In [ ]:
```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("lakshmi25npathi/imdb-dataset-of-50k-movie-reviews")

print("Path to dataset files:", path)
```

```
Using Colab cache for faster access to the 'imdb-dataset-of-50k-movie-reviews' dataset.
Path to dataset files: /kaggle/input/imdb-dataset-of-50k-movie-reviews
```

In [ ]:
```
!ls /kaggle/input/imdb-dataset-of-50k-movie-reviews
```

```
'IMDB Dataset.csv'
```

In [ ]:
```
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import re
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import matplotlib.pyplot as plt

# -------------------------
# 1️⃣ Device
# -------------------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# -------------------------
# 2️⃣ Load IMDB dataset
# -------------------------
df = pd.read_csv('/kaggle/input/imdb-dataset-of-50k-movie-reviews/IMDB Dataset.csv')

# Encode labels
le = LabelEncoder()
df['sentiment'] = le.fit_transform(df['sentiment'])

# -------------------------
# 3️⃣ Tokenize and Vocabulary
# -------------------------
def tokenize(text):
    return re.findall(r'\b\w+\b', text.lower())

df['tokens'] = df['review'].apply(tokenize)

all_tokens = [token for tokens in df['tokens'] for token in tokens]
vocab = {word: idx + 2 for idx, (word, _) in enumerate(Counter(all_tokens).items())}
vocab[''] = 0
vocab[''] = 1

def encode(tokens):
    return [vocab.get(token, 1) for token in tokens]

df['encoded'] = df['tokens'].apply(encode)

# -------------------------
# 4️⃣ Padding
# -------------------------
max_len = 300
def pad_sequence(seq):
    return seq[:max_len] + [0]*(max_len - len(seq)) if len(seq) < max_len else seq[:max_len]

df['padded'] = df['encoded'].apply(pad_sequence)

# -------------------------
# 5️⃣ Train/Test split
# -------------------------
X = np.array(df['padded'].tolist())
y = np.array(df['sentiment'].tolist())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train = torch.tensor(X_train, dtype=torch.long).to(device)
X_test = torch.tensor(X_test, dtype=torch.long).to(device)
y_train = torch.tensor(y_train, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test, dtype=torch.float32).to(device)

train_data = torch.utils.data.TensorDataset(X_train, y_train)
test_data = torch.utils.data.TensorDataset(X_test, y_test)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128)

# -------------------------
# 6️⃣ LSTM Model
# -------------------------
class LSTMClassifier(nn.Module):
```

```python
def encode(tokens):
    return [vocab.get(token, 1) for token in tokens]

df['encoded'] = df['tokens'].apply(encode)

# -------------------------
# 4️⃣ Padding
# -------------------------
max_len = 300
def pad_sequence(seq):
    return seq[:max_len] + [0]*(max_len - len(seq)) if len(seq) < max_len else seq[:max_len]

df['padded'] = df['encoded'].apply(pad_sequence)

# -------------------------
# 5️⃣ Train/Test split
# -------------------------
X = np.array(df['padded'].tolist())
y = np.array(df['sentiment'].tolist())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train = torch.tensor(X_train, dtype=torch.long).to(device)
X_test = torch.tensor(X_test, dtype=torch.long).to(device)
y_train = torch.tensor(y_train, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test, dtype=torch.float32).to(device)

train_data = torch.utils.data.TensorDataset(X_train, y_train)
test_data = torch.utils.data.TensorDataset(X_test, y_test)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128)

# -------------------------
# 6️⃣ LSTM Model
# -------------------------
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=200, hidden_dim=256, output_dim=1, n_layers=2, bidirectio
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers=n_layers, batch_first=True,
                            bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        _, (hidden, _) = self.lstm(x)
        if self.lstm.bidirectional:
            hidden = torch.cat((hidden[-2], hidden[-1]), dim=1)
        else:
            hidden = hidden[-1]
        out = self.fc(hidden)
        return self.sigmoid(out)

model = LSTMClassifier(vocab_size=len(vocab)).to(device)

# -------------------------
# 7️⃣ Training
# -------------------------
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=2)

epochs = 6
train_losses, test_losses = [], []

for epoch in range(epochs):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs).squeeze()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    train_loss = total_loss / len(train_loader)

    model.eval()
    total_loss = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, labels)
            total_loss += loss.item()
    test_loss = total_loss / len(test_loader)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    scheduler.step(test_loss)

    print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Test Loss: {test_loss:.4f}")

# -------------------------
# 8️⃣ Loss Curves
# -------------------------
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
```

8. Experiment using LSTM.

Predict next-day stock prices using LSTM

Aim: To build and evaluate an LSTM-based deep learning model capable of predicting future stock pri~~cing~~ based on past time-series data

objectives:-

1. to pre-process and normalize stock price data for model training
2. to create a sequential LSTM model using TensorFlow/Keras
3. To train and evaluate the model using a test dataset
4. To visualize and interpret model predictions.

3. Pseudo code:-

Begin

Load stock price dataset

Normalize data b/w 0 and 1

create time-series sequences.

splint dataset into training and testing sets.

Define LSTM model:-

Input layer: sequence of 60 time steps

LSTM layer with 50 units

Dense layer with 25 units

output layer with 1 neuron

compile model (optimizer = 'adam', loss = 'mean_squared_error')

Train model on training data (epochs = 50, batch_size = 32)

PREDICT on test data.

Inverse transform predictions to original price scale.

CALCULATE evaluation metrices (RMSE)

VISUALIZE true vs. predicted prices

observations:-

⇒ The LSTM successfully captures time dependencies in stock price movement

⇒ prediction lag is observed due to smoothing effect of LSTM memory.

⇒ model performance can be improved by:-

- Increasing epochs or LSTM layers
- using additional features (volume, open, high, low)
- using dropout regularization.

Result:-

Metric	Value.
RMSE	~ 3.42
Training	~3 mins
model Type	2-layer LSTM
Epochs	20.

Predicted vs Actual chart: clear upward and downward trend matching

output :-

After training for 20 epochs.

A line plot will show :-

- Blue line → Actual closing prices
- pencil line → predicted prices
- predictions follow the actual trend with small

Epoch 1/5 · step - loss : 0.0137

Epoch 2/5 step - loss : 2.8680e-04

3/5 step - loss : 2.5403e-04

4/5 step - loss : 2.3259e-04

5/5 step - loss 2.2624e-04.

LSTM Stock price prediction (AAPL)