IMDB Sentiment Classification: Model Benchmarking & Fine-Tuning

We experimented with the **IMDB 50K movie reviews dataset** to perform binary sentiment classification (positive vs. negative). To identify the best lightweight model for CPU execution, we:

1. **Prepared the dataset** from CSV, mapped sentiments to binary labels, and created a subset for fast experimentation.
2. **Fine-tuned 5 transformer models** (DistilBERT, TinyBERT, SmallBERT, ALBERT, MobileBERT) on the subset using a **custom F1 score** for evaluation.
3. **Selected the best-performing model** based on F1 score and retrained it on the **entire IMDB dataset**.
4. **Evaluated performance** on a held-out test set and reported precision, recall, and F1.
5. **Ran inference** on 10 randomly sampled reviews from the test set to demonstrate predictions.

This pipeline balances **speed and accuracy on CPU**, ensuring quick benchmarking while still scaling to the full dataset with the best model.

```
import os
import random
import math
import argparse
from dataclasses import dataclass
from typing import Dict, List
import numpy as np
import torch
from datasets import load_dataset, Dataset, DatasetDict
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    DataCollatorWithPadding,
    set_seed
)
from sklearn.metrics import f1_score, precision_score, recall_score

# -----------------------
# User-tunable / speed knobs
# -----------------------
NUM_EXAMPLES_SUBSET = 5000   # subset to quickly compare models (train+val)
NUM_VAL_SUBSET = 1000
NUM_EPOCHS_SUBSET = 2       # small epochs for fast runs
NUM_EPOCHS_FULL = 2         # when training best model on full dataset
BATCH_SIZE = 16             # CPU-friendly (decrease if OOM)
SEED = 42
MAX_LENGTH = 256            # max tokens (shorter = faster)
NUM_THREADS = 4             # torch threads
OUTPUT_DIR = "imdb_finetune_output"

# 5 CPU-friendly models to compare
MODEL_NAMES = [
    "distilbert-base-uncased",        # balanced speed & quality
    "prajjwal1/bert-tiny",            # tiny BERT (very fast)
    "prajjwal1/bert-small",           # small BERT
    "albert-base-v2",                 # parameter-efficient
    "google/mobilebert-uncased"       # mobileBERT (optimized)
]

# -----------------------
# Environment setup
# -----------------------
torch.set_num_threads(NUM_THREADS)
set_seed(SEED)
os.makedirs(OUTPUT_DIR, exist_ok=True)


def load_local_imdb_csv(csv_path: str = "IMDB_Dataset.csv"):
    """
    Expects CSV with columns 'review' and 'sentiment' (positive/negative).
    This Kaggle dataset uses these names.
    """
    if not os.path.exists(csv_path):
        raise FileNotFoundError(
            f"{csv_path} not found. Download the Kaggle dataset and place IMDB_Dataset.csv here."
        )
    ds = load_dataset("csv", data_files=csv_path)["train"]
    # rename to standard names if necessary
    if "review" not in ds.column_names or "sentiment" not in ds.column_names:
        raise ValueError("CSV must contain 'review' and 'sentiment' columns.")
    # map label string to int
    def map_label(example):
        example["label"] = 1 if example["sentiment"].lower().startswith("pos") else 0
        return example
    ds = ds.map(map_label)
    ds = ds.remove_columns([c for c in ds.column_names if c not in ("review", "label")])
    return ds

def prepare_datasets(full_ds, subset_size=NUM_EXAMPLES_SUBSET, val_size=NUM_VAL_SUBSET):
    """
    Create a quick subset for model selection:
      sample subset_size + val_size rows, then split into train/val/test.
    Also returns a held-out test set (20% of original).
    """
    # shuffle full dataset deterministically
    full_shuffled = full_ds.shuffle(seed=SEED)
    # held-out test set: 10k or 20% whichever smaller
    test_count = min(int(len(full_shuffled) * 0.2), 10000)
    test_ds = full_shuffled.select(range(test_count))
    # For subset training, sample from remainder
    remaining = full_shuffled.select(range(test_count, len(full_shuffled)))
    subset_total = subset_size + val_size
    subset_total = min(subset_total, len(remaining))
    subset = remaining.select(range(subset_total))
    # split
    train_subset = subset.select(range(subset_size))
    val_subset = subset.select(range(subset_size, subset_total))
    return DatasetDict({"train": train_subset, "validation": val_subset, "test": test_ds})

def tokenize_function(examples, tokenizer):
    return tokenizer(examples["review"], truncation=True, max_length=MAX_LENGTH)
```

```
def compute_metrics_custom(pred):
    labels = pred.label_ids
    preds = np.argmax(pred.predictions, axis=1)
    f1 = f1_score(labels, preds, average="binary")
    prec = precision_score(labels, preds, zero_division=0)
    rec = recall_score(labels, preds, zero_division=0)
    return {"f1": f1, "precision": prec, "recall": rec}



def train_and_eval_model(model_name: str, datasets: DatasetDict, tokenizer=None, epochs=NUM_EPOCHS_SUBSET, run_name=None):
    print(f"\n--- Training {model_name} (epochs={epochs}) ---")
    if tokenizer is None:
        tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
    # tokenize
    tokenized_train = datasets["train"].map(lambda x: tokenize_function(x, tokenizer), batched=True, remove_columns=datasets["train"].column_names)
    tokenized_val = datasets["validation"].map(lambda x: tokenize_function(x, tokenizer), batched=True, remove_columns=datasets["validation"].column_names)
    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

    # load model (num_labels=2)
    model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)

    training_args = TrainingArguments(
        output_dir=os.path.join(OUTPUT_DIR, f"{run_name or model_name.replace('/', '_')}"),
        num_train_epochs=epochs,
        per_device_train_batch_size=BATCH_SIZE,
        per_device_eval_batch_size=BATCH_SIZE,
        evaluation_strategy="epoch",
        save_strategy="no",         # do not save checkpoints for subset runs (speed)
        logging_strategy="epoch",
        learning_rate=2e-5,
        weight_decay=0.01,
        seed=SEED,
        disable_tqdm=False,
        dataloader_drop_last=False,
        fp16=False  # CPU -> no mixed precision
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_train,
        eval_dataset=tokenized_val,
        tokenizer=tokenizer,
        data_collator=data_collator,
        compute_metrics=compute_metrics_custom
    )

    trainer.train()
    metrics = trainer.evaluate()
    # clean up to free CPU memory
    del model
    torch.cuda.empty_cache()
    return metrics["eval_f1"], metrics


def train_best_on_full(model_name: str, full_ds, tokenizer=None, epochs=NUM_EPOCHS_FULL):
print(f"\n=== Final training on full dataset with {model_name} ===")
if tokenizer is None:
    tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
# Split full dataset into train/validation/test (80/10/10)
ds_shuf = full_ds.shuffle(seed=SEED)
n = len(ds_shuf)
n_train = int(n * 0.8)
n_val = int(n * 0.1)
train_full = ds_shuf.select(range(0, n_train))
val_full = ds_shuf.select(range(n_train, n_train + n_val))
test_full = ds_shuf.select(range(n_train + n_val, n))
tokenized_train = train_full.map(lambda x: tokenize_function(x, tokenizer), batched=True, remove_columns=train_full.column_names)
tokenized_val = val_full.map(lambda x: tokenize_function(x, tokenizer), batched=True, remove_columns=val_full.column_names)
tokenized_test = test_full.map(lambda x: tokenize_function(x, tokenizer), batched=True, remove_columns=test_full.column_names)

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)

training_args = TrainingArguments(
    output_dir=os.path.join(OUTPUT_DIR, f"final_{model_name.replace('/', '_')}"),
    num_train_epochs=epochs,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    weight_decay=0.01,
    seed=SEED,
    disable_tqdm=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_val,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics_custom
)

trainer.train()
eval_metrics = trainer.evaluate(eval_dataset=tokenized_test)
# Save final model
trainer.save_model(os.path.join(OUTPUT_DIR, "best_full_model"))
return trainer, eval_metrics, tokenized_test


def run_inference(trainer: Trainer, tokenizer, raw_test_dataset: Dataset, sample_n=10):
    print("\n--- Running inference on random samples ---")
    rng = random.Random(SEED)
    indices = rng.sample(range(len(raw_test_dataset)), k=min(sample_n, len(raw_test_dataset)))
    samples = [raw_test_dataset[i] for i in indices]
    texts = [s["review"] for s in samples]
    enc = tokenizer(texts, truncation=True, max_length=MAX_LENGTH, padding=True, return_tensors="pt")
    # move to CPU explicitly
    outputs = trainer.model(**{k: v for k, v in enc.items()})
```

```
    logits = outputs.logits.detach().cpu().numpy()
    preds = np.argmax(logits, axis=1)
    for i, txt in enumerate(texts):
        print(f"\n--- Sample {i+1} ---")
        print("Review:", txt[:400].replace("\n", " ").strip())
        print("Predicted label:", "positive" if preds[i]==1 else "negative")
        print("Logits:", logits[i])

# -----------------------
# Main pipeline
# -----------------------
def main():
    print("Loading dataset (local CSV)...")
    full_ds = load_local_imdb_csv("/content/IMDB Dataset.csv")  # raises if not present
    print(f"Total examples loaded: {len(full_ds)}")

    print("Preparing subset + test split for model selection...")
    datasets_sub = prepare_datasets(full_ds, subset_size=NUM_EXAMPLES_SUBSET, val_size=NUM_VAL_SUBSET)
    print("Subset sizes:", {k: len(datasets_sub[k]) for k in datasets_sub})

    # Iterate models (quick)
    model_scores = {}
    model_details = {}
    for model_name in MODEL_NAMES:
        try:
            tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
        except Exception as e:
            print(f"Tokenizer load failed for {model_name}: {e}. Skipping.")
            continue
    try:
        f1, metrics = train_and_eval_model(model_name, datasets_sub, tokenizer=tokenizer, epochs=NUM_EPOCHS_SUBSET, run_name=f"subset_{model_name.replace('/', '_')}")
        model_scores[model_name] = f1
        model_details[model_name] = metrics
        print(f"Model {model_name} subset F1 = {f1:.4f}")
    except Exception as e:
        print(f"Training failed for {model_name}: {e}. Skipping.")

if not model_scores:
    print("No models finished training successfully. Exiting.")
    return

# Choose best model by F1
best_model = max(model_scores.items(), key=lambda x: x[1])[0]
print(f"\nBest model on subset: {best_model} with F1={model_scores[best_model]:.4f}")

# Final training on full dataset with best model
tokenizer_best = AutoTokenizer.from_pretrained(best_model, use_fast=True)
trainer, eval_metrics, tokenized_test = train_best_on_full(best_model, full_ds, tokenizer=tokenizer_best, epochs=NUM_EPOCHS_FULL)
print(f"\nFinal test metrics (best model on full data): {eval_metrics}")

# Run inference on 10 random samples from test set
# Need raw test dataset (un-tokenized). In train_best_on_full we created tokenized_test; get the raw test split:
# We can reconstruct raw test by splitting full_ds again deterministically:
ds_shuf = full_ds.shuffle(seed=SEED)
n = len(ds_shuf)
n_train = int(n * 0.8)
n_val = int(n * 0.1)
    test_raw = ds_shuf.select(range(n_train + n_val, n))
    run_inference(trainer, tokenizer_best, test_raw, sample_n=10)
    print("\nDone.")

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

```
Loading dataset (local CSV)...
Generating train split:        50000/0 [00:01<00:00, 31424.02 examples/s]
Map: 100%                                                  50000/50000 [00:03<00:00, 19083.28 examples/s]
Total examples loaded: 50000
Preparing subset + test split for model selection...
Subset sizes: {'train': 5000, 'validation': 1000, 'test': 10000}
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%                                  48.0/48.0 [00:00<00:00, 4.63kB/s]
config.json: 100%                                          483/483 [00:00<00:00, 22.8kB/s]
vocab.txt: 100%                                          232k/232k [00:00<00:00, 4.80MB/s]
tokenizer.json: 100%                                       466k/466k [00:00<00:00, 8.65MB/s]

--- Training distilbert-base-uncased (epochs=2) ---
Map: 100%                                                  5000/5000 [00:10<00:00, 479.02 examples/s]
Map: 100%                                                  1000/1000 [00:01<00:00, 940.19 examples/s]
model.safetensors: 100%                                      268M/268M [00:02<00:00, 145MB/s]
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.we
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Training failed for distilbert-base-uncased: TrainingArguments.__init__() got an unexpected keyword argument 'evaluation_strategy'. Skipping.
config.json: 100%                                          285/285 [00:00<00:00, 25.9kB/s]
vocab.txt:       232k/? [00:00<00:00, 12.5MB/s]

--- Training prajjwal1/bert-tiny (epochs=2) ---
Map: 100%                                                  5000/5000 [00:11<00:00, 405.02 examples/s]
Map: 100%                                                  1000/1000 [00:01<00:00, 529.97 examples/s]
pytorch_model.bin: 100%                                      17.8M/17.8M [00:00<00:00, 21.6MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at prajjwal1/bert-tiny and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Training failed for prajjwal1/bert-tiny: TrainingArguments.__init__() got an unexpected keyword argument 'evaluation_strategy'. Skipping.
config.json: 100%                                          286/286 [00:00<00:00, 3.38kB/s]
model.safetensors: 100%                                      17.7M/17.7M [00:00<00:00, 39.8MB/s]
vocab.txt:       232k/? [00:00<00:00, 4.73MB/s]

--- Training prajjwal1/bert-small (epochs=2) ---
Map: 100%                                                  5000/5000 [00:11<00:00, 448.39 examples/s]
Map: 100%                                                  1000/1000 [00:01<00:00, 543.65 examples/s]
pytorch_model.bin: 100%                                      116M/116M [00:02<00:00, 62.3MB/s]
```

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
def run_inference_visual(trainer: Trainer, tokenizer, raw_test_dataset: Dataset, sample_n=10):
    print("\n--- Running visual inference on random samples ---")
    rng = random.Random(SEED)
    indices = rng.sample(range(len(raw_test_dataset)), k=min(sample_n, len(raw_test_dataset)))
    samples = [raw_test_dataset[i] for i in indices]
```