eval Attn-KNN:   0%|          | 0/157 [00:00<?,
?it/s]/Users/taher/Projects/attn_knn_repo/.venv/lib/python3.14/site-
packages/torch/utils/data/dataloader.py:692: UserWarning: 'pin_memory' argument
is set as true but not supported on MPS now, device pinned memory won't be used.
  warnings.warn(warn_msg)
Eval Attn-KNN: 100%|     | 157/157 [00:27<00:00,  5.61it/s]

CIFAR-100 Uniform kNN: {'accuracy': 0.1011, 'f1_macro': 0.08740133577835772,
'nll': 11.375288522467509, 'ece': 0.11406000821739437}
CIFAR-100 Attn-KNN:    {'accuracy': 0.0977, 'f1_macro': 0.08401176755311486,
'nll': 11.375306575732736, 'ece': 0.11740229175835847}

Image experiments complete!
```