Track 4: Text-Guided Image Clustering

Ananya Sharma (2023MAS7117)
Harshit Joshi (2023MAS7141)

April 18, 2025

1 Introduction

In this report we have explored various approaches for clustering image data from the provided subset of Food-101 dataset. We have compared traditional computer vision features with deep and text-guided representations. We have also tried Fine-Tuning the deep learning models to adapt to the domain distribution which significantly improved the cluster quality.

2 Data Preparation

- As the given dataset is quite large, to make it computationally feasible, we've worked with a subset. We randomly sampled 10% of the images from each of the given classes. This resulted in a dataset of $N = 2100$ images (70 images per class, $C = 30$).
- We used a fixed random seed (`seed = 782`) for this sampling and for initializing clustering algorithms like K-Means to ensure our results are reproducible.

```
seed = 782
np.random.seed(seed)
torch.manual_seed(seed)

# For KMeans
KMeans(n_clusters=30, random_state=seed)
```

3 Non-Competitive Section

3.1 Classical Feature Extraction

For this section we've used traditional computer vision features to try to cluster the images.

Preprocessing

- All images $\mathbf{I}$ from the $N = 2100$ sample were first resized to 224×224 pixels using OpenCV's `cv2.resize`, resulting in an 8-bit integer image $\mathbf{I}' \in \mathbb{Z}^{224\times224\times3}$ with pixel values between 0 and 255.
- For features like Color Histograms, LBP, and HOG, we normalized the pixel values to the range $[0, 1]$ by converting to float and dividing by 255. This normalized image is $\tilde{\mathbf{I}} \in [0, 1]^{224\times224\times3}$.
- SIFT and Canny edge detection operated on the 8-bit image $\mathbf{I}'$ and its grayscale conversion respectively.
- HOG worked on the grayscale version of $\tilde{\mathbf{I}}$

- The conversions for the image at path `img_path` is

```
img = cv2.imread(image_path)
img_resized = cv2.resize(img, (224, 224))
img_normalized = img_resized.astype(np.float32) / 255.0
img_gray = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)
```

Features Extracted

- **SIFT with Spatial Pooling (512D):** Computed SIFT descriptors (128D) on $\mathbf{I}'$. Divided the image into 4 quadrants ($q \in \{\text{TL, TR, BL, BR}\}$) and averaged the descriptors $\mathcal{D}_q$ within each: $\boldsymbol{\mu}_q = \text{mean}(\mathcal{D}_q)$ (or $\mathbf{0}$ if no descriptors). The final feature vector was the concatenation:

$$\mathbf{f}_{\text{SIFT}} = [\boldsymbol{\mu}_{\text{TL}}^T, \boldsymbol{\mu}_{\text{TR}}^T, \boldsymbol{\mu}_{\text{BL}}^T, \boldsymbol{\mu}_{\text{BR}}^T]^T \in \mathbb{R}^{512}.$$

```
sift = cv2.SIFT_create()
kp, des = sift.detectAndCompute(img_resized, None)
h, w = img_resized.shape[:2]
quadrant_features = []
quadrants = [
    (0, w//2, 0, h//2),        # top-left
    (w//2, w, 0, h//2),        # top-right
    (0, w//2, h//2, h),        # bottom-left
    (w//2, w, h//2, h)         # bottom-right
]
if kp is not None and des is not None and len(des) > 0:
    for (x_start, x_end, y_start, y_end) in quadrants:
        descs = [des[i] for i, point in enumerate(kp)
                 if x_start <= point.pt[0] < x_end and y_start <= point
                                                     .pt[1] < y_end
                                                     ]
        if descs:
            quadrant_features.append(np.mean(descs, axis=0))
        else:
            quadrant_features.append(np.zeros(128))
    sift_feat = np.concatenate(quadrant_features)
else:
    sift_feat = np.zeros(512)
```

- **Histogram of Oriented Gradients (HOG, 324D):** Computed HOG on the grayscale version of $\tilde{\mathbf{I}}$ using `skimage.feature.hog` (‘pixels_per_cell=(56, 56)‘, ‘cells_per_block=(2, 2)‘, ‘orientations=9‘), resulting in $\mathbf{f}_{\text{HOG}} \in \mathbb{R}^{324}$.

```
gray_norm = cv2.cvtColor(img_normalized, cv2.COLOR_BGR2GRAY)
hog_feat, hog_image = hog(gray_norm,
                          pixels_per_cell=(56, 56),
                          cells_per_block=(2, 2),
                          orientations=9,
                          visualize=True,
                          feature_vector=True)
```

- **Color Histogram (96D):** Computed 32-bin histograms (h_B, h_G, h_R) for each channel of $\tilde{\mathbf{I}}$ (range $[0, 1]$) and concatenated them:

$$\mathbf{f}_{\text{Color}} = [h_B^T, h_G^T, h_R^T]^T \in \mathbb{R}^{96}.$$

```
hist_b = cv2.calcHist([img_normalized], [0], None, [32], [0, 1]).
                                                 flatten()
```

```
    hist_g = cv2.calcHist([img_normalized], [1], None, [32], [0, 1]).
                                              flatten()
    hist_r = cv2.calcHist([img_normalized], [2], None, [32], [0, 1]).
                                              flatten()
    color_hist = np.concatenate([hist_b, hist_g, hist_r])
```

- **Canny Edge Histogram (64D):** Applied Canny edge detection (`cv2.Canny`) to the grayscale $\mathbf{I}'$, obtaining a binary edge map $\mathbf{E}$. Divided $\mathbf{E}$ into an 8×8 grid C_{ij} and computed the normalized edge density per cell (by summing over all pixel values (0:non-edge, 255:edge) and dividing it by the total number of pixels in the cell and the maximum pixel value (255)):

$$(\mathbf{f}_{\text{Edge}})_{8i+j} = \frac{\sum_{(x,y)\in C_{ij}} \mathbf{E}_{xy}}{255 \cdot |C_{ij}|}, \quad \mathbf{f}_{\text{Edge}} \in \mathbb{R}^{64}.$$

```
    gray_for_edges = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray_for_edges, 100, 200)    #thresholds=100 and 200
    edge_hist = []
    grid_size = 8
    cell_width = edges.shape[1] // grid_size
    cell_height = edges.shape[0] // grid_size
    for i in range(grid_size):
        for j in range(grid_size):
            cell = edges[i*cell_height:(i+1)*cell_height, j*cell_width:(j+1
                                            )*cell_width]
            edge_hist.append(np.sum(cell) / (255.0 * cell.size))
    edge_hist = np.array(edge_hist)
```

- **Local Binary Patterns (LBP, 10D):** Computed uniform LBPs ($P = 8, R = 1$) on grayscale $\tilde{\mathbf{I}}$ using `skimage.feature.local_binary_pattern`. The normalized histogram of the 10 uniform patterns gave $\mathbf{f}_{\text{LBP}} \in \mathbb{R}^{10}$.

```
    lbp = local_binary_pattern(gray_norm, P=8, R=1, method='uniform')
    (lbp_hist, _) = np.histogram(lbp.ravel(),
                                 bins=np.arange(0, 8 + 3),
                                 range=(0, 8 + 2))
    lbp_hist = lbp_hist.astype("float")
    lbp_hist /= (lbp_hist.sum() + 1e-7)
```

- **Concatenated Classical Features (1006D):** Concatenated all the above features:

$$\mathbf{f}_{\text{concat}} = [\mathbf{f}_{\text{SIFT}}^T, \mathbf{f}_{\text{HOG}}^T, \mathbf{f}_{\text{Color}}^T, \mathbf{f}_{\text{Edge}}^T, \mathbf{f}_{\text{LBP}}^T]^T \in \mathbb{R}^{1006}.$$

```
    concat_feat = np.concatenate([sift_feat, hog_feat, color_hist,
                                  edge_hist, lbp_hist])
```

Time Taken

- Feature extraction process took about 3 minutes for the 2100 images.

Results

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the features extracted from all the 2100 images. The Adjusted Rand Index (ARI) scores are shown below:

Technique	K-Means ARI	DBSCAN ARI
SIFT (Spatial Pool)	0.0210	0.0
HOG	0.0145	0.0
Color Histogram	0.0076	0.0
Canny Edge Histogram	0.0198	0.0060
LBP	0.0060	0.0
Concatenated	0.0077	0.0

- **Inferences:**
  - The ARI scores for K-Means were very low (close to 0, which indicates performance similar to random chance), suggesting that these classical features did not provide a sufficiently discriminative representation for clustering the food categories.
  - DBSCAN performed very poorly with the default parameters (`eps=0.5`, `min_samples=10`), yielding ARI scores of 0 or near 0. This suggests these parameters were unsuitable for the feature distributions, possibly due to high dimensionality and varying cluster densities inherent in these classical features.

3.2 Deep Feature Extraction (ResNet-50)

Now we extract features using ResNet-50 pretrained on ImageNet dataset to utilize their learned hierarchical representations.

Preprocessing

- We used the standard ImageNet preprocessing pipeline via `torchvision.transforms`: resize to 256x256, center crop to 224x224, convert to a PyTorch tensor $\mathbf{I}''$, and normalize using ImageNet's mean $\boldsymbol{\mu} = [0.485, 0.456, 0.406]$ and standard deviation $\boldsymbol{\sigma} = [0.229, 0.224, 0.225]$ to $\mathbf{I}_{\text{norm}}$ using:

$$\mathbf{I}_{\text{norm}} = \frac{\mathbf{I}'' - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

```
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Features Extracted

- We took the output from the penultimate layer (final average pooling layer, before the original classification layer) by replacing the final layer with an identity map resulting in $\mathbf{f}_{\text{ResNet50}} \in \mathbb{R}^{2048}$.

```
from torchvision import models

model = models.resnet50(pretrained=True)
model.fc = torch.nn.Identity()

#computing the features after preprocessing
img_tensor = preprocess(img).unsqueeze(0).to(device)
features = model(img_tensor)
```

Time Taken

- Feature extraction for ResNet-50 took about 15 minutes for 2100 images.

Results

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the extracted ResNet-50 features.

Technique	K-Means ARI	DBSCAN ARI
ResNet-50 (Pretrained)	0.2285	0.0

- **Inferences:**
  - K-Means clustering using pretrained ResNet-50 features yielded a significantly better ARI score (0.2285) compared to any of the classical features, demonstrating the power of learned representations from deep models.
  - DBSCAN again failed with the default parameters on these high-dimensional deep features, resulting in an ARI of 0.

3.3 Text-Guided Feature Extraction (BLIP Captions)

Here we tested converting images to text captions using BLIP and then clustering the textual embeddings obtained from these captions using SBERT.

Preprocessing

- Image preprocessing was handled internally by `transformers` BLIP model pipeline.

```python
from transformers import BlipProcessor
processor = BlipProcessor.from_pretrained("Salesforce/blip-image-
                                          captioning-base")
```

Features Extracted

- **Caption Generation:** We used the BLIP base model (`Salesforce/blip-image-captioning-base`) to generate a caption c_i for each image $\mathbf{I}_i$.
- **Text Embedding:** We encoded the captions $\{c_i\}$ using Sentence-BERT (`all-mpnet-base-v2`) into vectors $\mathbf{f}_{\text{BLIP+SBERT}} \in \mathbb{R}^{768}$.

```python
from transformers import BlipForConditionalGeneration
model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-
                                        image-captioning-base").to(
                                        device)

image = Image.open(image_path).convert("RGB")
inputs = processor(image, return_tensors="pt").to(device)
output_ids = model.generate(**inputs)
caption = processor.decode(output_ids[0], skip_special_tokens=True)

# once we have all the captions
embeddings = sbert_model.encode(captions, show_progress_bar=True)
```

NOTE: The quality of generated captions varied. Some were descriptive (e.g., "a plate with a hamburger and fries on it"), while others were vague ("a plate of food on a table") or contained repetitions/artifacts ("two taco ta ta ta ta...").

Time Taken

- Caption generation took approx. 10 mins for 2100 images. SBERT embedding time was negligible in comparison.

Results

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the SBERT embeddings of the BLIP-generated captions.

Technique	K-Means ARI	DBSCAN ARI
BLIP Captions + SBERT	0.2115	0.0140

- **Inferences:**
  - K-Means clustering based on BLIP caption embeddings performed comparably to using pretrained ResNet-50 features (ARI 0.21 vs 0.23), suggesting that text descriptions can capture semantically relevant information for clustering, despite some noise in the caption generation. Feature extraction was also faster.
  - DBSCAN showed only marginal improvement compared to ResNet-50 features (ARI 0.0140 vs 0), still yielding a very low ARI. The default parameters remained largely unsuitable.

4 Competitive Section

4.1 Baseline Deep Features (EfficientNet-B0)

As an alternative to ResNet-50, we extracted baseline features using EfficientNet-B0 which is another CNN pretrained on ImageNet, but much more lightweight.

Preprocessing

- The preprocessing was identical to that used for ResNet-50.

Features Extracted

- We used EfficientNet-B0 pretrained on ImageNet.
- The extracted features were the output of the `model.avgpool` layer, $\mathbf{f}_{\text{EffNetB0}} \in \mathbb{R}^{1280}$.

Time Taken

- Feature extraction took about 6 minutes for 2100 images.

Results and inferences

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the pretrained EfficientNet-B0 features.

Technique	K-Means ARI	DBSCAN ARI
EfficientNet-B0 (Pretrained)	0.2791	0.0

- **Inferences:**
  - EfficientNet-B0 features performed slightly better than ResNet-50 features for K-Means clustering (ARI 0.28 vs 0.23) and were faster to extract. This suggests EfficientNet-B0 provides a slightly more effective or efficient baseline visual representation for this dataset.
  - As with ResNet-50, DBSCAN failed with the default parameters (ARI = 0), indicating these parameters are not suitable for the distribution of these pretrained deep features either.

4.2 Text-Guided Features (VILT VQA)

Now we explored using Visual Question Answering (VQA) as an alternative text generation method for feature extraction.

Preprocessing

- Image preprocessing is handled internally by the VILT model pipeline.

```
from transformers import ViltProcessor
processor = ViltProcessor.from_pretrained("dandelin/vilt-b32-finetuned-
                                           vqa")
```

Tokens Extracted

- **VQA Generation:** We used the VILT model (`dandelin/vilt-b32-finetuned-vqa`) to answer the question `"Which dish is in this image ?"` for each image $\mathbf{I}_i$, yielding an answer a_i.
- **Text Embedding:** We encoded answers $\{a_i\}$ using SBERT (‘all-mpnet-base-v2‘) into $\mathbf{f}_{\text{VILT+SBERT}} \in \mathbb{R}^{768}$.

```
from transformers import ViltForQuestionAnswering
model = ViltForQuestionAnswering.from_pretrained("dandelin/vilt-b32-
                                           finetuned-vqa").to(device)

image = Image.open(image_path).convert("RGB")
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# question = "Which dish is in this image?"
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
inputs = processor(image, question, return_tensors="pt").to(device)
outputs = model(**inputs)
answer = processor.tokenizer.decode(outputs.logits.argmax(-1)[0])

# once we had all the answers
embeddings = sbert_model.encode(answers, show_progress_bar=True)
```

NOTE: The answers were very cryptic and contained special tokens (e.g., “[unused295]”, “Ëď”, “ÑŽ”, “ŬĎ”), possibly due to the model’s original training or limitations in generating specific dish names in response to our simple question.
NOTE: We wanted the question to be more specific; however, the limit was of 40 words, which made it difficult to specify all of the classes while being fair to all.

Time Taken

- VQA generation took approx. 25 mins for 2100 images. SBERT embedding time was negligible.

Results

- We clustered the SBERT embeddings of the VILT VQA answers using K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`).

Technique	K-Means ARI	DBSCAN ARI
VILT VQA + SBERT	0.1699	0.0426

- **Inferences:**

  - The K-Means ARI for VQA-based features was lower than both BLIP captioning (0.21) and the pretrained visual models (0.23-0.28). Combined with the longer computation time this approach seems less promising than BLIP captioning for generating clusterable text features.
  - DBSCAN performed slightly better here than on most other features (ARI 0.04), but the score was still very low to deem it meaningful.

4.3 Fine-tuning Deep Features (EfficientNet-B0)

We fine-tuned EfficientNet-B0 on our specific food dataset subset to investigate if domain adaptation improves visual feature quality for clustering.

Preprocessing

- We split the 2100 images into balanced train/test sets (1050 each, 35 per class) using the same random seed (`seed = 782`).
- Preprocessing for training and testing used the same ImageNet pipeline as the pretrained model (resize 256, crop 224, ToTensor, normalize), for training we also include `RandomHorizontalFlip`

```
train_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

- We replaced the final classifier of a pretrained EfficientNet-B0 with a new linear layer matching our 30 food classes.

```
model = models.efficientnet_b0(pretrained=True)
in_features = model.classifier[1].in_features
model.classifier[1] = nn.Linear(in_features, 30)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
EPOCHS = 15
```

Features Extracted

- The base model is EfficientNet-B0. The final layer is replaced for fine-tuning on 30 classes.
- After fine-tuning, features $\mathbf{f}_{\text{EffNetB0-FT}} \in \mathbb{R}^{1280}$ were extracted from the output of the `model.avgpool` layer of the fine-tuned model.

Training and time taken

- **Training:** We trained the adapted model for 15 epochs using 'CrossEntropyLoss' and the Adam optimizer (lr=1e-4). The model achieved a test accuracy of approximately 71%. Training took approx. 6 mins for the 1050 **train** images.

- **Feature Extraction:** We used the fine-tuned model (loading the saved state) to extract features from the 1050 **test** images. Feature extraction took approx. 1 min.

Results and inferences

- We clustered the features extracted from the 1050 test images using K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`).

Technique	K-Means ARI	DBSCAN ARI
EfficientNet-B0 (Fine-tuned)	0.4540	0.0

- **Inferences:**
  - Fine-tuning EfficientNet-B0 on the target domain significantly improved K-Means clustering performance (ARI 0.45) compared to the pretrained version (ARI 0.28). This clearly demonstrates that domain adaptation helps in learning more discriminative visual features tailored to the specific classes in our dataset.
  - Despite the improved feature quality evident from the K-Means result, DBSCAN still failed completely (ARI $= 0$) with the default parameters, further emphasizing its parameter sensitivity.

4.4 Fine-tuning Text-Guided Features (BLIP Captions)

We explored if fine-tuning the BLIP image captioning model on our dataset could enhance the quality of text-based features for clustering.

Preprocessing

- We used the same 50/50 train/test split (1050 images each) as for fine-tuning EfficientNet-B0 (`seed = 782`).
- For training, we paired images with simple template captions: "This is an image of [class_name]" (replaced all '_' by ' ').

```
caption = "This is an image of " + class_name.replace(’_’, ’ ’)
```

- Image preprocessing for BLIP fine-tuning is handled by its specific processor.

Feature Extracted

- **Caption Generation (Fine-tuned):** We generated captions c_i' for the 1050 **test** images using the fine-tuned BLIP model (‘Salesforce/blip-image-captioning-base‘ adapted).
- **Text Embedding:** We encoded these new captions $\{c_i'\}$ using the same SBERT model (‘all-mpnet-base-v2‘) into vectors $\mathbf{f}_{\text{BLIP-FT+SBERT}} \in \mathbb{R}^{768}$.

Training and time taken

- **Training:** We fine-tuned the ‘Salesforce/blip-image-captioning-base‘ model for 5 epochs using the Hugging Face ‘Trainer‘ library (configuration: batch size 4, gradient accumulation steps 2, learning rate 5×10^{-5}, mixed-precision fp16). Training took approx. 20 mins for the 1050 **train** images.

```
training_args = TrainingArguments(
    output_dir="/content/drive/My Drive/blip-finetuned_balanced",
    num_train_epochs=5,
    per_device_train_batch_size=4,
    evaluation_strategy="no",
    save_steps=500,
    logging_steps=100,
    learning_rate=5e-5,
    fp16=True,
    gradient_accumulation_steps=2,
)
```

- **Time Taken:** Caption generation using the fine-tuned model and subsequent SBERT embedding were performed on the 1050 test images. Time taken was about 5 minutes for the caption generation.

Results

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the SBERT embeddings derived from the fine-tuned BLIP captions (using the 1050 test images).

Technique	K-Means ARI	DBSCAN ARI
BLIP Captions (Fine-tuned) + SBERT	0.7726	0.7741

- **Inferences:**
  - Fine-tuning the BLIP model, even with very basic template captions during training, led to a big improvement in clustering performance using K-Means. The ARI score (0.77) was the highest achieved across all methods, substantially outperforming baseline BLIP (0.21) and even fine-tuned EfficientNet (0.45).
  - Surprisingly, DBSCAN also performed exceptionally well on these fine-tuned caption embeddings (ARI 0.77), using the same default parameters that failed on nearly all previous feature sets. This strongly indicates that fine-tuning the text generation process produced highly discriminative embeddings with a structure particularly well-suited for clustering (by both K-Means and density-based methods like DBSCAN with these parameters) within our specific food domain.

Visualization As this is the first approach which produced promising results, we visualized the features using `t-SNE`

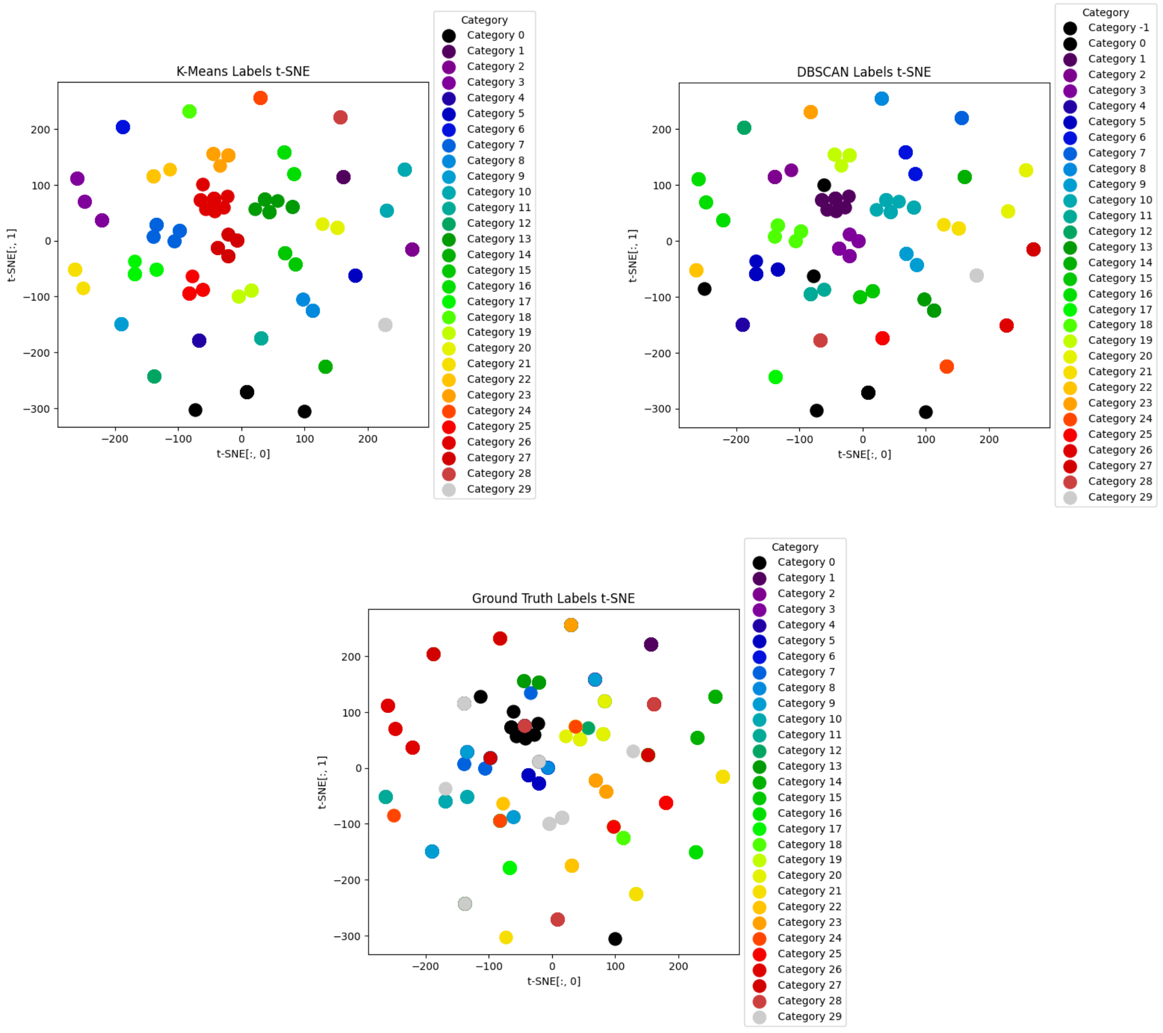

Figure 1: t-SNE visualizations for different clustering methods

We earlier planned on combining the fine-tuned deep (EfficientNet-B0) and fine-tuned text (BLIP+SBERT) features, potentially using a neural network architecture to learn multi-modal features. However, given the performance difference observed we decided to focus exclusively on the text-based features.

4.5 Dimensionality Reduction via Neural Network

We investigated whether compressing the highly effective fine-tuned BLIP+SBERT features ($\mathbf{f}_{\text{BLIP-FT+SBERT}} \in \mathbb{R}^{768}$) using a neural network could enhance clustering performance, potentially by isolating the most discriminative information.

Preprocessing

- The input data for this step consisted of the 1050 fine-tuned BLIP+SBERT feature vectors ($\mathbf{f}_{\text{BLIP-FT+SBERT}} \in \mathbb{R}^{768}$) generated from the **test** set images in Section 4.4.
- To split these embeddings using an 80/20 split, so:
  - Training set: 840 embeddings (28 per class)
  - Test set: 210 embeddings (7 per class)

```
le = LabelEncoder()
gt_labels_encoded = le.fit_transform(gt_labels)
```

```
X_train, X_val, y_train, y_val = train_test_split(
embeddings, gt_labels_encoded, test_size=0.2, random_state=782
)
```

Feature Extraction

- We designed a simple feed-forward neural network (CompressNet) with an architecture of 768 $\rightarrow$ 128 $\rightarrow$ 30. The intermediate 128-dimensional layer is taken as the compressed embedding space.
- After training the network on the 840 **train** images, we processed the fine-tuned BLIP+SBERT features from the 210 **test** images. Let these features be $\mathbf{f}_{\text{Compressed}} \in \mathbb{R}^{128}$.

```
class CompressNet(nn.Module):
    def __init__(self, input_dim, hidden_dim=128, output_dim=30):
        super(CompressNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        embeddings = x
        x = self.fc2(x)
        return x, embeddings


model = CompressNet(768, 128, 30)
```

Training and time taken

- **Training:** The CompressNet was trained for 20 epochs using 'CrossEntropyLoss' and an Adam optimizer.

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
EPOCHS = 20
```

- **Time Taken:** Training took about 5 seconds. Feature extraction time for the test set was negligible.

Results and inferences

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the extracted 128-dimensional features $\mathbf{f}_{\text{Compressed}}$ from the test set.

Technique	K-Means ARI	DBSCAN ARI
Compressed BLIP-FT+SBERT	0.7760	0.7741

- **Inferences:**
  - While the dimensionality reduction was successful in preserving the previous cluster structure quality, it offered no significant improvement in ARI scores.
  - We decided to continue with original embeddings for our final model.

5 Final Model

Based on the success of fine-tuning BLIP with template captions, we decided to train BLIP using a larger portion of the available data to potentially further improve the fine-tuned model's performance.

Preprocessing

- Instead of the initial 10% sample (2100 images), we sampled 20% of the images from each of the 30 classes in the given dataset. This resulted in a dataset of 4200 images (140 images per class).
- This new dataset (4200 images) was split 80% for training (3360 images, 112 per class) and 20% for testing (840 images, 28 per class).

Training and time taken

- **Training:** The 'Salesforce/blip-image-captioning-base' model was fine-tuned on this larger training set (3360 images) for 5 epochs using the same template captions ("This is an image of [class_name]") and hyperparameters as described previously.
- **Time Taken:** The training took about 40 minutes to complete

Results

- We applied K-Means ($k = 30$, `random_state=782`) and DBSCAN (`eps=0.5`, `min_samples=10`) to the extracted 768-dimensional features from the **test** set.

Technique	K-Means ARI	DBSCAN ARI
Final BLIP-FT+SBERT	0.8164	0.8164

- **Inferences:**
  - Fine-tuning the BLIP model on a larger dataset (3360 training images vs 1050 previously) yielded a further notable improvement in clustering performance.
  - This improvement was observed for both the clustering methods

5.1 Final Evaluation on Kaggle Test Set

- We used our final model to generate captions and then encode them using SBERT for the final evaluation on Kaggle Test Set provided in the kaggle cometition homepage
- The dataset consisted of 9000 images which we were tasked to cluster
- Generating the captions took about 30 minutes for the 9000 images
- We applied three different clustering algorithms to these 768D features to generate cluster assignments (labels from 0 to 29) for all 9000 images
- The ARI scores achieved were:

Algorithm	Kaggle Private ARI
`K-Means (n_clusters=30, random_state=782)` (K-Means)	**0.8036**
`DBSCAN (eps=0.5, min_samples=10)` (DBSCAN)	**0.8041**
`AgglomerativeClustering(n_clusters=30)` (Agglomerative)	**0.8036**

5.2 Visualization

- We've used `t-SNE` to visualize the clusters.
- As we lacked the ground truth labels the visualizations are not that meaningful themselves. Still they can be used to compare the clusters obtained through different clustering techniques.
- Here are the `t-SNE` clusters and the predicted labels for `KMeans`, `DBSCAN`, `AgglomerativeClustering`
- The -1 category in DBSCAN is for noise (these embeddings can belong to any cluster).

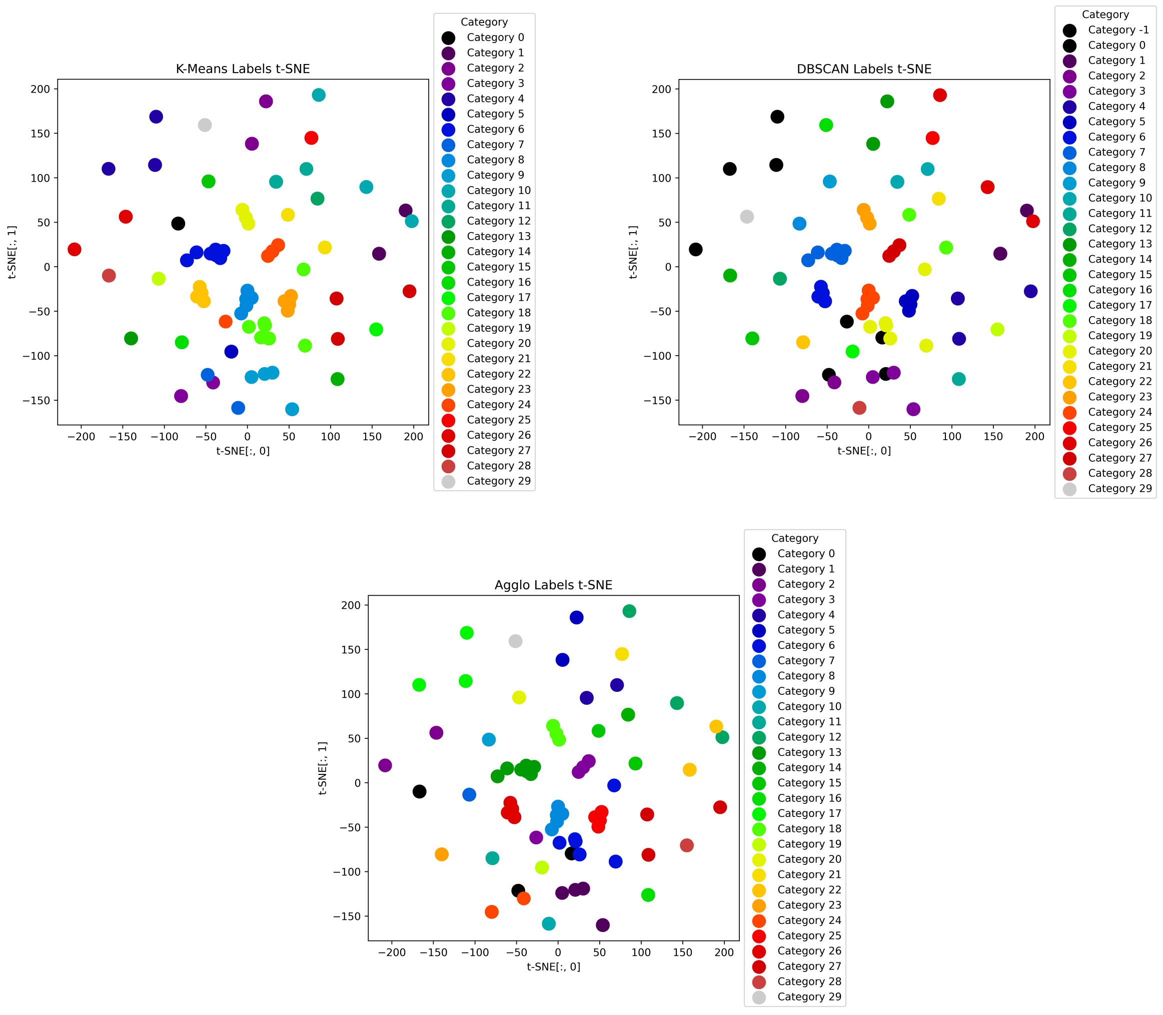

Figure 2: t-SNE visualizations for different clustering methods