# AI-Capstone: Project1 Report

111550093 朱驛庭

March 17, 2025

## 1 Introduction

In this project, our objective is to classify images of clothing items into predefined categories using various machine learning techniques. The task involves implementing and comparing the performance of three different models: ResNet, a deep learning-based approach; K-means, an unsupervised learning method; and SVM, a traditional supervised learning algorithm. The goal is to evaluate the effectiveness of these models in accurately classifying clothing items based on their visual characteristics.

## 2 Dataset

The dataset used in this project comprises clothing item images collected from Google Search and Unsplash. It is shared with a classmate (student ID: 111550160). The dataset can be accessed through the link (click).

### 2.1 Source

To assess the generalization capabilities of different models, the dataset includes images with varying levels of similarity within each category. For example, some images feature models wearing clothing items, while others showcase only clothing. The dataset comprises ten clothing categories: T-shirt, bag, dress, glasses, hat, hoodie, jacket, pants, shoes, and socks, with 50 images per category, totaling 500 images.

### 2.2 Train-test Split

The dataset is divided into training and testing sets, with 80% of the data used for training and 20% for testing. This split ensures that the models are trained on a substantial amount of data while retaining a separate set for evaluating their performance.

### 2.3 Preprocessing

I organized the images into separate folders based on their respective categories, naming each folder after the corresponding clothing type. Additionally, all images were converted to JPG format and sequentially renamed from 1 to 50.

# 3 Methods

## 3.1 Supervised Method1: Support Vector Machine

Support Vector Machine (SVM) is a supervised learning algorithm designed to find the optimal hyperplane that best separates data into different classes while maximizing the margin between them. The margin is defined as the distance between the hyperplane and the closest data points, known as **support vectors**. A larger margin generally leads to better generalization and robustness against overfitting. The illustration is show in Fig. 1. The SVM classifier seeks to minimize the following objective function:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{s.t. } y_i(w^T x_i + b) \geq 1, \forall i$$

where $w$ represents the weight vector, $b$ is the bias term, $x_i$ denotes the input feature vector, and $y_i$ is the corresponding class label ($+1$ or $-1$).

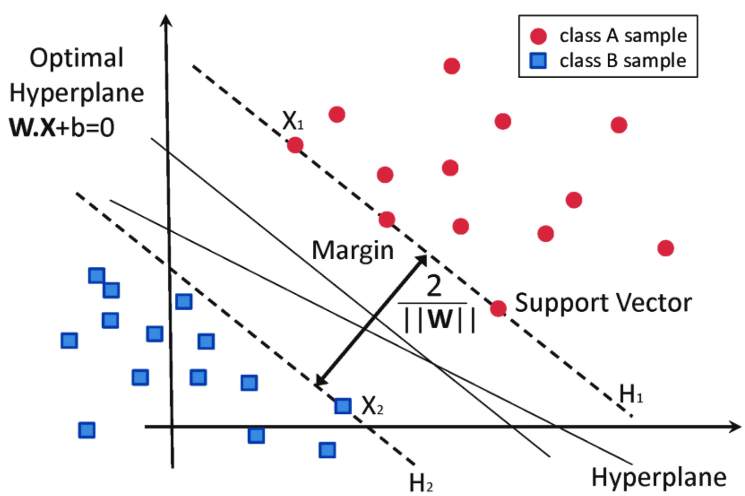


Figure 1: SVM Illustration [1].

### 3.1.1 Hinge Loss and Soft Margin SVM

To handle cases where data points may not be perfectly separable, SVM introduces **soft margin classification**, allowing for certain misclassifications by incorporating **slack variables** $\xi_i$. This is achieved by optimizing the **hinge loss function**, which penalizes misclassified points:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \forall i$$

where $C$ is a regularization parameter that balances the trade-off between maximizing the margin and minimizing classification errors.

### 3.1.2 Kernel Trick for Non-Linearly Separable Data

When the data is not linearly separable in the original feature space, SVM leverages the **kernel trick** to project the data into a higher-dimensional space where it becomes linearly separable. The kernel function replaces the direct computation of dot products, effectively mapping data into a transformed feature space:

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

where $\phi(x)$ is a mapping function that transforms the input space into a higher-dimensional feature space.

## 3.2 Supervised Method2: ResNet18

ResNet18 is a deep convolutional neural network (CNN) that belongs to the ResNet (Residual Network) family, introduced by He et al. in 2015. It is designed to address the vanishing gradient problem in deep networks by incorporating residual connections, allowing for deeper architectures while maintaining efficient training. ResNet18 consists of 18 layers, including convolutional, batch normalization, ReLU activation, and fully connected layers.

### 3.2.1 Architecture and Modification

ResNet18 follows a hierarchical structure with an initial convolutional layer, followed by 8 residual blocks grouped into 4 stages, and ending with a global average pooling and a fully connected layer. The modified version in this project replaces the original 1000-class output layer with a fully connected layer of size 10, corresponding to our clothing categories. The softmax activation is removed to output raw logits, which are processed by a cross-entropy loss function. The modified final layer is defined as:

$$\text{FC Layer: } y = Wx + b, \quad W \in \mathbb{R}^{10\times512}, \quad b \in \mathbb{R}^{10}$$

The overall structure of ResNet18 is illustrated in Fig. 2.

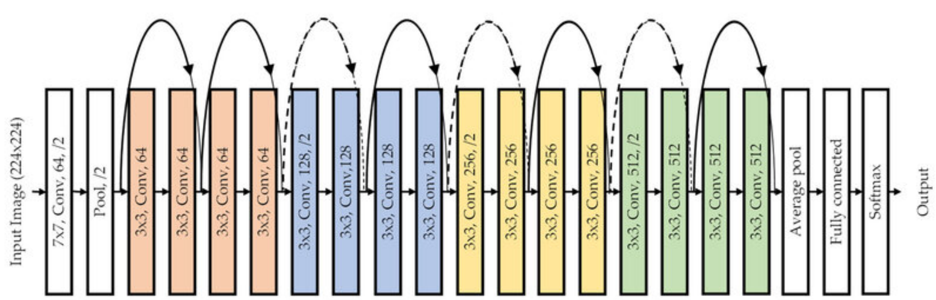


Figure 2: Structure of the ResNet18 Model [2].

### 3.2.2 Loss Function

The model is trained using the **cross-entropy loss function**, defined as:

$$L_{\text{CE}} = -\sum_{i=1}^{N} [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

where $y_i$ is the true class label, $p_i$ is the predicted probability, and $N$ is the total number of samples.

## 3.3 K-Means Clustering

K-Means is a widely used unsupervised learning algorithm for clustering, aiming to partition a dataset into $K$ clusters based on feature similarity. It operates iteratively to minimize intra-cluster variance, ensuring that data points within a cluster are as close as possible to the cluster centroid. The objective function, known as the **within-cluster sum of squares (WCSS)**, is given by:

$$\sum_{j=1}^{K} \sum_{i \in C_j} \|x_i - c_j\|^2$$

where $C_j$ represents the set of data points assigned to cluster $j$, and $c_j$ is the centroid of that cluster.

# 4 Implementation Details

## 4.1 Evaluation Metrics

### 4.1.1 Supervised Methods

For the supervised methods, namely ResNet and SVM, I employ several evaluation metrics to assess the performance of the models:

- **Confusion Matrix**: A confusion matrix is used to visualize the classification model's performance by organizing predictions into four categories:

$$\begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

  where:

  - **True Positives (TP)**: Correctly predicted positive instances.
  - **False Positives (FP)**: Incorrectly predicted positive instances.
  - **False Negatives (FN)**: Incorrectly predicted negative instances.
  - **True Negatives (TN)**: Correctly predicted negative instances.

- **Accuracy**: This metric measures the proportion of correctly classified instances over the total number of instances, providing a general sense of model performance. It is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**: Precision measures the proportion of correctly predicted positive instances among all instances classified as positive. It is given by:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity)**: Recall measures the model's ability to correctly identify all actual positive instances. It is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score**: The F1-score is the harmonic mean of precision and recall, balancing the trade-off between them. It is defined as:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

These metrics are computed using the `classification_report` function from scikit-learn, providing a comprehensive overview of the model' s performance across different classes.

#### 4.1.2 Unsupervised Method

For the unsupervised method, K-means, I use different metrics to evaluate clustering performance:

- **Silhouette Score**: This metric measures how similar an object is to its own cluster compared to other clusters. It ranges from -1 to 1, where a higher score indicates better-defined clusters.

- **Adjusted Rand Index (ARI)**: ARI measures the similarity between the clustering results and the true labels, adjusted for chance. It ranges from -1 to 1, with 1 indicating perfect agreement.

- **Calinski-Harabasz Score**: This metric evaluates the ratio of between-cluster dispersion to within-cluster dispersion. A higher score indicates well-separated and compact clusters.

- **Davies-Bouldin Score**: This metric assesses the average similarity between each cluster and its most similar cluster. A lower score indicates better clustering quality, as it suggests that clusters are well-separated.

### 4.2 Hyperparameters

- **ResNet**: The ResNet model's hyperparameters are carefully chosen to ensure effective training and convergence. The learning rate is initialized at 0.001, and a `StepLR` scheduler is used to dynamically adjust the learning rate during training. This scheduler reduces the learning rate by a factor of 0.1 every 10 epochs, helping the model converge more smoothly. The random seed is set to 40 to ensure reproducibility of results. The model is initialized with weights pre-trained on the ImageNet dataset, which provides a strong starting point for feature extraction. The batch size is set to 32, and the model is trained for 20 epochs, balancing computational efficiency with model performance.

- **SVM**: For the SVM model, I perform a grid search over a range of hyperparameters, including the regularization parameter $C$, kernel type (e.g., `rbf`, `poly`), and kernel-specific parameters like gamma and degree. This search is conducted using `GridSearchCV` from scikit-learn, which performs cross-validation to find the optimal hyperparameter combination.

- **K-means**: The primary hyperparameter for K-means is the number of clusters $K$. I experiment with different values of $K$ and use the ARI score to select the optimal number of clusters. Additionally, I use UMAP for dimensionality reduction before clustering, which involves tuning the number of components and the minimum distance between points.

## 5 Experiments

### 5.1 Data Augmentation

Data augmentation is a technique commonly used in machine learning to artificially expand the training dataset by applying transformations to the input images. The key idea is that by introducing non-identical inputs to the model in each epoch, data augmentation helps prevent overfitting and improves the generalization ability of the model. This is particularly beneficial for deep learning models, which tend to overfit when trained on small datasets. The comparison is shown below:

| Methods | Aug. | Accuracy(%) | Precision(%) | Recall(%) | F1-score(%) |
|---|---|---|---|---|---|
| SVM | ✗ | **73.74** | **76.30** | **73.80** | **73.80** |
| SVM | ✓ | 69.70 | 70.70 | 69.70 | 69.30 |
| ResNet18 | ✗ | 86.87 | 89.60 | 86.80 | 86.80 |
| ResNet18 | ✓ | **91.92** | **92.50** | **91.80** | **91.70** |

Table 1: Performance comparison of the effect of data augmentation on the two supervised methods.

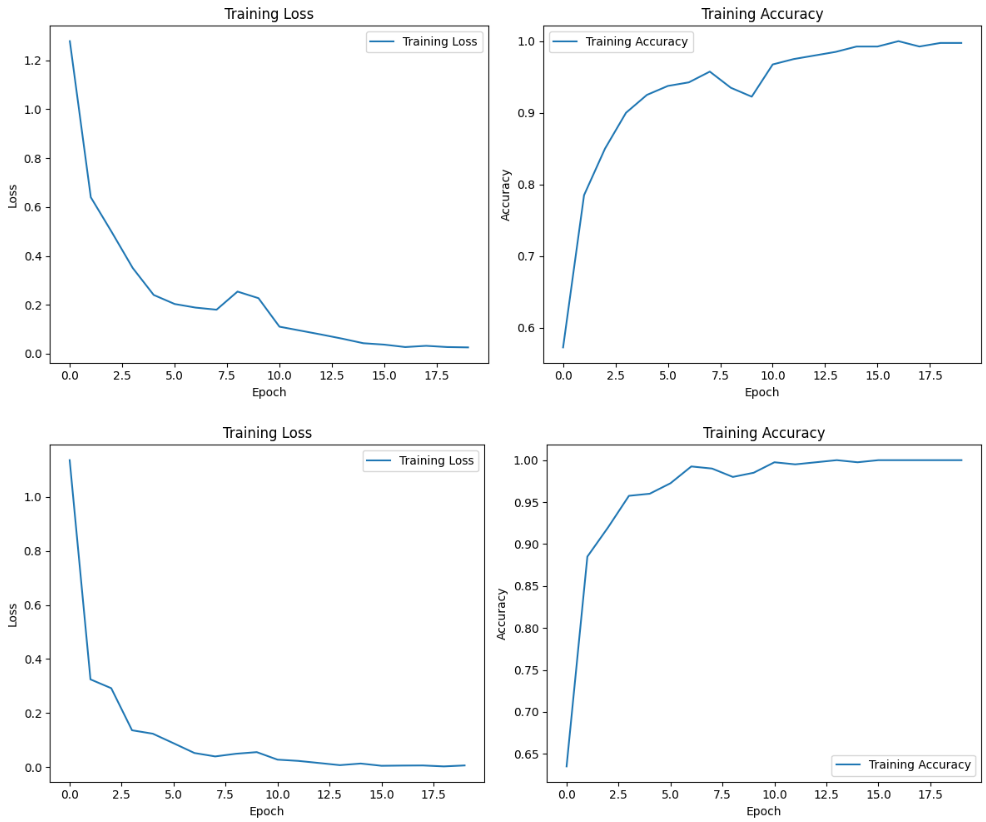


Figure 3: Training history of ResNet18 with and without data augmentation. The top row represents training with data augmentation, while the bottom row represents training without it.

From the results presented in Table 1, it is evident that data augmentation enhances the performance of ResNet18. This improvement suggests that ResNet18 benefits from the increased diversity introduced by augmentation, allowing it to generalize better to unseen data. However, the same approach does not yield positive results for SVM, as it is not a deep learning-based method. Instead of improving performance, the additional variations introduced by data augmentation negatively impact SVM, likely due to its reliance on hand-crafted features rather than learned representations.

Furthermore, as shown in Figure 3, data augmentation slows down the training process compared to the original setup. The model takes longer to converge since it needs to learn from a more diverse set of augmented samples. In typical cases, data augmentation may require additional training epochs to reach the same level of training loss as a model trained without augmentation.

## 5.2 t-SNE Visualization on ResNet18

t-SNE (t-distributed stochastic neighbor embedding) is a dimensionality reduction technique commonly used for visualization. By mapping high-dimensional data into a lower-dimensional space, t-SNE helps reveal underlying patterns and relationships within the data.

In Figure 4, we observe that as the data progresses through each layer of ResNet, the separation between different classes becomes more distinct. Notably, in the fully connected layer, the *hoodie*, *jacket*, and *t-shirt* classes are positioned relatively close to one another. This alignment is reasonable, as these three types of clothing share similar visual characteristics in the real world.

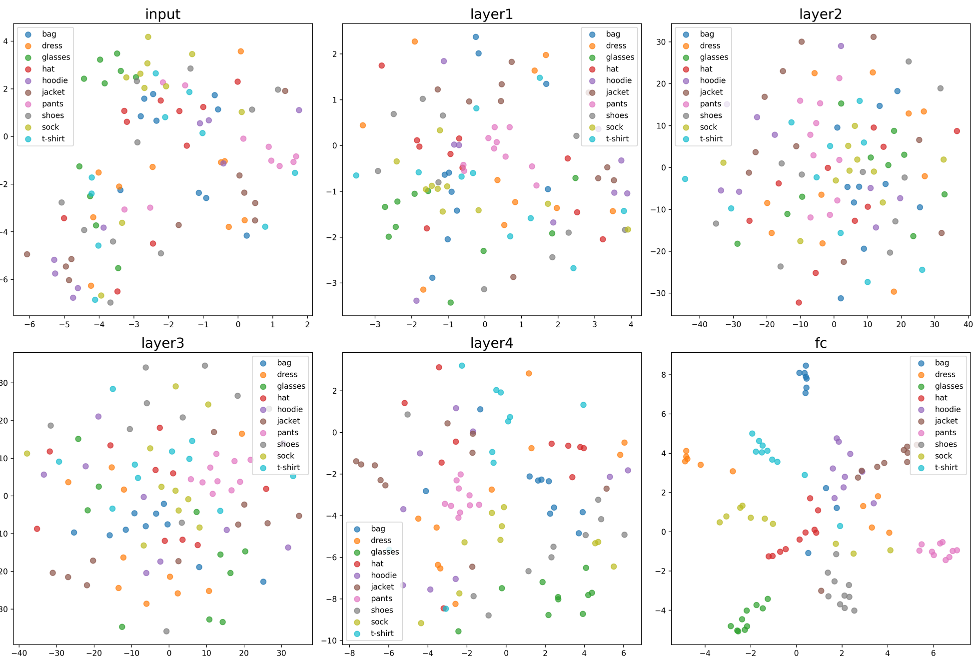


Figure 4: t-SNE visualization of different layers on ResNet18.

## 5.3 Multi-criteria Cluster Validation of K-means

Based on the result shown on Figure. 5. The analysis of K-means clustering metrics suggests that K=5 or K=6 provides the best balance across multiple validation metrics. The **Silhouette Score** and **Calinski-Harabasz Score** indicate that K=5 achieves strong separation, while K=6 has the highest **Adjusted Rand Index (ARI)**, suggesting better alignment with true labels. The **Davies-Bouldin Score** further supports K=5 or K=6 as optimal choices. At K=6, some clusters achieve high purity, such as bags (77.78%) and glasses (40.74%), while upper-body garments (t-shirts, jackets, and hoodies) tend to mix. The results indicate that while K-means effectively groups certain categories, visual similarities between some clothing items limit perfect separation. For general clothing classification, K=5 or K=6 is recommended, while K=2 or K=3 may suffice for broader category distinctions.

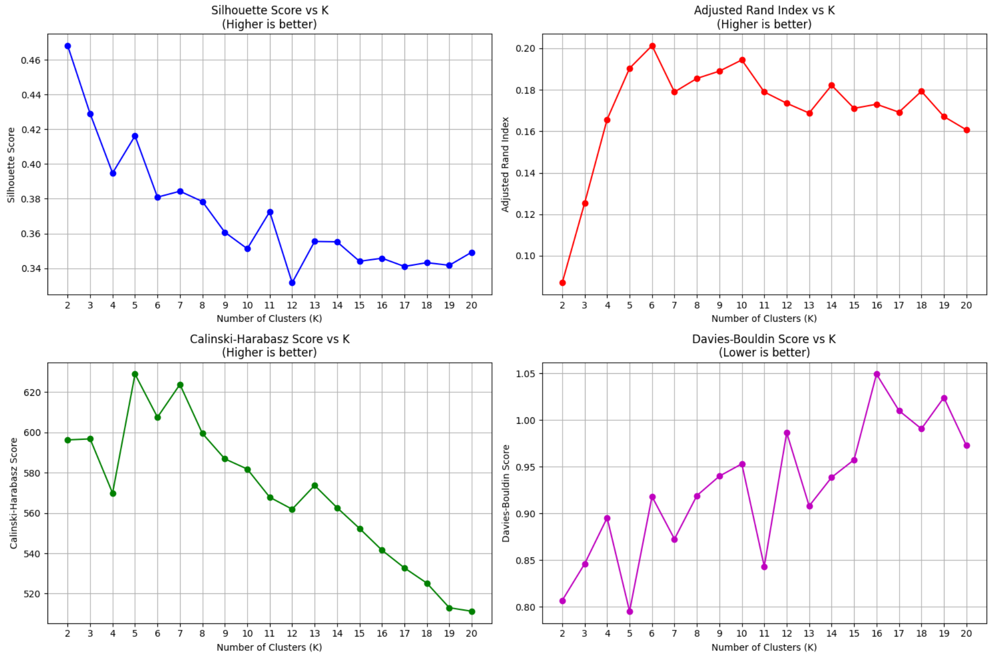


Figure 5: K-means clustering evaluation metrics for different numbers of clusters (K).

# 6 Discussion

## 6.1 Performance Comparison and Analysis

In this project, I have implemented and evaluated three different machine learning approaches for clothing image classification: ResNet18, SVM, and K-means. The experiments yielded several interesting observations:

- **ResNet18** demonstrated superior performance with a test accuracy of 91.92% when using data augmentation, outperforming the other methods. This result aligns with expectations, as deep learning methods typically excel at image classification tasks due to their ability to learn hierarchical representations directly from raw pixel data.
- **SVM** achieved a respectable accuracy of 73.74% without data augmentation. Interestingly, data augmentation decreased its performance, contradicting the common assumption that augmentation universally improves classification results. This suggests that traditional machine learning algorithms like SVM may struggle with the increased variance introduced by augmentation when using handcrafted features.
- **K-means**, as an unsupervised approach, showed limitations in perfectly separating the clothing categories, achieving a maximum ARI of 0.20 at K=6. This was expected, as unsupervised methods lack the guidance of class labels during training. However, the clustering revealed meaningful patterns, particularly in separating accessories from clothing items.

## 6.2 Factors Affecting Performance

Several factors influenced the performance of the different models:

- **Feature Representation**: ResNet18's superior performance can be attributed to its ability to learn complex, hierarchical features directly from raw images. In contrast, SVM relied on manually engineered features, which, while effective, may not capture all the nuances present in the data.

- **Dataset Characteristics**: The dataset's inherent challenges, such as varying backgrounds, lighting conditions, and item orientations, affected all models. Items with distinctive shapes (like bags) were generally easier to classify or cluster correctly compared to items with similar appearances (like t-shirts, hoodies, and jackets).

- **Visual Similarity**: The t-SNE visualization and K-means clustering results both revealed that visually similar categories (such as hoodie, jacket, and t-shirt) tend to overlap in the feature space. This suggests that even with sophisticated feature extraction, some ambiguity between similar clothing categories is inevitable.

- **Dimensionality Reduction**: For K-means, the UMAP dimensionality reduction played a crucial role in clustering performance. The optimal number of dimensions balanced between preserving data structure and reducing noise.

## 6.3 Lessons Learned

Overall, this project demonstrates the complementary nature of different machine learning approaches for image classification tasks. Each method offers unique insights and advantages, with deep learning-based approaches like ResNet18 providing state-of-the-art performance, traditional methods like SVM offering interpretability and computational efficiency, and unsupervised methods like K-means revealing natural data patterns without requiring labeled data.

# References

[1] D. Man, *Support vector machine in python*. [Online]. Available: https://medium.com/@dattanaman213/support-vector-machine-in-python-576eaac337ae.

[2] ResearchGate, *Structure of the resnet-18 model*. [Online]. Available: https://www.researchgate.net/figure/Structure-of-the-Resnet-18-Model_fig1_366608244.

# Appendix

Github link: https://github.com/ChuEating1005/AI-Capstone/tree/main/Project1

Listing 1: prepare_dataset.py

```
import os
import shutil
import random
from pathlib import Path
import numpy as np

def main():
    # Define paths
    SOURCE_DIR = Path('images')
    OUTPUT_DIR = Path('data')

    # Define split ratios
    TRAIN_RATIO = 0.8
    VAL_RATIO = 0
    TEST_RATIO = 0.2

    # Create output directories
    train_dir = OUTPUT_DIR / 'train'
    test_dir = OUTPUT_DIR / 'test'

    # Create directories if they don't exist
    for dir_path in [train_dir, test_dir]:
        for category in os.listdir(SOURCE_DIR):
            if category.startswith('.'):  # Skip hidden files like .DS_Store
                continue
            os.makedirs(dir_path / category, exist_ok=True)

    # Set random seed for reproducibility
    random.seed(42)
    np.random.seed(42)

    # Process each category
    for category in os.listdir(SOURCE_DIR):
        if category.startswith('.'):  # Skip hidden files like .DS_Store
            continue

        category_dir = SOURCE_DIR / category
        if not os.path.isdir(category_dir):
            continue

        # Get all image files
        image_files = [f for f in os.listdir(category_dir) if f.endswith(('.
    jpg', '.jpeg', '.png'))]

        # Shuffle the files
        random.shuffle(image_files)

        # Calculate split indices
```

```
        n_images = len(image_files)
        n_train = 40

        # Split the files
        train_files = image_files[:n_train]
        test_files = image_files[n_train:]

        # Copy files to respective directories
        for files, target_dir in zip([train_files, test_files], [train_dir, test_dir]):
            for file in files:
                src_path = category_dir / file
                dst_path = target_dir / category / file
                shutil.copy2(src_path, dst_path)

        print(f"Category {category}: {len(train_files)} train, {len(test_files)} test")

    print("Dataset preparation completed!")

if __name__ == "__main__":
    main()
```

Listing 2: dataset.py

```
from torch.utils.data import Dataset
from torchvision import transforms
from PIL import Image
import os
import numpy as np
import cv2
from skimage.feature import local_binary_pattern
from skimage.feature import hog
import umap

data_aug = True

train_tfm = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomGrayscale(p=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
]) if data_aug else transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

valid_tfm = transforms.Compose([
```

```python
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_tfm = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

svm_tfm = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue
=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomGrayscale(p=0.1),
])


class ResnetDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.samples = []
        self.classes = []
        self.loader = Image.open

        # Populate samples and classes
        for class_name in os.listdir(data_dir):
            class_path = os.path.join(data_dir, class_name)
            if os.path.isdir(class_path):
                self.classes.append(class_name)
                for file_name in os.listdir(class_path):
                    if file_name.lower().endswith(('.png', '.jpg', '.jpeg')):
                        file_path = os.path.join(class_path, file_name)
                        self.samples.append((file_path, self.classes.index(
class_name)))

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, index):
        """
        Override the __getitem__ method to skip corrupted images
        """
        path, label = self.samples[index]
        try:
            sample = self.loader(path)

            if self.transform is not None:
                sample = self.transform(sample)

```

```python
            return sample, label

        except Exception as e:
            print(f"Error loading image {path}: {e}")
            # Return a random valid image instead
            valid_idx = (index + 1) % len(self)
            return self.__getitem__(valid_idx)

class SVMDataset:
    def __init__(self, data_dir, augmentations=3):
        self.data_dir = data_dir
        self.augmentations = augmentations
        self.train_transform = svm_tfm
        self.test_transform = test_tfm
        self.class_names = []
        self.class_to_idx = {}

    def load_data(self):
        X_train, y_train = [], []
        X_test, y_test = [], []

        # Get class names from train directory
        self.class_names = [d for d in os.listdir(os.path.join(self.data_dir, 'train'))
                            if os.path.isdir(os.path.join(self.data_dir, 'train', d))]

        # Create class to index mapping
        self.class_to_idx = {cls_name: i for i, cls_name in enumerate(self.class_names)}

        # Process training data with augmentation
        print("Loading training data with augmentation...")
        for class_name in self.class_names:
            class_dir = os.path.join(self.data_dir, 'train', class_name)
            for img_file in os.listdir(class_dir):
                if img_file.lower().endswith(('.png', '.jpg', '.jpeg')):
                    img_path = os.path.join(class_dir, img_file)
                    img = Image.open(img_path).convert('RGB')

                    # Original image
                    features = self.extract_features(img, augment=False)
                    if features is not None:
                        X_train.append(features)
                        y_train.append(self.class_to_idx[class_name])

                    # Augmented versions
                    for _ in range(self.augmentations):
                        aug_features = self.extract_features(img, augment=True)
                        if aug_features is not None:
                            X_train.append(aug_features)
                            y_train.append(self.class_to_idx[class_name])

```

```
        # Process test data (no augmentation)
        print("Loading test data...")
        for class_name in self.class_names:
            class_dir = os.path.join(self.data_dir, 'test', class_name)
            for img_file in os.listdir(class_dir):
                if img_file.lower().endswith(('.png', '.jpg', '.jpeg')):
                    img_path = os.path.join(class_dir, img_file)
                    img = Image.open(img_path).convert('RGB')
                    features = self.extract_features(img, augment=False)
                    if features is not None:
                        X_test.append(features)
                        y_test.append(self.class_to_idx[class_name])

        return np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test)

    def extract_features(self, img, augment=False):
        if augment:
            img = self.train_transform(img)

        # Resize to 128x128
        img = img.resize((128, 128))
        img_array = np.array(img)

        # 1. Color Features (HSV space)
        img_hsv = cv2.cvtColor(img_array, cv2.COLOR_RGB2HSV)

        # HSV histograms
        hist_h = np.histogram(img_hsv[:,:,0], bins=16)[0]
        hist_s = np.histogram(img_hsv[:,:,1], bins=16)[0]
        hist_v = np.histogram(img_hsv[:,:,2], bins=16)[0]

        # Color statistics
        color_stats = np.concatenate([
            np.mean(img_hsv, axis=(0,1)),  # mean of each channel
            np.std(img_hsv, axis=(0,1)),   # std of each channel
            np.percentile(img_hsv, [25,75], axis=(0,1)).flatten()  # quartiles
        ])

        # 2. Texture Features
        gray = cv2.cvtColor(img_array, cv2.COLOR_RGB2GRAY)

        # Convert to 8-bit unsigned integer
        gray = (gray * 255).astype(np.uint8)

        # Multi-scale LBP
        lbp_features = []
        for radius in [1, 2, 3]:
            lbp = local_binary_pattern(gray, P=8*radius, R=radius, method='uniform')
            lbp_hist = np.histogram(lbp, bins=10)[0]
            lbp_features.extend(lbp_hist/lbp_hist.sum())  # 正規化

        # 3. Shape Features (HOG)
```

```
        hog_features = hog(gray,
                          orientations=9,
                          pixels_per_cell=(16, 16),
                          cells_per_block=(2, 2),
                          visualize=False)

        # 4. Edge Features
        edges = cv2.Canny(gray, 100, 200)
        edge_hist = np.histogram(edges, bins=16)[0]

        # Gradient direction histograms
        grad_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
        grad_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
        grad_mag = np.sqrt(grad_x**2 + grad_y**2)
        grad_dir = np.arctan2(grad_y, grad_x)
        grad_hist = np.histogram(grad_dir, bins=18, range=(-np.pi, np.pi))[0]

        # Combine all features and normalize
        features = np.concatenate([
            hist_h/hist_h.sum(),           # 16 features
            hist_s/hist_s.sum(),           # 16 features
            hist_v/hist_v.sum(),           # 16 features
            color_stats,                    # 12 features
            np.array(lbp_features),         # 30 features
            hog_features/np.linalg.norm(hog_features),  # normalized HOG
            edge_hist/edge_hist.sum(),      # 16 features
            grad_hist/grad_hist.sum()       # 18 features
        ])


        return features
```

Listing 3: main.py

```
import os
import argparse
import time
from pathlib import Path


def prepare_dataset():
    """Prepare the dataset by splitting it into train, validation, and test
   sets"""
    from prepare_dataset import main as prepare_main
    prepare_main()

def train_resnet():
    """Train the ResNet model (supervised learning with DL)"""
    from models.ResNet import ResNetTrainer

    print("\n" + "="*60)
    print("Training ResNet Model (Supervised Learning - Deep Learning)")
    print("="*60)

```

```
    start_time = time.time()
    trainer = ResNetTrainer(data_dir='data', batch_size=32, num_epochs=20, seed=40)
    trainer.train()
    accuracy, _ = trainer.evaluate()
    trainer.plot_training_history()
    trainer.visualize_tsne_comparison()
    end_time = time.time()

    print(f"\nResNet training completed in {end_time - start_time:.2f} seconds")
    # print(f"Test accuracy: {accuracy:.4f}")

    # return accuracy

def train_svm():
    """Train the SVM model (supervised learning without DL)"""
    from models.SVM import SVMClassifier

    print("\n" + "="*60)
    print("Training SVM Model (Supervised Learning - Traditional ML)")
    print("="*60)

    start_time = time.time()
    classifier = SVMClassifier(data_dir='data')
    accuracy, _ = classifier.train()
    end_time = time.time()

    print(f"\nSVM training completed in {end_time - start_time:.2f} seconds")
    print(f"Test accuracy: {accuracy:.4f}")

    return accuracy

def train_kmeans():
    """Train the K-means model (unsupervised learning)"""
    from models.K_means import KMeansClusterer

    print("\n" + "="*60)
    print("Training K-means Model (Unsupervised Learning)")
    print("="*60)

    start_time = time.time()
    clusterer = KMeansClusterer(data_dir='data')
    clusterer.train()
    end_time = time.time()

    print(f"\nK-means training completed in {end_time - start_time:.2f} seconds")

    return None  # No standard accuracy for unsupervised learning

def main():
    parser = argparse.ArgumentParser(description='Train and evaluate models for clothing classification')
```

```python
    parser.add_argument('--prepare', action='store_true', help='Prepare the dataset')
    parser.add_argument('--resnet', action='store_true', help='Train ResNet model')
    parser.add_argument('--svm', action='store_true', help='Train SVM model')
    parser.add_argument('--kmeans', action='store_true', help='Train K-means model')
    parser.add_argument('--all', action='store_true', help='Run all models')

    args = parser.parse_args()

    # Create results directory
    os.makedirs('results', exist_ok=True)

    # If no specific arguments are provided, run all
    if not (args.prepare or args.resnet or args.svm or args.kmeans or args.all):
        args.all = True

    # Prepare dataset if requested
    if args.prepare or args.all:
        print("\nPreparing dataset...")
        prepare_dataset()

    # Check if dataset exists
    if not Path('data').exists():
        print("\nDataset not found. Please run with --prepare first.")
        return

    # Ensure models directory exists
    if not Path('models').exists() and (args.resnet or args.svm or args.kmeans or args.all):
        print("\nModels directory not found. Please make sure the 'models' folder exists with model files.")
        return

    results = {}

    # Train models as requested
    if args.resnet or args.all:
        results['resnet'] = train_resnet()

    if args.svm or args.all:
        results['svm'] = train_svm()

    if args.kmeans or args.all:
        results['kmeans'] = train_kmeans()

    # Print summary of results
    print("\n" + "="*50)
    print("Summary of Results")
    print("="*50)

    for model, accuracy in results.items():
```

```
        if accuracy is not None:
            print(f"{model.upper()} Test Accuracy: {accuracy:.4f}")

    print("\nAll models have been trained and evaluated successfully!")

if __name__ == "__main__":
    main()
```

Listing 4: models/ResNet.py

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import numpy as np
import random
from torchvision import models
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
from tqdm import tqdm
from sklearn.manifold import TSNE
from dataset import ResnetDataset, train_tfm, valid_tfm, test_tfm

# Set random seeds for reproducibility
def set_seed(seed=40):
    """Set all random seeds for reproducibility"""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)  # For multi-GPU setups
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    os.environ['PYTHONHASHSEED'] = str(seed)


class ResNetTrainer:
    def __init__(self, data_dir='data', batch_size=32, num_epochs=40,
   learning_rate=0.001, seed=40):
        # Set seed for reproducibility
        set_seed(seed)

        self.data_dir = data_dir
        self.batch_size = batch_size
        self.num_epochs = num_epochs
        self.learning_rate = learning_rate
        self.device = torch.device("cuda:0" if torch.cuda.is_available() else
   "cpu")

        # Load datasets with custom class that handles corrupted images
        self.datasets = {
            'train': ResnetDataset(os.path.join(data_dir, 'train'), transform=
```

```
train_tfm),
            'test': ResnetDataset(os.path.join(data_dir, 'test'), transform=test_tfm)
        }

        # Create dataloaders
        self.dataloaders = {
            'train': DataLoader(self.datasets['train'], batch_size=batch_size, shuffle=True, num_workers=6),
            'test': DataLoader(self.datasets['test'], batch_size=batch_size, shuffle=False, num_workers=2)
        }

        self.dataset_sizes = {x: len(self.datasets[x]) for x in ['train', 'test']}
        self.class_names = self.datasets['train'].classes

        print(f"Classes: {self.class_names}")
        print(f"Dataset sizes: {self.dataset_sizes}")

        # Initialize model
        self.model = models.resnet18(weights='IMAGENET1K_V1')
        self.model.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(self.model.fc.in_features, len(self.class_names))
        )
        self.model = self.model.to(self.device)

        # Loss function and optimizer
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
        self.scheduler = optim.lr_scheduler.StepLR(self.optimizer, step_size=10, gamma=0.1)

        # For tracking metrics
        self.train_losses = []
        self.train_accs = []

    def train(self):
        best_test_acc = 0.0

        for epoch in range(self.num_epochs):
            print(f'Epoch {epoch+1}/{self.num_epochs}')
            print('-' * 10)

            # Each epoch has a training and testing phase
            for phase in ['train', 'test']:
                if phase == 'train':
                    self.model.train()  # Set model to training mode
                else:
                    self.model.eval()   # Set model to evaluate mode

                running_loss = 0.0
                running_corrects = 0
```

```python

            # Wrap dataloader with tqdm for progress bar
            pbar = tqdm(self.dataloaders[phase], desc=f'{phase} Epoch {epoch+1}/{self.num_epochs}')

            # Iterate over data
            for inputs, labels in pbar:
                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

                # Zero the parameter gradients
                self.optimizer.zero_grad()

                # Forward pass
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = self.model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = self.criterion(outputs, labels)

                    # Backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        self.optimizer.step()

                # Statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

                # Update progress bar
                batch_loss = loss.item()
                batch_acc = torch.sum(preds == labels.data).double() / inputs.size(0)
                pbar.set_postfix({'loss': f'{batch_loss:.4f}', 'acc': f'{batch_acc:.4f}'})

            epoch_loss = running_loss / self.dataset_sizes[phase]
            epoch_acc = running_corrects.double() / self.dataset_sizes[phase]

            if phase == 'train':
                self.train_losses.append(epoch_loss)
                self.train_accs.append(epoch_acc.item())
                self.scheduler.step()  # Step the scheduler
            else:
                # Save the best model
                if epoch_acc > best_test_acc:
                    best_test_acc = epoch_acc
                    torch.save(self.model.state_dict(), 'results/resnet_best_model.pth')

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

        print()

```

```
        print(f'Best test Acc: {best_test_acc:4f}')

        # Load best model weights
        self.model.load_state_dict(torch.load('results/resnet_best_model.pth'))
        return self.model

    def evaluate(self):
        # Set model to evaluate mode
        self.model.eval()

        # Initialize lists to store predictions and ground truth
        all_preds = []
        all_labels = []

        # No gradient calculation needed
        with torch.no_grad():
            # Wrap dataloader with tqdm for progress bar
            pbar = tqdm(self.dataloaders['test'], desc='Evaluating')

            for inputs, labels in pbar:
                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

                # Forward pass
                outputs = self.model(inputs)
                _, preds = torch.max(outputs, 1)

                # Store predictions and labels
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())

                # Update progress bar
                batch_acc = torch.sum(preds == labels.data).double() / inputs.size(0)
                pbar.set_postfix({'batch_acc': f'{batch_acc:.4f}'})

        # Calculate accuracy
        accuracy = np.mean(np.array(all_preds) == np.array(all_labels))
        print(f'Test Accuracy: {accuracy:.4f}')

        # Generate confusion matrix
        cm = confusion_matrix(all_labels, all_preds)

        # Plot confusion matrix
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=self.class_names,
                    yticklabels=self.class_names)
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.title('Confusion Matrix')
        plt.savefig('plots/resnet_confusion_matrix.png')

```

```
        # Print classification report
        print("\nClassification Report:")
        print(classification_report(all_labels, all_preds, target_names=self.class_names))

        return accuracy, cm

    def plot_training_history(self):
        plt.figure(figsize=(12, 5))

        # Plot loss
        plt.subplot(1, 2, 1)
        plt.plot(self.train_losses, label='Training Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Training Loss')
        plt.legend()

        # Plot accuracy
        plt.subplot(1, 2, 2)
        plt.plot(self.train_accs, label='Training Accuracy')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.title('Training Accuracy')
        plt.legend()

        plt.tight_layout()
        plt.savefig('plots/resnet_training_history.png')
        plt.show()


    def visualize_tsne_comparison(self, layers=['input', 'layer1', 'layer2', 'layer3', 'layer4', 'fc'], perplexity=30, n_iter=1000):
        """
        Visualize and compare feature representations from multiple layers using t-SNE

        Args:
            layers (list): List of layers to extract features from
            perplexity (int): Perplexity parameter for t-SNE
            n_iter (int): Number of iterations for t-SNE
        """
        self.model.eval()

        layer_indices = {
            'layer1': 4,
            'layer2': 5,
            'layer3': 6,
            'layer4': 7
        }

        fig, axes = plt.subplots(2, 3, figsize=(18, 12))
        axes = axes.flatten()

```

```
        for i, layer in enumerate(layers):
            if layer == 'input':
                feature_extractor = torch.nn.Identity()
            elif layer in layer_indices:
                feature_extractor = torch.nn.Sequential(*list(self.model.children())[:layer_indices[layer]])
            elif layer == 'fc':
                feature_extractor = torch.nn.Sequential(*list(self.model.children())[:-1])
            else:
                raise ValueError(f"Layer {layer} not supported for feature extraction")

            features = []
            labels = []

            with torch.no_grad():
                for inputs, targets in tqdm(self.dataloaders['test'], desc=f'Extracting features from {layer}'):
                    inputs = inputs.to(self.device)
                    feat = feature_extractor(inputs)
                    feat = feat.view(feat.size(0), -1)
                    features.append(feat.cpu().numpy())
                    labels.append(targets.numpy())

            features = np.concatenate(features, axis=0)
            labels = np.concatenate(labels, axis=0)

            print(f"Applying t-SNE on {features.shape[0]} samples with {features.shape[1]} features from {layer}...")
            tsne = TSNE(n_components=2, perplexity=perplexity, n_iter=n_iter, random_state=40)
            features_tsne = tsne.fit_transform(features)

            ax = axes[i]
            for _, label in enumerate(np.unique(labels)):
                idx = labels == label
                ax.scatter(features_tsne[idx, 0], features_tsne[idx, 1], label=self.class_names[label], alpha=0.7, s=50)

            ax.set_title(f'{layer}', fontsize=18)
            ax.legend()

        plt.tight_layout()
        plt.savefig('plots/resnet_tsne_comparison.png', dpi=300)
        plt.show()


if __name__ == "__main__":
    trainer = ResNetTrainer(data_dir='data', batch_size=32, num_epochs=40, seed=40)
    trainer.train()
    trainer.evaluate()
    trainer.plot_training_history()
```

```
    trainer.visualize_tsne_comparison()
```

Listing 5: models/SVM.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
import seaborn as sns
from PIL import Image
import joblib
from pathlib import Path
from dataset import SVMDataset

class SVMClassifier:
    def __init__(self, data_dir='dataset'):
        self.data_dir = Path(data_dir)
        self.class_names = None
        self.model = None
        self.scaler = None
        self.dataset = SVMDataset(data_dir, 0)

    def train(self):
        """Train the SVM model with hyperparameter tuning"""
        # Load data
        X_train, y_train, X_test, y_test = self.dataset.load_data()

        print(f"Training data shape: {X_train.shape}")
        print(f"Test data shape: {X_test.shape}")

        # Create a pipeline with scaling and SVM
        pipeline = Pipeline([
            ('scaler', StandardScaler()),
            ('svm', SVC(probability=True))
        ])

        # Define parameter grid for grid search
        param_grid = {
            'svm__C': [0.1, 1, 10, 100, 1000],
            'svm__gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
            'svm__kernel': ['rbf', 'poly'],
            'svm__degree': [2, 3] # for poly kernel
        }

        # Perform grid search with cross-validation
        print("Performing grid search for hyperparameter tuning...")
        grid_search = GridSearchCV(pipeline, param_grid, cv=3, n_jobs=-1,
    verbose=2)
        grid_search.fit(X_train, y_train)
```

```python

        # Get the best model
        self.model = grid_search.best_estimator_
        print(f"Best parameters: {grid_search.best_params_}")

        # Save the model
        joblib.dump(self.model, 'results/svm_model.pkl')

        # Evaluate on test set
        test_preds = self.model.predict(X_test)
        test_accuracy = accuracy_score(y_test, test_preds)
        print(f"Test accuracy: {test_accuracy:.4f}")

        # Generate confusion matrix
        cm = confusion_matrix(y_test, test_preds)

        # Check if class_names is populated
        if not self.class_names:
            self.class_names = [str(i) for i in range(len(set(y_test)))]

        # Plot confusion matrix
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=self.class_names,
                    yticklabels=self.class_names)
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.title('SVM Confusion Matrix')
        plt.savefig('plots/svm_confusion_matrix.png')

        # Print classification report
        print("\nClassification Report:")
        print(classification_report(y_test, test_preds, target_names=self.
    class_names))

        return test_accuracy, cm

if __name__ == "__main__":
    classifier = SVMClassifier(data_dir='dataset')
    classifier.train()
```

Listing 6: models/K_means.py

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, adjusted_rand_score,
    calinski_harabasz_score, davies_bouldin_score
import seaborn as sns
from PIL import Image
import joblib
```

```
from pathlib import Path
from collections import Counter
import pandas as pd
from skimage.feature import local_binary_pattern
from scipy.ndimage import sobel
from skimage.feature import hog
import cv2
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
import umap

class KMeansClusterer:
    def __init__(self, data_dir='dataset', n_clusters=10):
        self.data_dir = Path(data_dir)
        self.n_clusters = n_clusters
        self.model = None
        self.pca = None
        self.scaler = None
        self.class_names = None

    def extract_features(self, img_path):
        """
        Improved feature extraction with focus on clothing-specific
    characteristics
        """
        try:
            # Load and resize image
            img = Image.open(img_path).convert('RGB')
            img = img.resize((128, 128))  # Increase image size to preserve
    more details
            img_array = np.array(img)

            # 1. Enhanced Color Features
            # Use HSV color space
            img_hsv = cv2.cvtColor(img_array, cv2.COLOR_RGB2HSV)

            # Calculate HSV histograms
            hist_h = np.histogram(img_hsv[:,:,0], bins=16)[0]
            hist_s = np.histogram(img_hsv[:,:,1], bins=16)[0]
            hist_v = np.histogram(img_hsv[:,:,2], bins=16)[0]

            # Color statistics
            color_stats = np.concatenate([
                np.mean(img_hsv, axis=(0,1)),
                np.std(img_hsv, axis=(0,1))
            ])

            # 2. Improved texture features
            gray = cv2.cvtColor(img_array, cv2.COLOR_RGB2GRAY)

            # Use different scales of LBP
            lbp_features = []
            for radius in [1, 2, 3]:
                lbp = local_binary_pattern(gray, P=8*radius, R=radius, method=
```

```python
'uniform')
                lbp_hist = np.histogram(lbp, bins=10)[0]
                lbp_features.extend(lbp_hist)

            # 3. Improved shape features
            # Use denser HOG features
            hog_features = hog(gray,
                               orientations=9,
                               pixels_per_cell=(16, 16),
                               cells_per_block=(2, 2),
                               visualize=False)

            # 4. Edge features
            edges = cv2.Canny(gray, 100, 200)
            edge_hist = np.histogram(edges, bins=16)[0]

            # Combine all features
            features = np.concatenate([
                hist_h/hist_h.sum(), hist_s/hist_s.sum(), hist_v/hist_v.sum(),  # Normalized color histograms
                color_stats,                                                      # HSV statistics
                np.array(lbp_features)/sum(lbp_features),                        # Normalized LBP features
                hog_features/np.linalg.norm(hog_features),                       # Normalized HOG features
                edge_hist/edge_hist.sum()                                         # Normalized edge features
            ])

            return features

        except Exception as e:
            print(f"Error processing {img_path}: {e}")
            return None

    def load_data(self):
        """Load and prepare data for K-means clustering"""
        X = []  # Features
        y = []  # True labels (for evaluation only)
        img_paths = []  # Store paths for visualization

        # Get class names from train directory
        self.class_names = [d for d in os.listdir(self.data_dir / 'train')
                           if os.path.isdir(self.data_dir / 'train' / d)]

        # Create class to index mapping
        class_to_idx = {cls_name: i for i, cls_name in enumerate(self.class_names)}

        # Process all data (train, test combined for unsupervised learning)
        for split in ['train', 'test']:
            for class_name in self.class_names:
                class_dir = self.data_dir / split / class_name
```

```python
                for img_file in os.listdir(class_dir):
                    if img_file.lower().endswith(('.png', '.jpg', '.jpeg')):
                        img_path = class_dir / img_file
                        features = self.extract_features(img_path)
                        if features is not None:
                            X.append(features)
                            y.append(class_to_idx[class_name])
                            img_paths.append(str(img_path))

        return np.array(X), np.array(y), img_paths

    def train(self):
        """Improved training process"""
        # Load data
        X, y_true, img_paths = self.load_data()

        # Feature selection
        selector = SelectKBest(score_func=mutual_info_classif, k=100)
        X_selected = selector.fit_transform(X, y_true)

        # Standardization
        self.scaler = StandardScaler()
        X_scaled = self.scaler.fit_transform(X_selected)

        # Use UMAP for dimensionality reduction
        self.reducer = umap.UMAP(n_components=30)
        X_reduced = self.reducer.fit_transform(X_scaled)

        # Try different numbers of clusters
        silhouette_scores = []
        ari_scores = []
        ch_scores = []
        db_scores = []
        k_range = list(range(2, 21))  # Integers from 2 to 20

        # Create plot
        plt.figure(figsize=(15, 10))

        # Cluster for each K value and calculate metrics
        for k in k_range:
            kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
            labels = kmeans.fit_predict(X_reduced)

            # Calculate metrics
            silhouette = silhouette_score(X_reduced, labels)
            ari = adjusted_rand_score(y_true, labels)
            ch = calinski_harabasz_score(X_reduced, labels)
            db = davies_bouldin_score(X_reduced, labels)

            silhouette_scores.append(silhouette)
            ari_scores.append(ari)
            ch_scores.append(ch)
            db_scores.append(db)

```

```
            print(f"K={k}, Silhouette Score: {silhouette:.4f}, ARI: {ari:.4f}, CH: {ch:.4f}, DB: {db:.4f}")



        # Silhouette Score
        plt.subplot(2, 2, 1)
        plt.plot(k_range, silhouette_scores, 'bo-')
        plt.xlabel('Number of Clusters (K)')
        plt.ylabel('Silhouette Score')
        plt.title('Silhouette Score vs K\n(Higher is better)')
        plt.grid(True)
        plt.xticks(k_range, k_range)  # Set x-axis ticks to integers

        # ARI Score
        plt.subplot(2, 2, 2)
        plt.plot(k_range, ari_scores, 'ro-')
        plt.xlabel('Number of Clusters (K)')
        plt.ylabel('Adjusted Rand Index')
        plt.title('Adjusted Rand Index vs K\n(Higher is better)')
        plt.grid(True)
        plt.xticks(k_range, k_range)

        # Calinski-Harabasz Score
        plt.subplot(2, 2, 3)
        plt.plot(k_range, ch_scores, 'go-')
        plt.xlabel('Number of Clusters (K)')
        plt.ylabel('Calinski-Harabasz Score')
        plt.title('Calinski-Harabasz Score vs K\n(Higher is better)')
        plt.grid(True)
        plt.xticks(k_range, k_range)

        # Davies-Bouldin Score
        plt.subplot(2, 2, 4)
        plt.plot(k_range, db_scores, 'mo-')
        plt.xlabel('Number of Clusters (K)')
        plt.ylabel('Davies-Bouldin Score')
        plt.title('Davies-Bouldin Score vs K\n(Lower is better)')
        plt.grid(True)
        plt.xticks(k_range, k_range)

        plt.tight_layout()
        plt.savefig('plots/kmeans_metrics_comparison.png')

        # Find best K value for each metric
        best_k_silhouette = k_range[np.argmax(silhouette_scores)]
        best_k_ari = k_range[np.argmax(ari_scores)]
        best_k_ch = k_range[np.argmax(ch_scores)]
        best_k_db = k_range[np.argmin(db_scores)]  # Note: Lower DB score is better

        print("\nBest K values by different metrics:")
        print(f"Silhouette Score: K={best_k_silhouette}")
        print(f"Adjusted Rand Index: K={best_k_ari}")
```

```python
        print(f"Calinski-Harabasz Score: K={best_k_ch}")
        print(f"Davies-Bouldin Score: K={best_k_db}")

        # Use best K value (using best K from ARI score here)
        self.n_clusters = best_k_ari
        self.model = KMeans(n_clusters=self.n_clusters, random_state=42, n_init=10)
        cluster_labels = self.model.fit_predict(X_reduced)

        # Evaluate and visualize
        self._visualize_clusters(X_reduced, cluster_labels, y_true)
        self._analyze_clusters(cluster_labels, y_true)
        metrics = self.evaluate_clustering(X_reduced, cluster_labels, y_true)

        return self.model

    def _visualize_clusters(self, X_pca, cluster_labels, y_true):
        """Visualize the clusters in 2D using PCA"""
        # Further reduce to 2D for visualization
        pca_viz = PCA(n_components=2)
        X_pca_2d = pca_viz.fit_transform(X_pca)

        # Plot clusters
        plt.figure(figsize=(12, 10))

        # Plot by cluster assignment
        plt.subplot(1, 2, 1)
        scatter = plt.scatter(X_pca_2d[:, 0], X_pca_2d[:, 1], c=cluster_labels, cmap='viridis', alpha=0.6)
        plt.colorbar(scatter)
        plt.title('K-means Clustering Results')
        plt.xlabel('PCA Component 1')
        plt.ylabel('PCA Component 2')

        # Plot by true labels
        plt.subplot(1, 2, 2)
        scatter = plt.scatter(X_pca_2d[:, 0], X_pca_2d[:, 1], c=y_true, cmap='tab10', alpha=0.6)
        plt.colorbar(scatter)
        plt.title('True Class Labels')
        plt.xlabel('PCA Component 1')
        plt.ylabel('PCA Component 2')

        plt.tight_layout()
        plt.savefig('plots/kmeans_clusters_visualization.png')

    def _analyze_clusters(self, cluster_labels, y_true):
        """Analyze the composition of each cluster"""
        # Create a mapping from cluster labels to true labels
        cluster_to_label = {}

        for cluster_id in range(max(cluster_labels) + 1):  # Use actual cluster labels instead of self.n_clusters
            # Get indices of samples in this cluster
```

```
            indices = np.where(cluster_labels == cluster_id)[0]

            # Skip empty clusters
            if len(indices) == 0:
                print(f"\nCluster {cluster_id} is empty")
                continue

            # Get true labels of these samples
            true_labels = y_true[indices]

            # Count occurrences of each true label
            label_counts = Counter(true_labels)

            # Find the most common true label in this cluster
            most_common_label = label_counts.most_common(1)[0][0]
            cluster_to_label[cluster_id] = most_common_label

            # Print cluster composition
            print(f"\nCluster {cluster_id} composition:")
            for label, count in label_counts.most_common():
                percentage = count / len(indices) * 100
                print(f"  {self.class_names[label]}: {count} samples ({percentage:.2f}%)")

        # Create confusion matrix-like visualization
        n_clusters = max(cluster_labels) + 1
        confusion = np.zeros((n_clusters, len(self.class_names)))

        for i in range(len(cluster_labels)):
            cluster = cluster_labels[i]
            true_label = y_true[i]
            confusion[cluster, true_label] += 1

        # Normalize by cluster size
        for i in range(n_clusters):
            if np.sum(confusion[i, :]) > 0:
                confusion[i, :] = confusion[i, :] / np.sum(confusion[i, :])

        # Plot heatmap
        plt.figure(figsize=(12, 10))
        sns.heatmap(confusion, annot=True, fmt='.2f', cmap='Blues',
                   xticklabels=self.class_names,
                   yticklabels=[f'Cluster {i}' for i in range(n_clusters)])
        plt.xlabel('True Class')
        plt.ylabel('Cluster')
        plt.title('Cluster Composition')
        plt.tight_layout()
        plt.savefig('plots/kmeans_cluster_composition.png')

    def evaluate_clustering(self, X_pca, cluster_labels, y_true):
        """Evaluate the clustering performance"""
        # 1. Calculate ARI
        ari = adjusted_rand_score(y_true, cluster_labels)
        print(f"Adjusted Rand Index: {ari:.4f}")
```

```python

        # 2. Calculate Silhouette Score
        silhouette_avg = silhouette_score(X_pca, cluster_labels)
        print(f"Silhouette Score: {silhouette_avg:.4f}")

        # 3. Analyze the purity of each cluster
        purities = []
        for cluster_id in range(self.n_clusters):
            indices = np.where(cluster_labels == cluster_id)[0]
            if len(indices) > 0:
                true_labels = y_true[indices]
                most_common = Counter(true_labels).most_common(1)[0]
                purity = most_common[1] / len(indices)
                purities.append(purity)

        avg_purity = np.mean(purities)
        print(f"Average Cluster Purity: {avg_purity:.4f}")

        return {
            'ari': ari,
            'silhouette': silhouette_avg,
            'purity': avg_purity
        }

if __name__ == "__main__":
    clusterer = KMeansClusterer(data_dir='dataset', n_clusters=10)
    clusterer.train()
```