Q1: Implementing a Basic Autoencoder

Task: Autoencoders learn to reconstruct input data by encoding it into a lower-dimensional space. You will build a **fully connected autoencoder** and evaluate its performance on image reconstruction.

1. Load the **MNIST dataset** using tensorflow.keras.datasets.
2. Define a **fully connected (Dense) autoencoder**:
   - Encoder: Input layer (784), hidden layer (32).
   - Decoder: Hidden layer (32), output layer (784).
3. Compile and train the autoencoder with **binary cross-entropy loss**.
4. Plot **original vs. reconstructed images** after training.
5. Modify the latent dimension size (e.g., 16, 64) and analyze how it affects the quality of reconstruction.

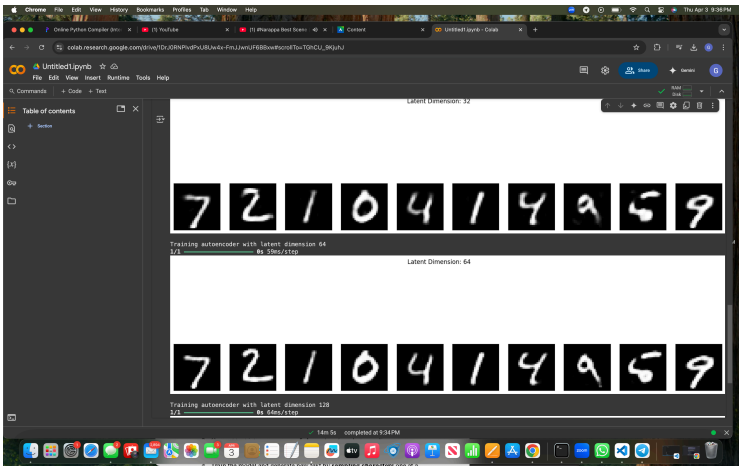
Hint: *Use Model() from tensorflow.keras.models and Dense() layers.*

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

# Load and preprocess MNIST data
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((len(x_train), 784))
x_test = x_test.reshape((len(x_test), 784))

# Define the autoencoder architecture
def build_autoencoder(latent_dim=32):
    # Encoder
    input_img = Input(shape=(784,))
    encoded = Dense(latent_dim, activation='relu')
(input_img)
```

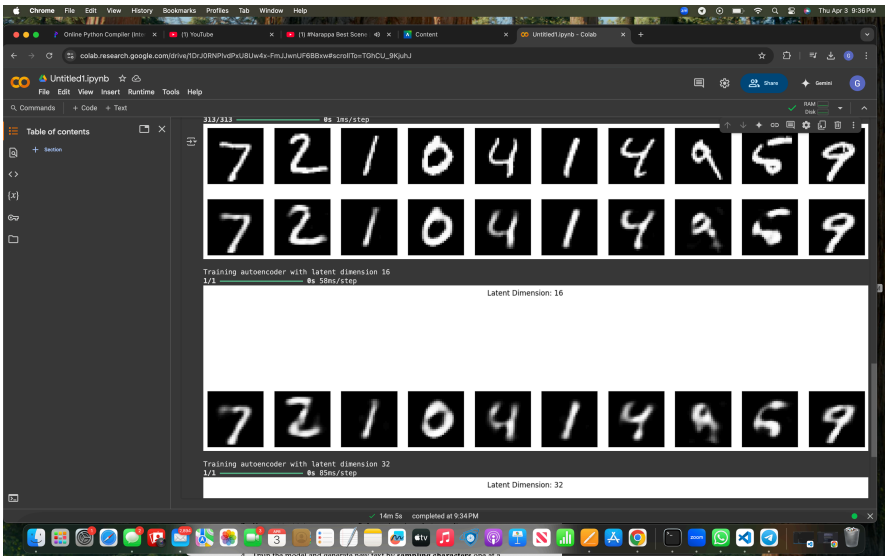
```
    # Decoder
    decoded = Dense(784, activation='sigmoid')(encoded)

    # Autoencoder model
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer=Adam(),
loss='binary_crossentropy')

    return autoencoder

# Train the autoencoder with latent dimension 32
autoencoder = build_autoencoder(latent_dim=32)
history = autoencoder.fit(x_train, x_train,
                          epochs=50,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test,
x_test))

# Reconstruct test images
decoded_imgs = autoencoder.predict(x_test)

# Plot original vs reconstructed images
n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
```

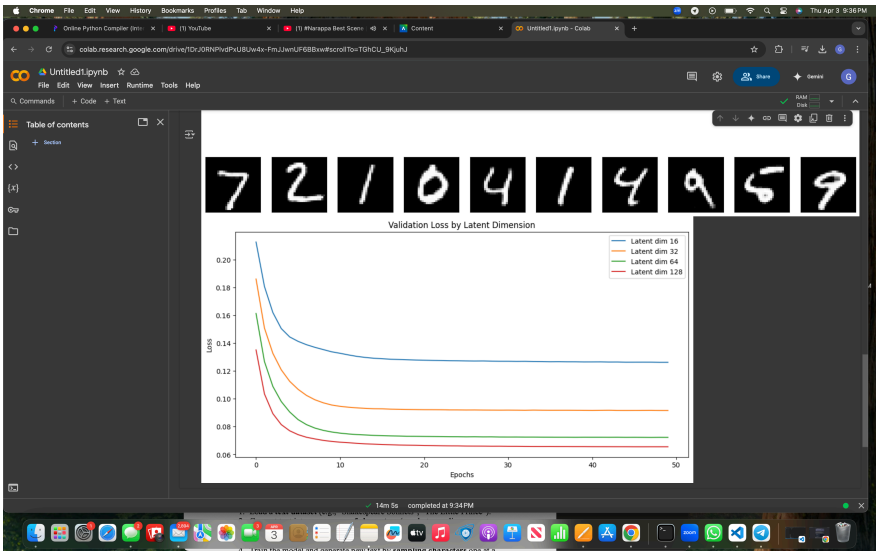
```
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()

# Now let's analyze different latent dimensions
latent_dims = [16, 32, 64, 128]
histories = []
reconstructions = []

for dim in latent_dims:
    print(f"\nTraining autoencoder with latent
dimension {dim}")
    model = build_autoencoder(latent_dim=dim)
    history = model.fit(x_train, x_train,
                        epochs=50,
                        batch_size=256,
                        shuffle=True,
                        validation_data=(x_test,
x_test),
                        verbose=0)
    histories.append(history)
    reconstructions.append(model.predict(x_test[:n]))

    # Plot sample reconstruction for this dimension
    plt.figure(figsize=(20, 4))
    plt.suptitle(f'Latent Dimension: {dim}', y=1.05)
    for i in range(n):
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(reconstructions[-1][i].reshape(28,
28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()

# Plot training curves for comparison
plt.figure(figsize=(12, 6))
for i, dim in enumerate(latent_dims):
```

```
    plt.plot(histories[i].history['val_loss'],
label=f'Latent dim {dim}')
plt.title('Validation Loss by Latent Dimension')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Home Assignment 3 Tadikonda Gnandeep Id st-70079736





Q2: Implementing a Denoising Autoencoder

Task: Denoising autoencoders can reconstruct clean data from noisy inputs. You will train a model to remove noise from images.

1. Modify the **basic autoencoder** from Q2 to a **denoising autoencoder** by adding **Gaussian noise** (mean=0, std=0.5) to input images.
2. Ensure that the **output remains the clean image** while training.
3. Train the model and visualize **noisy vs. reconstructed images**.
4. Compare the **performance of a basic vs. denoising autoencoder** in reconstructing images.
5. Explain one real-world scenario where denoising autoencoders can be useful (e.g., medical imaging, security).

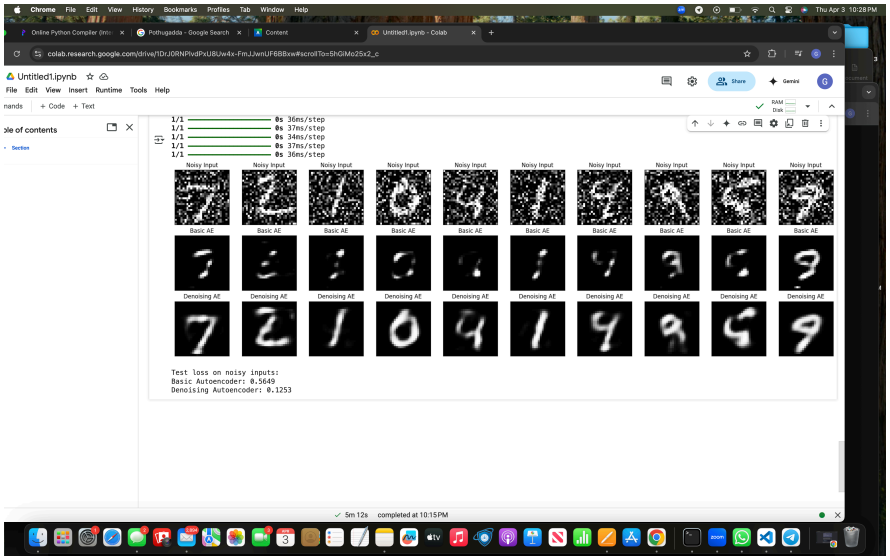
Hint: *Use np.random.normal() to add noise to images before training.*

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

## Load and preprocess MNIST data
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((len(x_train), 784))
x_test = x_test.reshape((len(x_test), 784))

## Add Gaussian noise to create noisy versions
def add_noise(images, mean=0, std=0.5):
    noise = np.random.normal(mean, std,
size=images.shape)
    noisy_images = images + noise
    # Clip to maintain valid pixel range [0,1]
    return np.clip(noisy_images, 0, 1)

x_train_noisy = add_noise(x_train)
```

```
x_test_noisy = add_noise(x_test)

## Build the denoising autoencoder (same architecture
as before)
def build_denoising_autoencoder(latent_dim=32):
    # Encoder
    input_img = Input(shape=(784,))
    encoded = Dense(latent_dim, activation='relu')
(input_img)

    # Decoder
    decoded = Dense(784, activation='sigmoid')(encoded)

    # Autoencoder model
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer=Adam(),
loss='binary_crossentropy')

    return autoencoder

## Train the denoising autoencoder
dae = build_denoising_autoencoder(latent_dim=32)
history_dae = dae.fit(x_train_noisy, x_train,  # Noisy
input, clean target
                      epochs=50,
                      batch_size=256,
                      shuffle=True,
                      validation_data=(x_test_noisy,
x_test))

## For comparison, let's also train a basic autoencoder
basic_ae = build_denoising_autoencoder(latent_dim=32)
history_basic = basic_ae.fit(x_train, x_train,  # Clean
input and target
                             epochs=50,
                             batch_size=256,
                             shuffle=True,
```

```
                                validation_data=(x_test,
x_test))

## Visualize results
def plot_comparisons(models, titles, test_images,
n=10):
    plt.figure(figsize=(20, 6))
    for i in range(n):
        # Display noisy input
        ax = plt.subplot(3, n, i + 1)
        plt.imshow(test_images[i].reshape(28, 28))
        plt.title("Noisy Input")
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        for j, model in enumerate(models):
            # Display reconstruction
            ax = plt.subplot(3, n, (j+1)*n + i + 1)
            reconstructed =
model.predict(test_images[i:i+1])
            plt.imshow(reconstructed.reshape(28, 28))
            plt.title(titles[j])
            plt.gray()
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
    plt.tight_layout()
    plt.show()

# Compare basic AE vs denoising AE on noisy inputs
plot_comparisons(
    models=[basic_ae, dae],
    titles=["Basic AE", "Denoising AE"],
    test_images=x_test_noisy
)

## Quantitative comparison
```

```
basic_loss = basic_ae.evaluate(x_test_noisy, x_test,
verbose=0)
dae_loss = dae.evaluate(x_test_noisy, x_test,
verbose=0)
print(f"\nTest loss on noisy inputs:")
print(f"Basic Autoencoder: {basic_loss:.4f}")
print(f"Denoising Autoencoder: {dae_loss:.4f}")
```



Q3: Implementing an RNN for Text Generation

Task: Recurrent Neural Networks (RNNs) can generate sequences of text. You will train an **LSTM-based RNN** to predict the next character in a given text dataset.

1. Load a **text dataset** (e.g., "Shakespeare Sonnets", "The Little Prince").
2. Convert text into a **sequence of characters** (one-hot encoding or embeddings).
3. Define an **RNN model** using LSTM layers to predict the next character.

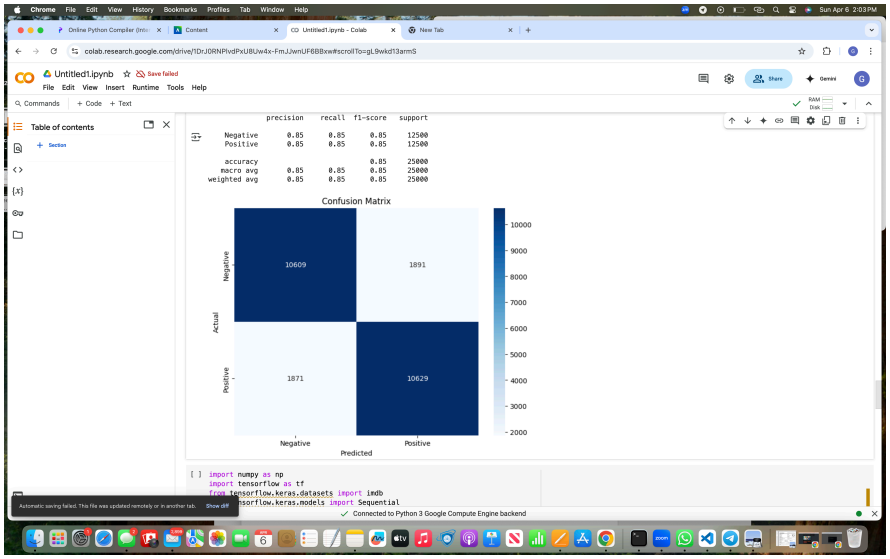
4. Train the model and generate new text by **sampling characters** one at a time.
5. Explain the role of **temperature scaling** in text generation and its effect on randomness.

Hint: *Use tensorflow.keras.layers.LSTM() for sequence modeling.*

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM,
Dense
from tensorflow.keras.preprocessing.sequence import
pad_sequences
from sklearn.metrics import classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Load and prepare the dataset
def load_imdb_data():
    try:
        # Load data with 10,000 most frequent words
        (x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=10000)

        # Pad sequences to 500 words
        max_len = 500
        x_train = pad_sequences(x_train,
maxlen=max_len)
        x_test = pad_sequences(x_test, maxlen=max_len)

        return (x_train, y_train), (x_test, y_test)
    except Exception as e:
        print(f"Error loading data: {e}")
        return None
```

```
# 2. Build the LSTM model
def build_model():
    model = Sequential([
        Embedding(10000, 128, input_length=500),
        LSTM(64, dropout=0.2, recurrent_dropout=0.2),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# 3. Main execution
def main():
    # Load data
    try:
        (x_train, y_train), (x_test, y_test) =
load_imdb_data()
        print("Data loaded successfully!")
        print(f"Training data shape: {x_train.shape}")
        print(f"Test data shape: {x_test.shape}")
    except:
        print("Failed to load data. Please check your
internet connection.")
        return

    # Build and train model
    model = build_model()
    model.summary()

    print("\nTraining model...")
    history = model.fit(x_train, y_train,
                        batch_size=128,
                        epochs=3,  # Reduced for
demonstration
                        validation_split=0.2)
```

```
    # Evaluate
    print("\nEvaluating model...")
    loss, accuracy = model.evaluate(x_test, y_test)
    print(f"Test Accuracy: {accuracy:.4f}")

    # Generate predictions
    y_pred = (model.predict(x_test) > 
0.5).astype("int32")

    # Classification report
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred, 
target_names=['Negative', 'Positive']))

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Negative', 'Positive'],
                yticklabels=['Negative', 'Positive'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

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



Q4: Sentiment Classification Using RNN

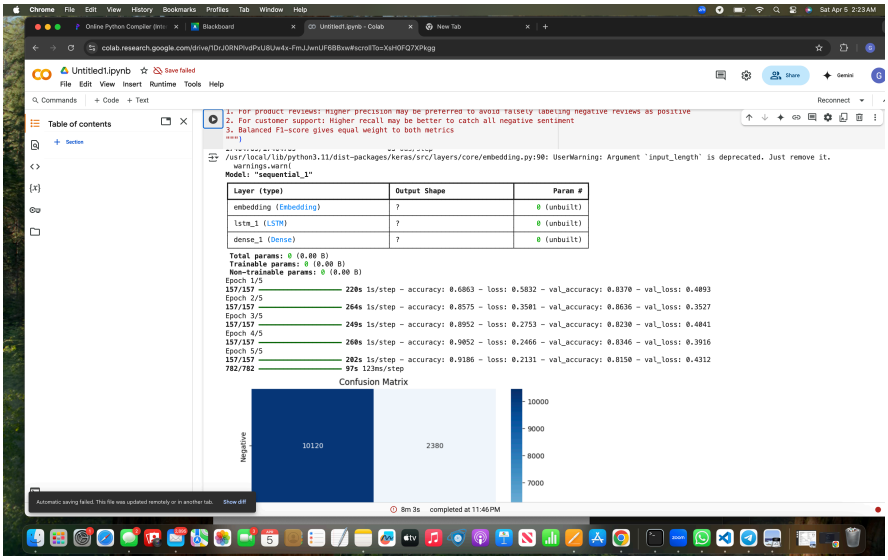
Task: Sentiment analysis determines if a given text expresses a positive or negative emotion. You will train an **LSTM-based sentiment classifier** using the IMDB dataset.

1. Load the **IMDB sentiment dataset** (tensorflow.keras.datasets.imdb).
2. Preprocess the text data by **tokenization** and **padding** sequences.
3. Train an **LSTM-based model** to classify reviews as **positive or negative**.
4. Generate a **confusion matrix** and classification report (accuracy, precision, recall, F1-score).
5. Interpret why **precision-recall tradeoff** is important in sentiment classification.

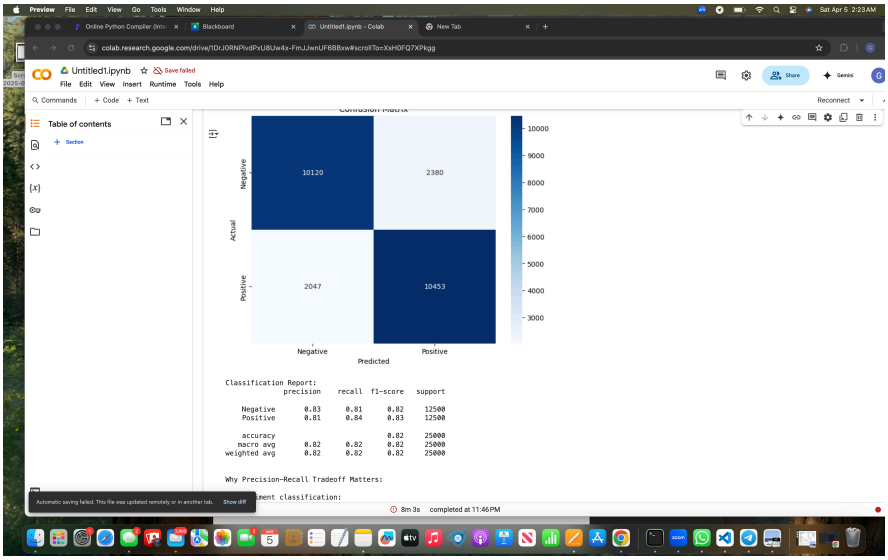
Hint: *Use confusion_matrix and classification_report from sklearn.metrics.*

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM,
Dense
from tensorflow.keras.preprocessing.sequence import
pad_sequences
from sklearn.metrics import confusion_matrix,
classification_report
import matplotlib.pyplot as plt
import seaborn as sns

## 1. Load and preprocess the IMDB dataset
vocab_size = 10000  # Keep top 10,000 most frequent
words
max_len = 500       # Maximum length of sequences

(x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure uniform length
x_train = pad_sequences(x_train, maxlen=max_len)
x_test = pad_sequences(x_test, maxlen=max_len)

## 2. Build the LSTM model
embedding_dim = 128
```

```
model = Sequential([
    Embedding(vocab_size, embedding_dim,
input_length=max_len),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()

## 3. Train the model
history = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=5,
                    validation_split=0.2)

## 4. Evaluate the model
# Get predictions
y_pred = (model.predict(x_test) > 0.5).astype("int32")

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Negative', 'Positive'],
            yticklabels=['Negative', 'Positive'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

# Classification Report
print("\nClassification Report:")
```




```
print(classification_report(y_test, y_pred,
target_names=['Negative', 'Positive']))

## 5. Precision-Recall Tradeoff Explanation
print("\nWhy Precision-Recall Tradeoff Matters:")
print("""
```



```
In sentiment classification:
- High Precision means when we predict positive, it's
likely correct (few false positives)
- High Recall means we capture most positive reviews
(few false negatives)

The tradeoff depends on application:
1. For product reviews: Higher precision may be
preferred to avoid falsely labeling negative reviews as
positive
2. For customer support: Higher recall may be better to
catch all negative sentiment
3. Balanced F1-score gives equal weight to both metrics
""")
```

Home Assignment 3 Tadikonda Gnandeep Id st-70079736