If you want to quickly **check your version**, run this in any Jupyter cell:

python
CopyEdit
```
import sklearn

print(sklearn.__version__)
```

If it shows something like `0.24`, `0.23`, `0.22`, then you definitely need to update.

✅ After updating, it should show something like `1.2.2`, `1.3.0`, `1.4.2`, etc.

Close your Jupyter Notebook.

Open **Anaconda Prompt**.

Run:

bash
CopyEdit
```
conda update scikit-learn
```

```
# Step 1: Install necessary libraries (if not installed)
# !pip install scikit-learn pandas matplotlib

# Step 2: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Step 3: Load the dataset
url = "https://raw.githubusercontent.com/selva86/datasets/master/BostonHousing.csv"
data = pd.read_csv(url)

X = data.drop('medv', axis=1).values  # features
y = data['medv'].values               # target

# Step 4: Preprocess (Scaling)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 5: Split the data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Step 6: Build the Neural Network model
model = MLPRegressor(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', max_iter=500, random_state=42)

# Step 7: Train the model
model.fit(X_train, y_train)

# Step 8: Predict
y_pred = model.predict(X_test)

# Step 9: Evaluate
print(f"Mean Absolute Error: {mean_absolute_error(y_test, y_pred):.2f}")
print(f"Mean Squared Error: {mean_squared_error(y_test, y_pred):.2f}")
print(f"R2 Score: {r2_score(y_test, y_pred):.2f}")

# Step 10: Plot Actual vs Predicted
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Housing Prices")
plt.plot([0, 50], [0, 50], '--r')
plt.show()

EXPLANATION:

🧠 What's this project about?

You're building a **Neural Network Regression model** (with `MLPRegressor`) to predict **Boston housing prices** using features like crime rate, number of rooms, tax rate, etc.

✅ Step 1: Install necessary libraries

python
CopyEdit

```
# !pip install scikit-learn pandas matplotlib
```

- This command is commented out, but it's used to install packages:
    - `scikit-learn`: For machine learning (training, evaluation).
    - `pandas`: To handle tabular data (CSV files).

- `matplotlib`: To visualize results (plots).

You don't need to run this unless you haven't installed the packages already.

✅ Step 2: Import libraries

python
CopyEdit
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

Library	Purpose
`numpy`	For numerical operations (arrays, math, etc.)
`pandas`	To load and manipulate datasets easily
`matplotlib.pyplot`	For plotting graphs (like actual vs predicted)
`train_test_split`	To split the data into training and testing sets
`StandardScaler`	For feature scaling (brings all features to similar range)
`MLPRegressor`	Neural network regressor (multi-layer perceptron)
`metrics`	To evaluate model performance (MAE, MSE, R^2)

✅ Step 3: Load the dataset

python
CopyEdit
```
url = "https://raw.githubusercontent.com/selva86/datasets/master/BostonHousing.csv"
data = pd.read_csv(url)
```

- Loads the **Boston Housing dataset** directly from GitHub.
- `data` is a Pandas DataFrame containing 14 columns.
    - 13 input features (e.g., `crim`, `rm`, `tax`, etc.)
    - 1 target column: `medv` (Median house price in $1000s)

✅ Step 4: Preprocess (Scaling)

python

CopyEdit
```
X = data.drop('medv', axis=1).values  # features
y = data['medv'].values               # target

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

- Separates **features (`X`)** from **target (`y`)**.
- `StandardScaler()` scales the data to **zero mean and unit variance**.
    - This helps the neural network learn faster and more accurately.

✅ Step 5: Split the data

python
CopyEdit
```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)
```

- Splits the data:
    - **80% for training**
    - **20% for testing**
- `random_state=42` makes sure results are reproducible.

✅ Step 6: Build the Neural Network

python
CopyEdit
```
model = MLPRegressor(
    hidden_layer_sizes=(64, 32),
    activation='relu',
    solver='adam',
    max_iter=500,
    random_state=42
)
```

- This creates a **Neural Network** with:
    - 2 hidden layers: one with 64 neurons, another with 32 neurons
    - `activation='relu'`: Non-linearity for better learning
    - `solver='adam'`: Optimizer for adjusting weights
    - `max_iter=500`: Number of training iterations

✅ Step 7: Train the model

python
CopyEdit
```
model.fit(X_train, y_train)
```

- This trains the neural network using the training data.

✅ Step 8: Predict

python
CopyEdit
```
y_pred = model.predict(X_test)
```

- Uses the trained model to predict prices for the test set.

✅ Step 9: Evaluate

python
CopyEdit
```
print(f"Mean Absolute Error: {mean_absolute_error(y_test, y_pred):.2f}")
print(f"Mean Squared Error: {mean_squared_error(y_test, y_pred):.2f}")
print(f"R2 Score: {r2_score(y_test, y_pred):.2f}")
```

Metric	Meaning
`MAE`	Average error (absolute) between predicted and actual prices
`MSE`	Squared error (penalizes big errors more)
`R² Score`	How well the model explains the data (closer to 1 is better)

✅ Step 10: Plot Actual vs Predicted

python
CopyEdit
```
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted Housing Prices")
plt.plot([0, 50], [0, 50], '--r')
plt.show()
```

- Plots a scatter plot of actual vs predicted house prices.
- The red dashed line shows perfect prediction (where actual = predicted).
- Points near the line → good predictions, far away → errors.

🎯 Summary Table

Step	Task	Purpose
1	Install libraries	Setup
2	Import	Bring in necessary tools
3	Load dataset	Read Boston housing data
4	Scale	Normalize features
5	Split	Divide data into training/testing
6	Build model	Create neural network
7	Train	Fit the model
8	Predict	Get predictions
9	Evaluate	Measure model performance
10	Plot	Visualize how well model predicted

DL 2

Classification using Deep neural network (Any One from the following) 1. Multiclass classification using Deep Neural Networks: Example: Use the OCR letter recognition dataset https://archive.ics.uci.edu/ml/datasets/letter+recognition

```
# Step 1: Import required libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Step 2: Load the dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recognition.data"
```

```
columns = ['letter','x-box','y-box','width','height','onpix','x-bar','y-bar',
           'x2bar','y2bar','xybar','x2ybr','xy2br','x-ege','xegvy','y-ege','yegvx']

data = pd.read_csv(url, names=columns)

# Step 3: Split features and labels
X = data.drop('letter', axis=1)
y = data['letter']

# Encode target letters (A–Z) to numbers (0–25)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Step 4: Preprocess the data (scaling)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 5: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2, random_state=42)

# Step 6: Define and train the DNN model
model = MLPClassifier(hidden_layer_sizes=(128, 64), activation='relu', solver='adam',
                      max_iter=300, random_state=42)

model.fit(X_train, y_train)

# Step 7: Predict
y_pred = model.predict(X_test)

# Step 8: Evaluation
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))

# Optional: Confusion matrix heatmap
plt.figure(figsize=(12,8))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_,
cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

🔵 Code Walkthrough:

Step-by-Step Process

Install Libraries (If not already installed):

python
CopyEdit

```
# Uncomment the following line if libraries are not installed
# !pip install scikit-learn pandas matplotlib
```

1. **Purpose**:
   Installs the required libraries:
   - **scikit-learn**: Machine learning algorithms.
   - **pandas**: For data handling and analysis.
   - **matplotlib**: For creating visualizations.

Import Libraries:

python
CopyEdit
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
import seaborn as sns
```

2. **Purpose**:
   - `numpy` and `pandas`: For handling numerical and data operations.
   - `matplotlib.pyplot`: For visualizing the results.
   - `sklearn.model_selection`: For splitting data into training/testing sets.
   - `sklearn.preprocessing`: For preprocessing data (scaling and encoding).
   - `sklearn.neural_network`: For building the Multi-Layer Perceptron (MLP) classifier.
   - `sklearn.metrics`: For model evaluation metrics (accuracy, confusion matrix, etc.).
   - `seaborn`: For creating beautiful visualizations (like heatmaps).

Loading and Preprocessing Data:

Load the Dataset:

python
CopyEdit
```
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recogniti
on.data"
data = pd.read_csv(url, header=None)
```

3. **Purpose**:
   Loads the **Letter Recognition dataset** from the UCI repository into a **pandas DataFrame**.

Assign Features (X) and Labels (y):

python
CopyEdit
```
X = data.drop(0, axis=1)  # Features (all columns except the first one)
```

```
y = data[0]               # Labels (the first column with letters)
```

4. **Purpose**:
   - `X`: The features (everything except the first column).
   - `y`: The target labels (the first column, which represents letters).

Data Preprocessing:

Encoding the Labels:

python
CopyEdit
```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

5. **Purpose**:
   - **Label Encoding**: Converts **categorical labels** (letters like A, B, C...) into **numeric values** (0, 1, 2...).
   - `fit_transform()` method learns the encoding and applies it to the data.
6. **Example**:
   - A → 0, B → 1, C → 2, ...

Scaling Features:

python
CopyEdit
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

6. **Purpose**:
   - **Standardization**: Scales the features to have **mean = 0** and **standard deviation = 1** using `StandardScaler`.
   - Helps models learn faster and more effectively when the features are on the same scale.

Splitting Data:

Splitting Data into Training and Testing:

python
CopyEdit
```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)
```

7. **Purpose**:

- Split the data into **training** (80%) and **testing** (20%) sets using `train_test_split`.
- `random_state=42`: Ensures the split is **reproducible** (i.e., same data every time).

Building and Training the Model:

Building the MLP Classifier:

python
CopyEdit
```
model = MLPClassifier(hidden_layer_sizes=(128, 64), activation='relu', solver='adam',
max_iter=300, random_state=42)
```

8. **Purpose**:
   - Creates a **Deep Neural Network** (MLP) with two hidden layers (128 and 64 neurons).
   - Uses the **ReLU** activation function and **Adam optimizer**.
   - `max_iter=300`: Limits training iterations to 300 for quicker training.

Training the Model:

python
CopyEdit
```
model.fit(X_train, y_train)
```

9. **Purpose**:
   - **Training the model** on the training data (`X_train` and `y_train`).

Evaluation and Visualization:

Making Predictions:

python
CopyEdit
```
y_pred = model.predict(X_test)
```

10. **Purpose**:
    - **Make predictions** on the test set (`X_test`).

Evaluating the Model:

python
CopyEdit
```
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print(f"Classification Report:\n{classification_report(y_test, y_pred)}")
```

11. **Purpose**:

- `accuracy_score`: Calculates the overall accuracy of the model.
- `classification_report`: Provides a detailed breakdown of **precision**, **recall**, and **F1-score** for each class.

Confusion Matrix:

python
CopyEdit
```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=label_encoder.classes_,
yticklabels=label_encoder.classes_)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

12. **Purpose**:
    - **Confusion Matrix**: Visualizes how well the model's predictions match the true labels.
    - `sns.heatmap()`: Creates a **heatmap** for better visualization.
    - `xticklabels` and `yticklabels`: Display the corresponding letters instead of numbers.

🔵 Key Concepts and Explanations:

🔵 `train_test_split`

- **Purpose**:
  Splits the dataset into **training** and **testing** datasets.
    - Training set: Used to train the model.
    - Test set: Used to evaluate the model's performance.

🔵 `StandardScaler`

- **Purpose**:
  Standardizes the features by scaling them to have **zero mean** and **unit variance**.
    - Helps prevent dominance of features with large scales (like height vs. weight).

🔵 `LabelEncoder` and Encoding

- **Purpose**:
  LabelEncoder converts **categorical labels** into **numeric labels**.
  Encoding is the process of converting text labels (like "A", "B", "C") into numeric values (0, 1, 2).

🔵 `MLPClassifier`

- **Purpose**:
  The **Multi-Layer Perceptron** is a type of **Deep Neural Network**.
  Used here for **classification**, as it can predict **categories** based on input features.

🔵 `classification_report`

- **Purpose**:
  Provides a comprehensive evaluation of model performance using **precision**, **recall**, and **F1-score** for each class.

🔵 `accuracy_score`

- **Purpose**:
  Measures the **overall accuracy** of the model by calculating the percentage of correct predictions.

🔵 `confusion_matrix`

- **Purpose**:
  Displays a table showing the counts of **true positives**, **false positives**, **true negatives**, and **false negatives**, allowing you to visually inspect the model's performance.

🔵 `plt` and `sns`

- **Purpose**:
  - `plt`: For creating **basic plots** (scatter, line, etc.).
  - `sns`: For creating **prettier and more informative visualizations** like heatmaps.

🔵 `random_state=42`

- **Purpose**:
  Ensures that the random operations (like splitting data) are **reproducible**. The choice of 42 is simply a convention (from *The Hitchhiker's Guide to the Galaxy*).

🛑 Quick Recap of the Process:

Stage	Action
1	Import needed libraries
2	Load and prepare data (X, y)
3	Encode labels (A → 0, B → 1...)
4	Scale features (standardize values)
5	Split into train/test sets
6	Build MLP Classifier (Deep Neural Network)
7	Train the Model
8	Predict on Test Data
9	Measure Performance (accuracy, precision, recall, confusion matrix)
10	Visualize Confusion Matrix

Dl2 -2
Classification using Deep neural network 2. Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset

```python
print("Prajwal Gadhave BACO21145")
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import imdb
from keras import models, layers, optimizers, losses, metrics
from sklearn.metrics import mean_absolute_error


# 1. Load the IMDB dataset
# Keep only the top 10,000 most frequent words
NUM_WORDS = 10000
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=NUM_WORDS)
```

```python
# 2. Decode a sample review (optional)
word_index = imdb.get_word_index()
reverse_word_index = {value + 3: key for key, value in word_index.items()}
reverse_word_index[0] = "<PAD>"
reverse_word_index[1] = "<START>"
reverse_word_index[2] = "<UNK>"

decoded_review = ' '.join([reverse_word_index.get(i, '?') for i in train_data[0]])
print("Sample Decoded Review:\n", decoded_review)

# 3. Vectorize input data
def vectorize_sequences(sequences, dimension=NUM_WORDS):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0
    return results

X_train = vectorize_sequences(train_data)
X_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')

# 4. Prepare validation set
X_val = X_train[:10000]
partial_X_train = X_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

# 5. Build the model
model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=(NUM_WORDS,)),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# 6. Compile the model
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001),
    loss=losses.binary_crossentropy,
    metrics=[metrics.binary_accuracy]
)

# 7. Train the model
history = model.fit(
    partial_X_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

```python
# 8. Plot training and validation loss
history_dict = history.history
epochs = range(1, len(history_dict['loss']) + 1)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs, history_dict['loss'], 'bo', label='Training Loss')
plt.plot(epochs, history_dict['val_loss'], 'b', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# 9. Plot training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs, history_dict['binary_accuracy'], 'ro', label='Training Accuracy')
plt.plot(epochs, history_dict['val_binary_accuracy'], 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# 10. Evaluate on test data
results = model.evaluate(X_test, y_test)
print(f"\nTest Loss: {results[0]:.4f}, Test Accuracy: {results[1]:.4f}")

# 11. Predict on test data
predictions = model.predict(X_test)
y_pred = (predictions > 0.5).astype("int").flatten()

# 12. Calculate Mean Absolute Error
mae = mean_absolute_error(y_test, y_pred)
print(f"\nMean Absolute Error on Test Set: {mae:.4f}")
```

The provided code is an example of a **binary sentiment analysis** problem using the **IMDB movie reviews dataset**. The goal of the model is to predict whether a given review is positive or negative based on its content. Here's a detailed explanation of each part of the code:

---

1. Loading the IMDB Dataset

python
CopyEdit
```python
NUM_WORDS = 10000
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=NUM_WORDS)
```

- **IMDB Dataset**: The dataset consists of movie reviews and associated sentiment labels (positive/negative). It contains 25,000 reviews for training and 25,000 for testing.

- **`num_words=10000`**: The `num_words` parameter limits the dataset to the 10,000 most frequent words in the training data. Words that occur less frequently are discarded.
- **`train_data`**, **`train_labels`**, **`test_data`**, **`test_labels`**: These are the training and test datasets, where:
  - `train_data` and `test_data`: Contain the word indices (integers representing words).
  - `train_labels` and `test_labels`: Contain the sentiment labels (0 for negative, 1 for positive).

---

2. Decoding a Sample Review (Optional)

python
CopyEdit

```
word_index = imdb.get_word_index()
reverse_word_index = {value + 3: key for key, value in word_index.items()}
reverse_word_index[0] = "<PAD>"
reverse_word_index[1] = "<START>"
reverse_word_index[2] = "<UNK>"

decoded_review = ' '.join([reverse_word_index.get(i, '?') for i in train_data[0]])
print("Sample Decoded Review:\n", decoded_review)
```

- **`word_index`**: A dictionary where keys are words and values are the corresponding indices.
- **`reverse_word_index`**: This dictionary reverses `word_index`, mapping indices back to words. We add special tokens:
  - **`0`**: Padding token (`<PAD>`) for short reviews.
  - **`1`**: Start token (`<START>`), indicating the beginning of the review.
  - **`2`**: Unknown token (`<UNK>`) for words not in the vocabulary.
- **`decoded_review`**: Converts the integer indices of the first review (`train_data[0]`) back into human-readable words. The result is printed out for inspection.

---

3. Vectorizing Input Data

python
CopyEdit

```
def vectorize_sequences(sequences, dimension=NUM_WORDS):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0
    return results

X_train = vectorize_sequences(train_data)
X_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

- **`vectorize_sequences` function**: Converts each review into a binary vector of length `NUM_WORDS` (10,000). Each word in the review is represented by a `1` at its corresponding index, and `0` everywhere else.
- **`X_train` and `X_test`**: These are the input feature matrices for the training and test sets.
- **`y_train` and `y_test`**: The sentiment labels are converted to `float32` for compatibility with TensorFlow.

---

4. Preparing Validation Set

python
CopyEdit
```
X_val = X_train[:10000]
partial_X_train = X_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

- The training data is split into two parts:
    - **Validation Set** (`X_val`, `y_val`): The first 10,000 examples are used for validation during training.
    - **Training Set** (`partial_X_train`, `partial_y_train`): The remaining data is used for training.

---

5. Building the Model

python
CopyEdit
```
model = models.Sequential([
    layers.Dense(16, activation='relu', input_shape=(NUM_WORDS,)),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

- **`Sequential model`**: A simple feedforward neural network with the following layers:
    - **First layer**: A `Dense` layer with 16 units and ReLU activation. It takes an input of size `NUM_WORDS` (10,000).
    - **Second layer**: Another `Dense` layer with 16 units and ReLU activation.
    - **Output layer**: A `Dense` layer with a single unit and a sigmoid activation function, which outputs a probability between 0 and 1 for binary classification (positive/negative sentiment).

---

6. Compiling the Model

python
CopyEdit
```
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001),
    loss=losses.binary_crossentropy,
    metrics=[metrics.binary_accuracy]
```

```
)
```

- **Optimizer**: `RMSprop` is used for optimizing the model with a learning rate of `0.001`.
- **Loss function**: `binary_crossentropy` is used for binary classification tasks.
- **Metrics**: The model will track `binary_accuracy` during training and evaluation.

---

7. Training the Model

python
CopyEdit
```
history = model.fit(
    partial_X_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

- **Training the model**: The model is trained using the `partial_X_train` and `partial_y_train` data for 20 epochs. The batch size is set to 512.
- **Validation data**: The model's performance is validated on the `X_val` and `y_val` set after each epoch.

---

8. Plotting Training and Validation Loss

python
CopyEdit
```
history_dict = history.history
epochs = range(1, len(history_dict['loss']) + 1)

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs, history_dict['loss'], 'bo', label='Training Loss')
plt.plot(epochs, history_dict['val_loss'], 'b', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

- **Training vs. Validation Loss**: This plot visualizes how the loss changes over each epoch for both the training and validation sets.

---

9. Plotting Training and Validation Accuracy

python
CopyEdit

```
plt.subplot(1, 2, 2)
plt.plot(epochs, history_dict['binary_accuracy'], 'ro', label='Training Accuracy')
plt.plot(epochs, history_dict['val_binary_accuracy'], 'r', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

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

- **Training vs. Validation Accuracy**: This plot shows how the model's accuracy improves during training on both the training and validation sets.

---

10. Evaluate on Test Data

python
CopyEdit
```
results = model.evaluate(X_test, y_test)
print(f"\nTest Loss: {results[0]:.4f}, Test Accuracy: {results[1]:.4f}")
```

- **Evaluate the model**: The model is evaluated on the test set (`X_test`, `y_test`). The results show the test loss and test accuracy.

---

11. Make Predictions on Test Data

python
CopyEdit
```
predictions = model.predict(X_test)
y_pred = (predictions > 0.5).astype("int").flatten()
```

- **Making predictions**: The model generates predictions for the test data. The predictions are then thresholded at 0.5 to convert the probabilities into binary labels (0 or 1).

---

12. Calculate Mean Absolute Error (MAE)

python
CopyEdit
```
mae = mean_absolute_error(y_test, y_pred)
print(f"\nMean Absolute Error on Test Set: {mae:.4f}")
```

- **MAE Calculation**: The Mean Absolute Error (MAE) is calculated to evaluate the difference between the predicted labels and the actual labels. The MAE is the average of the absolute differences between the predicted and actual values.

Conclusion:

This code demonstrates a typical workflow for binary sentiment analysis using deep learning:

1. **Load and preprocess data** (IMDB dataset).
2. **Build and compile the model**.
3. **Train the model** while monitoring validation loss and accuracy.
4. **Evaluate performance** using test data.
5. **Visualize the results** using plots for loss and accuracy.
6. **Evaluate Mean Absolute Error (MAE)** to assess prediction accuracy.

The model can be further improved by adjusting hyperparameters, using more complex architectures, or employing techniques like word embeddings for better representation of the text data.

Group 1 1. Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

📚 Here's how you can run C++ in Jupyter Notebook:

1. Install `xeus-cling` (C++ Jupyter Kernel)

If you are using Anaconda:

- Open **Anaconda Prompt** (not Jupyter Notebook).
- Run:

bash
CopyEdit

```
conda install -c conda-forge xeus-cling
```

It will install C++ kernels for **C++11**, **C++14**, and **C++17**.

dl - 3

Convolutional neural network (CNN) (Any One from the following) • Use any dataset of plant disease and design a plant disease detection system using CNN. • Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

```
print("Prajwal Gadhave BACO21145")
# Required for compatibility with Python 2/3 (optional in Python 3+)
# from __future__ import absolute_import, division, print_function

# TensorFlow and Keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

# Load Fashion MNIST dataset
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

# Class names for visualization
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Normalize pixel values to [0,1]
train_images = train_images / 255.0
test_images = test_images / 255.0

# Visualize first 25 training images
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([]), plt.yticks([]), plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()

# Build the neural network model
# model = keras.Sequential([
#     keras.layers.Flatten(input_shape=(28, 28)),       # Flatten 28x28 to
784
#     keras.layers.Dense(128, activation='relu'),       # Dense hidden layer
```

```
#     keras.layers.Dense(10, activation='softmax')     # Output layer for
10 classes
# ])
model = keras.Sequential([
    keras.layers.Reshape((28, 28, 1), input_shape=(28, 28)),  # Add channel
dimension
    keras.layers.Conv2D(32, (3,3), activation='relu'),
    keras.layers.MaxPooling2D((2,2)),
    keras.layers.Conv2D(64, (3,3), activation='relu'),
    keras.layers.MaxPooling2D((2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=5)

# Evaluate the model on test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('\nTest accuracy:', test_acc)

# Make predictions
predictions = model.predict(test_images)

# Visualize predictions for a single image
def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array[i], true_label[i],
img[i]
    plt.grid(False)
    plt.xticks([]), plt.yticks([])
    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    color = 'blue' if predicted_label == true_label else 'red'

    plt.xlabel(f"{class_names[predicted_label]}
{100*np.max(predictions_array):.2f}% ({class_names[true_label]})",
               color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    bars = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
```

```python
    predicted_label = np.argmax(predictions_array)
    bars[predicted_label].set_color('red')
    bars[true_label].set_color('blue')

# Display prediction for one image
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions, test_labels)
plt.show()

# Show multiple images and predictions
num_rows, num_cols = 5, 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions, test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions, test_labels)
plt.tight_layout()
plt.show()

# Predict a single image
img = test_images[0]
img = (np.expand_dims(img, 0))  # Add batch dimension

predictions_single = model.predict(img)

# Visualize prediction for the single image
plot_value_array(0, predictions_single, test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()

print("Predicted label:", np.argmax(predictions_single[0]))
```