AI Capstone HW1

Topic: Leetcode cheating detector

Github: AI-Capstone/HW1 at main · qwer521/AI-Capstone

111550034 黃皓君

1. Motivation

During LeetCode contests, some participants solve problems extremely quickly. However, upon reviewing submissions from top coders, I observed that their code often contains an unusual number of comments, which would be difficult to produce in a short time. This observation led me to suspect that some contestants might be using large language models (LLMs) to assist in solving problems. Motivated by this, I developed the idea of training a machine learning model to determine whether a piece of code is generated by AI and to further investigate whether the same individual consistently writes code in a particular style.

---

2. Dataset and Data Collection

Dataset Construction:

The dataset consists of C++ source code files collected from three distinct sources:

- **My own submissions:** Code I wrote for LeetCode problems.
- **GPT-o3-mini-high responses:** Code generated by a language model(GPT-o3-mini-high) prompted to solve LeetCode problems, with instructions to omit comments for increased ambiguity.
- **Public solutions:** Code from a LeetCode user (geetanjali-18) chose because of distinct, comment-free style.

Data Type:

Plain text files containing C++ source code (with comments removed).

Composition:

- **Training Set:** 27 documents per coding style.
- **Test Set:** 15 documents per coding style.
- The data is organized into folders labeled "0", "1", and "2", each corresponding to a different author or coding style.

Data Collection Process:

- Code samples were extracted from my submissions, GPT-generated responses, and public LeetCode solutions.

- Files were manually reviewed and labeled based on the source.
- Data cleanup involved removing extraneous spaces to ensure uniformity.

Dataset Documentation:
The complete dataset and its documentation are available on GitHub:
AI-Capstone/HW1/data at main · qwer521/AI-Capstone

---

3. Methods

3.1 Feature Extraction and Dimensionality Reduction

Feature Extraction:
I employed TF-IDF vectorization to transform raw C++ source code into numerical features. TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure that evaluates the importance of a term within a document relative to the corpus.

- **Term Frequency (TF):** The frequency of a term in a document, normalized by the total number of terms.
- **Inverse Document Frequency (IDF):** Down-weights terms that appear in many documents.
- The product, TF × IDF, assigns higher scores to terms that are frequent in a document yet rare in the corpus.

To capture subtle coding patterns, I used character-level n-grams (with an n-gram range of 3 to 5) extracted using the char_wb analyzer. This approach enables the model to capture common coding constructs (e.g., “int”, “num”, “sum”) and syntactic elements that may differ among authors.

Dimensionality Reduction:
In some experiments, Principal Component Analysis (PCA) was applied to reduce feature dimensionality. This reduction aided in visualizing clusters and improved computational efficiency.

3.2 Supervised Learning Approaches

I evaluated two main supervised learning models for code classification:

- **Support Vector Machines (SVM):**
  SVMs were applied on the TF-IDF features to capture stylistic patterns.
- **Deep Learning (CNN):**
  A convolutional neural network (CNN) built with TensorFlow/Keras, incorporating an embedding layer, 1D convolution, batch normalization, global max pooling, dropout, and L2 regularization. The output layer uses softmax for multi-class classification.

Both models were evaluated using 5-fold cross-validation and confusion matrices were generated to analyze misclassifications.

3.3 Unsupervised Learning Approaches

KMeans Clustering:

For unsupervised analysis, KMeans clustering was used to group the code files into clusters. To assess clustering performance, the following were performed:

- **Cluster Feature Extraction:**
  Top TF-IDF features for each cluster were identified to highlight distinctive coding patterns.
- **Mapping to True Labels:**
  Predicted clusters were mapped to true labels using the Hungarian algorithm for an optimal one-to-one assignment.
- **Performance Evaluation:**
  Overall cluster purity (accuracy) was computed, and a confusion matrix was generated comparing the mapped cluster labels to the ground-truth labels.

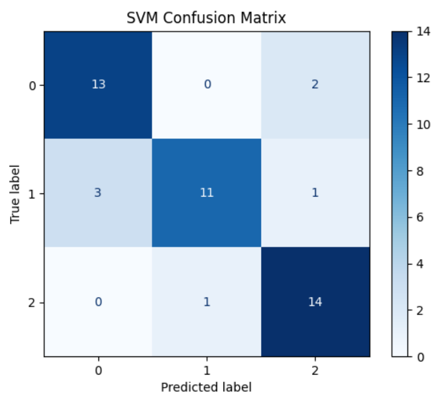
---

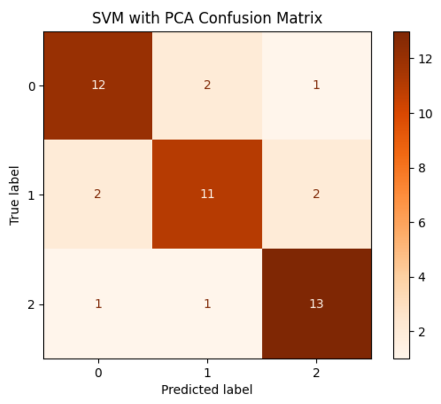
4. Experiments and Results

This section consolidates all experimental outcomes. It includes both supervised learning (with loss plots and confusion matrices) and unsupervised clustering results (including KMeans feature extraction).

4.1 Supervised Learning Results

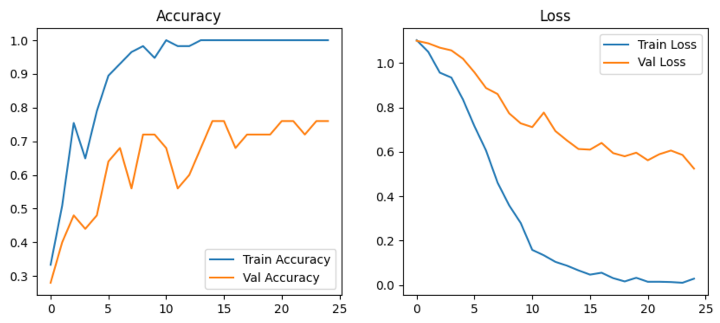
- **SVM:**
  - **Cross-Validation:** The SVM classifier achieved an accuracy of **75.4% ± 11.4%**, indicating effective learning of stylistic patterns.
  - **Test Set Evaluation:** The SVM model achieved approximately **84.4%** accuracy on the test set.
  - **Confusion Matrix:**





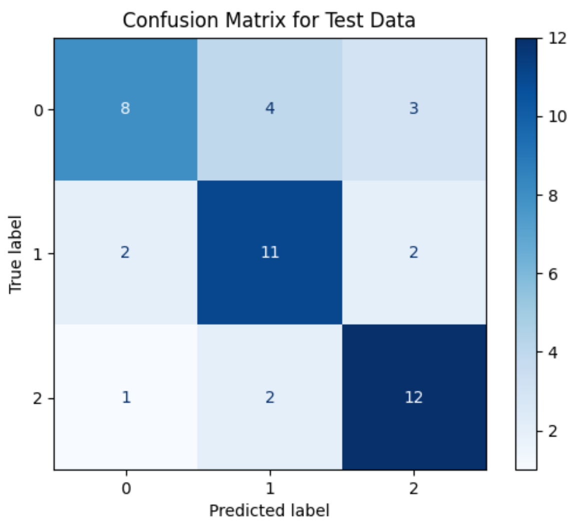
- **Analysis:**
  Most predictions lie on the diagonal, indicating that SVM correctly classifies a large portion of each class. After applying PCA for dimensionality reduction, SVM's performance is still respectable but can shift the error distribution. In some cases, PCA helps if it preserves discriminative features.

- **Deep Learning (CNN):**
  - **Test Accuracy:** The CNN model reached approximately **68.8%** test accuracy.
  - **Loss Plot:**



  Training accuracy rapidly approaches 100%, while validation accuracy lags behind (somewhere in the 70–80% range). Training loss heads toward zero, whereas validation loss decreases more slowly and eventually plateaus. The gap between training and validation curves indicates that the model memorizes training data and does not generalize as well.

  - **Confusion Matrix:**



True \ Predicted	0	1	2
0	8	4	3
1	2	11	2
2	1	2	12

  - **Analysis:**

The CNN is likely overfitting due to the small dataset size. Although CNN learns some patterns, it falls short of SVM's performance. The confusion matrix confirms that class 0 is the most challenging to classify, possibly indicating insufficient discriminative features for that style.

4.2 Unsupervised Learning (KMeans Clustering) Results

- **Cluster Distribution:**
  The clustering analysis produced the following distribution on the combined training and test data:
  - Cluster 0: 25 documents
  - Cluster 1: 39 documents
  - Cluster 2: 63 documents
- **Feature Extraction by KMeans:**
  For each cluster, the top TF-IDF features were extracted:
  - Cluster 0: The presence of terms like "node" and "nod" suggests frequent usage of tree structures.
  - Cluster 1: Features such as "ums", "nums", "num", "long" hint at numerical array manipulations.
  - Cluster 2: Contains a mix of syntax fragments (" } ", "){ ") and a frequent substring "ans", which might indicate code dealing with answers or outputs.
- **Representative Documents for Each Cluster:**
  - Cluster 0:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr) ...
```

    Deals with tree structures (e.g., TreeNode).

  - Cluster 1:

```
class Solution {
    public:
        long long countBadPairs(vector<int>& nums) {
            unordered_map<long long int,long long int> m;
            for(int i=0;i<nums.size();i++){
                    m[nums[i]-i]++; ...
```

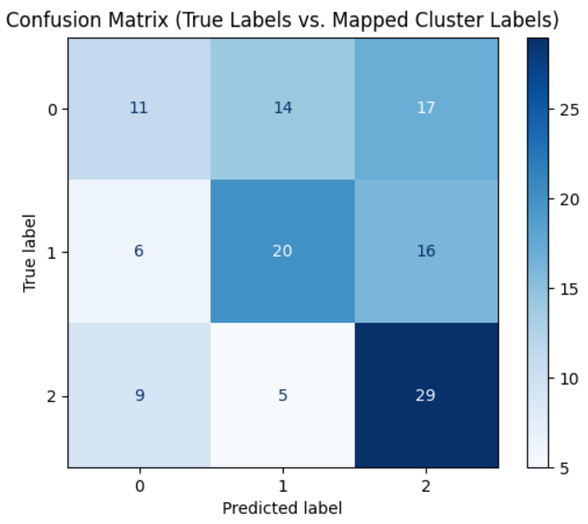
It focuses on array-based logic (nums and array operations)

- Cluster 2:

```
class Solution {
    public:
        vector<int> validSequence(string s, string t) {
            int m = s.length();
            int n = t.length();
            vector<int>vec(m);
            vec.push_back(n);
            vector<int>ans; ...
```

It contains code for string manipulation or general solution patterns.

- **Overall Cluster Purity and Confusion Matrix:**



Cluster Purity: **47.24%** (Accuracy)

- **Analysis**:
  The cluster purity indicates that KMeans did not strongly group the code by author style. Instead, the clusters appear to be influenced by problem type (e.g., tree vs. array vs. string operations) rather than purely by each coder's stylistic differences.
- **Possible Fixes**:
  - **Use Domain-Specific Features**: Parse the Abstract Syntax Tree (AST) or leverage CodeBERT-style embeddings to capture deeper syntactic and semantic patterns unique to each author.
  - **Expand the Dataset**: A larger dataset with more samples per author might help separate style from problem-specific keywords.
  - **Try Different Distance Metrics**: Switching to cosine distance or

performing further dimensionality reduction could better highlight style-based distinctions.

---

5. Discussion

Observations

- **Supervised Learning:**
  - The SVM classifier performed robustly on both cross-validation and test data, indicating that the character-level TF-IDF features capture meaningful stylistic differences in C++ code.
  - The expected CNN model should be more accurate compared to SVM. However the result is much lower. I think it is because when the training data is scarce, CNNs tend to overfit, memorizing the training examples rather than learning generalizable features.
- **Unsupervised Learning:**
  - Analysis of the top TF-IDF features and representative documents revealed that each cluster captured different aspects of the code. However, instead of isolating unique authorial styles, the clusters appear to be influenced by problem type—for instance, clusters were grouped by characteristics of string manipulation, tree structures, or array operations rather than solely by coding style.
  - This suggests that the current feature extraction method may be emphasizing problem-specific keywords and syntactic patterns over subtle stylistic nuances.
- **Why Supervised Learning Performs Better than Clustering:**
  - Supervised models can directly learn the differences between classes using label information, while unsupervised clustering is limited by the quality and nature of the extracted features and the high-dimensional, sparse representation provided by TF-IDF.

Factors Affecting Performance

- **Dataset Size and Balance:**
  - The limited number of documents (45 in training) and imbalance among classes affect both supervised and unsupervised performance.
- **Feature Extraction:**
  - While TF-IDF with character n-grams is effective, it may not fully capture the syntactic and semantic nuances of code. More advanced feature extraction (e.g., AST parsing or code embeddings like CodeBERT) could improve performance.

- **Dimensionality:**
    - High-dimensional, sparse data poses challenges for clustering algorithms. Dimensionality reduction and alternate distance metrics (e.g., cosine distance) might yield better separation.

Future Work

If more time were available, I would:

- **Collect More Data:** Expand the dataset to include more code samples per author.
- **Enhance Feature Extraction:** Explore domain-specific features or pretrained models for source code.
- **Experiment with Other Clustering Algorithms:** Investigate methods like HDBSCAN or spectral clustering that might better capture non-spherical cluster shapes.
- **Hyperparameter Tuning:** Conduct a more systematic hyperparameter search for both supervised and unsupervised models.

Lessons Learned

This project underscored the importance of quality data and appropriate feature engineering in machine learning. Although classical methods (SVM) performed well, unsupervised clustering in high-dimensional spaces remains challenging without careful tuning and potentially more sophisticated representations.

---

6. References

- TFIDF: TfidfVectorizer — scikit-learn 1.6.1 documentation
- Kmeans: KMeans — scikit-learn 1.6.1 documentation
- Keras. https://keras.io
- TensorFlow: TensorFlow Documentation
- Scikit-learn:Scikit-learn Documentation.

7. Appendix: Program Code

Github link: AI-Capstone/HW1/code at main · qwer521/AI-Capstone

Supervised-SVM.py:

```
import os
import glob
import numpy as np
import matplotlib.pyplot as plt
import re
```

```python
# Import necessary modules from scikit-learn
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix,
ConfusionMatrixDisplay
from sklearn.decomposition import PCA

def load_dataset(directory):
    """
    Loads text files from subdirectories.
    Each subdirectory name is used as the label.

    Parameters:
        directory (str): Path to the main directory.
        use_preprocessing (bool): Whether to remove spaces from the
text.

    Returns:
        texts (list): List of document strings.
        labels (np.array): Array of labels (as strings).
    """
    texts = []
    labels = []
    # Loop over each subdirectory (each represents a label)
    for label in os.listdir(directory):
        folder = os.path.join(directory, label)
        if os.path.isdir(folder):
            # Load all .txt files in the folder
            for filepath in glob.glob(os.path.join(folder, '*.txt')):
                with open(filepath, 'r', encoding='utf-8') as file:
                    text = file.read()
                    texts.append(text)
                    labels.append(label)
    return texts, np.array(labels)
```

```python
########################################################################
########
# Main Function for SVM-based Classification
########################################################################
########
def main():
    # Define dataset paths for training and test sets
    train_dir = './data/train'
    test_dir = './data/test'

    # Load training and test data
    X_train, y_train = load_dataset(train_dir)
    X_test, y_test = load_dataset(test_dir)

    # Display basic information about the dataset
    print(f"Loaded {len(X_train)} training documents and {len(X_test)}
test documents.")

    # ---------------------------
    # SVM Classification using TF-IDF
    # ---------------------------
    # Set TF-IDF parameters to capture character-level n-grams (3 to 5
characters)
    tfidf_params = {
        'stop_words': None,
        'analyzer': 'char_wb',  # Use word-boundary constrained
character n-grams
        'ngram_range': (3, 5)
    }

    # Create a pipeline that vectorizes text and applies a linear SVM
    svm_pipeline = Pipeline([
        ('tfidf', TfidfVectorizer(**tfidf_params)),
        ('clf', LinearSVC(max_iter=10000))
    ])

    # Evaluate using 5-fold cross-validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    print("=== Cross-Validation on Training Data (SVM) ===")
    scores = cross_val_score(svm_pipeline, X_train, y_train, cv=cv,
scoring='accuracy')
    print(f"SVM CV Accuracy: {np.mean(scores):.4f} ±
{np.std(scores):.4f}")

    # Train the SVM pipeline on the full training set and evaluate on
the test set
    print("\n=== Evaluation on Test Data (SVM) ===")
    svm_pipeline.fit(X_train, y_train)
    y_pred = svm_pipeline.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"SVM Test Accuracy: {acc:.4f}")

    # Plot confusion matrix for SVM predictions
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=svm_pipeline.classes_)
    disp.plot(cmap=plt.cm.Blues)
    plt.title("SVM Confusion Matrix")
    plt.show()

    # ---------------------------
    # SVM with PCA for Dimensionality Reduction
    # ---------------------------
    print("=== Experiment: SVM with PCA ===")
    # Vectorize text data using TF-IDF
    vectorizer = TfidfVectorizer(**tfidf_params)
    X_train_tfidf = vectorizer.fit_transform(X_train)
    X_test_tfidf = vectorizer.transform(X_test)

    # Convert sparse matrices to dense arrays for PCA processing
    X_train_dense = X_train_tfidf.toarray()
    X_test_dense = X_test_tfidf.toarray()

    # Apply PCA to reduce dimensionality; n_components can be tuned
based on dataset
    pca = PCA(n_components=70, random_state=42)
    X_train_pca = pca.fit_transform(X_train_dense)
    X_test_pca = pca.transform(X_test_dense)

    # Train a new SVM on the PCA-transformed data
    svm_pca = LinearSVC(max_iter=10000)
    svm_pca.fit(X_train_pca, y_train)
    y_pred_pca = svm_pca.predict(X_test_pca)
    acc_pca = accuracy_score(y_test, y_pred_pca)
    print(f"SVM with PCA Test Accuracy: {acc_pca:.4f}")

    # Plot confusion matrix for SVM with PCA predictions
    cm_pca = confusion_matrix(y_test, y_pred_pca)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm_pca,
display_labels=np.unique(y_test))
    disp.plot(cmap=plt.cm.Oranges)
    plt.title("SVM with PCA Confusion Matrix")
    plt.show()

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

Supervised-DeepLearning.py

```
import os
import glob
import numpy as np
import matplotlib.pyplot as plt
import re

# Deep Learning Imports
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D,
GlobalMaxPooling1D, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
```

```python
# Metrics for confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def load_dataset(directory):
    """
    Loads text files from subdirectories.
    Returns texts and corresponding labels.
    """
    texts = []
    labels = []
    for label in os.listdir(directory):
        folder = os.path.join(directory, label)
        if os.path.isdir(folder):
            for filepath in glob.glob(os.path.join(folder, '*.txt')):
                with open(filepath, 'r', encoding='utf-8') as file:
                    text = file.read()
                    texts.append(text)
                    labels.append(label)
    return texts, np.array(labels)

def main():
    train_dir = './data/train'
    test_dir = './data/test'

    X_train, y_train = load_dataset(train_dir)
    X_test, y_test = load_dataset(test_dir)

    y_train_int = np.array([int(label) for label in y_train])
    y_test_int = np.array([int(label) for label in y_test])
    num_classes = len(np.unique(y_train_int))
    print(f"Number of classes detected: {num_classes}")

    # Debug: Print label distributions for training and test sets
    unique_train, counts_train = np.unique(y_train_int,
return_counts=True)
    print("Training label distribution:", dict(zip(unique_train,
counts_train)))
```

```
    unique_test, counts_test = np.unique(y_test_int,
return_counts=True)
    print("Test label distribution:", dict(zip(unique_test,
counts_test)))

    # Shuffle training data to ensure validation split is
representative
    indices = np.arange(len(X_train))
    np.random.shuffle(indices)
    X_train = [X_train[i] for i in indices]
    y_train_int = y_train_int[indices]

    # Tokenize and pad sequences
    max_words = 10000  # Vocabulary size
    max_len = 500      # Maximum sequence length
    tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
    tokenizer.fit_on_texts(X_train)
    X_train_seq = tokenizer.texts_to_sequences(X_train)
    X_test_seq = tokenizer.texts_to_sequences(X_test)
    X_train_pad = pad_sequences(X_train_seq, maxlen=max_len,
padding='post', truncating='post')
    X_test_pad = pad_sequences(X_test_seq, maxlen=max_len,
padding='post', truncating='post')

    # Build a CNN for multi-class classification with modifications to
reduce overfitting
    model = Sequential([
        Embedding(input_dim=max_words, output_dim=128,
input_length=max_len),
        Conv1D(filters=128, kernel_size=5, activation='relu'),
        BatchNormalization(),
        GlobalMaxPooling1D(),
        Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.6),  # Increased dropout to reduce overfitting
        Dense(num_classes, activation='softmax')  # Multi-class output
    ])
    model.compile(loss='sparse_categorical_crossentropy',
optimizer=Adam(), metrics=['accuracy'])
```

```
    model.summary()

    # Early stopping callback to prevent overfitting (optional)
    # early_stopping =
tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)

    # Train the model with a validation split
    epochs = 25
    batch_size = 8
    history = model.fit(
        X_train_pad, y_train_int,
        validation_split=0.2,
        epochs=epochs,
        batch_size=batch_size,
        # callbacks=[early_stopping],
        verbose=1
    )

    # Evaluate on test data
    loss, accuracy = model.evaluate(X_test_pad, y_test_int, verbose=1)
    print(f"Deep Learning Model Test Accuracy: {accuracy:.4f}")

    # Generate predictions on test data for confusion matrix
    test_pred_probs = model.predict(X_test_pad)
    test_pred_labels = np.argmax(test_pred_probs, axis=1)

    # Plot confusion matrix for test data
    cm = confusion_matrix(y_test_int, test_pred_labels)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=np.unique(y_test_int))
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix for Test Data")
    plt.show()

    # Debug: Check a few predictions on validation samples from
training data
    val_split = 0.2
    split_index = int(len(X_train_pad) * (1 - val_split))
    X_val = X_train_pad[split_index:]
    y_val = y_train_int[split_index:]
    pred_probs = model.predict(X_val)
    pred_labels = np.argmax(pred_probs, axis=1)
    print("Sample validation predictions:", pred_labels[:10])
    print("Actual validation labels:", y_val[:10])

    # Plot training history
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Loss')
    plt.legend()

    plt.show()

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

Unsupervised-Kmeans.py

```
import os
import glob
import numpy as np
import matplotlib.pyplot as plt
import re

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics.pairwise import euclidean_distances
from scipy.optimize import linear_sum_assignment

def load_texts_and_labels(directory: str):
    """
    Loads .txt files from subdirectories of the given directory.
    The subdirectory name is used as the label for each .txt file.

    :param directory: Path to the directory containing labeled
subfolders.
    :param preprocess: Whether to remove spaces from the text.
    :return: (list of document strings, numpy array of integer labels)
    """
    texts, labels = [], []
    for label in os.listdir(directory):
        folder = os.path.join(directory, label)
        if os.path.isdir(folder):
            for filepath in glob.glob(os.path.join(folder, '*.txt')):
                with open(filepath, 'r', encoding='utf-8') as file:
                    content = file.read()
                    texts.append(content)
                    # Assumes folder names are valid integers (e.g.,
"0", "1", "2", ...)
                    labels.append(int(label))
    return texts, np.array(labels)

########################################################################
########
# KMeans Clustering Analysis with One-to-One Mapping
########################################################################
########
def analyze_kmeans_clusters(documents, true_labels, n_clusters=None,
top_n=5):
    """
    Performs TF-IDF vectorization, runs KMeans clustering, and prints
detailed analysis including:
      - Cluster distribution.
      - Top TF-IDF features per cluster.
      - Representative document snippet (closest to centroid).
      - One-to-one mapping of predicted clusters to true labels via
Hungarian algorithm.
      - Overall cluster purity (accuracy).
      - Confusion matrix of mapped predictions vs. true labels.

    :param documents: List of document strings.
    :param true_labels: Ground-truth labels as a numpy array of
integers.
    :param n_clusters: Number of clusters for KMeans. If None, set to
number of unique true labels.
    :param top_n: Number of top TF-IDF features to display per cluster.
    """
    # If n_clusters not provided, use number of unique true labels
    if n_clusters is None:
        n_clusters = len(np.unique(true_labels))

    # TF-IDF vectorization using character-level n-grams
    vectorizer = TfidfVectorizer(analyzer='char_wb', ngram_range=(3,5))
    X = vectorizer.fit_transform(documents)
    feature_names = vectorizer.get_feature_names_out()

    # Run KMeans clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    pred_labels = kmeans.fit_predict(X)

    print(f"\n=== KMeans with k={n_clusters} ===")

    # Print cluster distribution
    for c in range(n_clusters):
        count = np.sum(pred_labels == c)
        print(f"Cluster {c}: {count} documents")

    # Top TF-IDF features for each cluster
    centroids = kmeans.cluster_centers_
    print(f"\nTop {top_n} TF-IDF features per cluster:")
    for c in range(n_clusters):
        centroid = centroids[c]
```

```python
        top_indices = np.argsort(centroid)[::-1][:top_n]
        top_features = feature_names[top_indices]
        print(f"  Cluster {c}: " + ", ".join(top_features))

    # Representative Documents: find document closest to each centroid
    print("\nRepresentative documents for each cluster (closest to 
centroid):")
    distances = euclidean_distances(X, centroids)
    for c in range(n_clusters):
        cluster_mask = (pred_labels == c)
        if not np.any(cluster_mask):
            print(f"  Cluster {c} has no documents!")
            continue
        cluster_distances = distances[cluster_mask, c]
        closest_local_idx = np.argmin(cluster_distances)
        global_idx = np.where(cluster_mask)[0][closest_local_idx]
        snippet = documents[global_idx][:200].replace("\n", " ")
        print(f"  Cluster {c}: doc index={global_idx}, snippet:")
        print("    ", snippet, "...\n")

    # Create the contingency table (confusion matrix between clusters 
and true labels)
    cm = confusion_matrix(true_labels, pred_labels)

    # Use Hungarian algorithm to find one-to-one mapping between 
predicted clusters and true labels
    # We want to maximize correct assignments; convert to cost matrix 
by taking negative
    row_ind, col_ind = linear_sum_assignment(-cm)
    mapping = {col: row for row, col in zip(row_ind, col_ind)}

    # Ensure all clusters are mapped; if a cluster wasn't assigned, map 
it arbitrarily (should not happen if numbers match)
    for c in range(n_clusters):
        if c not in mapping:
            mapping[c] = -1  # assign a dummy label if missing

    print("\nOne-to-one mapping (predicted cluster -> true label):")
    for c in range(n_clusters):
        print(f"  Cluster {c} is mapped to true label: {mapping[c]}")

    # Map each predicted cluster to its corresponding true label using
the mapping
    mapped_pred = np.array([mapping[c] for c in pred_labels])

    # Compute overall purity (accuracy)
    overall_accuracy = np.sum(mapped_pred == true_labels) /
len(true_labels)
    print(f"\nOverall cluster purity (accuracy):
{overall_accuracy:.4f}")

    # Compute and display confusion matrix of true labels vs. mapped
predictions
    cm_mapped = confusion_matrix(true_labels, mapped_pred)
    all_labels = sorted(set(true_labels) | set(mapped_pred))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm_mapped,
display_labels=all_labels)
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix (True Labels vs. Mapped Cluster
Labels)")
    plt.show()

#######################################################################
########
# Main Function
#######################################################################
########
def main():
    train_dir = "./data/train"
    test_dir = "./data/test"

    # Load documents and true labels from both training and test
directories
    train_docs, train_labels = load_texts_and_labels(train_dir)
    test_docs, test_labels = load_texts_and_labels(test_dir)
    # Combine both datasets
    all_docs = train_docs + test_docs
    all_true_labels = np.concatenate((train_labels, test_labels))

    print(f"Loaded {len(all_docs)} documents from both train and test 
sets.")
    print("True label distribution:", 
dict(zip(*np.unique(all_true_labels, return_counts=True))))

    # Analyze KMeans clustering with one-to-one mapping and confusion 
matrix
    analyze_kmeans_clusters(all_docs, all_true_labels, n_clusters=None, 
top_n=5)

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