Clustering with Neural Networks using Hugging Face Datasets

Assignment Prepared By
Moin Mostakim

April 25, 2025

1. Introduction

This document outlines the process and requirements to develop a neural network model for clustering using open datasets available on Hugging Face. Clustering is an unsupervised task where the model identifies intrinsic groupings in the data without labeled outputs.

2. Objective

To design and implement a neural network capable of performing clustering on a selected Hugging Face dataset, with evaluation on metrics such as Silhouette Score, Davies-Bouldin Index, or cluster separation.

3. Requirements

3.1 Software and Tools

- Python (3.8+)
- PyTorch or TensorFlow
- Hugging Face `datasets` library
- Scikit-learn
- Matplotlib / Seaborn for visualization
- CUDA-enabled GPU (optional, for training acceleration)

3.2 Libraries

```
pip install torch torchvision datasets transformers scikit-learn
    matplotlib
```

3.3 Dataset

Choose a dataset from Hugging Face: https://huggingface.co/datasets

Examples:

- `ag_news` — for clustering news articles
- `glue/sst2` — for sentence semantic clustering
- `mnist` — for image-based clustering

4. Neural Network Design

Clustering with neural networks involves encoding data into a compact latent space where similar instances are closer together. This section describes the model architecture depending on the input modality.

4.1 Input Preprocessing

- **Text Data:** Use pre-trained transformers (e.g., BERT, RoBERTa) or sentence embeddings (e.g., Sentence-BERT) to convert text into dense vector representations.
- **Image Data:** Use convolutional encoders or pre-trained feature extractors (e.g., ResNet, VGG) to get high-dimensional embeddings.
- **Tabular Data:** Normalize features and use dense feed-forward layers.

4.2 Architecture Overview

We design an encoder network to project the input into a lower-dimensional latent space suitable for clustering. The core architectures include:

4.2.1 Autoencoder-based Clustering

- **Encoder:** Several dense or convolutional layers reducing dimensionality.
- **Latent Space:** Bottleneck layer representing data embedding.
- **Decoder:** Mirror of the encoder for reconstruction (used only during training).
- **Loss:** Combination of reconstruction loss and clustering-oriented loss.

$$L = L_{\text{reconstruction}} + \lambda \cdot L_{\text{clustering}}$$

4.2.2 Deep Embedding Clustering (DEC)

- Uses a deep autoencoder to learn representations.
- Learns a probability distribution over clusters using a Student's t-distribution.
- Loss is KL divergence between soft cluster assignments and auxiliary target distribution:

$$L = KL(P \,||\, Q) = \sum_i \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right)$$

Where:

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_k (1 + \|z_i - \mu_k\|^2/\alpha)^{-\frac{\alpha+1}{2}}}$$
$$p_{ij} = \frac{q_{ij}^2/\sum_i q_{ij}}{\sum_k q_{ik}^2/\sum_i q_{ik}}$$

4.2.3 Siamese Network for Contrastive Clustering

- Learns whether two samples are similar or dissimilar.
- Each branch processes a different input and compares their embeddings.
- **Loss:** Contrastive loss:

$$L = y \cdot \|z_1 - z_2\|^2 + (1 - y) \cdot \max(0, m - \|z_1 - z_2\|)^2$$

4.2.4 Triplet Network for Semantic Similarity

- Trained on triplets: anchor, positive, and negative examples.
- Embedding of anchor should be closer to positive than to negative.
- **Loss:** Triplet loss:

$$L = \max\left(0, \|f(x_a) - f(x_p)\|^2 - \|f(x_a) - f(x_n)\|^2 + \alpha\right)$$

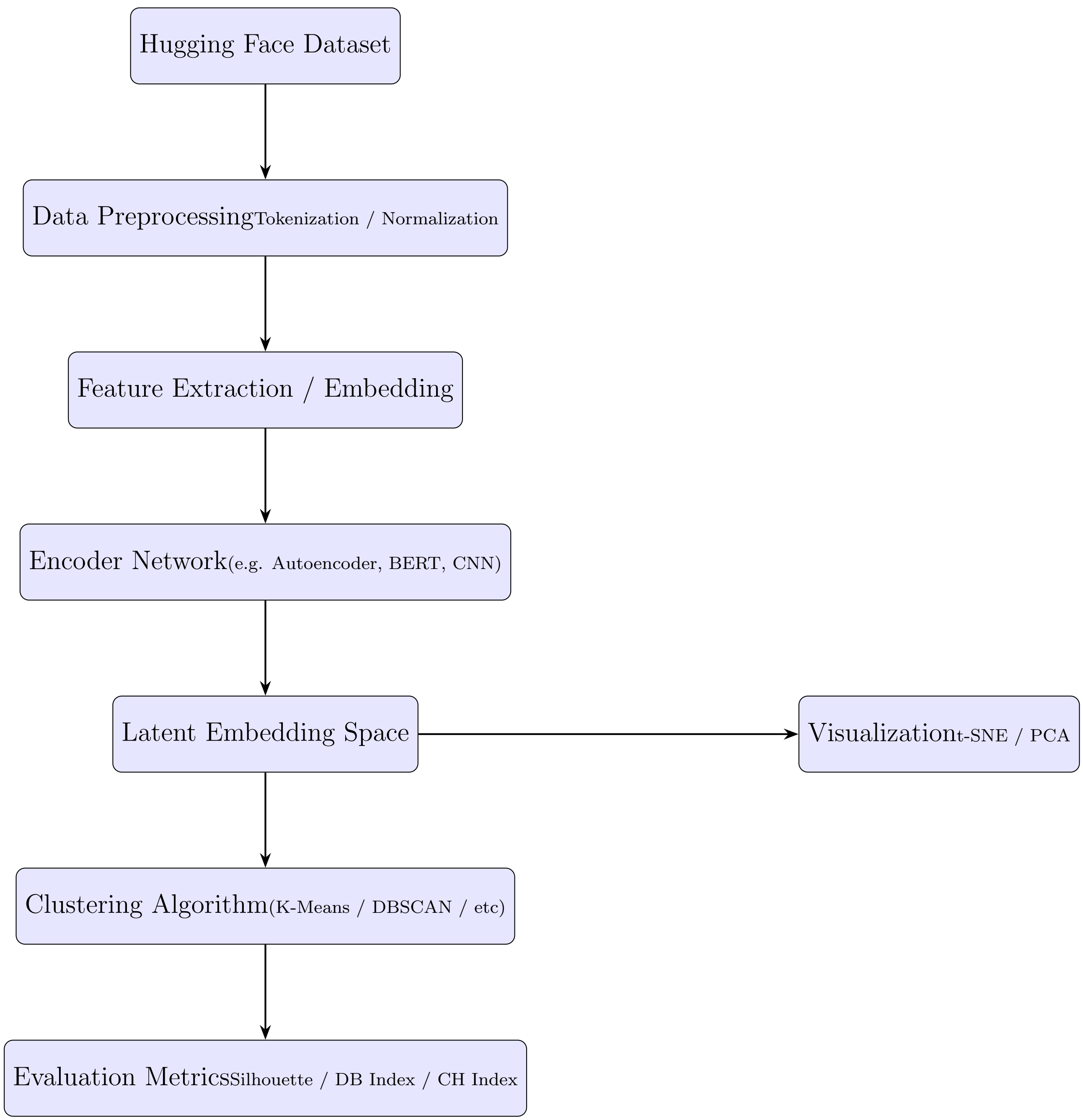
4.3 Embedding Size and Clustering

- The output of the encoder is a fixed-length embedding vector (e.g., 64 or 128 dimensions).
- Embeddings are passed to clustering algorithms such as K-Means or DBSCAN.

4.4 Training Strategy

- **Phase 1:** Pre-train encoder (and decoder if using autoencoder) with reconstruction or contrastive loss.
- **Phase 2:** Fine-tune embeddings with clustering loss or perform clustering on frozen embeddings.

1 Visualization



5. Clustering Algorithm

Apply a clustering algorithm to latent embeddings:

- K-Means
- DBSCAN
- Hierarchical Clustering

6. Evaluation Metrics

- Silhouette Score
- Davies-Bouldin Index
- Calinski-Harabasz Index
- Visual inspection via t-SNE or PCA

7. Expected Output

- Clustered representations
- Visualization of cluster separations
- Quantitative metric scores

2 Sample Code Introduction

This document outlines the basic structure of a neural network built using PyTorch. The example provided is for a simple Multi-Layer Perceptron (MLP) applied to the MNIST dataset.

3 Import Required Libraries

We begin by importing necessary libraries from PyTorch and Torchvision.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import torchvision.datasets as datasets
```

4 Define the Neural Network

We define a basic MLP with one hidden layer using `nn.Sequential` for clarity and modularity.

```
class MyMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyMLP, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        return self.net(x)
```

5 Set Hyperparameters

Set the size of layers, learning rate, batch size, and number of training epochs.

```
input_size = 784      # 28x28 images flattened
hidden_size = 128
output_size = 10      # Number of classes in MNIST
learning_rate = 0.001
batch_size = 64
epochs = 5
```

6 Load Dataset and Create Dataloaders

Here we use the MNIST dataset and apply transformations such as normalization.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=
    transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=
    transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
    True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=
    False)
```

7 Initialize Model, Loss Function, and Optimizer

Prepare the model for training and evaluation on GPU if available.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MyMLP(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

8 Training Loop

Perform training for the specified number of epochs.

```
for epoch in range(epochs):
    model.train()
    for batch_idx, (data, targets) in enumerate(train_loader):
        data = data.view(data.size(0), -1).to(device)
        targets = targets.to(device)

        scores = model(data)
        loss = criterion(scores, targets)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

9 Evaluation

Evaluate the trained model on the test dataset.

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, targets in test_loader:
        data = data.view(data.size(0), -1).to(device)
        targets = targets.to(device)

        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        total += targets.size(0)
        correct += (predicted == targets).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

References

- Hugging Face Datasets: `https://huggingface.co/docs/datasets/index`
- PyTorch: `https://pytorch.org`