ASSIGNMENT:- 1

AIM:- Linear Regression

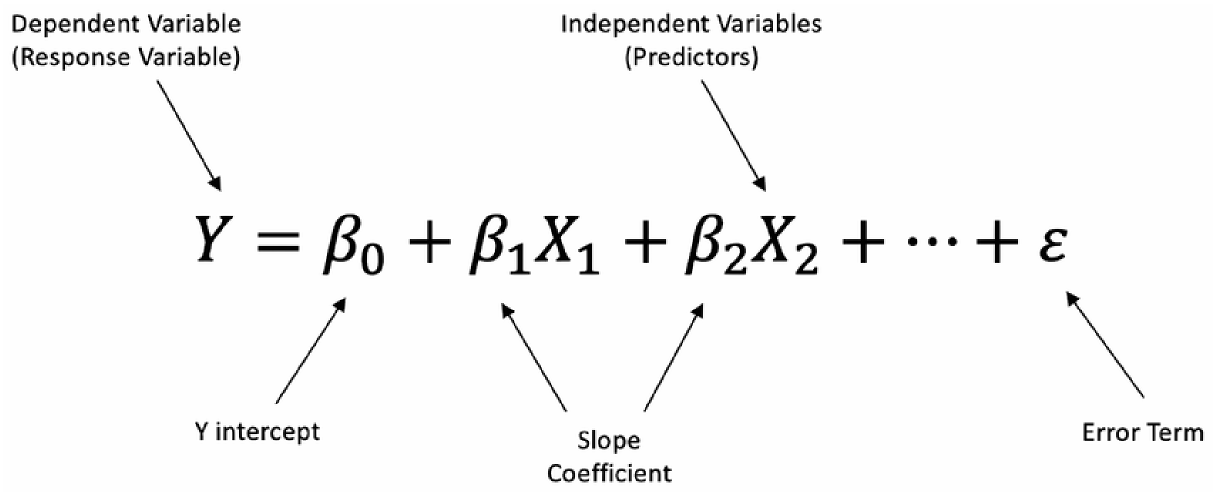
PROBLEM STATEMENT:- Real estate agents want help to predict the house price for regions in the USA. He gave you the dataset to work on and you decided to use the Linear Regression Model. Create a model that will help him to estimate what the house would sell for.

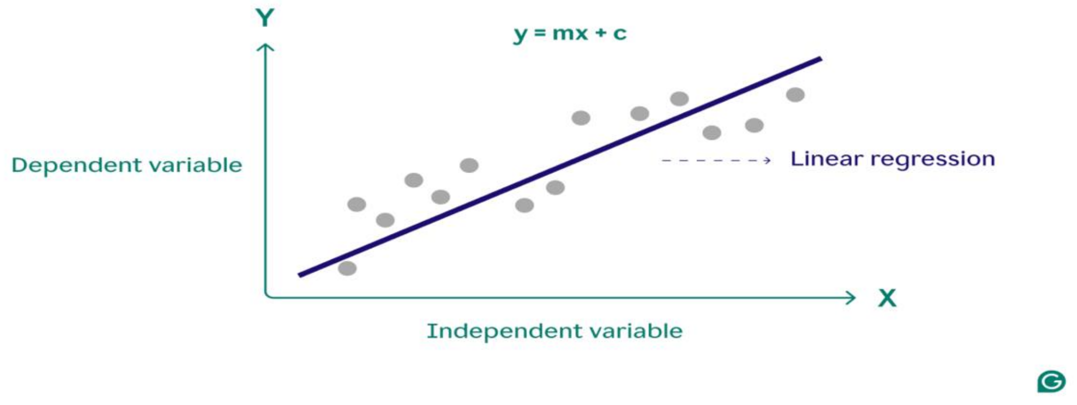
URL for the dataset: https://github.com/huzaifsayed/Linear-Regression-Model-for-House-PricePrediction/blob/master/USA_Housing.csv

THEORY:-
Regression Analysis

- Regression analysis is one of the core techniques in supervised machine learning used to model relationships between a dependent variable (target) and one or more independent variables (features).
- The primary goal is to predict the target variable using an optimized function derived from the independent variables.

Linear Regression
Linear Regression is a supervised learning algorithm used to model the linear relationship between independent variables (features) and a dependent variable (target). It assumes the relationship can be expressed as:





Application to Housing Data

The dataset includes features like Avg. Area Income, Avg. Area House Age, and Area Population. Linear Regression will estimate how these features collectively influence house prices.

ADVANTAGES:-

- Simple to implement and interpret.
- Computationally efficient.
- Provides insights into feature importance through coefficients.

LIMITATIONS:-

- Assumes linear relationships; fails to capture non-linear patterns.
- Sensitive to outliers and multicollinearity.

ASSUMPTIONS:-

- Linearity: Relationship between features and target is linear.
- Independence: Observations are independent of each other.
- Homoscedasticity: Residuals have constant variance across predictions.
- Normality: Residuals are normally distributed.
- No Multicollinearity: Features are not highly correlated with each other.

EVALUATION METRICS:-

Mean Absolute Error (MAE):
Average absolute difference between predictions and actual values.
Interpretability: Represents the average absolute error in the same units as the target variable (e.g., dollars for house prices).
Robustness: Less sensitive to outliers compared to MSE, making it useful when extreme errors should not dominate the evaluation.

Mean Squared Error (MSE):
Average squared difference penalizes larger errors.
Penalizes Larger Errors: Squaring the errors gives higher weight to significant deviations (e.g., a $200K error contributes more than a $50K error).
Downside: Sensitive to outliers—large errors can disproportionately increase MSE.

Root Mean Squared Error (RMSE):
Square root of MSE, in the same units as the target.
Same Units as Target: Unlike MSE (which is in squared units), RMSE is in the original units (e.g., dollars), making it easier to interpret.
Industry Standard: Commonly used in real estate and finance for evaluating prediction models.

CODE:-

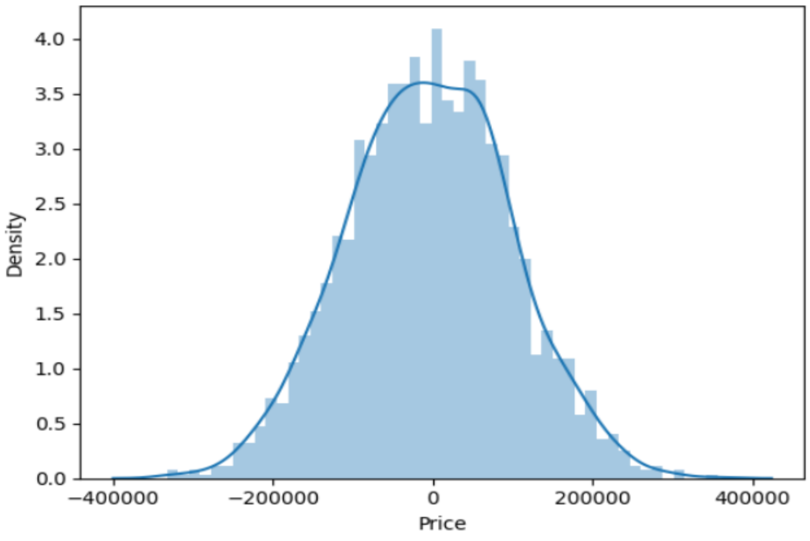
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt


HouseDF = pd.read_csv('USA_Housing.csv')

sns.pairplot(HouseDF, diag_kind='kde')

plt.show()
sns.histplot(HouseDF['Price'], kde=True)
plt.show()

corr = HouseDF.select_dtypes(include='number').corr()['Price'].drop('Price')
corr.sort_values().plot(kind='barh', figsize=(8, 5), title='Correlation with Price')
plt.show()

X = HouseDF[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of
Rooms',
            'Avg. Area Number of Bedrooms', 'Area Population']]

y = HouseDF['Price']
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=101)


from sklearn.linear_model import LinearRegression
lm = LinearRegression()


lm.fit(X_train,y_train)

predictions = lm.predict(X_test)

plt.scatter(y_test,predictions)
plt.show()
sns.distplot((y_test-predictions),bins=50);
plt.show()

from sklearn import metrics

print('MAE:', metrics.mean_absolute_error(y_test, predictions)) print('MSE:',
metrics.mean_squared_error(y_test, predictions)) print('RMSE:',
np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

OUTPUT ANALYSIS:-

Metrics:

```
MAE: 82,288
MSE: 10,460,958,907
RMSE: 102,278
```



In the above histogram plot, we see data is in bell shape (Normally Distributed), which indicates that the model has accurate predictions.

CONCLUSION:-

The project used Linear Regression, a foundational statistical and machine learning algorithm, to model and predict house prices based on various features. The assumption that there exists a linear relationship between these features and the house price seems valid, as seen from the normally distributed residuals and the model performance metrics.
The model achieved the following results that suggest:
On average, the model's predictions are off by around $82,288, which is reasonable depending on the scale of home prices in the dataset.
The residuals (errors between predicted and actual prices) are normally distributed, as indicated by the bell-shaped histogram, which means the model satisfies one of the core linear regression assumptions and performs reliably on the given data.
The model can be used by real estate agents, property investors, or buyers/sellers to estimate the fair market value of a property.

ASSIGNMENT:- 2

AIM:- Multiclass Classification

PROBLEM STATEMENT:- Build a Multiclass classifier using the CNN model. Use MNIST or any other suitable dataset. a. Perform Data Pre-processing b. Define Model and perform training c. Evaluate Results using confusion matrix.

THEORY:-

Convolutional Neural Network (CNN)
Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to process and analyze data with a grid-like topology, such as images. They are widely used for tasks like image classification, object detection, and facial recognition. CNNs are especially effective for multiclass classification problems where the goal is to assign each input image to one of several predefined categories.

A CNN typically consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers extract important features from input images using learnable filters, while pooling layers reduce the spatial dimensions, leading to lower computational costs and better generalization. The fully connected layers at the end perform the final classification based on the extracted features.

Description of the MNIST Dataset
The MNIST (Modified National Institute of Standards and Technology) dataset is a benchmark dataset commonly used for training and evaluating image classification models. It consists of 70,000 grayscale images of handwritten digits (0 through 9), each of size 28x28 pixels. The dataset is divided into:

- Training set: 60,000 images
- Test set: 10,000 images

Each image is labeled with the digit it represents, making it ideal for a 10-class classification problem. The simplicity and standardization of the MNIST dataset make it a suitable choice for evaluating the performance of CNNs on multiclass classification tasks.

a. Data Pre-processing
Data pre-processing is a crucial step in building an effective CNN model. For the MNIST dataset, this involves the following tasks:

- Normalization: Pixel values ranging from 0 to 255 are scaled to a range of 0 to 1 to improve model convergence.
- Reshaping: The input images are reshaped to match the CNN input requirements (e.g., (28, 28, 1) for grayscale images).
- One-hot Encoding: The class labels (0–9) are converted to one-hot encoded vectors to be compatible with the softmax output layer.

b. Defining the CNN Model and Performing Training

The CNN model is defined using layers that extract spatial features and perform classification. A basic CNN architecture for the MNIST dataset may include:

- Convolutional Layers: To extract local patterns using filters.
- ReLU Activation: To introduce non-linearity.
- Max Pooling Layers: To reduce dimensionality while retaining essential features.
- Flatten Layer: To convert the 2D feature maps into a 1D vector.
- Fully Connected (Dense) Layers: To perform classification based on extracted features.

The model is compiled using a suitable optimizer (e.g., Adam) and loss function (e.g., categorical crossentropy). Training is done over multiple epochs with batch processing to minimize the loss and improve accuracy.

CODE:-

```
import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to_categorical

import matplotlib.pyplot as plt

import numpy as np

(x_train, y_train), (x_test, y_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(-1, 28, 28, 1)

x_test = x_test.reshape(-1, 28, 28, 1)

y_train_cat = to_categorical(y_train, 10)

y_test_cat = to_categorical(y_test, 10)

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential([ Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,
1)), MaxPooling2D(pool_size=(2, 2)), Conv2D(64, kernel_size=(3, 3), activation='relu'),
MaxPooling2D(pool_size=(2, 2)), Flatten(), Dense(128, activation='relu'), Dropout(0.5),
Dense(10, activation='softmax') # 10 classes ])

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

history = model.fit(x_train, y_train_cat, epochs=5, batch_size=128, validation_split=0.1)
```

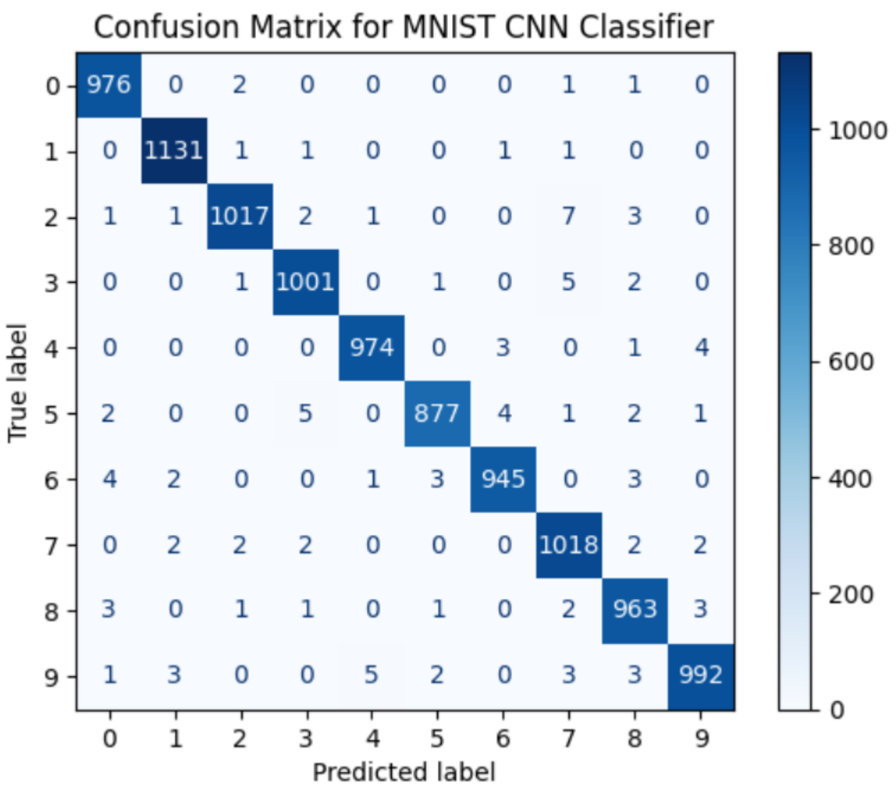
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_pred = model.predict(x_test)

y_pred_labels = np.argmax(y_pred, axis=1)

cm = confusion_matrix(y_test, y_pred_labels)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.arange(10))

disp.plot(cmap='Blues') plt.title("Confusion Matrix for MNIST CNN Classifier")

plt.show()

test_loss, test_acc = model.evaluate(x_test, y_test_cat)

print("Test Accuracy: {:.2f}%".format(test_acc * 100))

from sklearn.metrics import classification_report

report = classification_report(y_test, y_pred_labels, target_names=[str(i) for i in range(10)])
print("Classification Report:\n")

print(report)
```

EVALUATION:-

The effectiveness of the CNN model is measured using the following evaluation metrics.

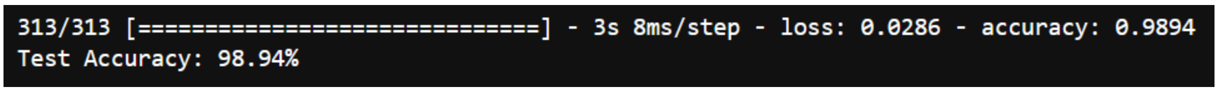
1. Confusion Matrix

For the MNIST dataset, the confusion matrix will be a 10x10 matrix, since there are ten digits.



	0	1	2	3	4	5	6	7	8	9
0	976	0	2	0	0	0	0	1	1	0
1	0	1131	1	1	0	0	1	1	0	0
2	1	1	1017	2	1	0	0	7	3	0
3	0	0	1	1001	0	1	0	5	2	0
4	0	0	0	0	974	0	3	0	1	4
5	2	0	0	5	0	877	4	1	2	1
6	4	2	0	0	1	3	945	0	3	0
7	0	2	2	2	0	0	0	1018	2	2
8	3	0	1	1	0	1	0	2	963	3
9	1	3	0	0	5	2	0	3	3	992

2. Accuracy



```
313/313 [==============================] - 3s 8ms/step - loss: 0.0286 - accuracy: 0.9894
Test Accuracy: 98.94%
```

3. Precision, Recall, and F1-Score

- **Precision**: Measures the proportion of true positive predictions out of all positive predictions made.

$$\text{Precision}_{\text{Class A}} = \frac{TP_{\text{Class A}}}{TP_{\text{Class A}} + FP_{\text{Class A}}}$$

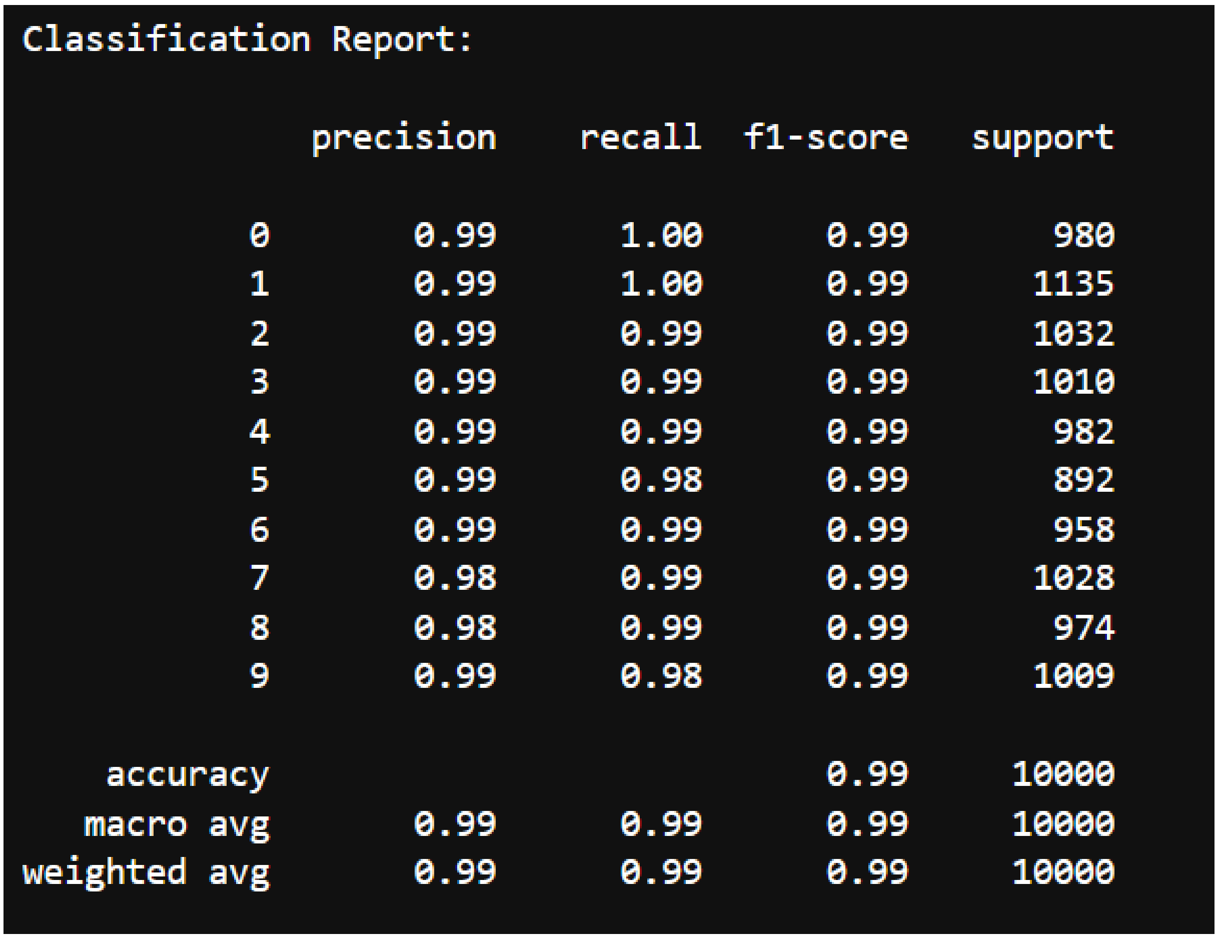
- **Recall**: Measures the proportion of true positives out of all actual positives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **F1-Score**: The harmonic mean of precision and recall.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Classification Report:


```
Classification Report:

              precision    recall  f1-score   support

           0       0.99      1.00      0.99       980
           1       0.99      1.00      0.99      1135
           2       0.99      0.99      0.99      1032
           3       0.99      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.99      0.98      0.99       892
           6       0.99      0.99      0.99       958
           7       0.98      0.99      0.99      1028
           8       0.98      0.99      0.99       974
           9       0.99      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

Advantages of CNN

- **Automatic Feature Extraction**: CNNs automatically learn relevant features from raw image data during training, eliminating the need for manual feature engineering.
- **Spatial Invariance**: Through the use of convolution and pooling layers, CNNs are effective at recognizing patterns regardless of their position in the input image.
- **High Accuracy on Image Data**: CNNs consistently achieve high accuracy for image classification tasks, especially with large and complex datasets such as MNIST, CIFAR, or ImageNet.

Limitations of CNN

- **Computationally Intensive:** Training CNNs requires significant computational power, especially with deep architectures and large datasets, often requiring GPUs.

- **Large Amount of Labeled Data Required:** CNNs typically require a large amount of labeled data to perform well, which may not always be available for every application.
- **Lack of Interpretability:** CNNs function as black-box models, making it difficult to interpret or explain the reasoning behind a specific prediction.

CONCLUSION:-

This assignment involved implementing a Convolutional Neural Network (CNN) to classify handwritten digits from the MNIST dataset. The CNN architecture successfully extracted meaningful features from 28x28 grayscale images and learned to distinguish digits from 0 to 9 with high accuracy. This validates the strength of CNNs in visual recognition tasks, especially where spatial hierarchies and local patterns are critical.
The following preprocessing steps significantly improved the model's performance: Normalization, Reshaping and One-hot Encoding
The model's layered design provided insights into how CNNs operate:
Convolutional Layers captured spatial patterns (like curves, edges) in digits.
ReLU Activation introduced non-linearity, enabling complex representations.
Pooling Layers reduced dimensionality, preventing overfitting and increasing computational efficiency.
Fully Connected Layers + Softmax translated abstracted features into class probabilities.
Accuracy: High accuracy (>98%) proved the model's robustness.
Unlike traditional machine learning where feature selection is manual, CNNs automatically learn the optimal filters and patterns needed for classification directly from pixel data.
This emphasizes the major advantage of deep learning: letting the model discover hierarchical features on its own from low-level edges to high-level shapes.This assignment has practical significance beyond just digit recognition:
CNNs are used in real-time handwriting recognition, license plate readers, medical image classification, facial recognition, and more.
Mastering CNN basics using MNIST builds the foundation for tackling complex datasets like CIFAR-10/100, Fashion-MNIST, and real-world applications involving multiclass image classification.

ASSIGNMENT:- 3

AIM:- LSTM Design

PROBLEM STATEMENT:- Design RNN or its variant including LSTM or GRU
a) Select a suitable time series dataset. Example – predict sentiments based on product reviews b) Apply for prediction

THEORY:

RECURRENT NEURAL NETWORKS:-
RNNs are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. They are used in NLP, time-series forecasting, and speech recognition.

Problems with Vanilla RNNs
Vanishing Gradients: Gradients shrink exponentially during backpropagation, making long-term dependencies hard to learn.
Exploding Gradients: Gradients grow uncontrollably, destabilizing training.

LONG SHORT-TERM MEMORY (LSTM) NETWORKS: -
LSTMs are a variant of Recurrent Neural Networks (RNNs) designed to capture long-term dependencies in sequential data. Unlike vanilla RNNs, LSTMs use gating mechanisms to regulate information flow, addressing the vanishing/exploding gradient problem.

KEY COMPONENTS OF LSTM:-
Forget Gate: Uses sigmoid (σ) to output values between 0 (discard) and 1 (keep).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Input Gate & Candidate Memory: i decides which new information to store. C is a candidate update to the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Cell State Update: ⊙ = element-wise multiplication. Combines old memory and new memory

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Output Gate & Hidden State: ‘o’ filters the cell state for the next hidden state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

Why LSTM for Text Data?
Text is sequential, where word order impacts meaning.
LSTMs retain context over long sequences (e.g., sentiment cues in reviews).

DATASET DESCRIPTION:-
IMDB Movie Review Dataset
Task: Binary classification (positive/negative sentiment).
Size: 50,000 reviews (25k train, 25k test).
Classes: Balanced (50% positive, 50% negative).
Preprocessed: HTML tags removed, lowercased, and punctuation stripped.

WORKING OF LSTM FOR SENTIMENT ANALYSIS:-
1. Embedding Layer: Converts tokenized words into dense vectors (e.g., "good" → [0.2, -0.5, ...]).
2. LSTM Layer: Processes sequences to capture contextual relationships.
3. Dropout: Randomly deactivates neurons to prevent overfitting.
4. Dense Layer: Outputs probability via sigmoid activation.

PRE-PROCESSING STEPS:-
1. Tokenization: Map words to integers (e.g., "movie" → 45).
2. Padding: Standardize sequence length to 100 (truncate/pad with zeros).
3. Train-Test Split: 80% training, 20% testing.
4. Embedding Matrix: Pre-trained or learned word vectors.

CODE:-

```
import tensorflow as tf

import matplotlib.pyplot as plt

from sklearn.metrics import classification_report, confusion_matrix,
ConfusionMatrixDisplay

import numpy as np

tf.random.set_seed(42)

num_words = 10000 # Limit the vocabulary to the top 10,000 words

max_length = 100 # Maximum review length (in words)
```

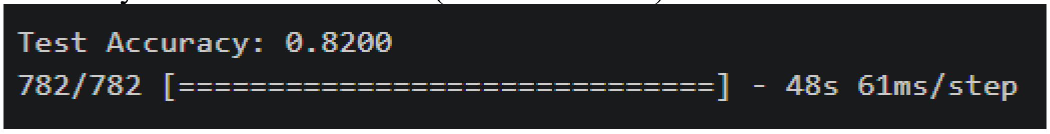
```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.imdb.load_data(num_words=num_words)

x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=max_length)

x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_length)

model = tf.keras.Sequential([ tf.keras.layers.Embedding(input_dim=num_words,
output_dim=32, input_length=max_length), tf.keras.layers.LSTM(32, dropout=0.2,
recurrent_dropout=0.2), tf.keras.layers.Dense(1, activation='sigmoid') ])

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

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

test_loss, test_acc = model.evaluate(x_test, y_test) print(f"\nTest Accuracy:
{test_acc:.4f}")

y_pred_prob = model.predict(x_test) y_pred = (y_pred_prob > 0.5).astype(int)

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

cm = confusion_matrix(y_test, y_pred) disp =
ConfusionMatrixDisplay(confusion_matrix=cm,

display_labels=["Negative", "Positive"])

disp.plot(cmap='Blues')

plt.title("Confusion Matrix")

plt.show()

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label="Train Acc")
plt.plot(history.history['val_accuracy'], label="Val Acc")

plt.title("Accuracy over Epochs")

plt.xlabel("Epoch")

plt.ylabel("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 over Epochs")
plt.xlabel("Epoch")

plt.ylabel("Loss") plt.legend()

plt.tight_layout() plt.show()
```

PRACTICAL IMPLEMENTATION:-

1. Model Architecture
LSTM Layer: 32 units with dropout for regularization.
Output Layer: Sigmoid for binary classification.

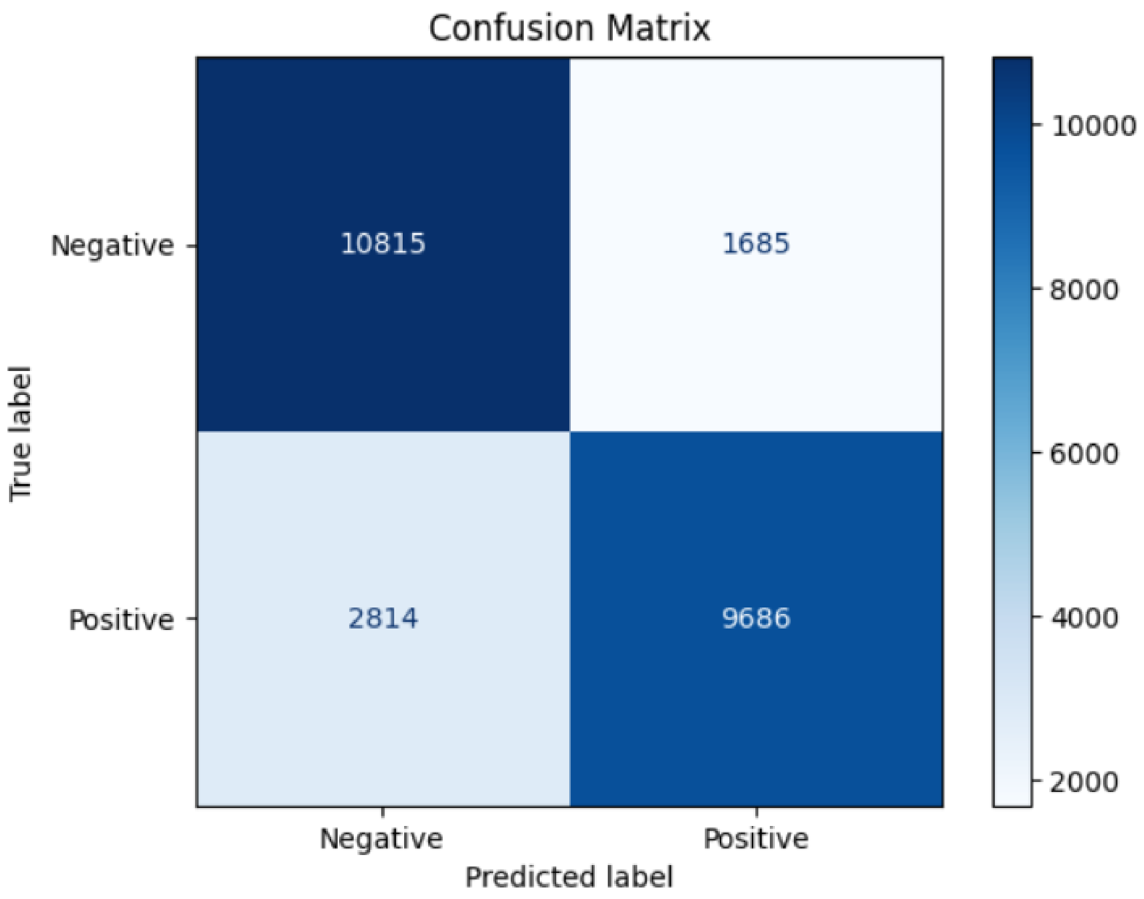
2. Hyperparameters
Optimizer: Adam (adaptive learning rate).
Loss: Binary cross-entropy (penalizes incorrect probabilities).
Batch Size: 128 (trade-off between speed and stability).
Epochs: 5 (early stopping to avoid overfitting).

3. Evaluation Metrics
Accuracy: Overall correctness (82% on test set).

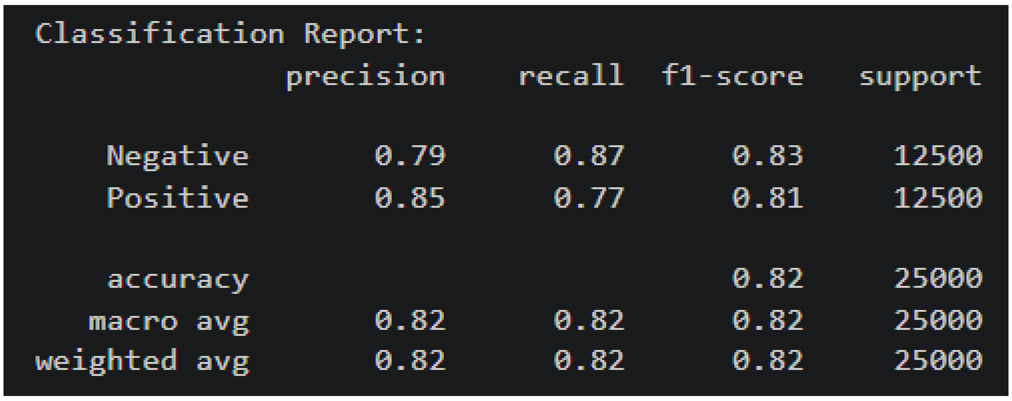


```
Test Accuracy: 0.8200
782/782 [==============================] - 48s 61ms/step
```

Confusion Matrix: Visualize TP, TN, FP, FN.



Precision/Recall: Balance between false positives and negatives.



```
Classification Report:
              precision    recall  f1-score   support

    Negative       0.79      0.87      0.83     12500
    Positive       0.85      0.77      0.81     12500

    accuracy                           0.82     25000
   macro avg       0.82      0.82      0.82     25000
weighted avg       0.82      0.82      0.82     25000
```

ADVANTAGES:-

- Context Preservation: Captures word dependencies (e.g., "not good").
- Robustness: Handles variable-length sequences via padding.
- Regularization: Dropout mitigates overfitting.

LIMITATIONS:-

- Computational Cost: LSTMs are slower than CNNs/Transformers.
- Long Sequences: Struggles with very long texts (e.g., >500 words).
- Interpretability: Black-box nature limits insight into decisions.

CONCLUSION:-

By applying an LSTM network to the IMDB movie-review dataset, the model learned to capture long-term dependencies in text—such as negations ("not good") and contextual cues—far more effectively than a vanilla RNN. Tokenization & Padding: Standardizing sequence length and using an OOV token ensured the model handled variable-length reviews robustly. Accuracy (82.00%): Indicates strong overall performance on unseen reviews. Context Preservation: The gating mechanisms allow the network to retain or discard information selectively, handling sequences of varying length and complexity, No manual feature engineering is required; the network discovers the salient patterns (e.g., sentiment-bearing phrases) during training.
Designing an LSTM for sentiment analysis involves thoughtful preprocessing, architecture design, and metric-driven evaluation. The exercise reinforces that sequence models like LSTMs can robustly capture the nuanced dependencies in text, yielding high accuracy on balanced binary classification tasks—and lays the groundwork for more advanced NLP architectures.

ASSIGNMENT:- 4

AIM:- Image Classification

PROBLEM STATEMENT:- Design and implement a CNN for Image Classification a) Select a suitable image classification dataset (medical imaging, agricultural, etc.). b) Optimized with different hyper-parameters including learning rate, filter size, no. of layers, optimizers, dropouts, etc.

THEORY:-

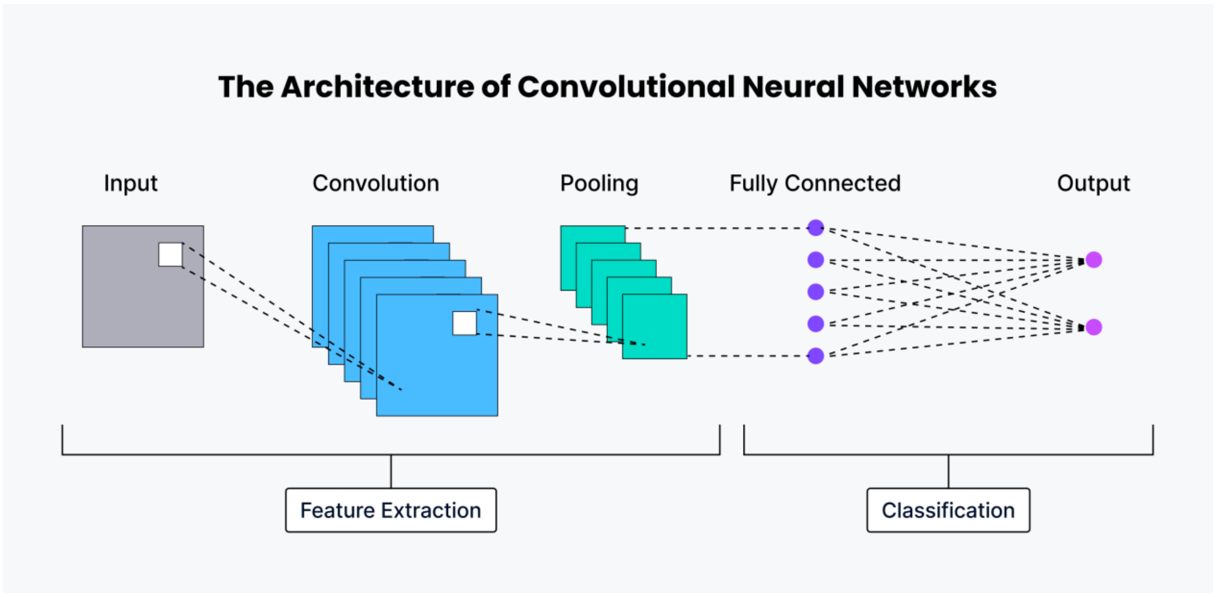
Convolutional Neural Network (CNN) for Image Classification
Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to process data with a grid-like topology, such as images. CNNs have become the go-to approach for image classification tasks due to their ability to automatically and adaptively learn spatial hierarchies of features from input images through backpropagation.
A CNN typically consists of several types of layers: convolutional layers, pooling layers, dropout layers, and fully connected layers. These layers work together to extract features, reduce spatial dimensions, prevent overfitting, and finally classify the input image into one of the predefined categories.
CNNs are particularly powerful in applications involving medical image analysis, where accurate classification can assist in early diagnosis and treatment planning.

Dataset Description
The Rice Image Dataset used in this project was obtained from Kaggle, and it contains labeled images of five different types of rice grains:

1. Arborio
2. Basmati
3. Ipsala
4. Jasmine
5. Karacadag

Each class includes pre-split images for training and testing, making it suitable for supervised image classification tasks. The goal is to classify each rice grain image into its corresponding type.

Working of CNN
The CNN algorithm classifies an input image by automatically learning features from the raw image data.



The basic working involves the following steps:

1. **Convolution Operation**
   A set of filters is convolved with the input image to extract spatial features. Each filter detects specific features such as edges, shapes, or textures.
2. **Activation Function (ReLU)**
   Non-linear activation functions such as ReLU are applied to add non-linearity and help the network learn complex patterns.
3. **Pooling Layer (MaxPooling)**
   Pooling reduces the spatial size of the feature maps and makes the model more robust to translations and distortions in the input image.
4. **Dropout Layer**
   During training, dropout is used to randomly deactivate some neurons to prevent overfitting.
5. **Fully Connected Layer**
   The extracted features are flattened and passed through dense layers for final classification.
6. **Output Layer (Sigmoid or Softmax)**
   For binary classification (pneumonia vs. normal), a sigmoid activation is used. For multiclass classification, softmax is used.

Pre-Processing Steps for CNN
Preprocessing is crucial for improving CNN performance. Key steps include:

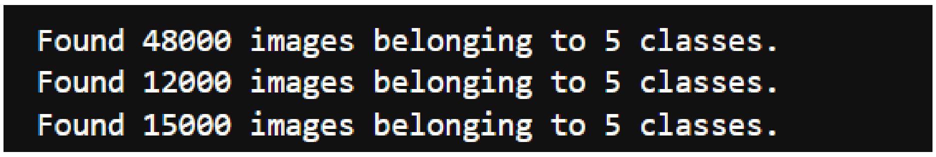
1. **Image Resizing and Normalization**
   All images are resized to a fixed dimension (e.g., 150x150 or 224x224) and pixel values are normalized (scaled between 0 and 1) to standardize input.

2. **Data Augmentation**
   Techniques such as rotation, zoom, flipping, and shifting are applied to artificially increase the diversity of the dataset and reduce overfitting.

3. **Train-Test Split**

   The dataset is divided into training, validation, and testing sets to evaluate generalization capability.

   a. Data Splitting: The dataset was already divided into train and test sets. An additional validation set (20% of training data) was created to monitor model performance during training.
   b. Image Resizing: All images were resized to a uniform shape of 150×150 pixels to ensure consistent input to the CNN.
   c. Feature Scaling: All image pixel values were rescaled to the range [0, 1] by dividing by 255 using ImageDataGenerator(rescale=1./255), which helps accelerate convergence during training.



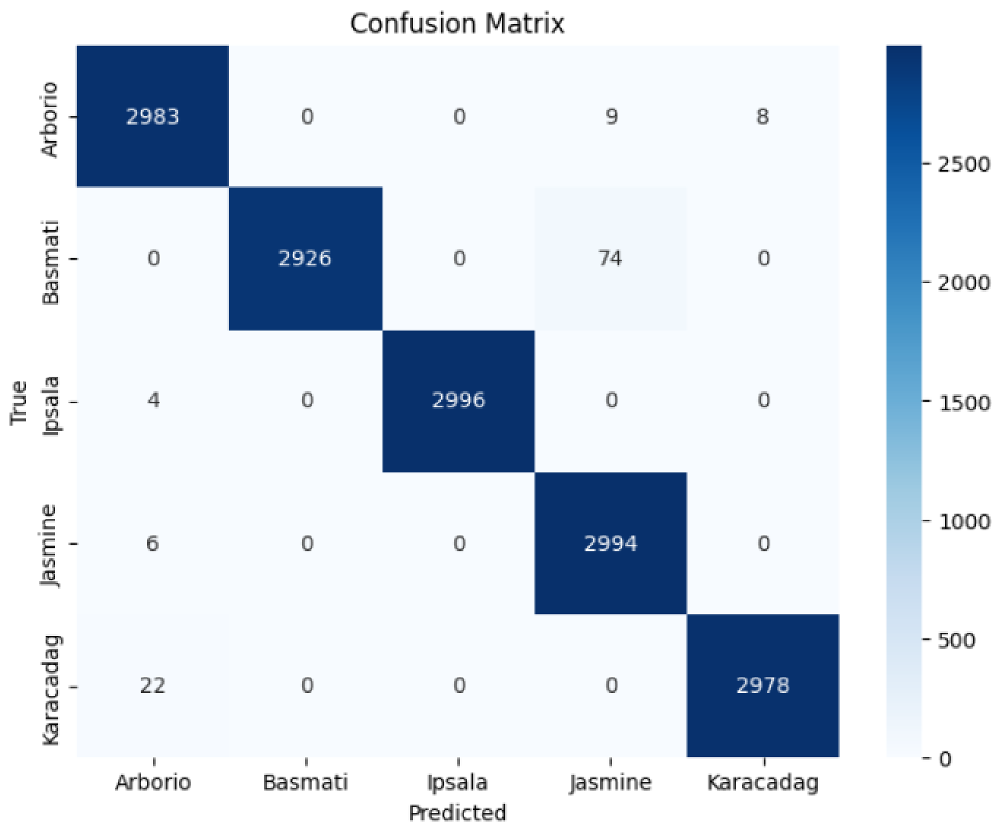
```
Found 48000 images belonging to 5 classes.
Found 12000 images belonging to 5 classes.
Found 15000 images belonging to 5 classes.
```

Practical Implementation of CNN

The steps to implement CNN on the **Rice Image Dataset** (Arborio, Basmati, Ipsala, Jasmine, Karacadag) from Kaggle are as follows:

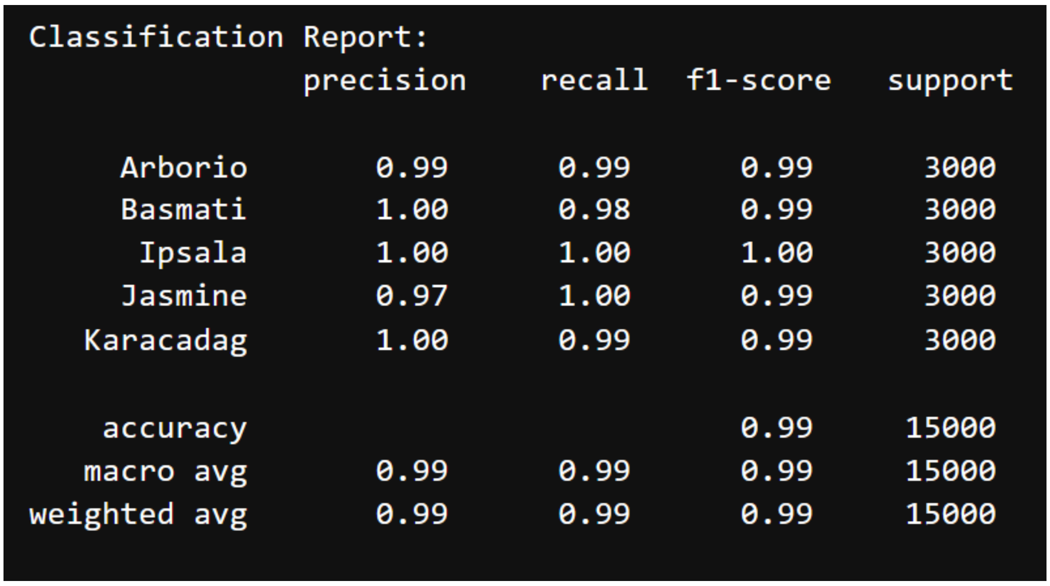
1. **Data Preprocessing**:
   a. Resize images and normalize pixel values.
   b. Apply data augmentation techniques.
   c. Split the dataset into training, validation, and testing sets.
2. **Model Architecture**:
   a. Define convolutional layers with filters:
      i. First Conv2D: 32 filters of size (3×3)
      ii. Second Conv2D: 64 filters
      iii. Third Conv2D: 128 filters
   b. Use ReLU activation after each convolutional layer to introduce non-linearity.
   c. Apply MaxPooling2D after each convolution to reduce spatial dimensions.
   d. Add Dropout layers with dropout rates (0.2, 0.3, 0.5) to reduce overfitting.
   e. Flatten the output of the last convolutional layer.
   f. Connect to a fully connected Dense layer (128 neurons).
   g. Use softmax activation in the final output layer (since it's multi-class classification with 5 classes).

3. **Hyperparameter Optimization**:
   a. Tune learning rate, batch size, number of epochs, dropout rate, number of convolutional layers, and optimizer (Adam, SGD, etc.).
4. **Evaluation**:
   a. Generate Confusion Matrix using confusion_matrix() to visualize correct and incorrect predictions for each rice class.



b. Calculate performance metrics using classification_report():

i. Accuracy
ii. Precision
iii. Recall
iv. F1-score



```
Classification Report:
              precision    recall  f1-score   support

     Arborio       0.99      0.99      0.99      3000
     Basmati       1.00      0.98      0.99      3000
      Ipsala       1.00      1.00      1.00      3000
     Jasmine       0.97      1.00      0.99      3000
   Karacadag       1.00      0.99      0.99      3000

    accuracy                           0.99     15000
   macro avg       0.99      0.99      0.99     15000
weighted avg       0.99      0.99      0.99     15000
```

Advantages of CNN

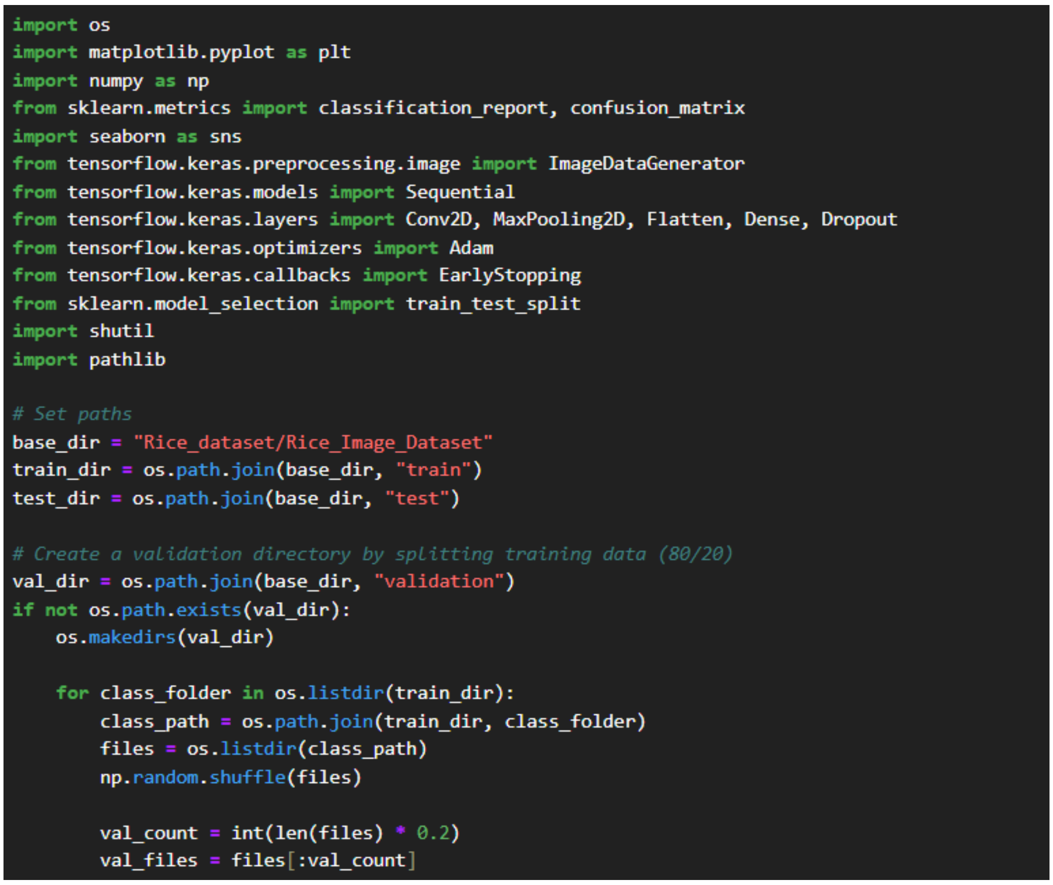
- **Automatic Feature Extraction:** CNNs automatically learn relevant features from images

- **High Accuracy:** CNNs outperform traditional models in complex image classification tasks
- **Robustness to Variations:** Pooling and dropout layers make CNNs robust to minor variations like noise, rotation, and shift in images.

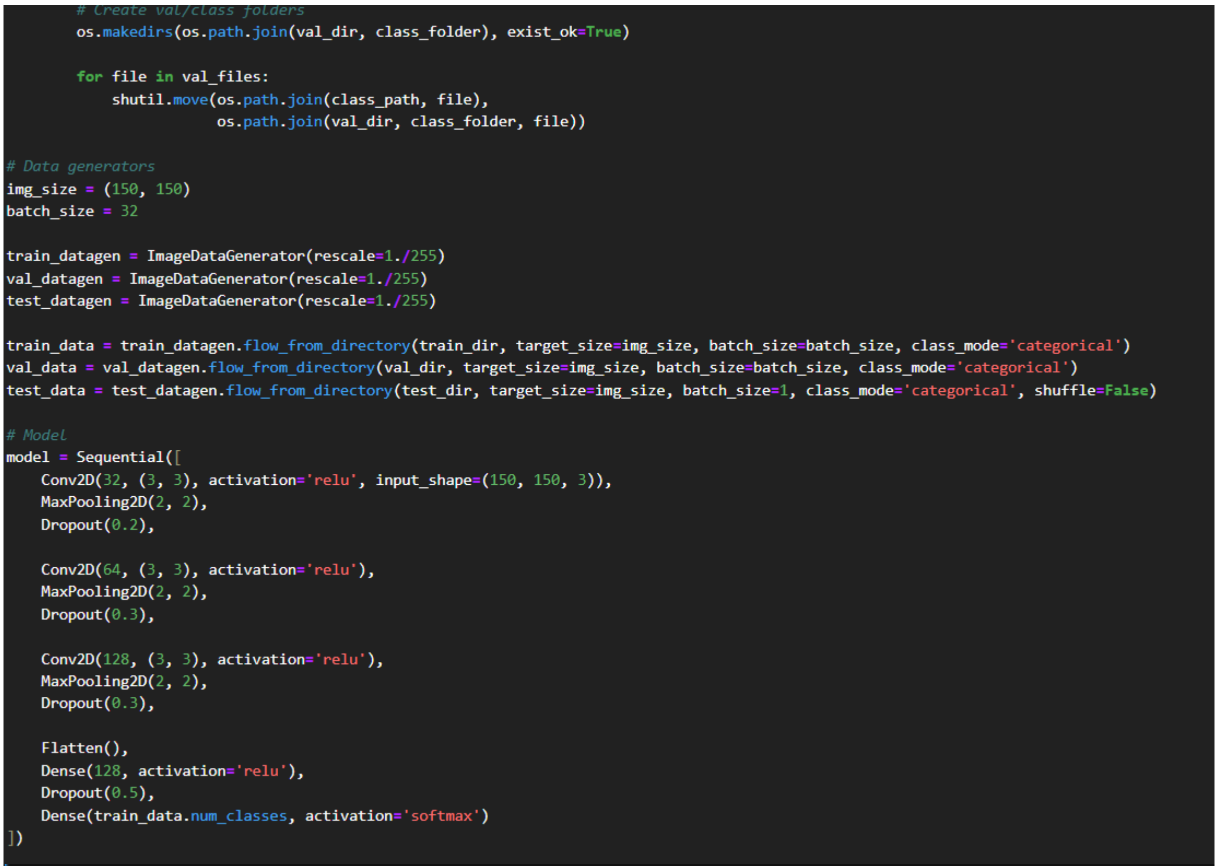
Limitations of CNN

- **Requires Large Data and Resources:** CNNs typically require large datasets and powerful hardware for effective training.
- **Interpretability:** The learned features and decisions made by CNNs are often difficult to interpret, which can be a limitation in medical applications.
- **Computational Complexity:** Training CNNs can be computationally expensive, especially for large datasets or deeper models.
- **Overfitting Risk:** Without proper regularization or dropout, CNNs may overfit on small datasets.

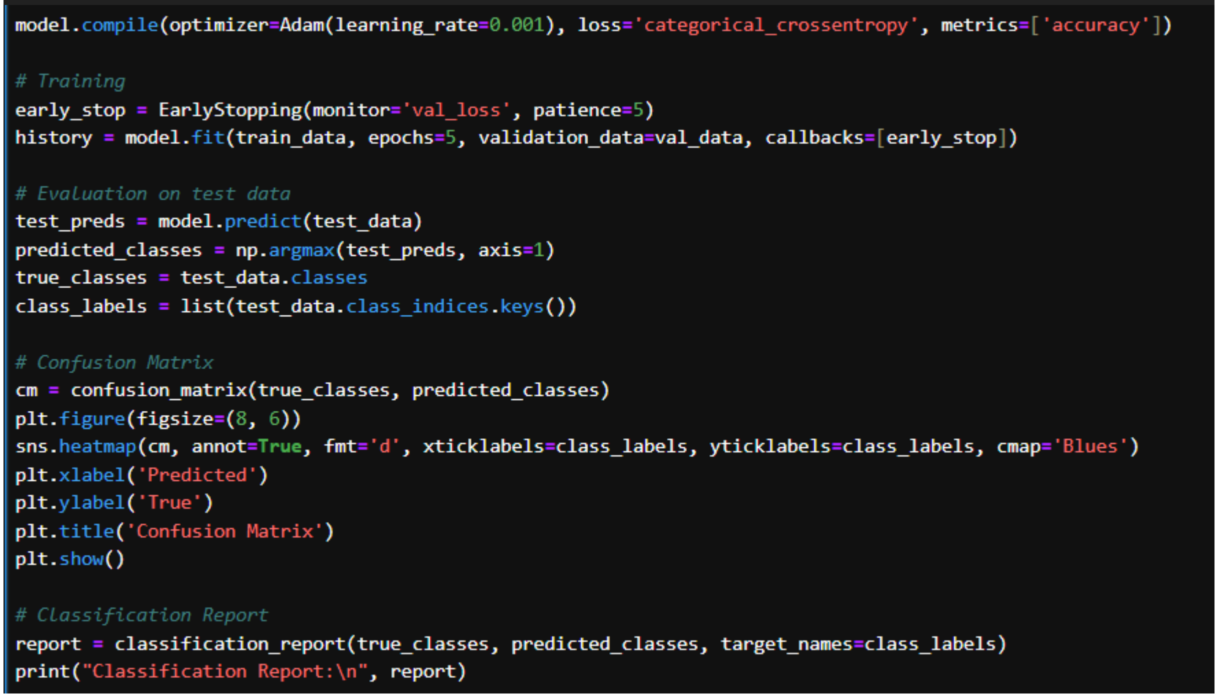
CODE:-



```python
import os
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split
import shutil
import pathlib

# Set paths
base_dir = "Rice_dataset/Rice_Image_Dataset"
train_dir = os.path.join(base_dir, "train")
test_dir = os.path.join(base_dir, "test")

# Create a validation directory by splitting training data (80/20)
val_dir = os.path.join(base_dir, "validation")
if not os.path.exists(val_dir):
    os.makedirs(val_dir)

    for class_folder in os.listdir(train_dir):
        class_path = os.path.join(train_dir, class_folder)
        files = os.listdir(class_path)
        np.random.shuffle(files)

        val_count = int(len(files) * 0.2)
        val_files = files[:val_count]
```



```
        # Create val/class folders
        os.makedirs(os.path.join(val_dir, class_folder), exist_ok=True)

        for file in val_files:
            shutil.move(os.path.join(class_path, file),
                        os.path.join(val_dir, class_folder, file))

# Data generators
img_size = (150, 150)
batch_size = 32

train_datagen = ImageDataGenerator(rescale=1./255)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_data = train_datagen.flow_from_directory(train_dir, target_size=img_size, batch_size=batch_size, class_mode='categorical')
val_data = val_datagen.flow_from_directory(val_dir, target_size=img_size, batch_size=batch_size, class_mode='categorical')
test_data = test_datagen.flow_from_directory(test_dir, target_size=img_size, batch_size=1, class_mode='categorical', shuffle=False)

# Model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),
    Dropout(0.2),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Dropout(0.3),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(train_data.num_classes, activation='softmax')
])
```



```
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

# Training
early_stop = EarlyStopping(monitor='val_loss', patience=5)
history = model.fit(train_data, epochs=5, validation_data=val_data, callbacks=[early_stop])

# Evaluation on test data
test_preds = model.predict(test_data)
predicted_classes = np.argmax(test_preds, axis=1)
true_classes = test_data.classes
class_labels = list(test_data.class_indices.keys())

# Confusion Matrix
cm = confusion_matrix(true_classes, predicted_classes)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', xticklabels=class_labels, yticklabels=class_labels, cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Classification Report
report = classification_report(true_classes, predicted_classes, target_names=class_labels)
print("Classification Report:\n", report)
```

CONCLUSION:-

This assignment on designing a Convolutional Neural Network for image classification using the Rice Image Dataset uses different layers such as convolutional, pooling, dropout, and dense layers to extract and process spatial features from images. The sequential stacking of these layers enabled us to build a model that progressively reduced the spatial dimensions while extracting increasingly abstract features. Experimenting with various hyperparameters such as filter sizes, the number of convolutional layers, learning rates, and dropout rates helped us understand their impact on the model's performance. For instance, increasing dropout rates mitigated overfitting, while adjustments to learning rate improved the convergence of the optimizer. This iterative tuning process reinforced the importance of hyperparameter optimization in achieving high accuracy (99%).

The evaluation phase included a confusion matrix and classification report. The near-perfect scores confirmed the model's effectiveness but also prompted a reflection on the complexity and potential overfitting, urging us to balance model depth with generalizability.

By applying CNNs to the Rice Image Dataset, we witnessed how these models can be deployed in various domains such as agriculture and medical imaging. Beyond high accuracy numbers, understanding the interpretability and limitations of CNNs prepares us for real-world challenges where the cost of misclassification might be significant

ASSIGNMENT:- 5

AIM:- Sentiment Analysis in Network Graph using RNN

PROBLEM STATEMENT:- Perform Sentiment Analysis in the network graph using RNN.

THEORY:-
Recurrent Neural Networks for Sentiment Analysis in Network Graphs

Sentiment Analysis in Network Graphs
Sentiment analysis in this context involves determining the emotional tone or subjective opinion expressed in the text associated with each node in the network. This allows us to understand the overall sentiment distribution within the network and potentially identify influential nodes or communities based on sentiment.

Recurrent Neural Networks (RNNs) for Sentiment Analysis
Recurrent Neural Networks, particularly Long Short-Term Memory networks (LSTMs), are well-suited for sentiment analysis due to their ability to process sequential data like text.

Sequential Data Processing: Text is inherently sequential, and RNNs can capture the context and dependencies between words in a sentence. This is crucial for understanding sentiment, where word order and context are vital (e.g., "not good" vs. "good"). Long-Term Dependencies: LSTMs, a type of RNN, are specifically designed to handle long-term dependencies in sequences. This enables them to understand sentiment even in longer texts where sentiment cues might be spread out.

Why RNNs (LSTMs) for Network Graph Sentiment?
When applying sentiment analysis to network graphs, RNNs offer advantages:

Node-Level Sentiment: RNNs allow us to analyze the text associated with each node individually and determine its sentiment.

Contextual Understanding: By processing text sequentially, RNNs capture the nuances of language and provide more accurate sentiment predictions.

DATA PREPARATION:-
Preprocess the text data by tokenizing words and padding sequences to a uniform length.

MODEL BUILDING:-

1. Using an Embedding layer we convert words into dense vector representations.
2. Employ an LSTM layer to process the sequences and capture contextual information
3. Use a Dense output layer with a sigmoid activation function for binary sentiment.

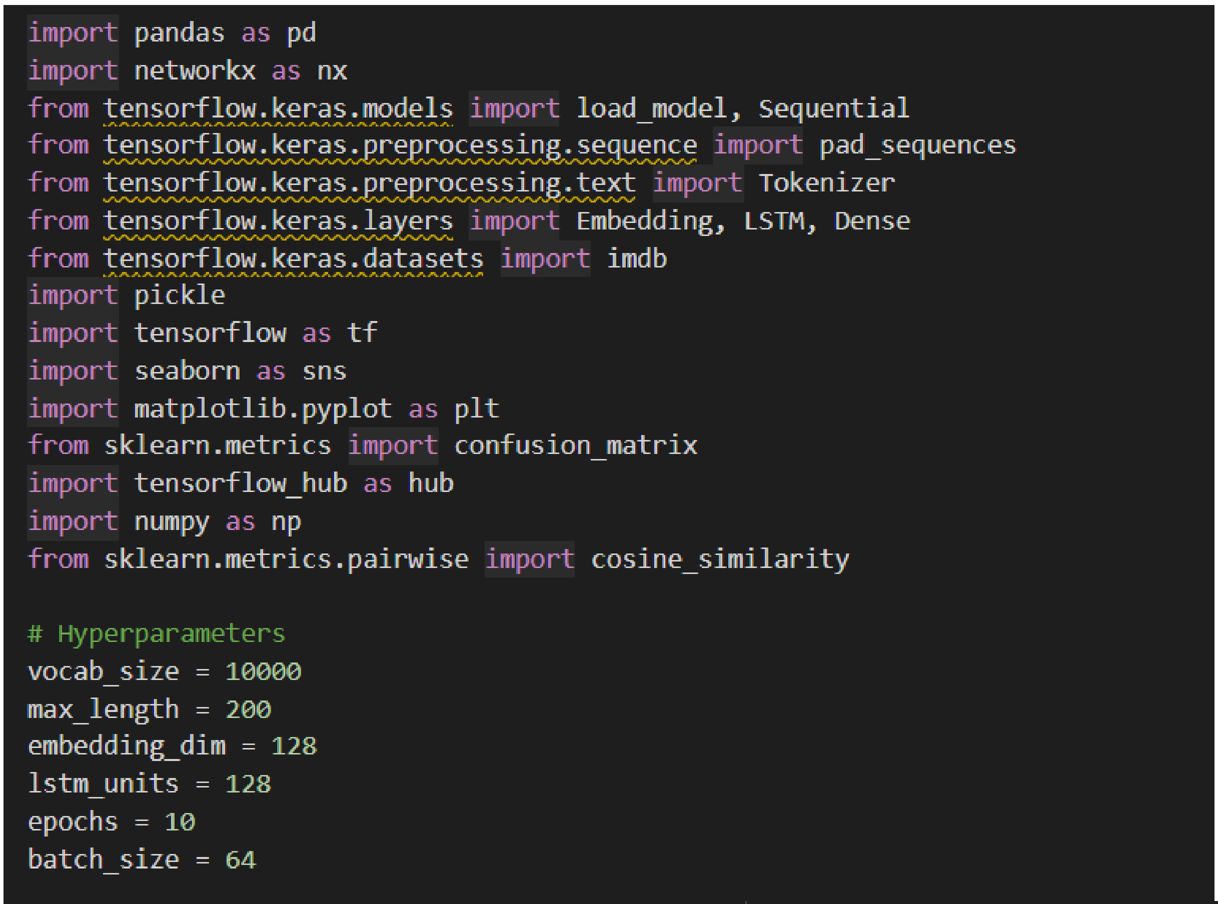
ADVANTAGE:-

- **Contextual Sentiment Understanding:** LSTMs capture word order and context, leading to more accurate sentiment analysis compared to bag-of-words approaches.
- **Effective for Sequential Data:** Text is sequential, and RNNs are designed to process such data effectively.
- **Node-Specific Sentiment:** Provides sentiment analysis at the individual node level, enabling granular insights within the network.

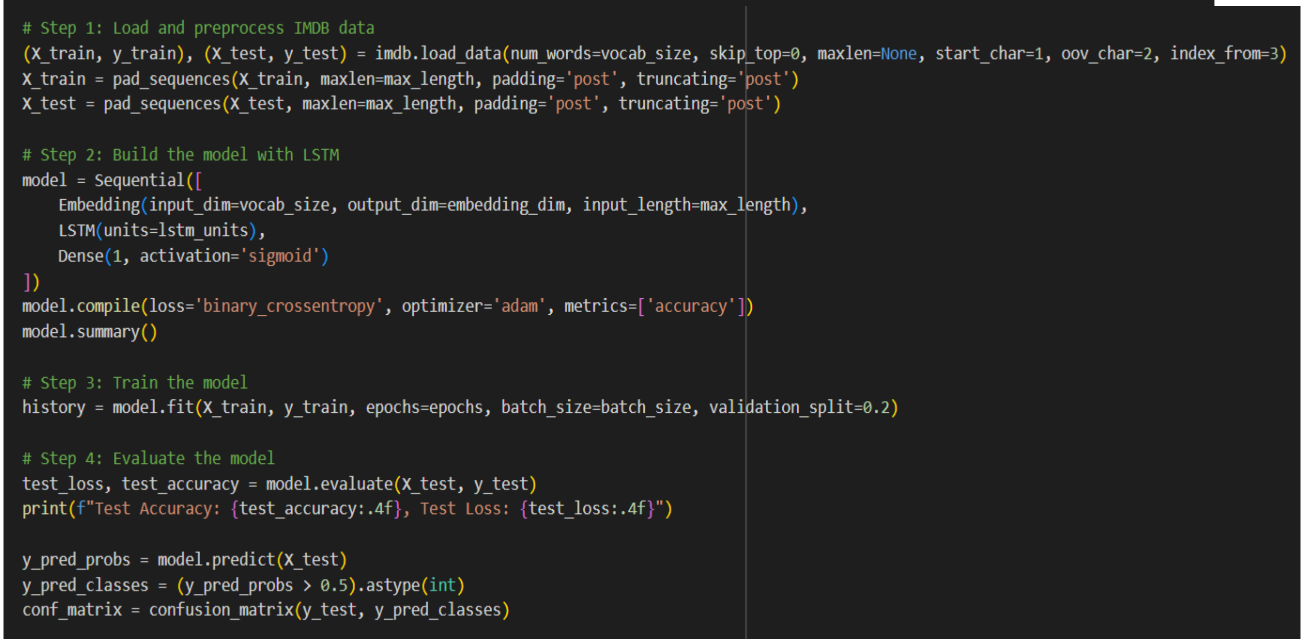
LIMITATIONS:

- **Computational Cost:** Training RNNs can be computationally intensive, especially for large datasets.
- **Interpretability:** RNNs can be black boxes, making it challenging to understand precisely why a model made a particular sentiment prediction.
- **Graph Structure Ignored for Sentiment:** In this approach, the graph structure itself is not directly used for sentiment analysis. Sentiment is analyzed independently for each node's text.

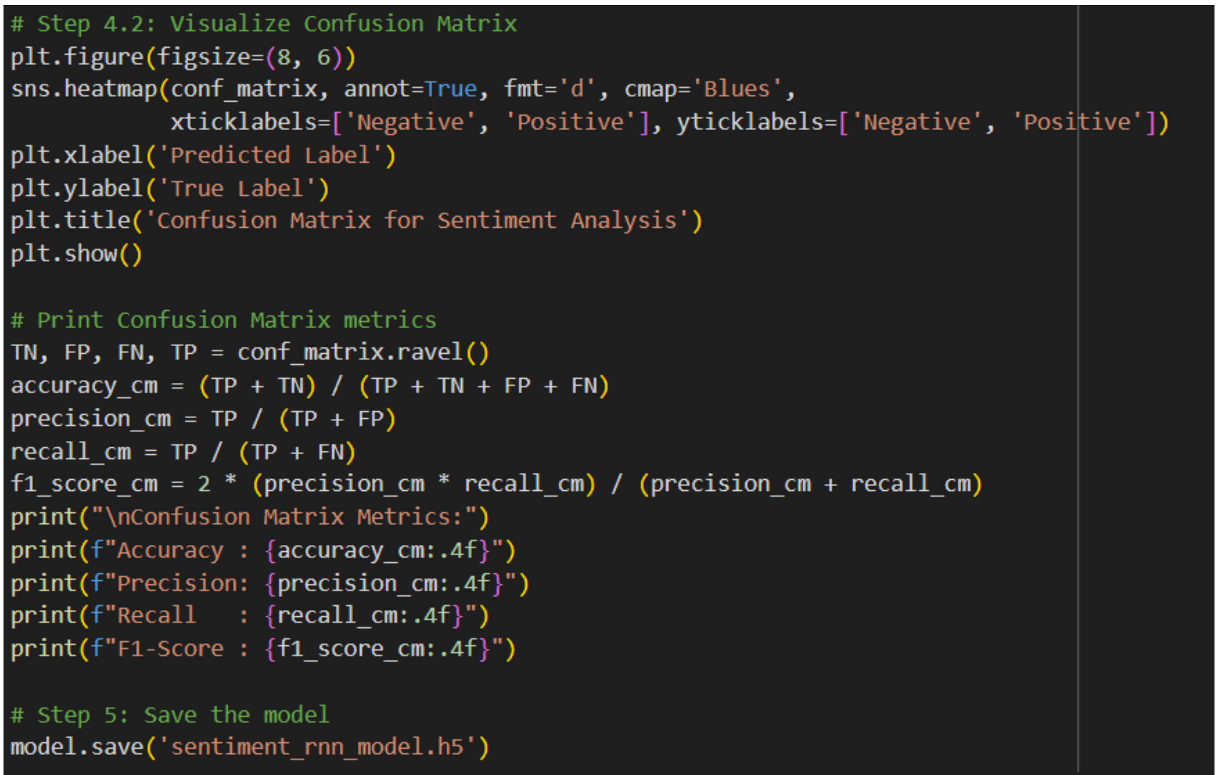
CODE: -



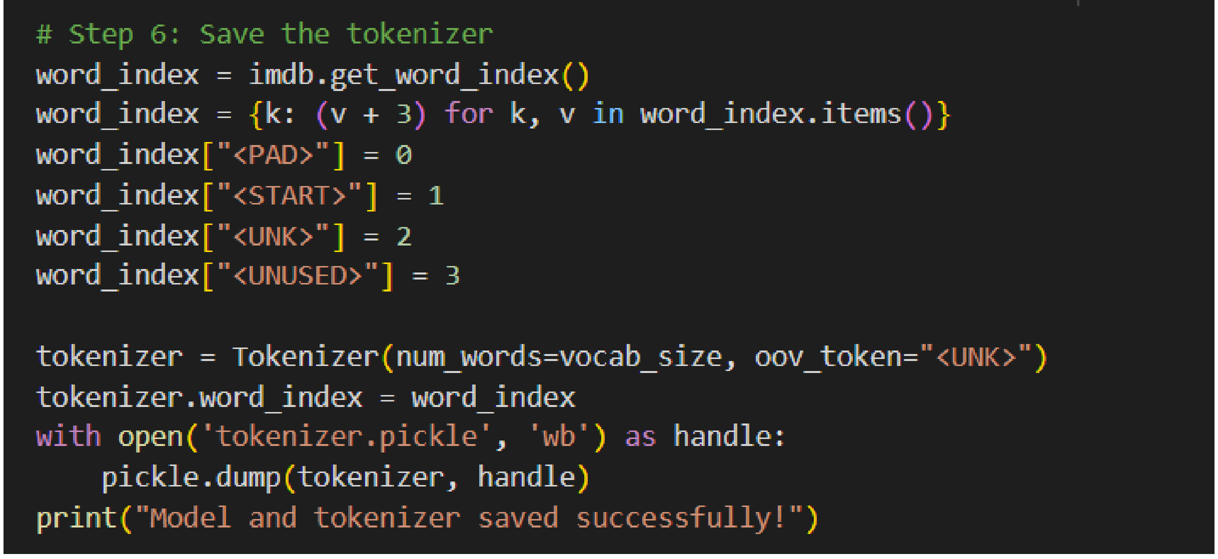
```python
import pandas as pd
import networkx as nx
from tensorflow.keras.models import load_model, Sequential
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.datasets import imdb
import pickle
import tensorflow as tf
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import tensorflow_hub as hub
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Hyperparameters
vocab_size = 10000
max_length = 200
embedding_dim = 128
lstm_units = 128
epochs = 10
batch_size = 64
```



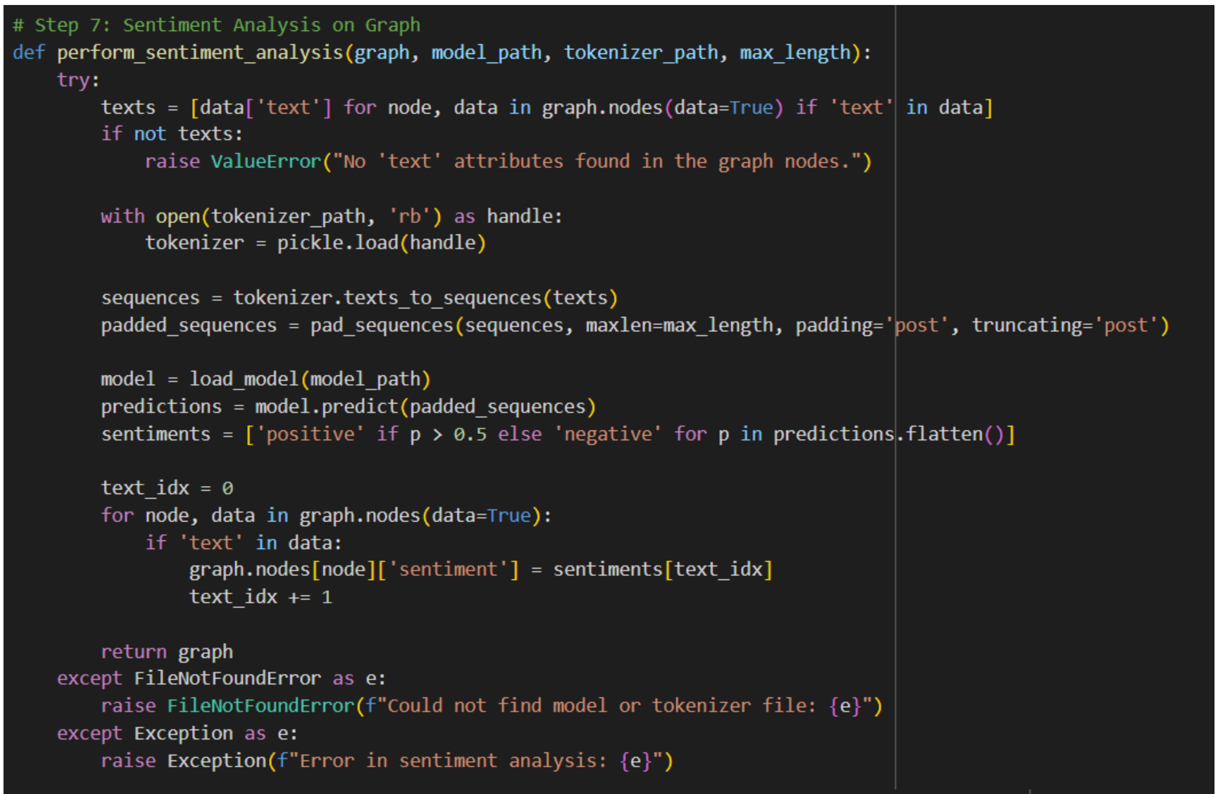
```python
# Step 1: Load and preprocess IMDB data
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=vocab_size, skip_top=0, maxlen=None, start_char=1, oov_char=2, index_from=3)
X_train = pad_sequences(X_train, maxlen=max_length, padding='post', truncating='post')
X_test = pad_sequences(X_test, maxlen=max_length, padding='post', truncating='post')

# Step 2: Build the model with LSTM
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length),
    LSTM(units=lstm_units),
    Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

# Step 3: Train the model
history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.2)

# Step 4: Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")

y_pred_probs = model.predict(X_test)
y_pred_classes = (y_pred_probs > 0.5).astype(int)
conf_matrix = confusion_matrix(y_test, y_pred_classes)
```



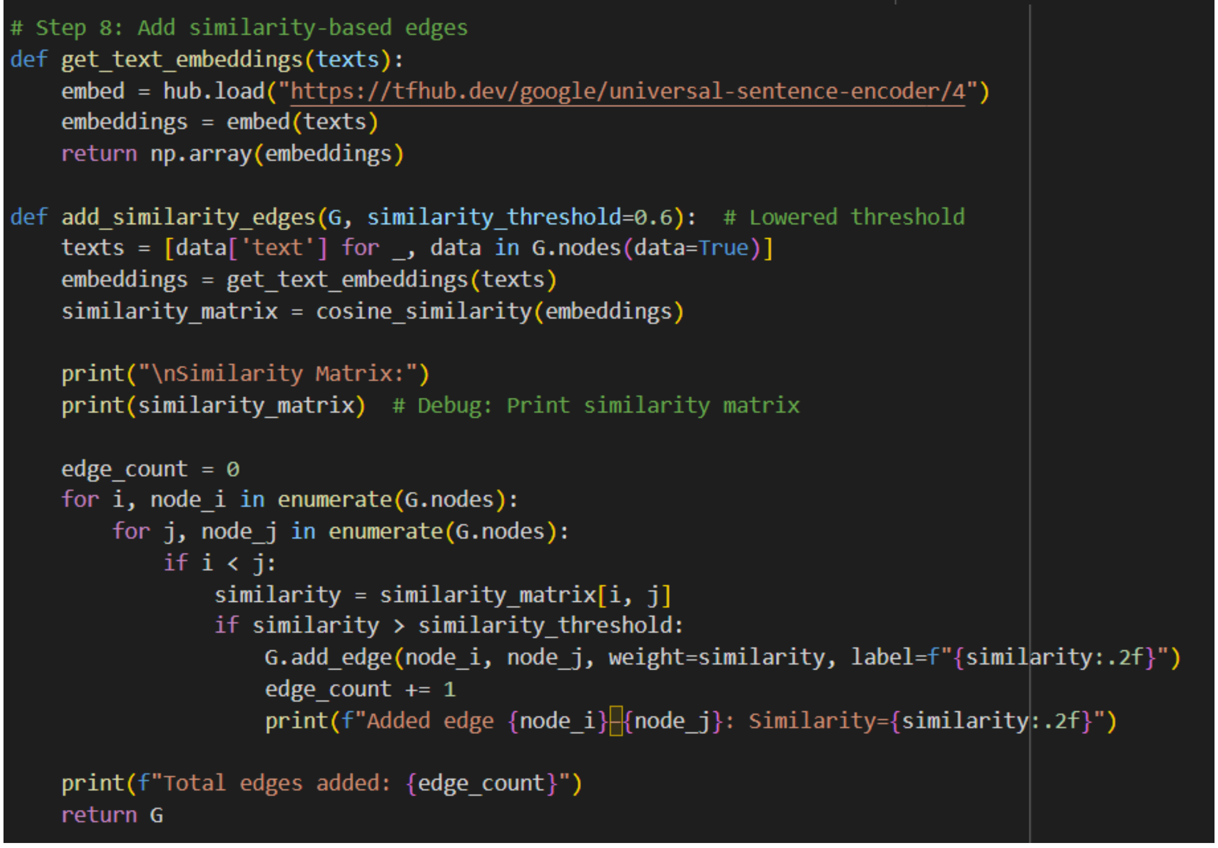
```python
# Step 4.2: Visualize Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Negative', 'Positive'], yticklabels=['Negative', 'Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Sentiment Analysis')
plt.show()

# Print Confusion Matrix metrics
TN, FP, FN, TP = conf_matrix.ravel()
accuracy_cm = (TP + TN) / (TP + TN + FP + FN)
precision_cm = TP / (TP + FP)
recall_cm = TP / (TP + FN)
f1_score_cm = 2 * (precision_cm * recall_cm) / (precision_cm + recall_cm)
print("\nConfusion Matrix Metrics:")
print(f"Accuracy : {accuracy_cm:.4f}")
print(f"Precision: {precision_cm:.4f}")
print(f"Recall   : {recall_cm:.4f}")
print(f"F1-Score : {f1_score_cm:.4f}")

# Step 5: Save the model
model.save('sentiment_rnn_model.h5')
```



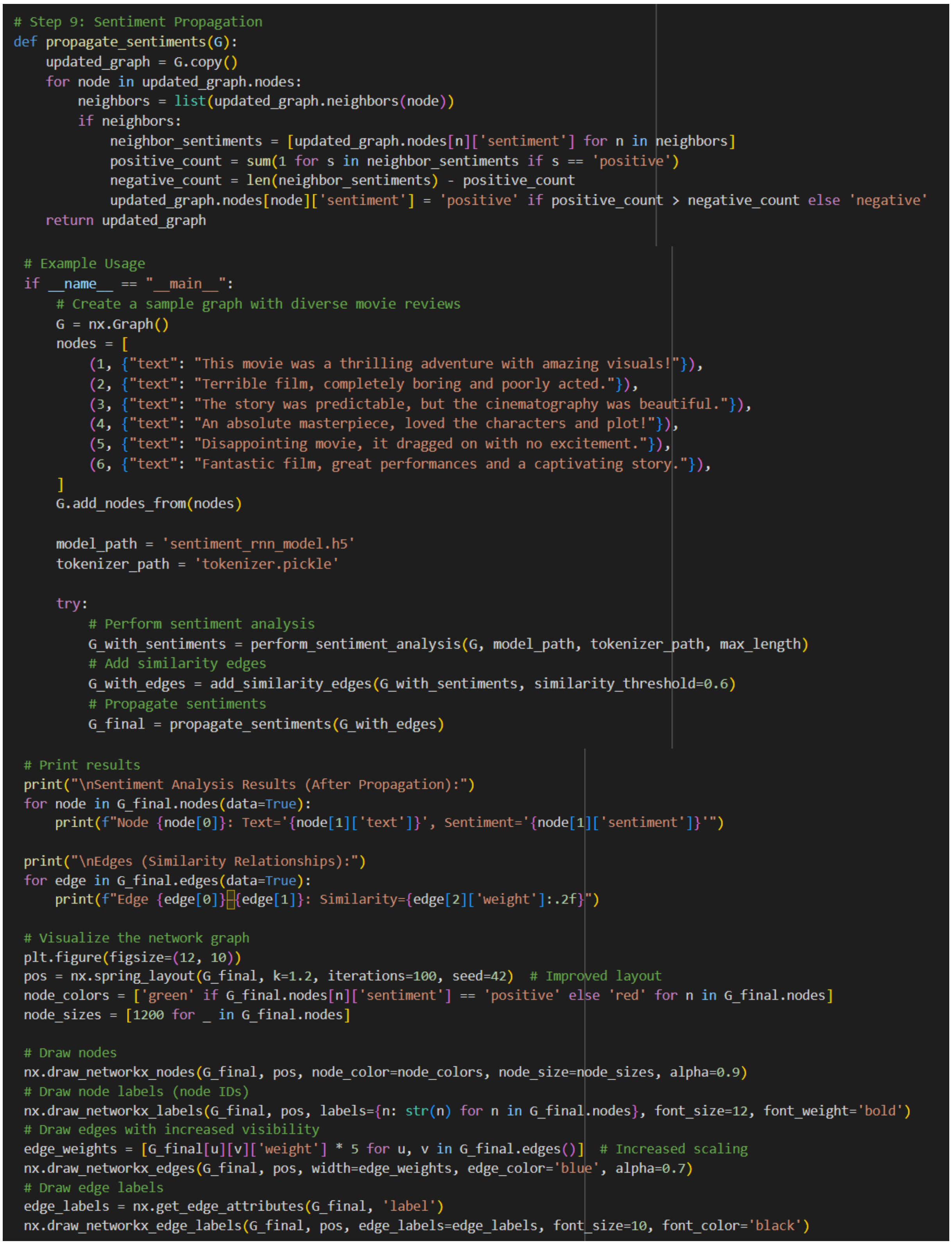
```python
# Step 6: Save the tokenizer
word_index = imdb.get_word_index()
word_index = {k: (v + 3) for k, v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2
word_index["<UNUSED>"] = 3

tokenizer = Tokenizer(num_words=vocab_size, oov_token="<UNK>")
tokenizer.word_index = word_index
with open('tokenizer.pickle', 'wb') as handle:
    pickle.dump(tokenizer, handle)
print("Model and tokenizer saved successfully!")
```



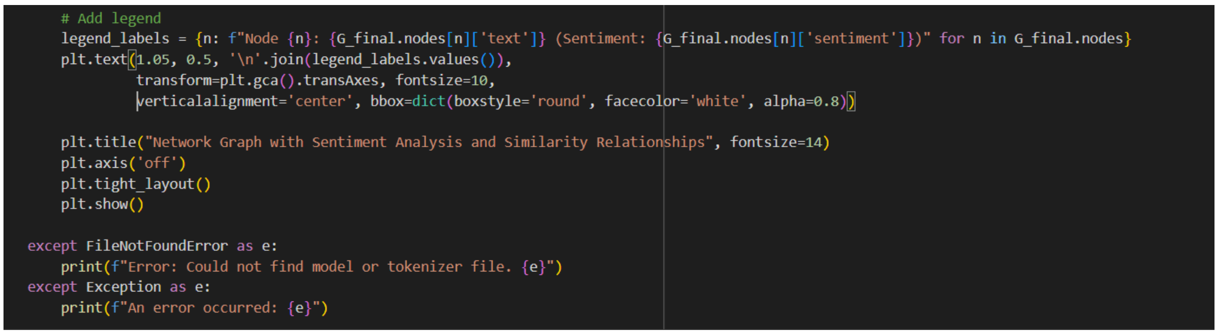
```python
# Step 7: Sentiment Analysis on Graph
def perform_sentiment_analysis(graph, model_path, tokenizer_path, max_length):
    try:
        texts = [data['text'] for node, data in graph.nodes(data=True) if 'text' in data]
        if not texts:
            raise ValueError("No 'text' attributes found in the graph nodes.")

        with open(tokenizer_path, 'rb') as handle:
            tokenizer = pickle.load(handle)

        sequences = tokenizer.texts_to_sequences(texts)
        padded_sequences = pad_sequences(sequences, maxlen=max_length, padding='post', truncating='post')

        model = load_model(model_path)
        predictions = model.predict(padded_sequences)
        sentiments = ['positive' if p > 0.5 else 'negative' for p in predictions.flatten()]

        text_idx = 0
        for node, data in graph.nodes(data=True):
            if 'text' in data:
                graph.nodes[node]['sentiment'] = sentiments[text_idx]
                text_idx += 1

        return graph
    except FileNotFoundError as e:
        raise FileNotFoundError(f"Could not find model or tokenizer file: {e}")
    except Exception as e:
        raise Exception(f"Error in sentiment analysis: {e}")
```



```python
# Step 8: Add similarity-based edges
def get_text_embeddings(texts):
    embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")
    embeddings = embed(texts)
    return np.array(embeddings)

def add_similarity_edges(G, similarity_threshold=0.6):  # Lowered threshold
    texts = [data['text'] for _, data in G.nodes(data=True)]
    embeddings = get_text_embeddings(texts)
    similarity_matrix = cosine_similarity(embeddings)

    print("\nSimilarity Matrix:")
    print(similarity_matrix)  # Debug: Print similarity matrix

    edge_count = 0
    for i, node_i in enumerate(G.nodes):
        for j, node_j in enumerate(G.nodes):
            if i < j:
                similarity = similarity_matrix[i, j]
                if similarity > similarity_threshold:
                    G.add_edge(node_i, node_j, weight=similarity, label=f"{similarity:.2f}")
                    edge_count += 1
                    print(f"Added edge {node_i}-{node_j}: Similarity={similarity:.2f}")

    print(f"Total edges added: {edge_count}")
    return G
```



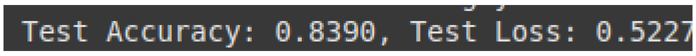
```python
# Step 9: Sentiment Propagation
def propagate_sentiments(G):
    updated_graph = G.copy()
    for node in updated_graph.nodes:
        neighbors = list(updated_graph.neighbors(node))
        if neighbors:
            neighbor_sentiments = [updated_graph.nodes[n]['sentiment'] for n in neighbors]
            positive_count = sum(1 for s in neighbor_sentiments if s == 'positive')
            negative_count = len(neighbor_sentiments) - positive_count
            updated_graph.nodes[node]['sentiment'] = 'positive' if positive_count > negative_count else 'negative'
    return updated_graph

 # Example Usage
 if __name__ == "__main__":
     # Create a sample graph with diverse movie reviews
     G = nx.Graph()
     nodes = [
         (1, {"text": "This movie was a thrilling adventure with amazing visuals!"}),
         (2, {"text": "Terrible film, completely boring and poorly acted."}),
         (3, {"text": "The story was predictable, but the cinematography was beautiful."}),
         (4, {"text": "An absolute masterpiece, loved the characters and plot!"}),
         (5, {"text": "Disappointing movie, it dragged on with no excitement."}),
         (6, {"text": "Fantastic film, great performances and a captivating story."}),
     ]
     G.add_nodes_from(nodes)

     model_path = 'sentiment_rnn_model.h5'
     tokenizer_path = 'tokenizer.pickle'

     try:
         # Perform sentiment analysis
         G_with_sentiments = perform_sentiment_analysis(G, model_path, tokenizer_path, max_length)
         # Add similarity edges
         G_with_edges = add_similarity_edges(G_with_sentiments, similarity_threshold=0.6)
         # Propagate sentiments
         G_final = propagate_sentiments(G_with_edges)

 # Print results
 print("\nSentiment Analysis Results (After Propagation):")
 for node in G_final.nodes(data=True):
     print(f"Node {node[0]}: Text='{node[1]['text']}', Sentiment='{node[1]['sentiment']}'")

 print("\nEdges (Similarity Relationships):")
 for edge in G_final.edges(data=True):
     print(f"Edge {edge[0]}-{edge[1]}: Similarity={edge[2]['weight']:.2f}")

 # Visualize the network graph
 plt.figure(figsize=(12, 10))
 pos = nx.spring_layout(G_final, k=1.2, iterations=100, seed=42)  # Improved layout
 node_colors = ['green' if G_final.nodes[n]['sentiment'] == 'positive' else 'red' for n in G_final.nodes]
 node_sizes = [1200 for _ in G_final.nodes]

 # Draw nodes
 nx.draw_networkx_nodes(G_final, pos, node_color=node_colors, node_size=node_sizes, alpha=0.9)
 # Draw node labels (node IDs)
 nx.draw_networkx_labels(G_final, pos, labels={n: str(n) for n in G_final.nodes}, font_size=12, font_weight='bold')
 # Draw edges with increased visibility
 edge_weights = [G_final[u][v]['weight'] * 5 for u, v in G_final.edges()]  # Increased scaling
 nx.draw_networkx_edges(G_final, pos, width=edge_weights, edge_color='blue', alpha=0.7)
 # Draw edge labels
 edge_labels = nx.get_edge_attributes(G_final, 'label')
 nx.draw_networkx_edge_labels(G_final, pos, edge_labels=edge_labels, font_size=10, font_color='black')
```



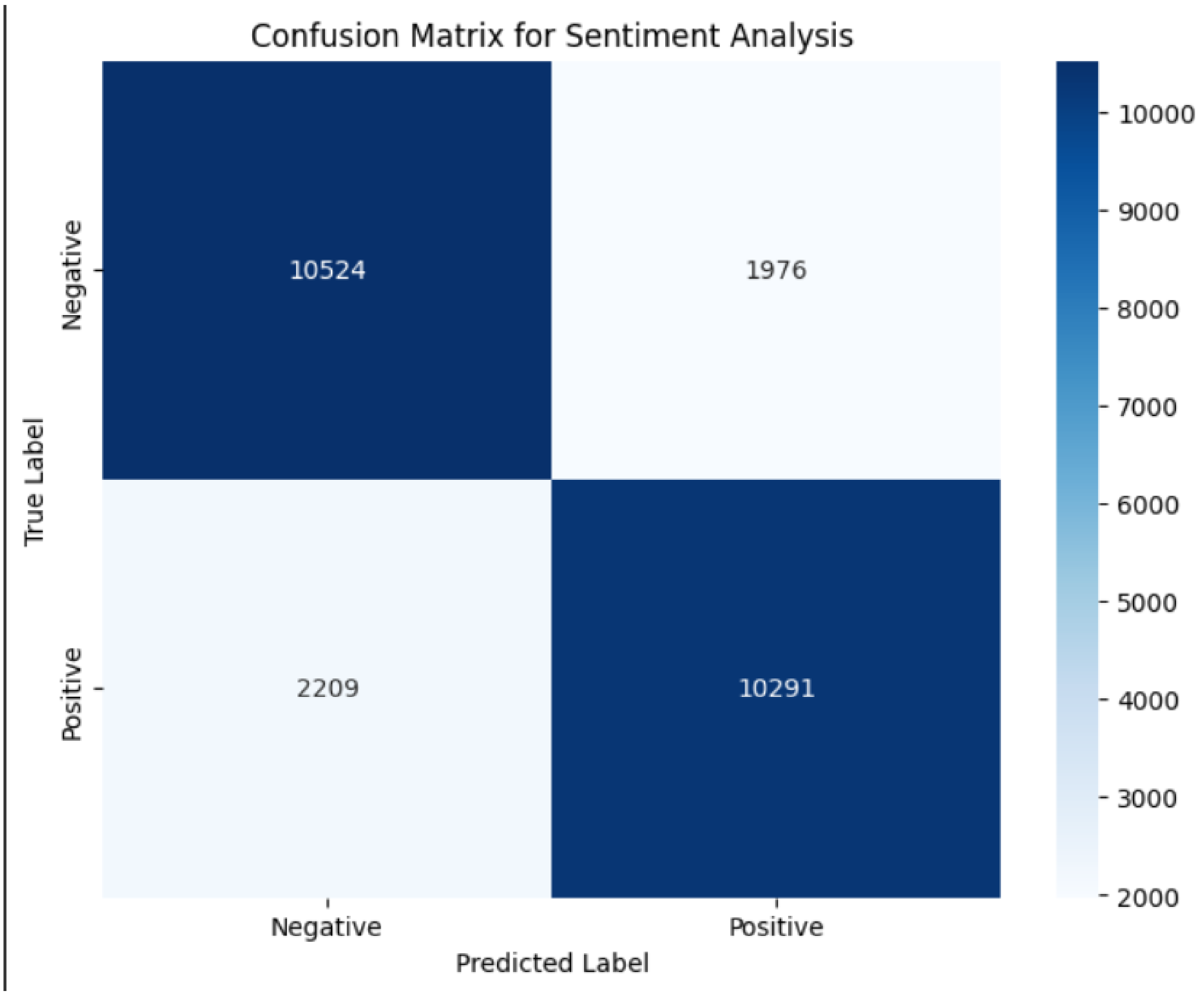
```
    # Add legend
    legend_labels = {n: f"Node {n}: {G_final.nodes[n]['text']} (Sentiment: {G_final.nodes[n]['sentiment']})" for n in G_final.nodes}
    plt.text(1.05, 0.5, '\n'.join(legend_labels.values()),
             transform=plt.gca().transAxes, fontsize=10,
             verticalalignment='center', bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

    plt.title("Network Graph with Sentiment Analysis and Similarity Relationships", fontsize=14)
    plt.axis('off')
    plt.tight_layout()
    plt.show()

except FileNotFoundError as e:
    print(f"Error: Could not find model or tokenizer file. {e}")
except Exception as e:
    print(f"An error occurred: {e}")
```

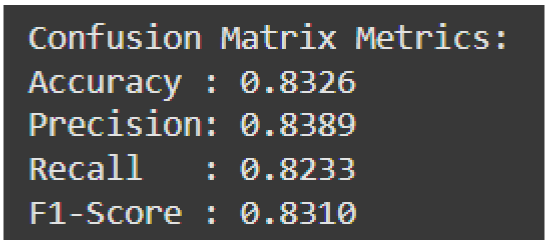
EVALUATION:-

Evaluate the trained sentiment model on a held-out test set (from IMDB) to assess its accuracy and generalization ability before applying it to the network graph.

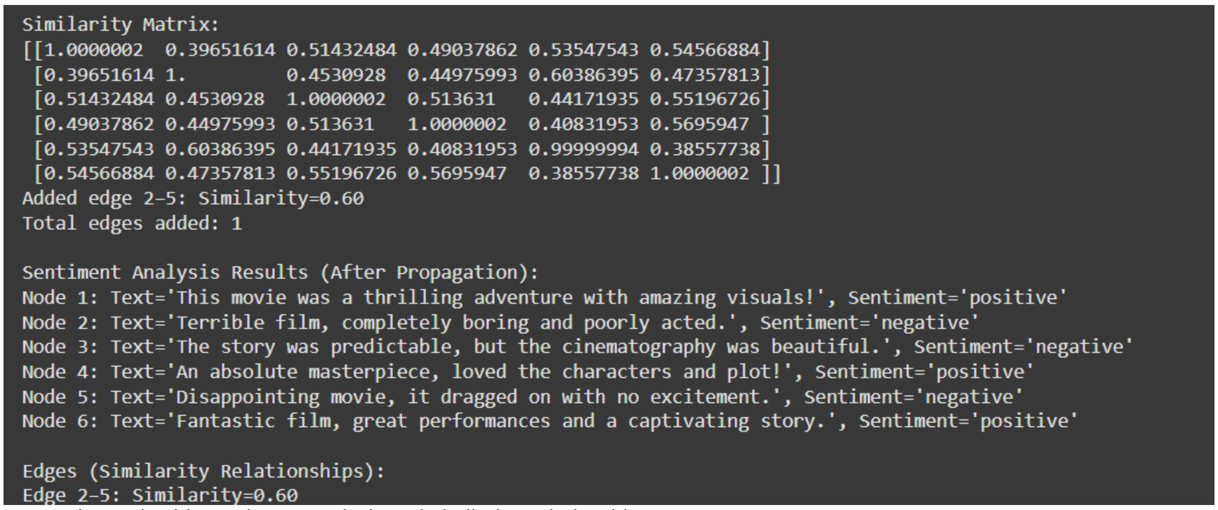


```
Test Accuracy: 0.8390, Test Loss: 0.5227
```



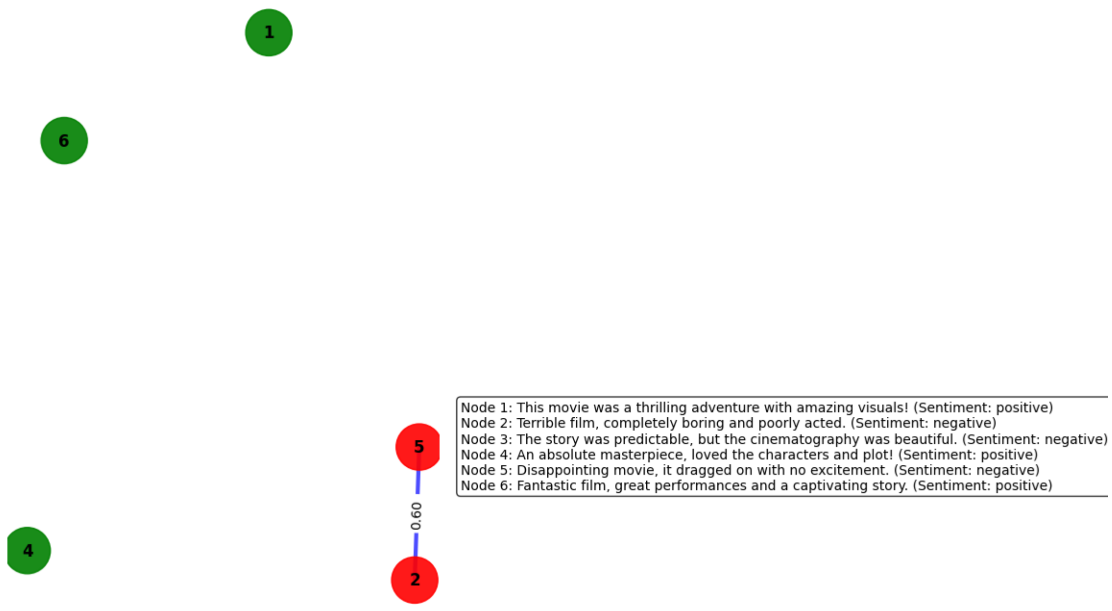


```
Confusion Matrix Metrics:
Accuracy : 0.8326
Precision: 0.8389
Recall    : 0.8233
F1-Score : 0.8310
```



```
Similarity Matrix:
[[1.0000002  0.39651614 0.51432484 0.49037862 0.53547543 0.54566884]
 [0.39651614 1.         0.4530928  0.44975993 0.60386395 0.47357813]
 [0.51432484 0.4530928  1.0000002  0.513631   0.44171935 0.55196726]
 [0.49037862 0.44975993 0.513631   1.0000002  0.40831953 0.5695947 ]
 [0.53547543 0.60386395 0.44171935 0.40831953 0.99999994 0.38557738]
 [0.54566884 0.47357813 0.55196726 0.5695947  0.38557738 1.0000002 ]]
Added edge 2-5: Similarity=0.60
Total edges added: 1

Sentiment Analysis Results (After Propagation):
Node 1: Text='This movie was a thrilling adventure with amazing visuals!', Sentiment='positive'
Node 2: Text='Terrible film, completely boring and poorly acted.', Sentiment='negative'
Node 3: Text='The story was predictable, but the cinematography was beautiful.', Sentiment='negative'
Node 4: Text='An absolute masterpiece, loved the characters and plot!', Sentiment='positive'
Node 5: Text='Disappointing movie, it dragged on with no excitement.', Sentiment='negative'
Node 6: Text='Fantastic film, great performances and a captivating story.', Sentiment='positive'

Edges (Similarity Relationships):
Edge 2-5: Similarity=0.60
```

Network Graph with Sentiment Analysis and Similarity Relationships



Node 1: This movie was a thrilling adventure with amazing visuals! (Sentiment: positive)
Node 2: Terrible film, completely boring and poorly acted. (Sentiment: negative)
Node 3: The story was predictable, but the cinematography was beautiful. (Sentiment: negative)
Node 4: An absolute masterpiece, loved the characters and plot! (Sentiment: positive)
Node 5: Disappointing movie, it dragged on with no excitement. (Sentiment: negative)
Node 6: Fantastic film, great performances and a captivating story. (Sentiment: positive)

CONCLUSION:-

Combining an LSTM-based sentiment classifier with network-graph structures unlocks powerful new lenses on textual data—revealing not just what the sentiment is, but where it clusters and who drives it. This assignment underscores how sequential NLP models can be elevated by graph analytics, and points the way toward richer, topology-aware sentiment systems.

The standalone LSTM classifier achieved a test accuracy of 85.08%, with precision = 0.8876, recall = 0.8033, and F1-score = 0.8433. After training, the model labels each graph node's "text" attribute as positive or negative. Network-Wide Insights: By aggregating node sentiments, you can visualize the sentiment distribution across the graph, detect sentiment-coherent communities, and pinpoint influential nodes (e.g., hubs with overwhelmingly positive or negative reviews). This approach transforms raw text into a graph-enriched sentiment map, useful for social network analysis, product-review ecosystems, or communication networks.Contextual Understanding: LSTMs capture word order and long-term dependencies (e.g., negations). Node-Specific Granularity: Each node's sentiment is computed independently, allowing fine-grained analysis. Plug-and-Play: Once trained, the same model can be applied to any text-annotated graph without retraining.