Technical Assessment for AI intern role – Smart Data Solutions – ML TASK

SUBMITTED BY – NANDAKISHORE B

To solve this problem, I developed a multimodal document classification system that combines both image features from scanned TIF files and textual features extracted from OCR. I started by setting up the Python environment in Google Colab, installing the required libraries, and mounting Google Drive to access the dataset. The dataset was organized into class-specific folders, each containing images and their corresponding OCR text files.

For the **image preprocessing**, I extracted several statistical and structural features: mean and standard deviation of pixel intensity, histogram distributions (10 bins), edge density (via Canny edge detection), texture density (pixel transitions), and geometric properties like aspect ratio. For the **text preprocessing**, I applied TF-IDF vectorization with bigram support, a maximum of 5000 features, and English stop word removal. After preprocessing, I encoded the class labels, scaled the image features, and split the data into training and test sets with balanced class distribution.

I trained a **Random Forest Classifier** for the image features and a **Logistic Regression model** for the text features. Then, I created an **ensemble model** by combining their probability outputs with a weighted average (0.6 image, 0.4 text). This approach allowed me to benefit from the strengths of both modalities.

For **evaluation**, I measured accuracy, precision, recall, and F1-score for all three models (image, text, and ensemble). I also generated confusion matrices to visualize classification performance per class. Cross-validation was performed (5-fold) to check model stability across different data splits. Below are the generated figures for reference:

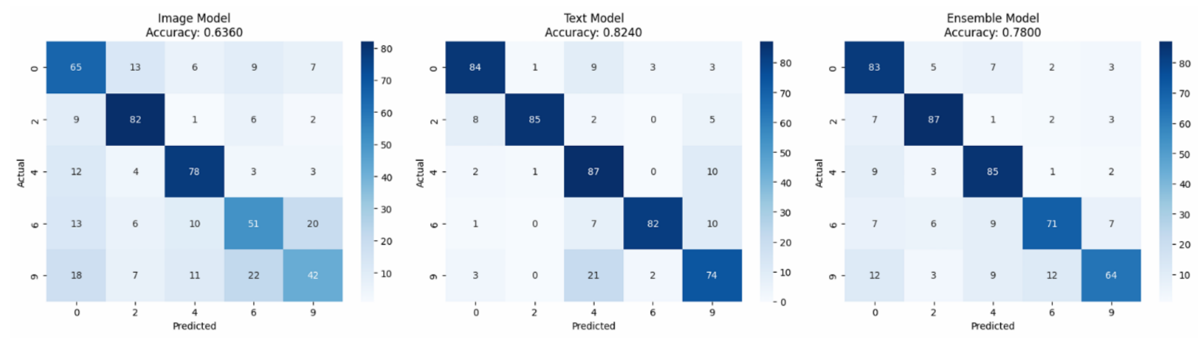

Image Model (Accuracy: 0.6360)

Actual \ Predicted	0	2	4	6	9
0	65	13	6	9	7
2	9	82	1	6	2
4	12	4	78	3	3
6	13	6	10	51	20
9	18	7	11	22	42

Text Model (Accuracy: 0.8240)

Actual \ Predicted	0	2	4	6	9
0	84	1	9	3	3
2	8	85	2	0	5
4	2	1	87	0	10
6	1	0	7	82	10
9	3	0	21	2	74

Ensemble Model (Accuracy: 0.7800)

Actual \ Predicted	0	2	4	6	9
0	83	5	7	2	3
2	7	87	1	2	3
4	9	3	85	1	2
6	7	6	9	71	7
9	12	3	9	12	64

Figure 1: Model performance comparison (Accuracy, Precision, Recall, F1-score)

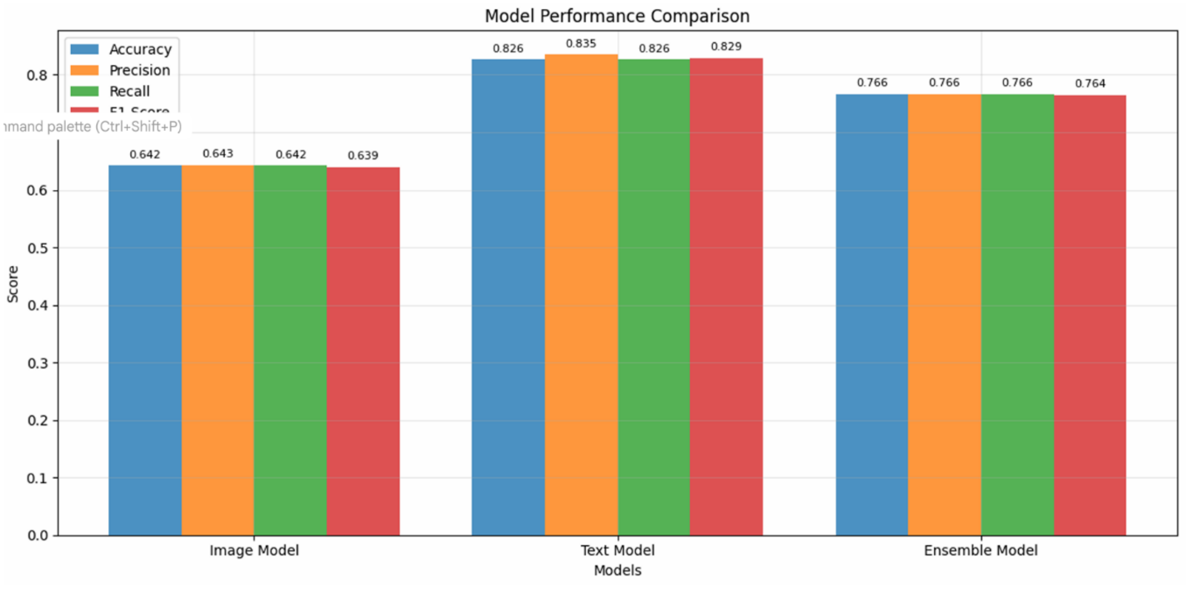

Figure 2: Confusion matrices for Image Model, Text Model, and Ensemble Model

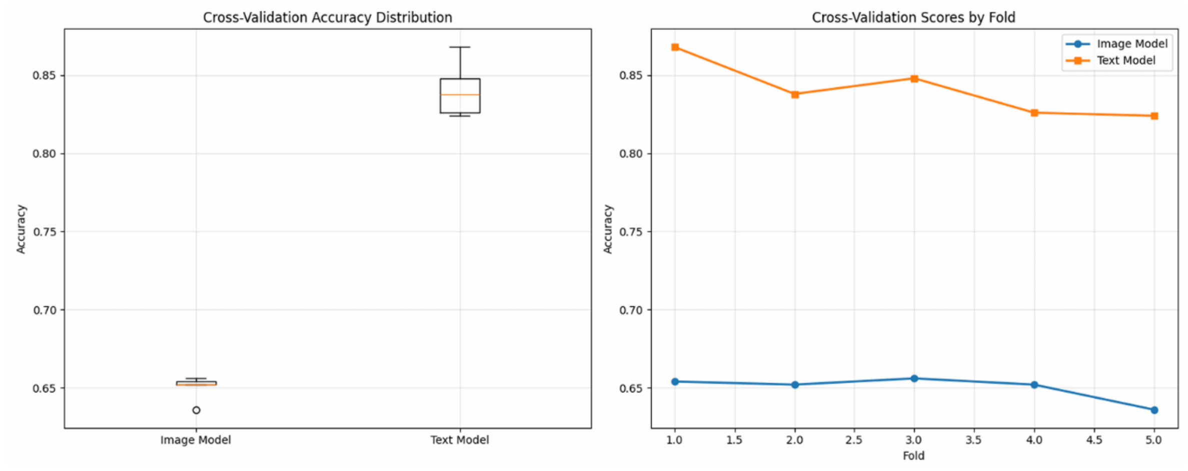

Figure 3: Cross-validation accuracy distribution and fold-wise performance

From the results, the text model performed best with around **82% accuracy**, followed by the ensemble at **78%**, and the image model at **64%**. While the ensemble was slightly lower than the text model, it provided more robustness by considering both modalities.

I chose this method because **text and images capture different aspects** of the documents: OCR text captures the semantic meaning, while image features capture formatting, layout, and patterns that text alone cannot represent. Using separate models for each allowed independent optimization, and combining them helped reduce weaknesses from relying on only one type of feature.

Challenges faced included handling empty or corrupted OCR text files, managing the imbalance in feature sizes between image (20 features) and text (5000 features), and the long processing time for extracting image features from 2500 files. Choosing the ensemble weights also required manual testing due to time constraints.

For a more optimized approach, I would implement deep learning architectures that leverage the complementary strengths of visual and textual modalities. This would involve using Convolutional Neural Networks (CNNs) to extract rich visual embeddings from images, capturing spatial hierarchies and visual patterns that traditional feature extraction methods might miss. Simultaneously, transformer-based models like BERT would process textual data to understand semantic relationships, context, and linguistic nuances.
The architecture would feature separate encoders for each modality - a CNN backbone (such as ResNet or EfficientNet) pre-trained on ImageNet for image feature extraction, and a pre-trained BERT model for text encoding. These modality-specific embeddings would then be fused through attention mechanisms or concatenation layers, allowing the model to learn cross-modal relationships and dependencies.
To optimize ensemble performance, I would implement automated weight optimization using techniques like Bayesian optimization or grid search with cross-validation, rather than manual tuning. This would systematically explore the hyperparameter space to find optimal combinations of learning rates, batch sizes, dropout rates, and fusion strategies.

I will also provide the code for the optimized approach below --

```python
"""
Multimodal Document Classifier (CNN image embeddings + Transformer text embeddings)
Improvements implemented:
 - Image augmentation + ResNet50 feature extractor (optional fine-tuning)
 - DistilBERT for text embeddings
 - Classifier training with hyperparameter tuning examples
 - Automated ensemble weight search (grid search)
 - Error analysis and saving of models / scalers / vectorizers
 - FastAPI endpoint for serving predictions

Usage:
 - Update BASE_PATH and OCR_PATH to your dataset paths
 - Run in Colab or any Python env with GPU if fine-tuning
"""

import os
import numpy as np
import joblib
from pathlib import Path
from tqdm.auto import tqdm

# Torch + torchvision
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms, models
from PIL import Image

# Transformers for text embeddings
from transformers import AutoTokenizer, AutoModel

# scikit-learn
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, RandomizedSearchCV, ParameterGrid
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.preprocessing import StandardScaler, LabelEncoder

# FastAPI
from fastapi import FastAPI
from pydantic import BaseModel
import uvicorn

# ----------------------
# Config / Hyperparams
# ----------------------
BASE_PATH = "/content/drive/MyDrive/SDS Task2/ML/data/images"  # folder per class with images
OCR_PATH = "/content/drive/MyDrive/SDS Task2/ML/data/ocr"     # folder per class with txt files
BATCH_SIZE = 32
NUM_WORKERS = 2
IMG_SIZE = 224
EMBED_DIM_IMG = 2048  # ResNet50 avgpool output
EMBED_DIM_TXT = 768   # DistilBERT hidden size
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
TRAIN_IMAGE_ENCODER = False  # set True to fine-tune CNN
SEED = 42

# Where to save artifacts
MODEL_DIR = Path("./models")
MODEL_DIR.mkdir(parents=True, exist_ok=True)

# ----------------------
# Utilities
# ----------------------
torch.manual_seed(SEED)
np.random.seed(SEED)

def list_image_files(base_path):
    files = []
    classes = sorted([d for d in os.listdir(base_path) if os.path.isdir(os.path.join(base_path, d))])
    for cls in classes:
        class_dir = os.path.join(base_path, cls)
        for fname in os.listdir(class_dir):
            if fname.lower().endswith((".tif", ".tiff", ".png", ".jpg", ".jpeg")):
                files.append((os.path.join(class_dir, fname), cls, fname))
    return files, classes

# ----------------------
# Dataset
# ----------------------
class DocDataset(Dataset):
    def __init__(self, rows, ocr_root, transform=None, tokenizer=None, max_len=128):
        """
        rows: list of tuples (img_path, class_label, filename)
        ocr_root: path to OCR folders (class/filename.txt)
        transform: torchvision transforms for images
        tokenizer: transformers tokenizer for text
        """
        self.rows = rows
        self.ocr_root = ocr_root
        self.transform = transform
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def _load_ocr(self, cls, fname):
        # OCR stored as filename + '.txt' inside class folder
        path = os.path.join(self.ocr_root, cls, fname + ".txt")
        if os.path.exists(path):
            try:
                with open(path, "r", encoding="utf-8", errors="ignore") as f:
                    return f.read().strip()
            except:
                return ""
        return ""

    def __getitem__(self, idx):
        img_path, cls, fname = self.rows[idx]
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        text = self._load_ocr(cls, fname)
        if self.tokenizer:
            encoded = self.tokenizer(
                text,
                padding="max_length",
                truncation=True,
                max_length=self.max_len,
                return_tensors="pt",
            )
            # squeeze to remove batch dim
            for k in encoded:
                encoded[k] = encoded[k].squeeze(0)
        else:
            encoded = None
        return {
            "image": image,
            "text_enc": encoded,
            "label": cls,
            "img_path": img_path,
            "fname": fname
        }

# ----------------------
# Transforms and Augmentation
# ----------------------
train_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(5),
    transforms.ColorJitter(brightness=0.1, contrast=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

eval_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

# ----------------------
# Feature extractors
# ----------------------
def get_resnet_feature_extractor(trainable=False):
    model = models.resnet50(pretrained=True)
    # remove classifier head
    modules = list(model.children())[:-1]  # remove fc layer
    feat_net = nn.Sequential(*modules)
    for p in feat_net.parameters():
        p.requires_grad = trainable
    feat_net.eval()
    feat_net.to(DEVICE)
    return feat_net

print("Loading ResNet50 feature extractor...")
img_encoder = get_resnet_feature_extractor(trainable=TRAIN_IMAGE_ENCODER)

print("Loading DistilBERT tokenizer + model...")
TXT_MODEL_NAME = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(TXT_MODEL_NAME)
txt_model = AutoModel.from_pretrained(TXT_MODEL_NAME).to(DEVICE)
txt_model.eval()

# ----------------------
# Extract embeddings helper (batched)
# ----------------------
def extract_image_embeddings(dataloader, encoder):
    encoder.eval()
    embeddings = []
    paths = []
    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Image Embeddings"):
            imgs = batch["image"].to(DEVICE)
            feat = encoder(imgs)            # shape [B, 2048, 1, 1]
            feat = feat.view(feat.size(0), -1).cpu().numpy()  # [B, 2048]
            embeddings.append(feat)
            paths.extend(batch["img_path"])
    embeddings = np.vstack(embeddings)
    return embeddings, paths

def extract_text_embeddings(dataloader, model, tokenizer):
    model.eval()
    embeddings = []
    texts = []
    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Text Embeddings"):
            enc = batch["text_enc"]
            # enc is a dict of tensors
            input_ids = enc["input_ids"].to(DEVICE)
            attention_mask = enc["attention_mask"].to(DEVICE)
            out = model(input_ids=input_ids, attention_mask=attention_mask)
            # use last_hidden_state[:,0,:] as CLS-like pooling for DistilBERT
            cls_emb = out.last_hidden_state[:, 0, :].cpu().numpy()  # [B, hidden]
            embeddings.append(cls_emb)
            # accumulate text (for debugging)
            texts.extend([None] * cls_emb.shape[0])
    embeddings = np.vstack(embeddings)
    return embeddings, texts

# ----------------------
# Load file list and create datasets
# ----------------------
rows, class_names = list_image_files(BASE_PATH)
print(f"Found {len(rows)} images across {len(class_names)} classes: {class_names}")

# split into train/val/test
train_val_rows, test_rows = train_test_split(rows, test_size=0.2, random_state=SEED, stratify=[r[1] for r in rows])
train_rows, val_rows = train_test_split(train_val_rows, test_size=0.2, random_state=SEED, stratify=[r[1] for r in train_val_rows])

train_ds = DocDataset(train_rows, OCR_PATH, transform=train_transform, tokenizer=tokenizer)
val_ds = DocDataset(val_rows, OCR_PATH, transform=eval_transform, tokenizer=tokenizer)
test_ds = DocDataset(test_rows, OCR_PATH, transform=eval_transform, tokenizer=tokenizer)

train_loader_img = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader_img = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
test_loader_img = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)

train_loader_txt = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader_txt = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
test_loader_txt = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)

# ----------------------
# Extract embeddings
# ----------------------
print("Extracting image embeddings ...")
X_img_train, _ = extract_image_embeddings(train_loader_img, img_encoder)
X_img_val, _ = extract_image_embeddings(val_loader_img, img_encoder)
X_img_test, _ = extract_image_embeddings(test_loader_img, img_encoder)

print("Extracting text embeddings ...")
X_txt_train, _ = extract_text_embeddings(train_loader_txt, txt_model, tokenizer)
X_txt_val, _ = extract_text_embeddings(val_loader_txt, txt_model, tokenizer)
```

```
X_txt_val, _ = extract_text_embeddings(val_loader_txt, txt_model, tokenizer)
X_txt_test, _ = extract_text_embeddings(test_loader_txt, txt_model, tokenizer)

# Encode labels
le = LabelEncoder()
y_train = le.fit_transform([r[1] for r in train_rows])
y_val = le.transform([r[1] for r in val_rows])
y_test = le.transform([r[1] for r in test_rows])

# Scale image features
img_scaler = StandardScaler()
X_img_train_scaled = img_scaler.fit_transform(X_img_train)
X_img_val_scaled = img_scaler.transform(X_img_val)
X_img_test_scaled = img_scaler.transform(X_img_test)

# Optionally reduce text dim with PCA/UMAP if needed (not done here)
print(f"Image embed shape: {X_img_train_scaled.shape}, Text embed shape: {X_txt_train.shape}")

# ----------------------
# Train classifiers
# ----------------------
# Image classifier (simple RF or small MLP)
img_clf = RandomForestClassifier(n_estimators=200, random_state=SEED, n_jobs=-1)
img_clf.fit(X_img_train_scaled, y_train)
img_val_pred = img_clf.predict(X_img_val_scaled)
img_val_acc = accuracy_score(y_val, img_val_pred)
print(f"Image classifier validation accuracy: {img_val_acc:.4f}")

# Text classifier (Logistic Regression with l2)
txt_clf = LogisticRegression(max_iter=1000, random_state=SEED)
txt_clf.fit(X_txt_train, y_train)
txt_val_pred = txt_clf.predict(X_txt_val)
txt_val_acc = accuracy_score(y_val, txt_val_pred)
print(f"Text classifier validation accuracy: {txt_val_acc:.4f}")

# Hyperparameter tuning example (randomized search for text clf)
# Comment/uncomment as needed - can be slow
# param_dist = {"C": [0.01, 0.1, 1, 10, 100]}
# rsearch = RandomizedSearchCV(LogisticRegression(max_iter=1000), param_dist, n_iter=5, cv=3, scoring='accuracy', n_jobs=1, random_state=
# rsearch.fit(X_txt_train, y_train)
# print("Best text params:", rsearch.best_params_)
# txt_clf = rsearch.best_estimator_

# ----------------------
# Ensemble weight optimization (grid search over weights)
# ----------------------
img_probs_val = img_clf.predict_proba(X_img_val_scaled)
txt_probs_val = txt_clf.predict_proba(X_txt_val)

best_w = None
best_acc = -1.0
best_pred = None
weight_grid = np.linspace(0.0, 1.0, 11)  # 0.0..1.0 step 0.1 for image weight
for w in weight_grid:
    ensemble_proba = w * img_probs_val + (1 - w) * txt_probs_val
    preds = np.argmax(ensemble_proba, axis=1)
    acc = accuracy_score(y_val, preds)
    if acc > best_acc:
        best_acc = acc
        best_w = w
        best_pred = preds

print(f"Best ensemble weight (image): {best_w}, val acc: {best_acc:.4f}")

# Evaluate on test set using best weight
img_probs_test = img_clf.predict_proba(X_img_test_scaled)
txt_probs_test = txt_clf.predict_proba(X_txt_test)
ensemble_proba_test = best_w * img_probs_test + (1 - best_w) * txt_probs_test
ensemble_pred_test = np.argmax(ensemble_proba_test, axis=1)

# ----------------------
# Metrics & Error analysis
# ----------------------
print("\n--- Test Performance ---")
print("Image model:")
print(classification_report(y_test, img_clf.predict(X_img_test_scaled), target_names=le.classes_))
print("Text model:")
print(classification_report(y_test, txt_clf.predict(X_txt_test), target_names=le.classes_))
print("Ensemble model:")
print(classification_report(y_test, ensemble_pred_test, target_names=le.classes_))

cm = confusion_matrix(y_test, ensemble_pred_test)
print("Ensemble Confusion Matrix:\n", cm)
[illegible]
```

```
print("Ensemble Accuracy:", accuracy_score(y_test, ensemble_pred_test))

# Optionally save confusion matrix figure using matplotlib in your doc

# ----------------------
# Save artifacts
# ----------------------
joblib.dump(img_clf, MODEL_DIR / "img_clf_rf.pkl")
joblib.dump(txt_clf, MODEL_DIR / "txt_clf_lr.pkl")
joblib.dump(img_scaler, MODEL_DIR / "img_scaler.pkl")
joblib.dump(le, MODEL_DIR / "label_encoder.pkl")
# Save weight config and info
joblib.dump({"img_weight": best_w, "txt_weight": 1 - best_w}, MODEL_DIR / "ensemble_config.pkl")

print("Models and artifacts saved to", MODEL_DIR)

# ----------------------
# Minimal FastAPI for serving
# ----------------------
# Note: In production you should load models once at startup and handle concurrency.
app = FastAPI()

class PredictRequest(BaseModel):
    image_path: str = None   # or allow upload
    ocr_text: str = ""

# load artifacts (for API)
IMG_MODEL = joblib.load(MODEL_DIR / "img_clf_rf.pkl")
TXT_MODEL = joblib.load(MODEL_DIR / "txt_clf_lr.pkl")
IMG_SCALER = joblib.load(MODEL_DIR / "img_scaler.pkl")
LABEL_ENCODER = joblib.load(MODEL_DIR / "label_encoder.pkl")
ENSEMBLE_CFG = joblib.load(MODEL_DIR / "ensemble_config.pkl")

def preprocess_single_image(path):
    img = Image.open(path).convert("RGB")
    img_t = eval_transform(img).unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        feat = img_encoder(img_t).view(1, -1).cpu().numpy()
    feat_scaled = IMG_SCALER.transform(feat)
    return feat_scaled

def preprocess_single_text(text):
    enc = tokenizer(text, truncation=True, padding='max_length', max_length=128, return_tensors='pt')
    input_ids = enc['input_ids'].to(DEVICE)
    attention_mask = enc['attention_mask'].to(DEVICE)
    with torch.no_grad():
        out = txt_model(input_ids=input_ids, attention_mask=attention_mask)
        emb = out.last_hidden_state[:, 0, :].cpu().numpy()
    return emb

@app.post("/predict")
def predict(req: PredictRequest):
    # Basic inference path - expects image_path to be accessible by server
    if req.image_path:
        try:
            img_feat = preprocess_single_image(req.image_path)
            img_proba = IMG_MODEL.predict_proba(img_feat)[0]
        except Exception as e:
            return {"error": f"Image processing failed: {e}"}
    else:
        img_proba = np.ones(len(LABEL_ENCODER.classes_)) / len(LABEL_ENCODER.classes_)  # fallback uniform

    if req.ocr_text:
        txt_feat = preprocess_single_text(req.ocr_text)
        txt_proba = TXT_MODEL.predict_proba(txt_feat)[0]
    else:
        txt_proba = np.ones(len(LABEL_ENCODER.classes_)) / len(LABEL_ENCODER.classes_)

    w = ENSEMBLE_CFG["img_weight"]
    ensemble_proba = w * img_proba + (1 - w) * txt_proba
    pred_idx = int(np.argmax(ensemble_proba))
    pred_label = LABEL_ENCODER.classes_[pred_idx]
    confidence = float(ensemble_proba[pred_idx])

    return {
        "predicted_class": pred_label,
        "confidence": confidence,
        "probabilities": {c: float(p) for c, p in zip(LABEL_ENCODER.classes_, ensemble_proba)}
    }

# To run the API (uncomment below when running as script)
# if __name__ == "__main__":
#     uvicorn.run(app, host="0.0.0.0", port=8000)
```