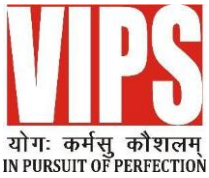


VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

BTECH Programme: AIDS-A

Course Title: Advances in Data Science Lab

Course Code: AIDS411P

Submitted To

Dr. Divya Taneja

Submitted By

Name: Parina Garg

Enrollment No:02917711922

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

VISION OF INSTITUTE

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

MISSION OF INSTITUTE

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building
capabilities to enable them to make significant contributions to the world.

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S.No	Experiment Name	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

EXPERIMENT-1

Aim: To implement and train deep learning models (e.g., CNN, RNN) on real-world datasets for various applications.

Objectives:

- To develop and apply CNN and RNN models on real-world datasets (such as MNIST or Fashion-MNIST) for performing classification tasks.
- To assess and compare the performance of these models using evaluation metrics and by interpreting results through visual analysis of predictions.

Theory:

Convolutional Neural Networks (CNNs)

CNNs are a type of deep learning model specifically built for analyzing structured grid-like data, such as images. They rely on convolutional layers that apply filters to local regions of the input, allowing the model to automatically learn spatial hierarchies of features like edges, textures, and shapes. Pooling layers help reduce the dimensionality while retaining essential details, and fully connected layers are used at the end for classification tasks.
CNNs are widely applied in **image recognition, medical imaging, and object detection**, as they efficiently capture local spatial relationships in the data.

Recurrent Neural Networks (RNNs)

RNNs are designed for handling **sequential data**, where previous information directly influences future outputs. By maintaining a hidden state that updates as new inputs are processed, RNNs are well-suited for tasks such as **time-series forecasting, speech recognition, and natural language processing**.
In the context of images, RNNs can treat each row or column as a sequence, enabling them to capture dependencies across dimensions in a temporal-like fashion. Unlike CNNs, they are explicitly structured to model sequential relationships.

Comparing CNNs and RNNs on Image Datasets

On datasets like **MNIST** or **Fashion-MNIST**, CNNs typically outperform by effectively capturing spatial structures, whereas RNNs offer an alternative by treating images as pixel or row sequences. Both architectures demonstrate the adaptability of deep learning, where the choice of model depends on the **data characteristics** and **task requirements**.

Performance Evaluation

The effectiveness of CNNs and RNNs is measured using metrics such as **accuracy, precision, recall, and confusion matrices**. Additionally, visualizing predictions provides an intuitive check of model performance. Together, these methods ensure that results are both **quantitatively reliable** and **visually interpretable**.

Code and Output:

#CNN

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.image as mpimg
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

train = pd.read_csv("train.csv")

test = pd.read_csv("test.csv")

train.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0

5 rows × 785 columns

```
test.head()
```

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0

5 rows × 784 columns

```
print(train.isna().sum().sum())
print(test.isna().sum().sum())
```

```
0
0
```

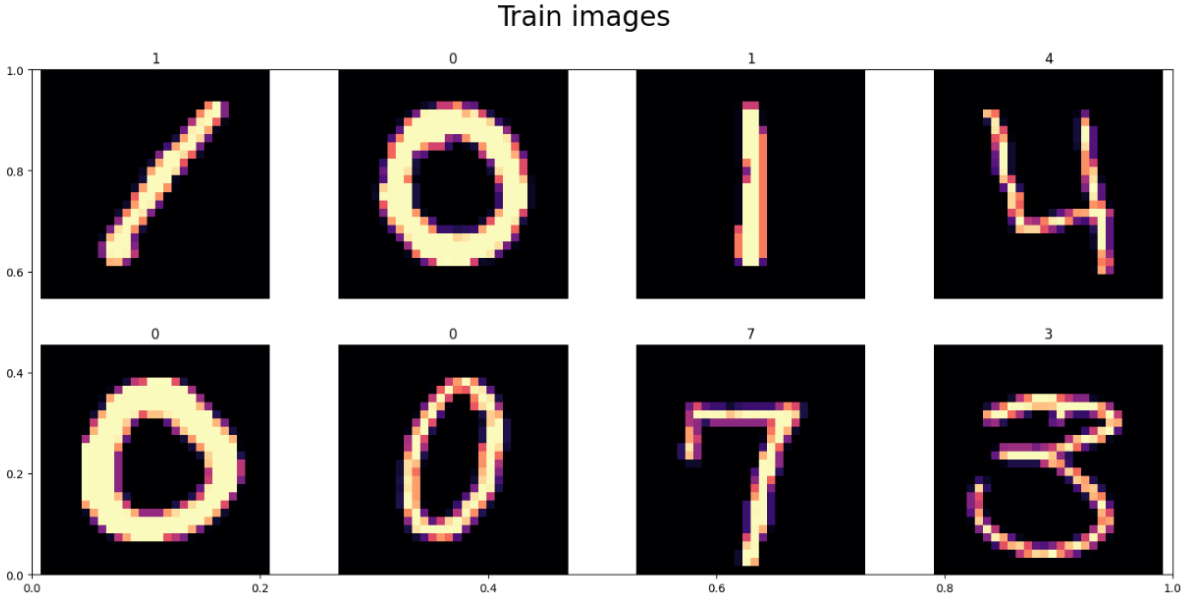
```
train['label'].value_counts().sort_index()
```

label	count
0	4132
1	4684
2	4177
3	4351
4	4072
5	3795
6	4137
7	4401
8	4063
9	4188

dtype: int64

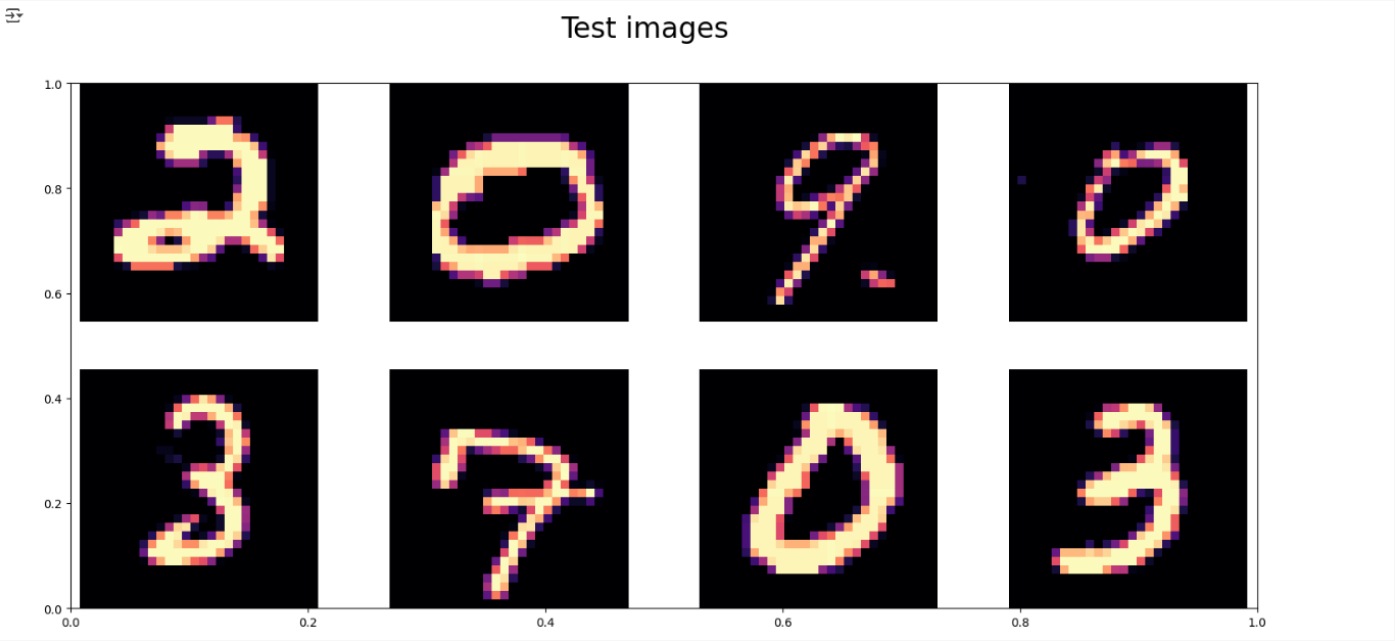
```
fig, ax = plt.subplots(figsize=(18, 8))
for ind, row in train.iloc[:8, :].iterrows():
    plt.subplot(2, 4, ind+1)
    plt.title(row[0])
    img = row.to_numpy()[1:].reshape(28, 28)
    fig.suptitle('Train images', fontsize=24)
    plt.axis('off')

    plt.imshow(img, cmap='magma')
```

```
/tmp/ipython-input-2252787088.py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior).
  plt.title(row[0])
```




```
fig, ax = plt.subplots(figsize=(18, 8))
for ind, row in test.iloc[:8, :].iterrows():
    plt.subplot(2, 4, ind+1)
    img = row.to_numpy()[:].reshape(28, 28)
    fig.suptitle('Test images', fontsize=24)
```

```
    plt.axis('off')
    plt.imshow(img, cmap='magma')
```




```
X = train.iloc[:, 1:].to_numpy()
y = train['label'].to_numpy()
test = test.loc[:, :].to_numpy()

for i in [X, y, test]:
    print(i.shape)
```

```
(42000, 784)
(42000,)
(28000, 784)
```

```
X = X / 255.0
test = test / 255.0

print(X.shape)
print(test.shape)
```

```
(42000, 28, 28, 1)
(28000, 28, 28, 1)
```

```
X = X.reshape(-1,28,28,1)
test = test.reshape(-1,28,28,1)

print(X.shape)
print(test.shape)
```

```
(42000, 28, 28, 1)
(28000, 28, 28, 1)
```

```
print(y.shape)
print(y[0])
```

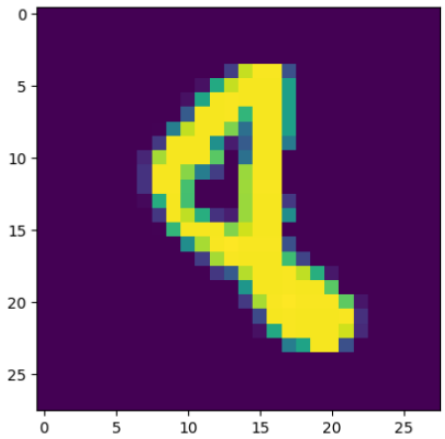
```
(42000,)
1
```

```
y_enc = to_categorical(y, num_classes = 10)

print(y_enc.shape)
print(y_enc[0])
```

```
(29400, 28, 28, 1)
(29400, 10)
(12600, 28, 28, 1)
(12600, 10)
```

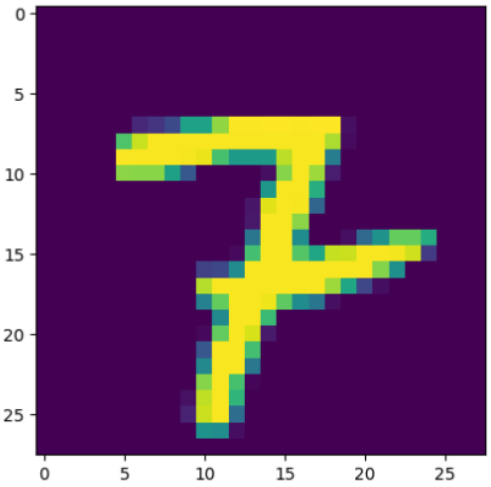
```
random_seed = 2

X_train, X_val, y_train_enc, y_val_enc = train_test_split(X, y_enc, test_size=0.3)

for i in [X_train, y_train_enc, X_val, y_val_enc]:
    print(i.shape)

g = plt.imshow(X_train[0][:,:,0])
print(y_train_enc[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```




```
g = plt.imshow(X_train[9][:,:,0])
print(y_train_enc[9])
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```




```
INPUT_SHAPE = (28,28,1)
OUTPUT_SHAPE = 10
BATCH_SIZE = 128
EPOCHS = 10
VERBOSE = 2

model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3), activation='relu',
input_shape=INPUT_SHAPE))
model.add(MaxPool2D((2,2)))
model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))
model.add(Conv2D(128, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

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

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. 
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 128)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 256)	33,024
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8,256
dropout_2 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

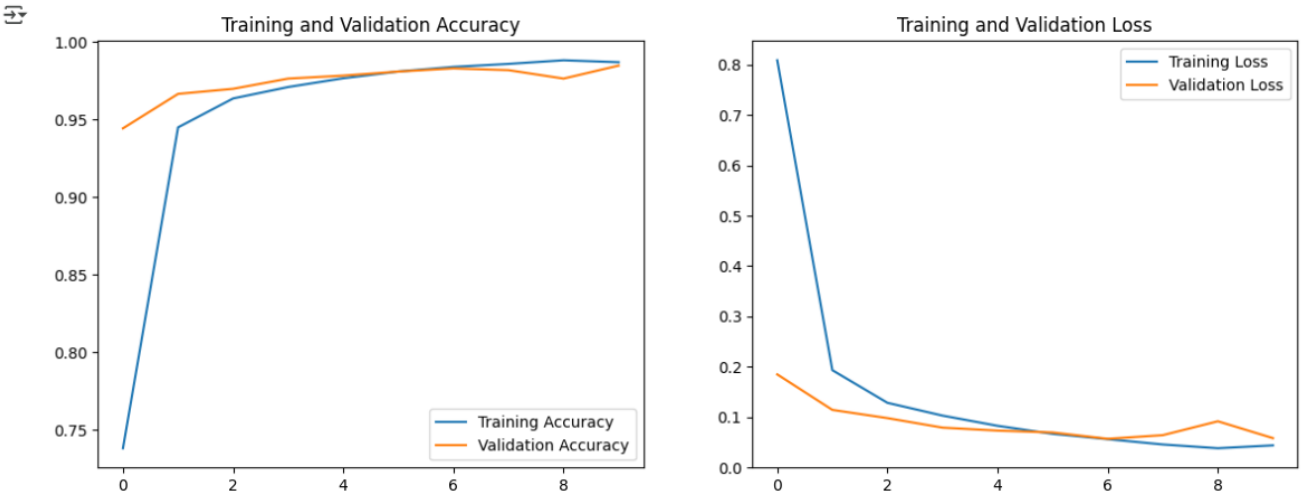
```
Total params: 167,498 (654.29 KB)
Trainable params: 167,498 (654.29 KB)
Non-trainable params: 0 (0.00 B)
```

```
history = model.fit(X_train, y_train_enc,
                    epochs=EPOCHS,
                    batch_size=BATCH_SIZE,
                    verbose=VERBOSE,
                    validation_split=0.3)
```

```
Epoch 1/10
161/161 - 21s - 132ms/step - accuracy: 0.7380 - loss: 0.8086 - val_accuracy: 0.9443 - val_loss: 0.1849
Epoch 2/10
161/161 - 20s - 121ms/step - accuracy: 0.9449 - loss: 0.1933 - val_accuracy: 0.9666 - val_loss: 0.1144
Epoch 3/10
161/161 - 23s - 145ms/step - accuracy: 0.9636 - loss: 0.1285 - val_accuracy: 0.9698 - val_loss: 0.0981
Epoch 4/10
161/161 - 17s - 105ms/step - accuracy: 0.9710 - loss: 0.1030 - val_accuracy: 0.9764 - val_loss: 0.0791
Epoch 5/10
161/161 - 22s - 137ms/step - accuracy: 0.9766 - loss: 0.0828 - val_accuracy: 0.9783 - val_loss: 0.0732
Epoch 6/10
161/161 - 18s - 110ms/step - accuracy: 0.9810 - loss: 0.0668 - val_accuracy: 0.9810 - val_loss: 0.0696
Epoch 7/10
161/161 - 21s - 131ms/step - accuracy: 0.9840 - loss: 0.0563 - val_accuracy: 0.9830 - val_loss: 0.0570
Epoch 8/10
161/161 - 19s - 118ms/step - accuracy: 0.9859 - loss: 0.0457 - val_accuracy: 0.9819 - val_loss: 0.0642
Epoch 9/10
161/161 - 18s - 111ms/step - accuracy: 0.9882 - loss: 0.0382 - val_accuracy: 0.9764 - val_loss: 0.0918
Epoch 10/10
161/161 - 20s - 125ms/step - accuracy: 0.9870 - loss: 0.0439 - val_accuracy: 0.9848 - val_loss: 0.0583
```

```
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
```

```
plt.savefig('./foo.png')
plt.show()
```




```
model.evaluate(X_val, y_val_enc, verbose=False)
```

```
[0.0577397383749485, 0.9844444394111633]
```

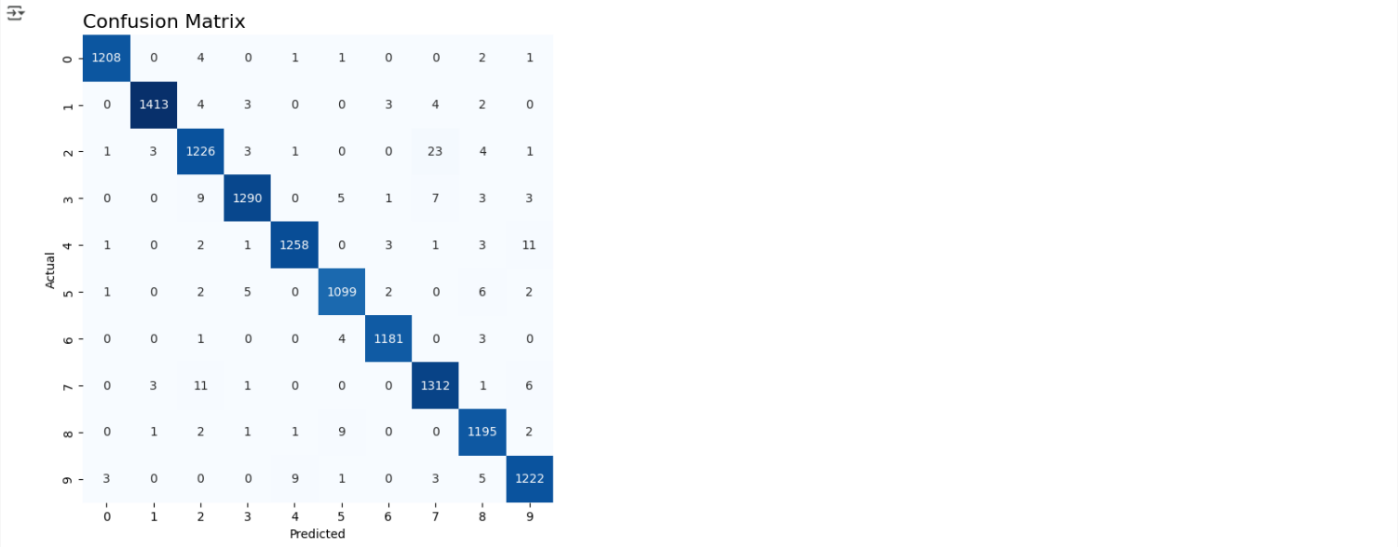
```
y_pred_enc = model.predict(X_val)

y_act = [np.argmax(i) for i in y_val_enc]
y_pred = [np.argmax(i) for i in y_pred_enc]

print(y_pred_enc[0])
print(y_pred[0])
```

```
394/394 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step
[1.5263842e-09 3.0734038e-06 1.9192817e-06 7.6141032e-07 3.6560451e-08
 2.9495922e-10 1.6512483e-10 9.9998790e-01 1.4909288e-09 6.2943968e-06]
7
```

```
print(classification_report(y_act, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      1217
           1       1.00      0.99      0.99      1429
           2       0.97      0.97      0.97      1262
           3       0.99      0.98      0.98      1318
           4       0.99      0.98      0.99      1280
           5       0.98      0.98      0.98      1117
           6       0.99      0.99      0.99      1189
           7       0.97      0.98      0.98      1334
           8       0.98      0.99      0.98      1211
           9       0.98      0.98      0.98      1243

    accuracy                           0.98     12600
   macro avg       0.98      0.98      0.98     12600
weighted avg       0.98      0.98      0.98     12600
```

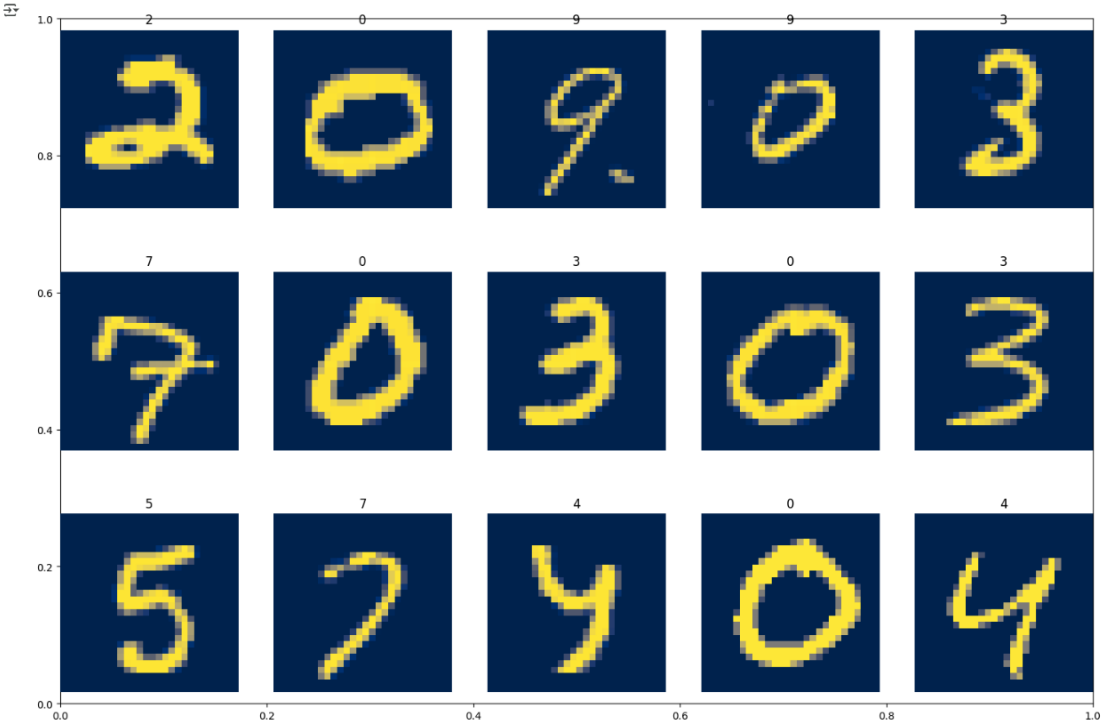
```
fig, ax = plt.subplots(figsize=(7, 7))
sns.heatmap(confusion_matrix(y_act, y_pred), annot=True,
            cbar=False, fmt='1d', cmap='Blues', ax=ax)
ax.set_title('Confusion Matrix', loc='left', fontsize=16)
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.show()
```




```
y_pred_enc = model.predict(test)
y_pred = [np.argmax(i) for i in y_pred_enc]

print(y_pred_enc[0])
print(y_pred[0])
```

```
875/875 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step
[1.5045844e-08 1.4467790e-08 9.9999911e-01 2.5524567e-08 1.4395856e-09
 1.0495399e-12 8.9408023e-11 6.6161988e-07 1.3435741e-07 4.6921750e-10]
2
```

```
fig, ax = plt.subplots(figsize=(18, 12))
for ind, row in enumerate(test[:15]):
    plt.subplot(3, 5, ind+1)
    plt.title(y_pred[ind])
    img = row.reshape(28, 28)
    fig.suptitle('Predicted values', fontsize=24)
    plt.axis('off')
    plt.imshow(img, cmap='cividis')
```

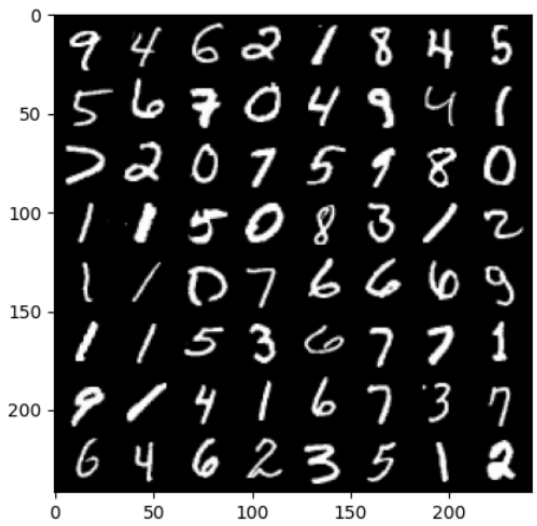


#RNN

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms

BATCH_SIZE = 64

transform = transforms.Compose([transforms.ToTensor()])


trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE,
shuffle=False, num_workers=2)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 56.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 1.87MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 14.5MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 6.91MB/s]
```

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

dataiter = iter(trainloader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))
```




```
N_STEPS = 28
N_INPUTS = 28
N_NEURONS = 150
N_OUTPUTS = 10
N_EPHOCS = 10

class ImageRNN(nn.Module):
    def __init__(self, batch_size, n_steps, n_inputs, n_neurons, n_outputs):
        super(ImageRNN, self).__init__()
        self.n_neurons = n_neurons
        self.batch_size = batch_size
        self.n_steps = n_steps
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.basic_rnn = nn.RNN(self.n_inputs, self.n_neurons)
        self.FC = nn.Linear(self.n_neurons, self.n_outputs)

    def init_hidden(self,):
        return (torch.zeros(1, self.batch_size, self.n_neurons))

    def forward(self, X):
        X = X.permute(1, 0, 2)
```

```
        self.batch_size = X.size(1)
        self.hidden = self.init_hidden()
        lstm_out, self.hidden = self.basic_rnn(X, self.hidden)
        out = self.FC(self.hidden)
        return out.view(-1, self.n_outputs)

dataiter = iter(trainloader)
images, labels = next(dataiter)
model = ImageRNN(BATCH_SIZE, N_STEPS, N_INPUTS, N_NEURONS, N_OUTPUTS)
logits = model(images.view(-1, 28,28))
print(logits[0:10])
```

```
tensor([[ 0.0030,  0.0100, -0.0206, -0.1471,  0.1211, -0.0253,  0.0621,  0.1325,
          0.0851,  0.0622],
        [ 0.0010, -0.0341,  0.0018, -0.1489,  0.0546, -0.0042,  0.0423,  0.1359,
          0.0844,  0.0715],
        [ 0.0069, -0.0313,  0.0045, -0.1393,  0.0620,  0.0039,  0.0482,  0.1359,
          0.0735,  0.0641],
        [ 0.0086, -0.0333,  0.0043, -0.1343,  0.0588,  0.0090,  0.0442,  0.1352,
          0.0743,  0.0551],
        [ 0.0014, -0.0386, -0.0049, -0.1433,  0.0496,  0.0062,  0.0428,  0.1327,
          0.0873,  0.0700],
        [ 0.0058, -0.0370, -0.0010, -0.1390,  0.0574,  0.0072,  0.0487,  0.1373,
          0.0719,  0.0634],
        [-0.0018, -0.0392, -0.0012, -0.1488,  0.0522, -0.0054,  0.0432,  0.1354,
          0.0873,  0.0716],
        [ 0.0012, -0.0436, -0.0104, -0.1476,  0.0448,  0.0018,  0.0343,  0.1387,
          0.0782,  0.0641],
        [ 0.0054, -0.0076, -0.0340, -0.1311,  0.0892, -0.0131,  0.0560,  0.1338,
          0.1139,  0.0467],
        [ 0.0062, -0.0361,  0.0010, -0.1393,  0.0577,  0.0075,  0.0487,  0.1381,
          0.0710,  0.0630]], grad_fn=<SliceBackward0>)
```

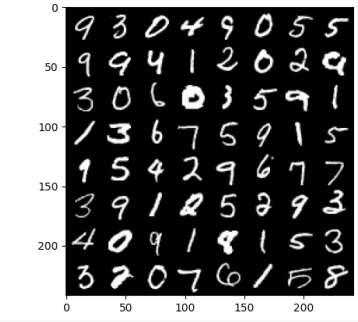
```
import torch.optim as optim

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = ImageRNN(BATCH_SIZE, N_STEPS, N_INPUTS, N_NEURONS, N_OUTPUTS)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

def get_accuracy(logit, target, batch_size):
    corrects = (torch.max(logit, 1)[1].view(target.size()).data ==
target.data).sum()
    accuracy = 100.0 * corrects/batch_size
    return accuracy.item()

for epoch in range(N_EPHOCS):
    train_running_loss = 0.0
    train_acc = 0.0
    model.train()
    for i, data in enumerate(trainloader):
        optimizer.zero_grad()
        model.hidden = model.init_hidden()
        inputs, labels = data
        inputs = inputs.view(-1, 28,28)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
        optimizer.step()
        train_running_loss += loss.detach().item()
        train_acc += get_accuracy(outputs, labels, BATCH_SIZE)
    model.eval()
    print('Epoch:  %d | Loss: %.4f | Train Accuracy: %.2f'
          %(epoch, train_running_loss / i, train_acc/i))
```

```
Epoch:  0 | Loss: 0.7330 | Train Accuracy: 76.08
Epoch:  1 | Loss: 0.3336 | Train Accuracy: 90.33
Epoch:  2 | Loss: 0.2443 | Train Accuracy: 93.04
Epoch:  3 | Loss: 0.2006 | Train Accuracy: 94.34
Epoch:  4 | Loss: 0.1733 | Train Accuracy: 95.05
Epoch:  5 | Loss: 0.1496 | Train Accuracy: 95.80
Epoch:  6 | Loss: 0.1319 | Train Accuracy: 96.28
Epoch:  7 | Loss: 0.1367 | Train Accuracy: 96.15
Epoch:  8 | Loss: 0.1303 | Train Accuracy: 96.33
Epoch:  9 | Loss: 0.1157 | Train Accuracy: 96.77
```

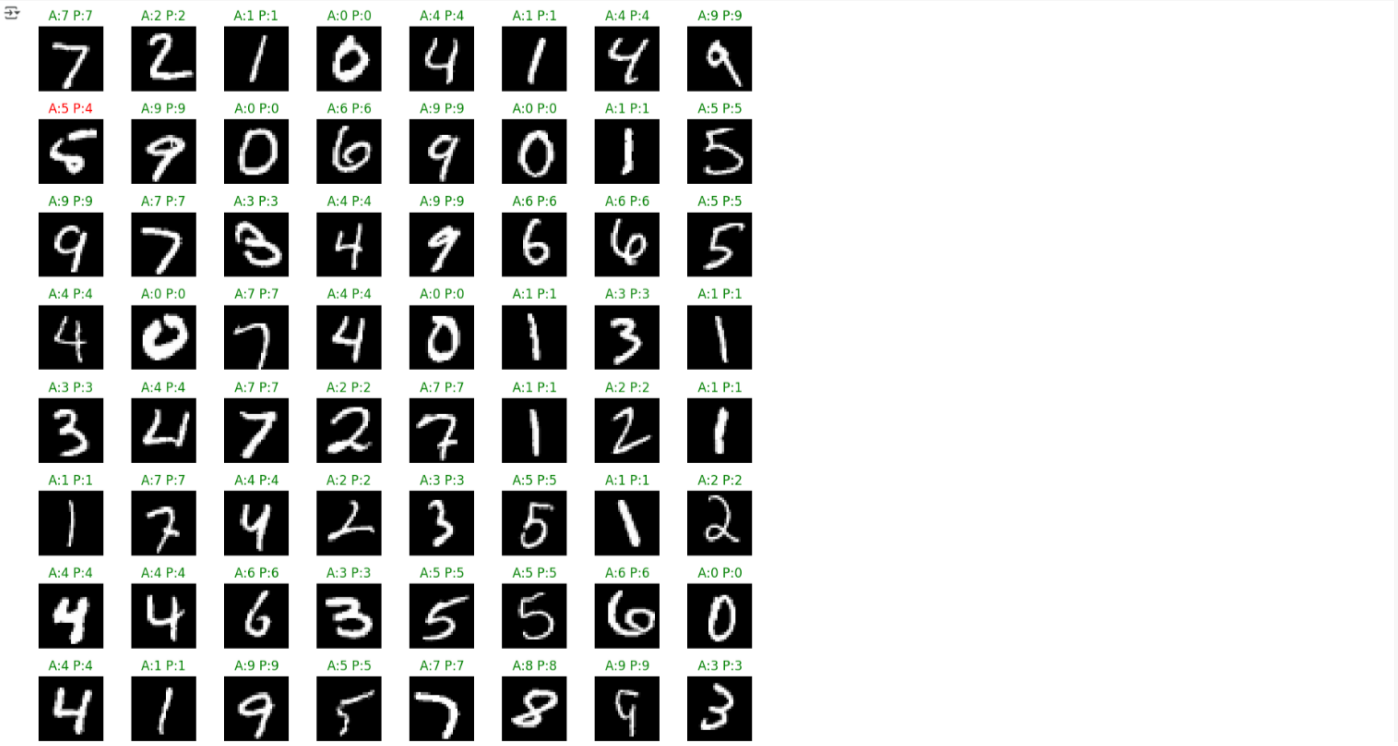
```
test_acc = 0.0
for i, data in enumerate(testloader, 0):
    inputs, labels = data
    inputs = inputs.view(-1, 28, 28)
    outputs = model(inputs)
    test_acc += get_accuracy(outputs, labels, BATCH_SIZE)

print('Test Accuracy: %.2f'%( test_acc/i))

model.eval()
dataiter = iter(testloader)
images, labels = next(dataiter)
images = images.view(-1, 28, 28)
outputs = model(images)
_, predicted = torch.max(outputs.data, 1)

print('Predicted: ', ' '.join('%5s' % predicted[j].item() for j in
range(BATCH_SIZE)))
print('Actual:    ', ' '.join('%5s' % labels[j].item() for j in range(BATCH_SIZE)))
```

```
Test Accuracy: 96.89
Predicted:     7     2     1     0     4     1     4     9     4     9     0     6     9     0     1     5     9     7     3     4     9     6     6     5     4     0     7     4     0     1     3     1     3     4     7     2
Actual:        7     2     1     0     4     1     4     9     5     9     0     6     9     0     1     5     9     7     3     4     9     6     6     5     4     0     7     4     0     1     3     1     3     4     7     2
```




```
fig, axes = plt.subplots(8, 8, figsize=(10, 10))
axes = axes.flatten()
```

```
for i in range(BATCH_SIZE):
    img = images[i].squeeze().numpy()
    axes[i].imshow(img, cmap='gray')
    axes[i].set_title(f'A:{labels[i]} P:{predicted[i].item()}', color='green' if
predicted[i] == labels[i] else 'red')
    axes[i].axis('off')

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



Learning Outcomes:

1. Learn the design and training workflow of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) using real-world datasets.
2. Acquire hands-on experience in building deep learning models for applications like image classification and sequence modeling with TensorFlow/Keras and PyTorch.
3. Build the capability to evaluate and interpret model performance through metrics, confusion matrices, and visual analysis of predictions.

EXPERIMENT-2

Aim: To build and train GAN models for generating realistic images and evaluate the quality of the generated samples.

Objectives:

- To develop and train a GAN on the Fashion-MNIST dataset to generate realistic images of fashion items.
- To assess and evaluate the GAN-generated images through visual inspection and quantitative metrics, gaining deeper insights into generative modeling.

Theory:

Introduction to GANs

Generative Adversarial Networks (GANs) are a type of deep learning model introduced by Ian Goodfellow in 2014, aimed at generating realistic synthetic data. GANs consist of two neural networks—the **Generator** and the **Discriminator**—which are trained in a competitive, adversarial setting. The generator's role is to transform random noise vectors into images that resemble real data, while the discriminator's role is to differentiate between real and generated images. This adversarial framework enables the generator to progressively improve its outputs until they are nearly indistinguishable from real samples.

Training Process

GANs are trained using a **minimax game** approach. The generator seeks to minimize the discriminator's ability to correctly identify generated images as fake, while the discriminator tries to maximize its classification accuracy. This alternating optimization process forms a feedback loop that enhances the performance of both networks.

- The generator usually employs **upsampling layers** and **non-linear activation functions** such as ReLU or Tanh.
- The discriminator typically uses **convolutional layers**, along with LeakyReLU activations and dropout, to stabilize training.

Proper **weight initialization**, **hyperparameter tuning**, and **balanced training** are critical to prevent common issues like **mode collapse** or unstable convergence.

Evaluation Metrics

Evaluating GAN performance involves both quantitative and qualitative measures:

- **Inception Score (IS):** Assesses the diversity and quality of generated images.
- **Fréchet Inception Distance (FID):** Measures the similarity between the distributions of real and generated images, with lower values indicating higher fidelity.

In addition to these metrics, **visual inspection** is crucial for detecting artifacts and ensuring structural realism in generated samples. Combining both quantitative and qualitative evaluations ensures a comprehensive assessment of GAN performance.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import os

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

BATCH_SIZE = 64
IMAGE_SIZE = 28
CHANNELS = 1
NOISE_DIM = 100
LEARNING_RATE = 0.0002
BETA1 = 0.5
NUM_EPOCHS = 10

FASHION_CLASSES = [
    'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
    'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
]

class Generator(nn.Module):
    def __init__(self, noise_dim=100, img_size=28):
        super(Generator, self).__init__()
        self.img_size = img_size
        self.model = nn.Sequential(
            nn.Linear(noise_dim, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(True),
            nn.Linear(1024, img_size * img_size),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.shape[0], 1, self.img_size, self.img_size)
        return img

class Discriminator(nn.Module):
    def __init__(self, img_size=28):
```

```
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(img_size * img_size, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.shape[0], -1)
        validity = self.model(img_flat)
        return validity

class FashionGAN:
    def __init__(self, noise_dim=100, img_size=28, lr=0.0002, beta1=0.5):
        self.noise_dim = noise_dim
        self.img_size = img_size
        self.generator = Generator(noise_dim, img_size).to(device)
        self.discriminator = Discriminator(img_size).to(device)
        self.criterion = nn.BCELoss()
        self.optimizer_G = optim.Adam(self.generator.parameters(), lr=lr, betas=(beta1, 0.999))
        self.optimizer_D = optim.Adam(self.discriminator.parameters(), lr=lr, betas=(beta1, 0.999))
        self.g_losses = []
        self.d_losses = []
        self.d_acc_real = []
        self.d_acc_fake = []
        self._init_weights()

    def _init_weights(self):
        for module in [self.generator, self.discriminator]:
            for m in module.modules():
                if isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight.data, 0.0, 0.02)
                    nn.init.constant_(m.bias.data, 0)
                elif isinstance(m, nn.BatchNorm1d):
                    nn.init.normal_(m.weight.data, 1.0, 0.02)
                    nn.init.constant_(m.bias.data, 0)

    def train(self, dataloader, num_epochs):
        print(f"Starting training for {num_epochs} epochs...")
        print(f"Dataset size: {len(dataloader.dataset)} images")
        print(f"Batches per epoch: {len(dataloader)}")
        fixed_noise = torch.randn(16, self.noise_dim, device=device)

        for epoch in range(num_epochs):
```

```
        epoch_g_loss = 0
        epoch_d_loss = 0
        d_real_acc = 0
        d_fake_acc = 0

        for i, (real_images, _) in enumerate(dataloader):
            batch_size = real_images.size(0)
            real_labels = torch.ones(batch_size, 1, device=device)
            fake_labels = torch.zeros(batch_size, 1, device=device)
            real_images = real_images.to(device)

            self.optimizer_D.zero_grad()
            real_output = self.discriminator(real_images)
            d_loss_real = self.criterion(real_output, real_labels)
            d_real_acc += ((real_output > 0.5).float() == real_labels).float().mean().item()
            noise = torch.randn(batch_size, self.noise_dim, device=device)
            fake_images = self.generator(noise)
            fake_output = self.discriminator(fake_images.detach())
            d_loss_fake = self.criterion(fake_output, fake_labels)
            d_fake_acc += ((fake_output <= 0.5).float() == (1 - fake_labels)).float().mean().item()
            d_loss = d_loss_real + d_loss_fake
            d_loss.backward()
            self.optimizer_D.step()

            self.optimizer_G.zero_grad()
            fake_output = self.discriminator(fake_images)
            g_loss = self.criterion(fake_output, real_labels)
            g_loss.backward()
            self.optimizer_G.step()

            epoch_g_loss += g_loss.item()
            epoch_d_loss += d_loss.item()

        avg_g_loss = epoch_g_loss / len(dataloader)
        avg_d_loss = epoch_d_loss / len(dataloader)
        avg_d_real_acc = d_real_acc / len(dataloader)
        avg_d_fake_acc = d_fake_acc / len(dataloader)
        self.g_losses.append(avg_g_loss)
        self.d_losses.append(avg_d_loss)
        self.d_acc_real.append(avg_d_real_acc)
        self.d_acc_fake.append(avg_d_fake_acc)

        if epoch % 5 == 0 or epoch == num_epochs - 1:
            print(f"Epoch [{epoch+1:3d}/{num_epochs}] "
                  f"D_loss: {avg_d_loss:.4f} G_loss: {avg_g_loss:.4f} "
                  f"D_real_acc: {avg_d_real_acc:.3f} D_fake_acc: {avg_d_fake_acc:.3f}")
            self.save_sample_images(fixed_noise, epoch)

    print("\nTraining completed!")
    self.print_training_summary()

def save_sample_images(self, noise, epoch):
```

```
        self.generator.eval()
        with torch.no_grad():
            fake_images = self.generator(noise)
            fake_images = (fake_images + 1) / 2
            os.makedirs("fashion_samples", exist_ok=True)
            grid = torchvision.utils.make_grid(fake_images, nrow=4, normalize=True, padding=2)
            torchvision.utils.save_image(grid, f"fashion_samples/epoch_{epoch:03d}.png")
        self.generator.train()

    def generate_fashion_items(self, num_items=16):
        self.generator.eval()
        with torch.no_grad():
            noise = torch.randn(num_items, self.noise_dim, device=device)
            fake_images = self.generator(noise)
            fake_images = (fake_images + 1) / 2
        self.generator.train()
        return fake_images

    def plot_training_progress(self):
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))
        axes[0, 0].plot(self.g_losses, label='Generator Loss', color='blue', linewidth=2)
        axes[0, 0].plot(self.d_losses, label='Discriminator Loss', color='red', linewidth=2)
        axes[0, 0].set_xlabel('Epoch')
        axes[0, 0].set_ylabel('Loss')
        axes[0, 0].set_title('Training Losses')
        axes[0, 0].legend()
        axes[0, 0].grid(True, alpha=0.3)
        axes[0, 1].plot(self.d_acc_real, label='Real Images Accuracy', color='green', linewidth=2)
        axes[0, 1].plot(self.d_acc_fake, label='Fake Images Accuracy', color='orange', linewidth=2)
        axes[0, 1].axhline(y=0.5, color='black', linestyle='--', alpha=0.5, label='Random Guess')
        axes[0, 1].set_xlabel('Epoch')
        axes[0, 1].set_ylabel('Accuracy')
        axes[0, 1].set_title('Discriminator Accuracy')
        axes[0, 1].legend()
        axes[0, 1].grid(True, alpha=0.3)
        axes[0, 1].set_ylim(0, 1)
        fake_images = self.generate_fashion_items(16)
        grid = torchvision.utils.make_grid(fake_images, nrow=4, normalize=True, padding=2)
        grid_np = grid.permute(1, 2, 0).cpu().numpy()
        axes[1, 0].imshow(grid_np.squeeze(), cmap='gray')
        axes[1, 0].set_title('Generated Fashion Items')
        axes[1, 0].axis('off')
        if len(self.g_losses) > 5:
            window_size = min(5, len(self.g_losses) // 4)
            g_smooth = np.convolve(self.g_losses, np.ones(window_size)/window_size, mode='valid')
            d_smooth = np.convolve(self.d_losses, np.ones(window_size)/window_size, mode='valid')
            axes[1, 1].plot(g_smooth, label='Generator (Smoothed)', color='blue', linewidth=2)
            axes[1, 1].plot(d_smooth, label='Discriminator (Smoothed)', color='red', linewidth=2)
            axes[1, 1].set_xlabel('Epoch')
            axes[1, 1].set_ylabel('Loss')
            axes[1, 1].set_title('Smoothed Training Losses')
            axes[1, 1].legend()
```

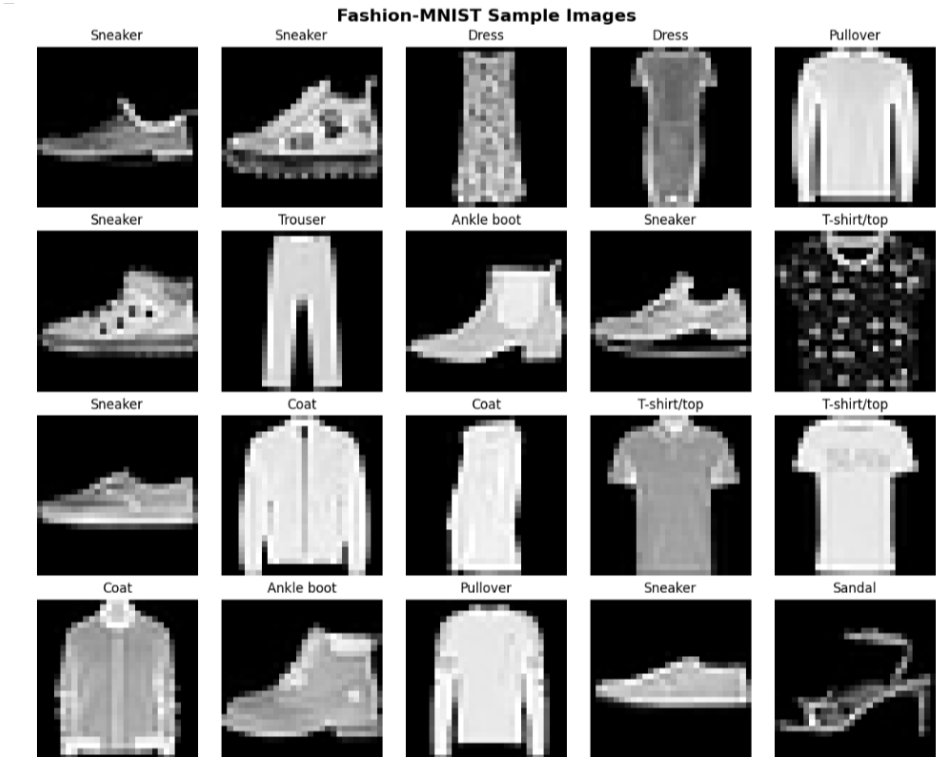
```
            axes[1, 1].grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()

    def compare_real_vs_fake(self, real_dataloader):
        real_batch = next(iter(real_dataloader))[0][:16]
        fake_batch = self.generate_fashion_items(16)
        fig, axes = plt.subplots(4, 8, figsize=(16, 8))
        for i in range(16):
            row = i // 8
            col = i % 8
            axes[row, col].imshow(real_batch[i].squeeze(), cmap='gray')
            axes[row, col].set_title('Real' if i < 8 else '')
            axes[row, col].axis('off')
        for i in range(16):
            row = (i // 8) + 2
            col = i % 8
            axes[row, col].imshow(fake_batch[i].squeeze().cpu(), cmap='gray')
            axes[row, col].set_title('Generated' if i < 8 else '')
            axes[row, col].axis('off')
        plt.suptitle('Real vs Generated Fashion Items', fontsize=16, fontweight='bold')
        plt.tight_layout()
        plt.show()

    def print_training_summary(self):
        print("\n" + "="*50)
        print("TRAINING SUMMARY")
        print("="*50)
        print(f"Final Generator Loss: {self.g_losses[-1]:.4f}")
        print(f"Final Discriminator Loss: {self.d_losses[-1]:.4f}")
        print(f"Final D Real Accuracy: {self.d_acc_real[-1]:.3f}")
        print(f"Final D Fake Accuracy: {self.d_acc_fake[-1]:.3f}")
        print(f"Average Generator Loss: {np.mean(self.g_losses):.4f}")
        print(f"Average Discriminator Loss: {np.mean(self.d_losses):.4f}")
        print("="*50)

    def save_models(self, path="fashion_gan_models"):
        os.makedirs(path, exist_ok=True)
        torch.save(self.generator.state_dict(), f"{path}/generator.pth")
        torch.save(self.discriminator.state_dict(), f"{path}/discriminator.pth")
        print(f"Models saved to {path}/")

    def load_models(self, path="fashion_gan_models"):
        self.generator.load_state_dict(torch.load(f"{path}/generator.pth"))
        self.discriminator.load_state_dict(torch.load(f"{path}/discriminator.pth"))
        print(f"Models loaded from {path}/")

def create_fashion_mnist_dataloader(batch_size=64, num_workers=2):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5])
    ])
```

```
    dataset = torchvision.datasets.FashionMNIST(
        root='./data',
        train=True,
        download=True,
        transform=transform
    )
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        drop_last=True,
        pin_memory=True if torch.cuda.is_available() else False
    )
    return dataloader, dataset

def show_fashion_samples(dataloader, num_samples=20):
    data_iter = iter(dataloader)
    images, labels = next(data_iter)
    fig, axes = plt.subplots(4, 5, figsize=(12, 10))
    axes = axes.ravel()
    for i in range(num_samples):
        img = images[i].squeeze()
        label = labels[i].item()
        axes[i].imshow(img, cmap='gray')
        axes[i].set_title(f'{FASHION_CLASSES[label]}')
        axes[i].axis('off')
    plt.suptitle('Fashion-MNIST Sample Images', fontsize=16, fontweight='bold')
    plt.tight_layout()
    plt.show()

def main():
    print("Fashion-MNIST GAN Training")
    print("=" * 40)
    print("Loading Fashion-MNIST dataset...")
    dataloader, dataset = create_fashion_mnist_dataloader(batch_size=BATCH_SIZE, num_workers=2)
    print(f"Dataset loaded successfully!")
    print(f"Total images: {len(dataset)}")
    print(f"Image size: {IMAGE_SIZE}x{IMAGE_SIZE}")
    print(f"Classes: {len(FASHION_CLASSES)}")
    print(f"Batch size: {BATCH_SIZE}")
    print(f"Batches per epoch: {len(dataloader)}")
    print("\nShowing sample Fashion-MNIST images...")
    show_fashion_samples(dataloader)
    print("\nInitializing Fashion GAN...")
    gan = FashionGAN(noise_dim=NOISE_DIM, img_size=IMAGE_SIZE, lr=LEARNING_RATE, beta1=BETA1)
    print(f"Generator parameters: {sum(p.numel() for p in gan.generator.parameters()):,}")
    print(f"Discriminator parameters: {sum(p.numel() for p in gan.discriminator.parameters()):,}")
    print(f"\nStarting training for {NUM_EPOCHS} epochs...")
    gan.train(dataloader, NUM_EPOCHS)
    print("\nDisplaying training results...")
    gan.plot_training_progress()
```

```
print("\nComparing real vs generated images...")
    gan.compare_real_vs_fake(dataloader)
    gan.save_models()
    print("\n" + "=" * 40)
    print("Fashion-MNIST GAN training completed!")
  print("Check 'fashion_samples' folder for generated images during training.")
    print("Models saved in 'fashion_gan_models' folder.")
if __name__ == "__main__":
    main()
```

Output:

```
Using device: cpu
Fashion-MNIST GAN Training
========================================
Loading Fashion-MNIST dataset...
100%|██████████| 26.4M/26.4M [00:00<00:00, 115MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 3.45MB/s]
100%|██████████| 4.42M/4.42M [00:00<00:00, 54.9MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 7.20MB/s]
Dataset loaded successfully!
Total images: 60000
Image size: 28x28
Classes: 10
Batch size: 64
Batches per epoch: 937
```

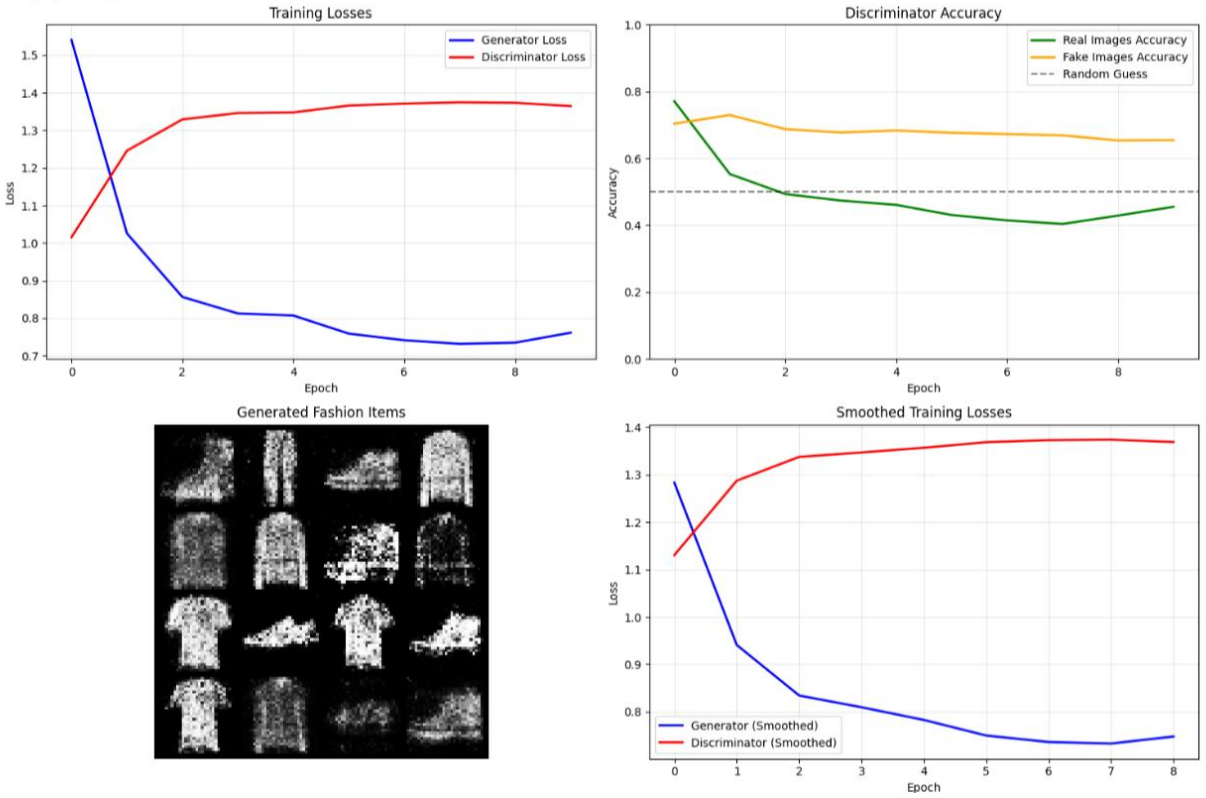


```
Initializing Fashion GAN...
Generator parameters: 1,489,424
Discriminator parameters: 566,273
```

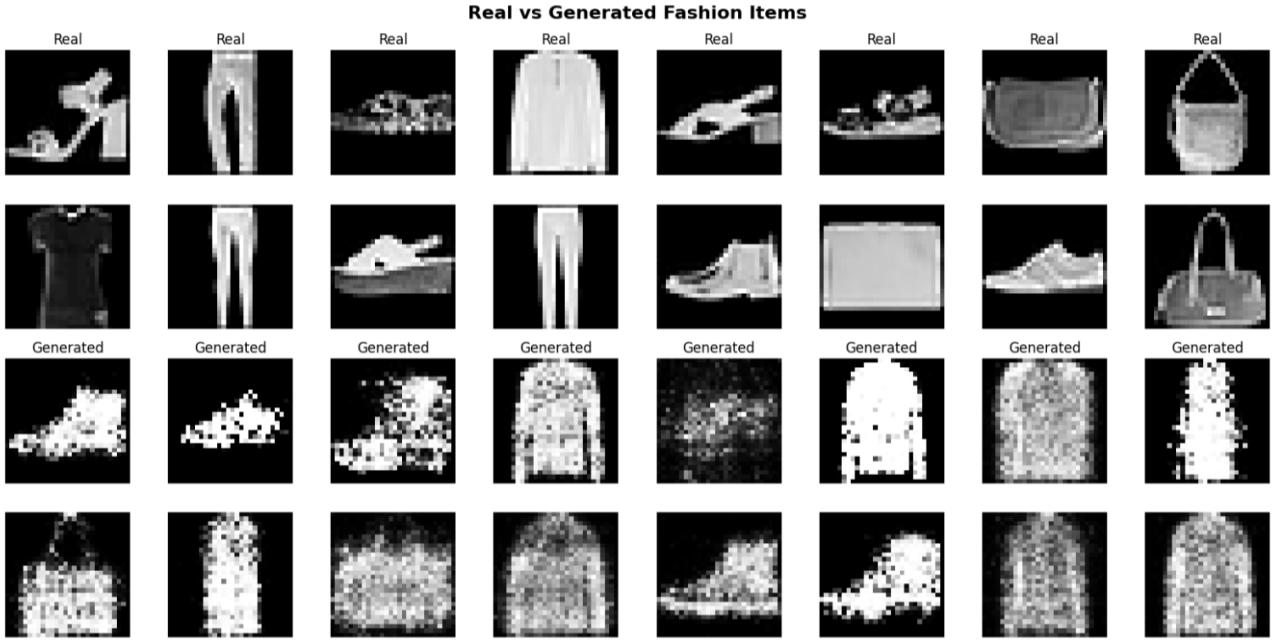
```
Starting training for 10 epochs...
Starting training for 10 epochs...
Dataset size: 60000 images
Batches per epoch: 937
Epoch [  1/10] D_loss: 1.0155 G_loss: 1.5408 D_real_acc: 0.771 D_fake_acc: 0.704
Epoch [  6/10] D_loss: 1.3661 G_loss: 0.7587 D_real_acc: 0.431 D_fake_acc: 0.677
Epoch [ 10/10] D_loss: 1.3647 G_loss: 0.7611 D_real_acc: 0.455 D_fake_acc: 0.655

Training completed!
```

```
==================================================
TRAINING SUMMARY
==================================================
Final Generator Loss: 0.7611
Final Discriminator Loss: 1.3647
Final D Real Accuracy: 0.455
Final D Fake Accuracy: 0.655
Average Generator Loss: 0.8769
Average Discriminator Loss: 1.3135
==================================================
```

```
Displaying training results...
```



```
Comparing real vs generated images...
```




```
Models saved to fashion_gan_models/

========================================
Fashion-MNIST GAN training completed!
Check 'fashion_samples' folder for generated images during training.
Models saved in 'fashion_gan_models' folder.
```

Learning Outcomes:

1. Learned the structure and operational principles of Generative Adversarial Networks (GANs), focusing on the functions of the Generator and Discriminator.
2. Acquired practical experience in training GAN models with PyTorch, tracking their training progress, and visualizing the outputs they produce.
3. Built the ability to assess the quality of generated images through both visual inspection and established quantitative metrics like FID and IS.

EXPERIMENT-3

Aim: To perform text classification tasks using NLP techniques and compare different algorithms for accuracy and efficiency.

Objectives:

- To preprocess textual data using NLP techniques and convert it into suitable feature representations.
- To perform text classification with different algorithms and compare their accuracy and efficiency using standard evaluation metrics.

Theory:

Introduction

Text classification is a key task in Natural Language Processing where the objective is to categorize text into predefined labels. In this project, the **20 Newsgroups dataset** is used to classify documents into multiple news categories. This type of task is widely applied in spam filtering, sentiment analysis, and intent recognition systems.

Preprocessing of Data

Raw text is unstructured, so systematic preprocessing is required. The text is converted into lowercase, punctuation and numbers are removed, stop words are filtered out, and lemmatization is applied to normalize words. These steps ensure that irrelevant details are removed while the semantic meaning is preserved. After cleaning, the text is represented numerically using **TF-IDF vectorization**, which captures the importance of words relative to the dataset.

Key preprocessing steps:

- Lowercasing
- Removing punctuation, numbers, and spaces
- Stop-word removal
- Lemmatization
- TF-IDF vectorization

Models Used

Different machine learning models are trained to compare accuracy and efficiency. Naïve Bayes provides fast and simple classification. Logistic Regression performs well with sparse features from TF-IDF. Linear SVM is highly effective in high-dimensional data. Decision Trees capture non-linear patterns, while Random Forests improve on them using ensemble learning. Finally, the Neural Network (MLP) introduces deeper representation learning but requires more computational time.

Evaluation and Visualization

The performance of each model is measured using accuracy, precision, recall, and F1-score, along with runtime to assess efficiency. These results are compared through bar plots, showing accuracy and training time side by side. Such analysis highlights the trade-off between model effectiveness and computational cost.

Conclusion

The study demonstrates that preprocessing and TF-IDF play a vital role in improving classification accuracy. Logistic Regression, Naïve Bayes, and SVM achieve strong performance with efficiency, while tree-based methods are slower. The MLP provides deeper learning but consumes more time. Thus, the best choice of model depends on the balance required between **accuracy and efficiency** in the application.

Code and Output:

```
import re
import time
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk
import random

nltk.download("stopwords")
nltk.download("wordnet")

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"[^a-z\s]", "", text)
    text = re.sub(r"\s+", " ", text).strip()
    lemmatizer = WordNetLemmatizer()
    tokens = text.split()
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in
stopwords.words("english")]
    return " ".join(tokens)

newsgroups = fetch_20newsgroups(subset='all', shuffle=True, random_state=42)
```

```
X, y = newsgroups.data, newsgroups.target
X_cleaned = [preprocess_text(doc) for doc in X]
for i in random.sample(range(len(X_cleaned)), 5):
    print(f"\nRandom Sample {i}:\n", X_cleaned[i][:300])
```

```
=== Sample 13793 ===
Original:
 From: dtmedin@catbyte.b30.ingr.com (Dave Medin)
Subject: TDR plug-in
Reply-To: dtmedin@catbyte.b30.ingr.com
Organization: Intergraph Corporation, Huntsville AL
Lines: 15

If anyone out there has an HP180 series scope or mainframe, I
have the TDR plug in (the 1810, I believe) for it and have no need


Preprocessed:
 dtmedincatbytebingrcom dave medin subject tdr plugin replyto dtmedincatbytebingrcom organization intergraph corporation huntsville al line anyone hp series scope mainframe tdr

=== Sample 15247 ===
Original:
 From: jesup@cbmvax.cbm.commodore.com (Randell Jesup)
Subject: Re: Products to handle HDTV moving pircture (180MB/sec)
Reply-To: jesup@cbmvax.cbm.commodore.com (Randell Jesup)
Organization: Commodore, West Chester, PA
Lines: 31

kazsato@twics.co.jp writes:
>I'd like to know if there is any system (CP

Preprocessed:
 jesupcbmvaxcbmcommodorecom randell jesup subject product handle hdtv moving pircture mbsec replyto jesupcbmvaxcbmcommodorecom randell jesup organization commodore west chester

=== Sample 3855 ===
Original:
 From: Lawrence Curcio <lc2b+@andrew.cmu.edu>
Subject: Athlete's Heart
Organization: Doctoral student, Public Policy and Management, Carnegie Mellon, Pittsburgh, PA
Lines: 16
NNTP-Posting-Host: andrew.cmu.edu

I've read that exercise makes the heart pump more blood at a stroke, and
that it also makes

Preprocessed:
 lawrence curcio lcbandrewcmuedu subject athlete heart organization doctoral student public policy management carnegie mellon pittsburgh pa line nntppostinghost andrewcmuedu ive
```

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

vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

models = {
    "Naive Bayes": MultinomialNB(),
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Linear SVM": LinearSVC(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "Neural Network (MLP)": MLPClassifier(hidden_layer_sizes=(100,), max_iter=300)
}

results = {}
for name, model in models.items():
    start = time.time()
    model.fit(X_train_tfidf, y_train)
    y_pred = model.predict(X_test_tfidf)
    end = time.time()
    acc = accuracy_score(y_test, y_pred)
    runtime = end - start
```

```
    results[name] = {"accuracy": acc, "runtime": runtime}
    print(f"\n=== {name} ===")
    print(f"Accuracy: {acc:.4f}")
    print(f"Training + Prediction Time: {runtime:.2f} seconds")
    print(classification_report(y_test, y_pred,
target_names=newsgroups.target_names))
```

```
=== Naive Bayes ===
Accuracy: 0.8483
Training + Prediction Time: 0.05 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.81      0.81      0.81       151
           comp.graphics       0.71      0.85      0.77       202
 comp.os.ms-windows.misc       0.74      0.78      0.76       195
comp.sys.ibm.pc.hardware       0.62      0.77      0.68       183
   comp.sys.mac.hardware       0.88      0.83      0.85       205
          comp.windows.x       0.86      0.82      0.84       215
            misc.forsale       0.87      0.72      0.79       193
               rec.autos       0.88      0.90      0.89       196
         rec.motorcycles       0.90      0.92      0.91       168
      rec.sport.baseball       0.96      0.94      0.95       211
        rec.sport.hockey       0.94      0.97      0.96       198
               sci.crypt       0.94      0.95      0.95       201
         sci.electronics       0.88      0.75      0.81       202
                 sci.med       0.95      0.91      0.93       194
               sci.space       0.94      0.96      0.95       189
  soc.religion.christian       0.73      0.96      0.83       202
      talk.politics.guns       0.81      0.94      0.87       188
   talk.politics.mideast       0.93      0.95      0.94       182
      talk.politics.misc       0.91      0.74      0.81       159
      talk.religion.misc       0.90      0.35      0.50       136

                accuracy                           0.85      3770
               macro avg       0.86      0.84      0.84      3770
            weighted avg       0.86      0.85      0.85      3770
```

```
=== Logistic Regression ===
Accuracy: 0.8634
Training + Prediction Time: 7.49 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.82      0.83      0.83       151
           comp.graphics       0.74      0.81      0.77       202
 comp.os.ms-windows.misc       0.77      0.78      0.78       195
comp.sys.ibm.pc.hardware       0.66      0.70      0.68       183
   comp.sys.mac.hardware       0.87      0.81      0.84       205
          comp.windows.x       0.84      0.81      0.83       215
            misc.forsale       0.82      0.77      0.79       193
               rec.autos       0.88      0.90      0.89       196
         rec.motorcycles       0.96      0.94      0.95       168
      rec.sport.baseball       0.96      0.94      0.95       211
        rec.sport.hockey       0.96      0.98      0.97       198
               sci.crypt       0.96      0.96      0.96       201
         sci.electronics       0.80      0.79      0.79       202
                 sci.med       0.93      0.92      0.93       194
               sci.space       0.91      0.96      0.93       189
  soc.religion.christian       0.86      0.95      0.90       202
      talk.politics.guns       0.89      0.94      0.91       188
   talk.politics.mideast       0.95      0.96      0.95       182
      talk.politics.misc       0.88      0.83      0.85       159
      talk.religion.misc       0.84      0.60      0.70       136

                accuracy                           0.86      3770
               macro avg       0.86      0.86      0.86      3770
            weighted avg       0.86      0.86      0.86      3770
```

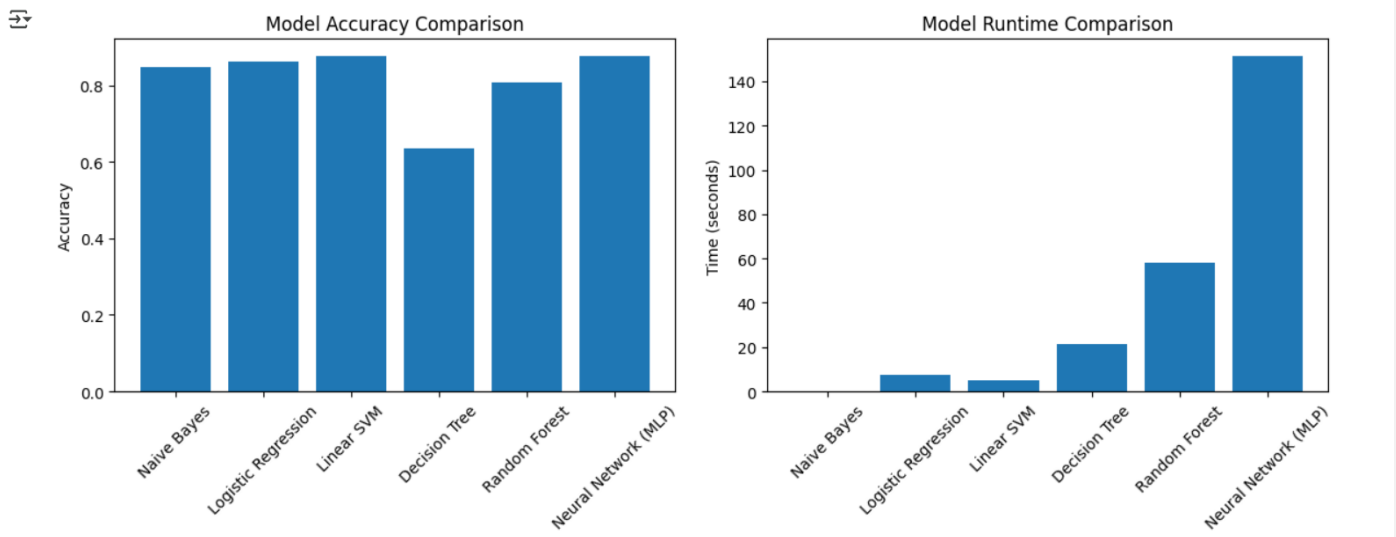
```
=== Linear SVM ===
Accuracy: 0.8780
Training + Prediction Time: 4.98 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.86      0.86      0.86       151
           comp.graphics       0.74      0.81      0.77       202
 comp.os.ms-windows.misc       0.82      0.79      0.81       195
comp.sys.ibm.pc.hardware       0.70      0.72      0.71       183
   comp.sys.mac.hardware       0.86      0.83      0.85       205
          comp.windows.x       0.84      0.84      0.84       215
            misc.forsale       0.83      0.79      0.81       193
               rec.autos       0.91      0.93      0.92       196
         rec.motorcycles       0.95      0.93      0.94       168
      rec.sport.baseball       0.95      0.93      0.94       211
        rec.sport.hockey       0.96      0.99      0.97       198
               sci.crypt       0.95      0.98      0.96       201
         sci.electronics       0.84      0.78      0.81       202
                 sci.med       0.95      0.93      0.94       194
               sci.space       0.92      0.96      0.94       189
  soc.religion.christian       0.89      0.95      0.92       202
      talk.politics.guns       0.92      0.94      0.93       188
   talk.politics.mideast       0.94      0.95      0.95       182
      talk.politics.misc       0.88      0.89      0.88       159
      talk.religion.misc       0.84      0.72      0.77       136

                accuracy                           0.88      3770
               macro avg       0.88      0.88      0.88      3770
            weighted avg       0.88      0.88      0.88      3770
```

```
=== Decision Tree ===
Accuracy: 0.6355
Training + Prediction Time: 21.61 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.55      0.58      0.57       151
           comp.graphics       0.44      0.49      0.46       202
 comp.os.ms-windows.misc       0.56      0.65      0.60       195
comp.sys.ibm.pc.hardware       0.41      0.45      0.43       183
   comp.sys.mac.hardware       0.63      0.55      0.59       205
          comp.windows.x       0.65      0.57      0.61       215
            misc.forsale       0.63      0.60      0.61       193
               rec.autos       0.65      0.65      0.65       196
         rec.motorcycles       0.70      0.75      0.72       168
      rec.sport.baseball       0.68      0.70      0.69       211
        rec.sport.hockey       0.75      0.73      0.74       198
               sci.crypt       0.83      0.78      0.80       201
         sci.electronics       0.47      0.44      0.46       202
                 sci.med       0.69      0.69      0.69       194
               sci.space       0.75      0.78      0.76       189
  soc.religion.christian       0.73      0.75      0.74       202
      talk.politics.guns       0.64      0.72      0.68       188
   talk.politics.mideast       0.85      0.77      0.81       182
      talk.politics.misc       0.60      0.58      0.59       159
      talk.religion.misc       0.47      0.43      0.45       136

                accuracy                           0.64      3770
               macro avg       0.64      0.63      0.63      3770
            weighted avg       0.64      0.64      0.64      3770
```

```
=== Random Forest ===
Accuracy: 0.8093
Training + Prediction Time: 57.89 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.83      0.80      0.81       151
           comp.graphics       0.63      0.71      0.67       202
 comp.os.ms-windows.misc       0.71      0.84      0.77       195
comp.sys.ibm.pc.hardware       0.61      0.68      0.64       183
   comp.sys.mac.hardware       0.87      0.80      0.83       205
          comp.windows.x       0.85      0.73      0.78       215
            misc.forsale       0.74      0.76      0.75       193
               rec.autos       0.85      0.83      0.84       196
         rec.motorcycles       0.90      0.90      0.90       168
      rec.sport.baseball       0.86      0.91      0.89       211
        rec.sport.hockey       0.89      0.90      0.90       198
               sci.crypt       0.93      0.90      0.91       201
         sci.electronics       0.75      0.67      0.70       202
                 sci.med       0.84      0.88      0.86       194
               sci.space       0.87      0.92      0.89       189
  soc.religion.christian       0.77      0.95      0.85       202
      talk.politics.guns       0.79      0.88      0.83       188
   talk.politics.mideast       0.94      0.88      0.91       182
      talk.politics.misc       0.80      0.67      0.73       159
      talk.religion.misc       0.90      0.48      0.62       136

                accuracy                           0.81      3770
               macro avg       0.82      0.80      0.80      3770
            weighted avg       0.82      0.81      0.81      3770


=== Neural Network (MLP) ===
Accuracy: 0.8769
Training + Prediction Time: 151.55 seconds
                          precision    recall  f1-score   support

             alt.atheism       0.86      0.88      0.87       151
           comp.graphics       0.77      0.81      0.79       202
 comp.os.ms-windows.misc       0.82      0.79      0.80       195
comp.sys.ibm.pc.hardware       0.66      0.76      0.71       183
   comp.sys.mac.hardware       0.86      0.84      0.85       205
          comp.windows.x       0.82      0.83      0.83       215
            misc.forsale       0.82      0.75      0.78       193
               rec.autos       0.87      0.93      0.90       196
         rec.motorcycles       0.96      0.95      0.95       168
      rec.sport.baseball       0.96      0.94      0.95       211
        rec.sport.hockey       0.95      0.97      0.96       198
               sci.crypt       0.96      0.95      0.95       201
         sci.electronics       0.87      0.75      0.81       202
                 sci.med       0.95      0.92      0.93       194
               sci.space       0.95      0.96      0.96       189
  soc.religion.christian       0.91      0.94      0.92       202
      talk.politics.guns       0.91      0.94      0.92       188
   talk.politics.mideast       0.94      0.94      0.94       182
      talk.politics.misc       0.88      0.91      0.90       159
      talk.religion.misc       0.84      0.76      0.80       136

                accuracy                           0.88      3770
               macro avg       0.88      0.88      0.88      3770
            weighted avg       0.88      0.88      0.88      3770
```

```
model_names = list(results.keys())
accuracies = [results[m]["accuracy"] for m in model_names]
runtimes = [results[m]["runtime"] for m in model_names]

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.bar(model_names, accuracies)
plt.title("Model Accuracy Comparison")
plt.ylabel("Accuracy")
plt.xticks(rotation=45)

plt.subplot(1,2,2)
plt.bar(model_names, runtimes)
```

```
plt.title("Model Runtime Comparison")
plt.ylabel("Time (seconds)")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Learning Outcomes:

1. Gained understanding of text preprocessing steps like cleaning, stop-word removal, lemmatization, and TF-IDF representation.
2. Learned to apply and evaluate multiple machine learning and neural models for text classification tasks.
3. Developed skills to compare models based on accuracy, precision, recall, F1-score, and runtime efficiency.

EXPERIMENT-4

Aim: To process large-scale datasets using Spark's distributed computing capabilities and run machine learning algorithms on them.

Objectives:

- To process and analyze large-scale datasets efficiently using Apache Spark's distributed computing framework, leveraging its speed, fault tolerance, and parallelism.
- To implement and evaluate scalable machine learning models on distributed data using Spark MLlib.

Theory:

Apache Spark is an open-source distributed computing framework for fast processing of large datasets. It uses in-memory computation and parallel processing across multiple nodes, making it scalable, fault-tolerant, and suitable for both batch and streaming workloads.

Spark MLlib

MLlib is Spark's machine learning library, offering distributed algorithms for classification, regression, clustering, and recommendation. It supports Python, Scala, and Java, enabling large-scale model training that a single machine cannot handle.

Dataset and Exploration

The pipeline creates a synthetic dataset with numeric, categorical, binary, and continuous features. Exploratory analysis inspects schema, statistics, missing values, and feature distributions, supported by visualizations like histograms, bar charts, and correlation heatmaps.

Data Preprocessing

Categorical features are indexed and one-hot encoded, numeric features standardized, and age bucketized. Features are assembled into a single vector, and PCA is optionally applied for dimensionality reduction.

Model Training and Evaluation

Classification models (Logistic Regression, Random Forest, Gradient Boosting) and regression models (Linear Regression, Random Forest Regressor) are trained and evaluated using metrics like AUC, accuracy, RMSE, and R^2. Feature importance highlights key predictors.

K-Means clustering is performed for different k values, evaluating compactness via WSSSE. Results and metrics are visualized in dashboards summarizing model performance, training times, and feature importance.

The full pipeline integrates dataset creation, exploration, preprocessing, modeling, clustering, and feature analysis. Results can be saved to Google Drive, and the Spark session is closed to release resources, enabling efficient, scalable ML workflows.

Code:

```
print("Installing Apache Spark and dependencies...")
!apt-get update -qq
!apt-get install -y openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://archive.apache.org/dist/spark/spark-3.4.0/spark-3.4.0-bin-
hadoop3.tgz
!tar xf spark-3.4.0-bin-hadoop3.tgz
!pip install -q findspark pyspark==3.4.0
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.4.0-bin-hadoop3"

import findspark
findspark.init()
print("☑ Spark installation completed!")

import sys
sys.path.append('/content/spark-3.4.0-bin-hadoop3/python')
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import (VectorAssembler, StandardScaler, StringIndexer,
OneHotEncoder, MinMaxScaler, Bucketizer, PCA)
from pyspark.ml.classification import (
LogisticRegression, RandomForestClassifier, GBTClassifier)
from pyspark.ml.regression import (LinearRegression, RandomForestRegressor,
GBTRegressor)
from pyspark.ml.clustering import KMeans, GaussianMixture
from pyspark.ml.evaluation import (BinaryClassificationEvaluator,
MulticlassClassificationEvaluator,RegressionEvaluator)
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
import time
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import builtins

class ColabSparkMLProcessor:
    def __init__(self, app_name="ColabLargeScaleML"):
        """Initialize Spark session optimized for Colab environment"""
        print(f"🚀 Initializing Spark session: {app_name}")
        self.spark = SparkSession.builder \
            .appName(app_name) \
            .master("local[*]") \
            .config("spark.driver.memory", "2g") \
            .config("spark.driver.maxResultSize", "1g") \
            .config("spark.sql.adaptive.enabled", "true") \
```

```
            .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
            .config("spark.serializer",
"org.apache.spark.serializer.KryoSerializer") \
            .config("spark.sql.execution.arrow.pyspark.enabled", "true") \
            .getOrCreate()
        self.spark.sparkContext.setLogLevel("ERROR")

        cores = self.spark.sparkContext.defaultParallelism
        print(f"☑ Spark initialized with {cores} cores")
        print(f"📊 Spark UI: {self.spark.sparkContext.uiWebUrl}")

    def create_sample_dataset(self, num_rows=100000):
        """Create sample dataset optimized for Colab memory limits"""
        print(f"📁 Creating sample dataset with {num_rows:,} rows...")
        start_time = time.time()

        df = self.spark.range(num_rows) \
            .select(
                col("id"),

                (rand() * 100).alias("feature1"),
                (randn() * 10 + 50).alias("feature2"),
                (rand() * 1000).alias("feature3"),
                (randn() * 5 + 25).alias("feature4"),

                when(rand() > 0.7, "Category_A")
                .when(rand() > 0.4, "Category_B")
                .otherwise("Category_C").alias("category"),
                when(rand() > 0.6, "High")
                .when(rand() > 0.3, "Medium")
                .otherwise("Low").alias("priority"),

                (rand() * 60 + 18).cast("int").alias("age"),
                (rand() * 80000 + 20000).alias("income"),

                (rand() > 0.4).cast("int").alias("target_binary"),

                (rand() * 1000 + randn() * 100).alias("target_continuous")
            )

        df = df.repartition(4)
        df.cache()

        count = df.count()
        creation_time = time.time() - start_time
        print(f"☑ Dataset created in {creation_time:.2f}s")
        print(f"📊 Shape: {count:,} rows × {len(df.columns)} columns")
        return df

    def explore_dataset(self, df):
```

```
        """Comprehensive dataset exploration with visualizations"""
        print("\n" + "="*50)
        print("📊 DATASET EXPLORATION")
        print("="*50)

        print("\n🔍 Dataset Schema:")
        df.printSchema()
        print(f"\n🏷️ Dataset size: {df.count():,} rows")

        numeric_cols = [f.name for f in df.schema.fields
                        if f.dataType in [IntegerType(), LongType(), DoubleType(),
FloatType()]]
        print(f"\n📈 Statistical Summary ({len(numeric_cols)} numeric columns):")
        stats_df = df.select(numeric_cols).describe()
        stats_df.show()

        stats_pandas = stats_df.toPandas()

        print("\n❌ Missing Values Analysis:")
        null_counts = df.select([sum(col(c).isNull().cast("int")).alias(c) for c in
df.columns])
        null_df = null_counts.collect()[0].asDict()
        for col_name, null_count in null_df.items():
            print(f" {col_name}: {null_count} nulls")

        categorical_cols = [f.name for f in df.schema.fields if f.dataType ==
StringType()]
        print(f"\n🏷️ Categorical Features ({len(categorical_cols)} columns):")
        for col_name in categorical_cols:
            print(f"\n {col_name.upper()} Distribution:")
            cat_dist = df.groupBy(col_name).count().orderBy("count",
ascending=False)
            cat_dist.show(10)

        self._create_exploration_plots(df, numeric_cols, categorical_cols)
        return {
            'numeric_columns': numeric_cols,
            'categorical_columns': categorical_cols,
            'row_count': df.count(),
            'column_count': len(df.columns)
        }
    def _create_exploration_plots(self, df, numeric_cols, categorical_cols):
        """Create exploration visualizations"""
        print("\n🎨 Creating visualizations...")

        sample_df = df.sample(0.1, seed=42).toPandas()

        plt.style.use('seaborn-v0_8')
        fig = plt.figure(figsize=(20, 15))
```

```
        if len(numeric_cols) > 0:
            for i, col in enumerate(numeric_cols[:6]):
                plt.subplot(3, 4, i + 1)
                plt.hist(sample_df[col], bins=30, alpha=0.7, edgecolor='black')
                plt.title(f'Distribution of {col}')
                plt.xlabel(col)
                plt.ylabel('Frequency')

        plot_idx = 7
        for col in categorical_cols[:2]:
            plt.subplot(3, 4, plot_idx)
            value_counts = sample_df[col].value_counts()
            plt.bar(value_counts.index, value_counts.values)
            plt.title(f'Distribution of {col}')
            plt.xlabel(col)
            plt.ylabel('Count')
            plt.xticks(rotation=45)
            plot_idx += 1

        if len(numeric_cols) > 1:
            plt.subplot(3, 4, 9)
            numeric_sample = sample_df[numeric_cols[:6]]
            corr_matrix = numeric_sample.corr()
            sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0,
                        square=True, fmt='.2f', cbar_kws={"shrink": .8})
            plt.title('Feature Correlations')
        plt.tight_layout()
        plt.show()
    def preprocess_data(self, df, target_col="target_binary"):
        """Advanced preprocessing pipeline"""
        print("\n" + "="*50)
        print("🔧 DATA PREPROCESSING")
        print("="*50)
        start_time = time.time()

        numeric_cols = [f.name for f in df.schema.fields
                        if f.dataType in [IntegerType(), LongType(), DoubleType(),
FloatType()]
                        and f.name not in [target_col, "target_continuous", "id"]]
        categorical_cols = [f.name for f in df.schema.fields if f.dataType ==
StringType()]
        print(f"📊 Processing {len(numeric_cols)} numeric and
{len(categorical_cols)} categorical features")
        stages = []
        encoded_cols = []
        for col_name in categorical_cols:
            indexer = StringIndexer(
                inputCol=col_name,
                outputCol=f"{col_name}_indexed",
                handleInvalid="keep"
```

```
            )
            stages.append(indexer)
            encoder = OneHotEncoder(
                inputCol=f"{col_name}_indexed",
                outputCol=f"{col_name}_encoded"
            )
            stages.append(encoder)
            encoded_cols.append(f"{col_name}_encoded")
        if "age" in df.columns:
            age_bucketizer = Bucketizer(
                splits=[-float('inf'), 25, 35, 50, 65, float('inf')],
                inputCol="age",
                outputCol="age_group"
            )
            stages.append(age_bucketizer)
            numeric_cols = [col for col in numeric_cols if col != "age"]
            numeric_cols.append("age_group")
        feature_cols = numeric_cols + encoded_cols
        assembler = VectorAssembler(
            inputCols=feature_cols,
            outputCol="raw_features",
            handleInvalid="keep"
        )
        stages.append(assembler)
        scaler = StandardScaler(
            inputCol="raw_features",
            outputCol="scaled_features",
            withStd=True,
            withMean=True
        )
        stages.append(scaler)
        max_components = builtins.min(10, len(feature_cols))
        if max_components > 1:
            pca = PCA(
                k=max_components,
                inputCol="scaled_features",
                outputCol="features"
            )
            stages.append(pca)
        else:
            from pyspark.ml.feature import SQLTransformer
            sql_transformer = SQLTransformer(
                statement="SELECT *, scaled_features as features FROM __THIS__"
            )
            stages.append(sql_transformer)
        print("🏗️ Building preprocessing pipeline...")
        pipeline = Pipeline(stages=stages)
        print("⚙️ Fitting preprocessing pipeline...")
        pipeline_model = pipeline.fit(df)
        print("🔄 Transforming data...")
```

```
        processed_df = pipeline_model.transform(df)
        processed_df.cache()

        processed_count = processed_df.count()
        processing_time = time.time() - start_time
        print(f"☑ Preprocessing completed in {processing_time:.2f}s")
        print(f"📊 Processed {processed_count:,} rows")
        print(f"🎯 Final feature vector size: {len(feature_cols)}")
        return processed_df, pipeline_model

    def train_classification_models(self, df, target_col="target_binary"):
        """Train multiple classification models"""
        print("\n" + "="*50)
        print("🤖 CLASSIFICATION MODEL TRAINING")
        print("="*50)
        print("📂 Splitting data...")
        train_df, test_df = df.randomSplit([0.8, 0.2], seed=42)
        train_df.cache()
        test_df.cache()
        train_count = train_df.count()
        test_count = test_df.count()
        print(f"📊 Training set: {train_count:,} rows")
        print(f"📊 Test set: {test_count:,} rows")
        models = {}
        evaluator_binary = BinaryClassificationEvaluator(labelCol=target_col)
        evaluator_multi = MulticlassClassificationEvaluator(labelCol=target_col)

        print("\n🎯 Training Logistic Regression...")
        start_time = time.time()
        lr = LogisticRegression(
            labelCol=target_col,
            featuresCol="features",
            maxIter=100,
            regParam=0.01
        )
        lr_model = lr.fit(train_df)
        lr_predictions = lr_model.transform(test_df)
        lr_predictions.cache()
        lr_auc = evaluator_binary.evaluate(lr_predictions)
        lr_accuracy = evaluator_multi.evaluate(lr_predictions,
                                               {evaluator_multi.metricName:
"accuracy"})
        models['Logistic Regression'] = {
            'model': lr_model,
            'predictions': lr_predictions,
            'auc': lr_auc,
            'accuracy': lr_accuracy,
            'training_time': time.time() - start_time
        }
```

```
        print(f" ⏱️ Training time: {models['Logistic
Regression']['training_time']:.2f}s")
        print(f" 📈 AUC: {lr_auc:.4f}")
        print(f" 🎯 Accuracy: {lr_accuracy:.4f}")

        print("\n🌲 Training Random Forest...")
        start_time = time.time()
        rf = RandomForestClassifier(
            labelCol=target_col,
            featuresCol="features",
            numTrees=50,
            maxDepth=10,
            seed=42
        )
        rf_model = rf.fit(train_df)
        rf_predictions = rf_model.transform(test_df)
        rf_predictions.cache()
        rf_auc = evaluator_binary.evaluate(rf_predictions)
        rf_accuracy = evaluator_multi.evaluate(rf_predictions,
{evaluator_multi.metricName: "accuracy"})
        models['Random Forest'] = {
            'model': rf_model,
            'predictions': rf_predictions,
            'auc': rf_auc,
            'accuracy': rf_accuracy,
            'training_time': time.time() - start_time
        }
        print(f" ⏱️ Training time: {models['Random
Forest']['training_time']:.2f}s")
        print(f" 📈 AUC: {rf_auc:.4f}")
        print(f" 🎯 Accuracy: {rf_accuracy:.4f}")

        print("\n🚀 Training Gradient Boosting...")
        start_time = time.time()
        gbt = GBTClassifier(
            labelCol=target_col,
            featuresCol="features",
            maxIter=30,
            maxDepth=5,
            seed=42
        )
        gbt_model = gbt.fit(train_df)
        gbt_predictions = gbt_model.transform(test_df)
        gbt_predictions.cache()
        gbt_auc = evaluator_binary.evaluate(gbt_predictions)
        gbt_accuracy = evaluator_multi.evaluate(gbt_predictions,
                                                {evaluator_multi.metricName:
"accuracy"})
        models['Gradient Boosting'] = {
            'model': gbt_model,
```

```
            'predictions': gbt_predictions,
            'auc': gbt_auc,
            'accuracy': gbt_accuracy,
            'training_time': time.time() - start_time
        }
        print(f" ⏱️ Training time: {models['Gradient
Boosting']['training_time']:.2f}s")
        print(f" 📈 AUC: {gbt_auc:.4f}")
        print(f" 🎯 Accuracy: {gbt_accuracy:.4f}")

        self._plot_classification_results(models)
        return models

    def train_regression_models(self, df, target_col="target_continuous"):
        """Train regression models"""
        print("\n" + "="*50)
        print("📈 REGRESSION MODEL TRAINING")
        print("="*50)

        train_df, test_df = df.randomSplit([0.8, 0.2], seed=42)
        train_df.cache()
        test_df.cache()
        models = {}
        evaluator = RegressionEvaluator(labelCol=target_col)

        print("\n📏 Training Linear Regression...")
        start_time = time.time()
        lr = LinearRegression(
            labelCol=target_col,
            featuresCol="features",
            maxIter=100,
            regParam=0.01
        )
        lr_model = lr.fit(train_df)
        lr_predictions = lr_model.transform(test_df)
        lr_rmse = evaluator.evaluate(lr_predictions, {evaluator.metricName:
"rmse"})
        lr_r2 = evaluator.evaluate(lr_predictions, {evaluator.metricName: "r2"})
        models['Linear Regression'] = {
            'model': lr_model,
            'predictions': lr_predictions,
            'rmse': lr_rmse,
            'r2': lr_r2,
            'training_time': time.time() - start_time
        }
        print(f" ⏱️ Training time: {models['Linear
Regression']['training_time']:.2f}s")
        print(f" 📊 RMSE: {lr_rmse:.4f}")
        print(f" 📈 R²: {lr_r2:.4f}")
```

```
        print("\n🌳 Training Random Forest Regression...")
        start_time = time.time()
        rfr = RandomForestRegressor(
            labelCol=target_col,
            featuresCol="features",
            numTrees=50,
            maxDepth=10,
            seed=42
        )
        rfr_model = rfr.fit(train_df)
        rfr_predictions = rfr_model.transform(test_df)
        rfr_rmse = evaluator.evaluate(rfr_predictions, {evaluator.metricName:
"rmse"})
        rfr_r2 = evaluator.evaluate(rfr_predictions, {evaluator.metricName: "r2"})
        models['Random Forest Regression'] = {
            'model': rfr_model,
            'predictions': rfr_predictions,
            'rmse': rfr_rmse,
            'r2': rfr_r2,
            'training_time': time.time() - start_time
        }
        print(f" ⏱️ Training time: {models['Random Forest
Regression']['training_time']:.2f}s")
        print(f" 📊 RMSE: {rfr_rmse:.4f}")
        print(f" 📈 R²: {rfr_r2:.4f}")
        return models

    def perform_clustering(self, df, k_values=[3, 5, 7]):
        """Perform clustering analysis"""
        print("\n" + "="*50)
        print("🎯 CLUSTERING ANALYSIS")
        print("="*50)
        clustering_results = {}
        for k in k_values:
            print(f"\n🔍 K-Means clustering with k={k}...")
            start_time = time.time()
            kmeans = KMeans(
                k=k,
                featuresCol="features",
                predictionCol="cluster",
                seed=42,
                maxIter=100
            )
            kmeans_model = kmeans.fit(df)
            clustered_df = kmeans_model.transform(df)
            clustered_df.cache()

            wssse = kmeans_model.summary.trainingCost
            cluster_centers = kmeans_model.clusterCenters()
            clustering_results[f'K-Means (k={k})'] = {
```

```
                'model': kmeans_model,
                'predictions': clustered_df,
                'wssse': wssse,
                'cluster_centers': cluster_centers,
                'training_time': time.time() - start_time
            }
            print(f" ⏱️ Training time: {clustering_results[f'K-Means
(k={k})']['training_time']:.2f}s")
            print(f" 📊 WSSSE: {wssse:.4f}")

            cluster_dist =
clustered_df.groupBy("cluster").count().orderBy("cluster")
            print(" 🎯 Cluster distribution:")
            cluster_dist.show()
        return clustering_results

    def _plot_classification_results(self, models):
        """Plot classification results"""
        print("\n🎨 Creating results visualization...")

        model_names = list(models.keys())
        aucs = [models[name]['auc'] for name in model_names]
        accuracies = [models[name]['accuracy'] for name in model_names]
        times = [models[name]['training_time'] for name in model_names]

        fig, axes = plt.subplots(1, 3, figsize=(18, 6))

        bars1 = axes[0].bar(model_names, aucs, color=['skyblue', 'lightgreen',
'salmon'])
        axes[0].set_title('Model AUC Comparison', fontsize=14, fontweight='bold')
        axes[0].set_ylabel('AUC Score')
        axes[0].set_ylim([0, 1])
        axes[0].tick_params(axis='x', rotation=45)

        for bar, auc in zip(bars1, aucs):
            axes[0].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                         f'{auc:.3f}', ha='center', va='bottom', fontweight='bold')

        bars2 = axes[1].bar(model_names, accuracies, color=['skyblue',
'lightgreen', 'salmon'])
        axes[1].set_title('Model Accuracy Comparison', fontsize=14,
fontweight='bold')
        axes[1].set_ylabel('Accuracy')
        axes[1].set_ylim([0, 1])
        axes[1].tick_params(axis='x', rotation=45)
        for bar, acc in zip(bars2, accuracies):
            axes[1].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                         f'{acc:.3f}', ha='center', va='bottom', fontweight='bold')
```

```
        bars3 = axes[2].bar(model_names, times, color=['skyblue', 'lightgreen',
'salmon'])
        axes[2].set_title('Training Time Comparison', fontsize=14,
fontweight='bold')
        axes[2].set_ylabel('Training Time (seconds)')
        axes[2].tick_params(axis='x', rotation=45)
        for bar, time_val in zip(bars3, times):
            axes[2].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.1,
                    f'{time_val:.1f}s', ha='center', va='bottom',
fontweight='bold')
        plt.tight_layout()
        plt.show()

    def feature_importance_analysis(self, rf_model, feature_names=None):
        """Analyze and visualize feature importance"""
        print("\n" + "="*50)
        print("🔍 FEATURE IMPORTANCE ANALYSIS")
        print("="*50)

        importances = rf_model.featureImportances.toArray()
        if feature_names is None:
            feature_names = [f"Feature_{i}" for i in range(len(importances))]

        importance_data = list(zip(feature_names, importances))
        importance_df = pd.DataFrame(importance_data, columns=['Feature',
'Importance'])
        importance_df = importance_df.sort_values('Importance', ascending=False)
        print("🏆 Top 10 Most Important Features:")
        print(importance_df.head(10).to_string(index=False))

        plt.figure(figsize=(12, 8))
        top_features = importance_df.head(15)
        bars = plt.bar(range(len(top_features)), top_features['Importance'],
                       color='steelblue', alpha=0.8)
        plt.title('Top 15 Feature Importances', fontsize=16, fontweight='bold')
        plt.xlabel('Features')
        plt.ylabel('Importance')
        plt.xticks(range(len(top_features)), top_features['Feature'], rotation=45,
ha='right')

        for i, bar in enumerate(bars):
            plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.001,
                    f'{top_features.iloc[i]["Importance"]:.3f}',
                    ha='center', va='bottom', fontsize=10)
        plt.tight_layout()
        plt.show()
        return importance_df

    def run_complete_pipeline(self, dataset_size=100000):
        """Execute the complete ML pipeline"""
```

```
        print("🚀" * 20)
        print("COMPLETE SPARK ML PIPELINE FOR GOOGLE COLAB")
        print("🚀" * 20)
        pipeline_start = time.time()
        try:

            print("\n📁 STEP 1: Creating Dataset")
            df = self.create_sample_dataset(dataset_size)

            print("\n🔍 STEP 2: Dataset Exploration")
            exploration_results = self.explore_dataset(df)

            print("\n🔧 STEP 3: Data Preprocessing")
            processed_df, preprocessing_model = self.preprocess_data(df)

            print("\n🤖 STEP 4: Classification Models")
            classification_models = self.train_classification_models(processed_df)

            print("\n📈 STEP 5: Regression Models")
            regression_models = self.train_regression_models(processed_df)

            print("\n🎯 STEP 6: Clustering Analysis")
            clustering_results = self.perform_clustering(processed_df)

            print("\n🔍 STEP 7: Feature Importance")
            rf_model = classification_models['Random Forest']['model']
            importance_df = self.feature_importance_analysis(rf_model)

            total_time = time.time() - pipeline_start
            print("\n" + "🎉" * 20)
            print("PIPELINE COMPLETED SUCCESSFULLY!")
            print("🎉" * 20)
            print(f"⏱️ Total execution time: {total_time:.2f} seconds")
            print(f"📊 Dataset processed: {dataset_size:,} rows")
            print(f"🤖 Models trained: {len(classification_models) +
len(regression_models)} ML models")
            print(f"🎯 Clustering algorithms: {len(clustering_results)}")
            return {
                'dataset': processed_df,
                'preprocessing_model': preprocessing_model,
                'classification_models': classification_models,
                'regression_models': regression_models,
                'clustering_results': clustering_results,
                'feature_importance': importance_df,
                'execution_time': total_time,
                'exploration_results': exploration_results
            }
        except Exception as e:
            print(f"❌ Error in pipeline execution: {str(e)}")
            raise e
```

```
    def model_performance_dashboard(self, results):
        """Create a comprehensive performance dashboard"""
        print("\n" + "="*50)
        print("📊 MODEL PERFORMANCE DASHBOARD")
        print("="*50)

        classification_models = results['classification_models']
        regression_models = results['regression_models']
        clustering_results = results['clustering_results']

        fig = plt.figure(figsize=(20, 12))

        ax1 = plt.subplot(2, 3, 1)
        model_names = list(classification_models.keys())
        aucs = [classification_models[name]['auc'] for name in model_names]
        colors = ['#FF6B6B', '#4ECDC4', '#45B7D1']
        bars = ax1.bar(model_names, aucs, color=colors, alpha=0.8)
        ax1.set_title('Classification AUC Scores', fontsize=14, fontweight='bold')
        ax1.set_ylabel('AUC Score')
        ax1.set_ylim([0, 1])
        ax1.tick_params(axis='x', rotation=45)
        for bar, auc in zip(bars, aucs):
            ax1.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                     f'{auc:.3f}', ha='center', va='bottom', fontweight='bold')

        ax2 = plt.subplot(2, 3, 2)
        reg_names = list(regression_models.keys())
        r2_scores = [regression_models[name]['r2'] for name in reg_names]
        bars = ax2.bar(reg_names, r2_scores, color=['#96CEB4', '#FECA57'],
alpha=0.8)
        ax2.set_title('Regression R² Scores', fontsize=14, fontweight='bold')
        ax2.set_ylabel('R² Score')
        ax2.tick_params(axis='x', rotation=45)
        for bar, r2 in zip(bars, r2_scores):
            ax2.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                     f'{r2:.3f}', ha='center', va='bottom', fontweight='bold')

        ax3 = plt.subplot(2, 3, 3)
        all_models = list(classification_models.keys()) +
list(regression_models.keys())
        all_times = ([classification_models[name]['training_time'] for name in
                      classification_models.keys()] +
                     [regression_models[name]['training_time'] for name in
regression_models.keys()])
        bars = ax3.bar(all_models, all_times,
                       color=['#FF6B6B', '#4ECDC4', '#45B7D1', '#96CEB4',
'#FECA57'], alpha=0.8)
        ax3.set_title('Training Time Comparison', fontsize=14, fontweight='bold')
        ax3.set_ylabel('Time (seconds)')
```

```
        ax3.tick_params(axis='x', rotation=45)
        for bar, time_val in zip(bars, all_times):
            ax3.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.1,
                     f'{time_val:.1f}s', ha='center', va='bottom',
fontweight='bold')

        ax4 = plt.subplot(2, 3, 4)
        cluster_names = list(clustering_results.keys())
        wssse_values = [clustering_results[name]['wssse'] for name in
cluster_names]
        ax4.plot(range(len(cluster_names)), wssse_values, 'o-', linewidth=2,
markersize=8,
                 color='#E17055')
        ax4.set_title('K-Means WSSSE by K Value', fontsize=14, fontweight='bold')
        ax4.set_xlabel('K Value')
        ax4.set_ylabel('WSSSE')
        ax4.set_xticks(range(len(cluster_names)))
        ax4.set_xticklabels([name.split('=')[1].rstrip(')') for name in
cluster_names])
        ax4.grid(True, alpha=0.3)

        ax5 = plt.subplot(2, 3, 5)
        importance_df = results['feature_importance']
        top_10 = importance_df.head(10)
        bars = ax5.barh(range(len(top_10)), top_10['Importance'], color='#74B9FF',
alpha=0.8)
        ax5.set_title('Top 10 Feature Importances', fontsize=14, fontweight='bold')
        ax5.set_xlabel('Importance')
        ax5.set_yticks(range(len(top_10)))
        ax5.set_yticklabels(top_10['Feature'])
        ax5.invert_yaxis()

        ax6 = plt.subplot(2, 3, 6)
        accuracies = [classification_models[name]['accuracy'] for name in
model_names]

        wedges, texts, autotexts = ax6.pie(accuracies, labels=model_names,
autopct='%1.3f',
                                           colors=colors, startangle=90)
        ax6.set_title('Classification Accuracy Distribution', fontsize=14,
fontweight='bold')
        plt.tight_layout()
        plt.show()

        print("\n PERFORMANCE SUMMARY:")
        print("-" * 40)
        best_classifier = builtins.max(classification_models.items(), key=lambda x:
x[1]['auc'])
        best_regressor = builtins.max(regression_models.items(), key=lambda x:
x[1]['r2'])
```

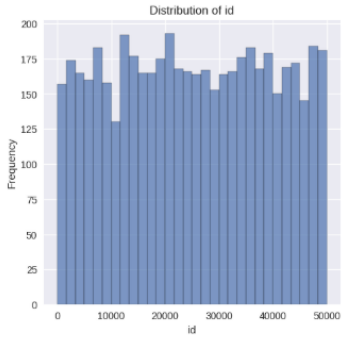
```
        fastest_model = builtins.min(classification_models.items(), key=lambda x:
x[1]['training_time'])
        print(f"🏆 Best Classifier: {best_classifier[0]} (AUC:
{best_classifier[1]['auc']:.4f})")
        print(f"📈 Best Regressor: {best_regressor[0]} (R²:
{best_regressor[1]['r2']:.4f})")
        print(f"⚡ Fastest Training: {fastest_model[0]}
({fastest_model[1]['training_time']:.2f}s)")
        print(f"⏱️ Total Pipeline Time: {results['execution_time']:.2f}s")

    def save_results_to_drive(self, results,
base_path="/content/drive/MyDrive/spark_ml_results/"):
        """Save results to Google Drive (if mounted)"""
        try:
            import os
            if not os.path.exists("/content/drive"):
                print("📁 Google Drive not mounted. To save results:")
                print(" 1. Run: from google.colab import drive")
                print(" 2. Run: drive.mount('/content/drive')")
                return
            os.makedirs(base_path, exist_ok=True)
            print(f"💾 Saving results to {base_path}...")
            importance_df = results['feature_importance']
            importance_df.to_csv(f"{base_path}feature_importance.csv", index=False)

            summary_data = []

            for name, metrics in results['classification_models'].items():
                summary_data.append({
                    'Model': name,
                    'Type': 'Classification',
                    'AUC': metrics['auc'],
                    'Accuracy': metrics['accuracy'],
                    'Training_Time': metrics['training_time']
                })

            for name, metrics in results['regression_models'].items():
                summary_data.append({
                    'Model': name,
                    'Type': 'Regression',
                    'RMSE': metrics['rmse'],
                    'R2': metrics['r2'],
                    'Training_Time': metrics['training_time']
                })
            summary_df = pd.DataFrame(summary_data)
            summary_df.to_csv(f"{base_path}model_performance_summary.csv",
index=False)

            execution_summary = {
                'Total_Execution_Time': results['execution_time'],
```

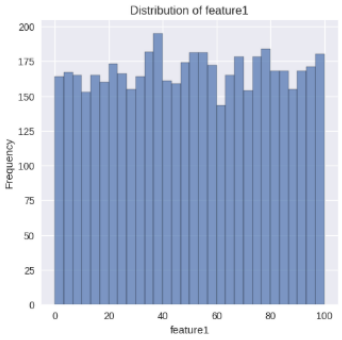
```
                'Dataset_Rows': results['exploration_results']['row_count'],
                'Dataset_Columns': results['exploration_results']['column_count'],
                'Models_Trained': len(results['classification_models']) +
len(results['regression_models']),
                'Clustering_Algorithms': len(results['clustering_results'])
            }
            summary_text = "\n".join([f"{k}: {v}" for k, v in
execution_summary.items()])
            with open(f"{base_path}execution_summary.txt", 'w') as f:
                f.write("SPARK ML PIPELINE EXECUTION SUMMARY\n")
                f.write("="*40 + "\n\n")
                f.write(summary_text)
            print("☑ Results saved successfully!")
            print(f"🗁 Files saved:")
            print(f" - feature_importance.csv")
            print(f" - model_performance_summary.csv")
            print(f" - execution_summary.txt")
        except Exception as e:
            print(f"❌ Error saving results: {str(e)}")

    def close(self):
        """Clean up Spark session"""
        print("\n🧹 Cleaning up Spark session...")
        self.spark.stop()
        print("☑ Spark session closed successfully!")

def main():
    print("🎯 Starting Spark ML Pipeline in Google Colab")
    print("="*60)
    processor = ColabSparkMLProcessor(app_name="ColabSparkMLDemo")
    try:
        print("\n🚀 Executing complete ML pipeline...")
        results = processor.run_complete_pipeline(dataset_size=50000)
        processor.model_performance_dashboard(results)




        processor.save_results_to_drive(results)
        print("\n" + "🎉" * 25)
        print("COLAB SPARK ML PIPELINE COMPLETED!")
        print("🎉" * 25)
        print(f"📊 Processed {results['exploration_results']['row_count']:,}
rows")
        print(f"🤖 Trained {len(results['classification_models']) +
len(results['regression_models'])} models")
        print(f"⏱ Total time: {results['execution_time']:.2f} seconds")
        print(f"🔗 Spark UI: {processor.spark.sparkContext.uiWebUrl}")
        return results, processor
    except Exception as e:
```

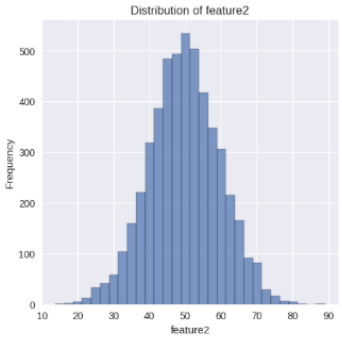
```
        print(f"❌ Pipeline failed: {str(e)}")
        processor.close()
        raise e
if __name__ == "__main__":
    results, processor = main()
```

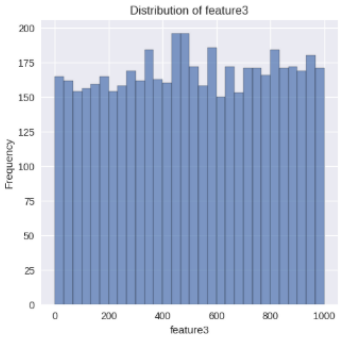
Output:

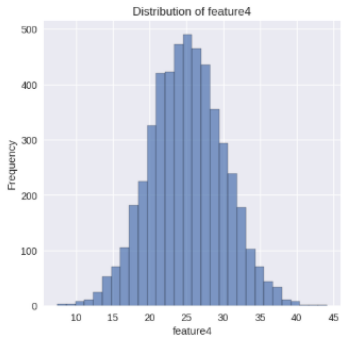
```
🎯 Starting Spark ML Pipeline in Google Colab
============================================================
🚀 Initializing Spark session: ColabSparkMLDemo
✅ Spark initialized with 2 cores
📊 Spark UI: http://5a7456b12781:4040

🚀 Executing complete ML pipeline...
🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀
COMPLETE SPARK ML PIPELINE FOR GOOGLE COLAB
🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀🚀

📁 STEP 1: Creating Dataset
📁 Creating sample dataset with 50,000 rows...
✅ Dataset created in 0.95s
📊 Shape: 50,000 rows x 11 columns

🔍 STEP 2: Dataset Exploration

==================================================
📊 DATASET EXPLORATION
==================================================

🔍 Dataset Schema:
root
 |-- id: long (nullable = false)
 |-- feature1: double (nullable = false)
 |-- feature2: double (nullable = false)
 |-- feature3: double (nullable = false)
 |-- feature4: double (nullable = false)
 |-- category: string (nullable = false)
 |-- priority: string (nullable = false)
 |-- age: integer (nullable = true)
 |-- income: double (nullable = false)
 |-- target_binary: integer (nullable = false)
 |-- target_continuous: double (nullable = false)


🏷️ Dataset size: 50,000 rows

📈 Statistical Summary (9 numeric columns):
+-------+-----------------+------------------+-----------------+------------------+-----------------+-----------------+-----------------+------------------+------------------+
|summary|               id|          feature1|         feature2|          feature3|         feature4|              age|           income|      target_binary|  target_continuous|
+-------+-----------------+------------------+-----------------+------------------+-----------------+-----------------+-----------------+------------------+------------------+
|  count|            50000|             50000|            50000|             50000|            50000|            50000|            50000|             50000|             50000|
|   mean|          24999.5|  50.11013260761266|49.99915524881761|  498.8876482081904|24.999216458401847|          47.4871| 59899.18818058937|            0.5988|  498.5694451401103|
| stddev|14433.901066586257| 28.879705312223553|9.995486475688281|  288.1188996428885| 5.001336848050958|17.352780064575224|23083.839687505915|0.49014626884971507|  304.5908042762995|
|    min|                0|0.001384451569519...|6.729869653551816|0.024430154580978858| 5.486603050077971|               18|20000.912748263334|                 0|-298.99609050634757|
|    max|            49999|  99.99758566401019|93.98361777878564|  999.9925829853719| 49.25912312088123|               77| 99999.91926388569|                 1| 1341.7655584415998|
+-------+-----------------+------------------+-----------------+------------------+-----------------+-----------------+-----------------+------------------+------------------+


❌ Missing Values Analysis:
 id: 0 nulls
 feature1: 0 nulls
 feature2: 0 nulls
 feature3: 0 nulls
 feature4: 0 nulls
 category: 0 nulls
 priority: 0 nulls
 age: 0 nulls
 income: 0 nulls
 target_binary: 0 nulls
 target_continuous: 0 nulls

🏷️ Categorical Features (2 columns):

 CATEGORY Distribution:
+----------+-----+
|  category|count|
+----------+-----+
|Category_B|21182|
|Category_A|14836|
|Category_C|13982|
+----------+-----+


 PRIORITY Distribution:
+--------+-----+
|priority|count|
+--------+-----+
|  Medium|21036|
|    High|19921|
|     Low| 9043|
+--------+-----+
```

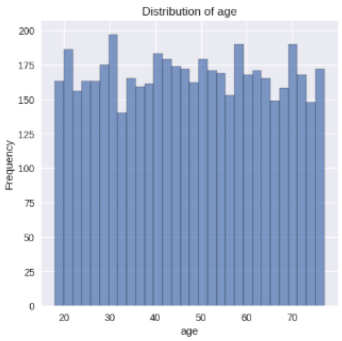


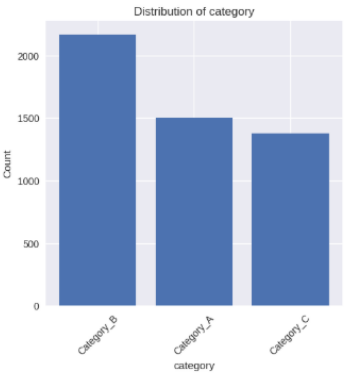


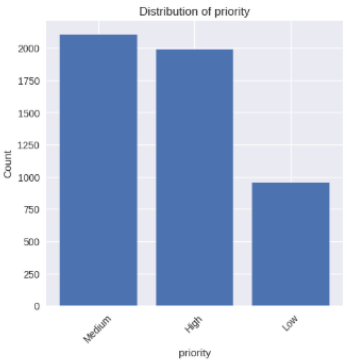


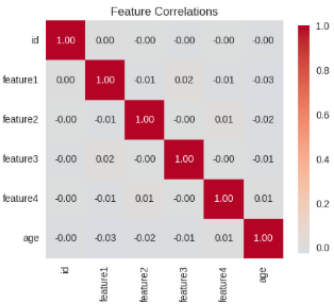










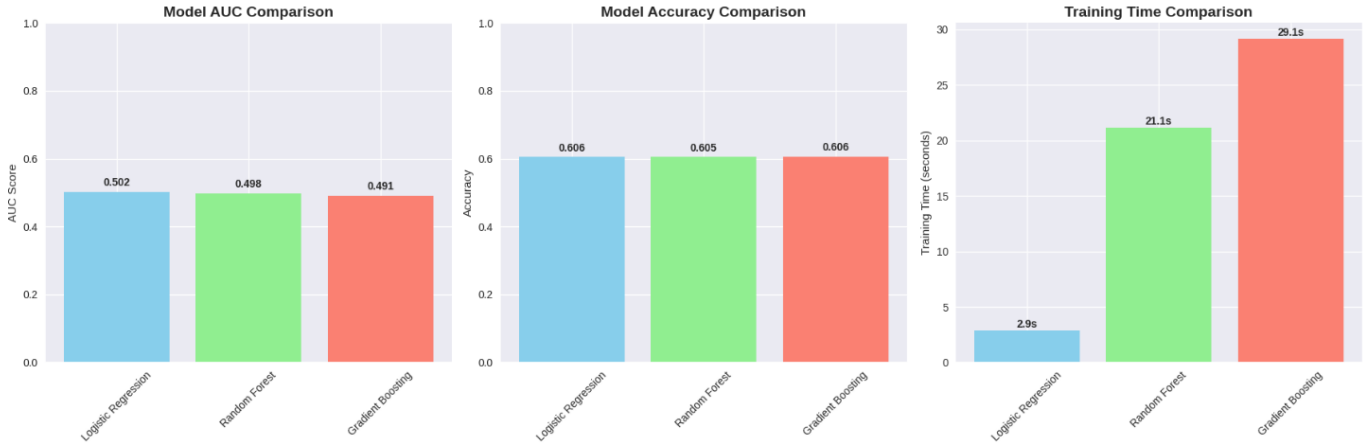


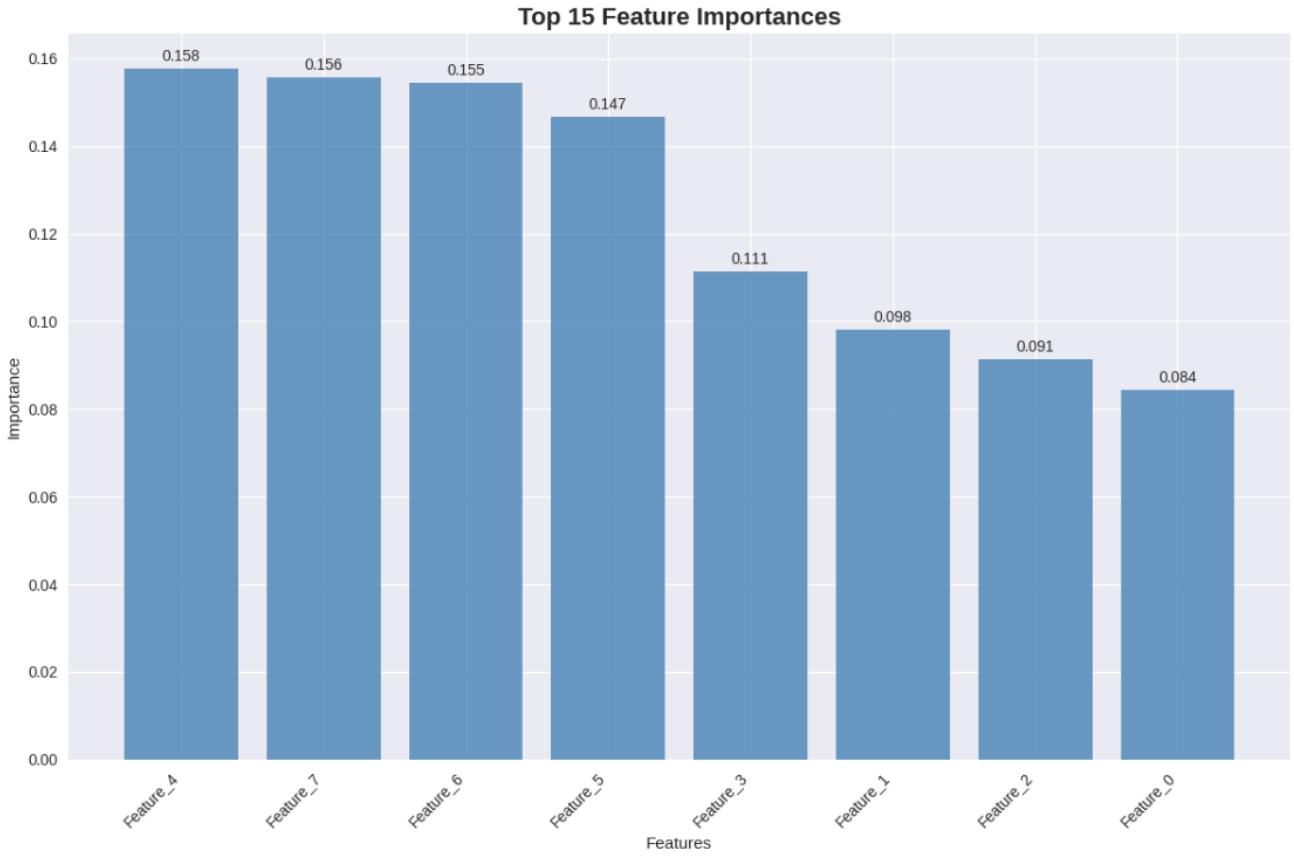

```
🔧 STEP 3: Data Preprocessing

==================================================
🔧 DATA PREPROCESSING
==================================================
📊 Processing 6 numeric and 2 categorical features
🏗️ Building preprocessing pipeline...
⚙️ Fitting preprocessing pipeline...
🔄 Transforming data...
✅ Preprocessing completed in 3.89s
📊 Processed 50,000 rows
🎯 Final feature vector size: 8

🚂 STEP 4: Classification Models

==================================================
🚂 CLASSIFICATION MODEL TRAINING
==================================================
📁 Splitting data...
📊 Training set: 40,144 rows
📊 Test set: 9,856 rows

🎯 Training Logistic Regression...
  ⏱️ Training time: 2.86s
  📈 AUC: 0.5024
  🎯 Accuracy: 0.6059

🌲 Training Random Forest...
  ⏱️ Training time: 21.13s
  📈 AUC: 0.4981
  🎯 Accuracy: 0.6054

🚀 Training Gradient Boosting...
  ⏱️ Training time: 29.14s
  📈 AUC: 0.4914
  🎯 Accuracy: 0.6062
```

```
📈 STEP 5: Regression Models

==================================================
📈 REGRESSION MODEL TRAINING
==================================================

🔪 Training Linear Regression...
  ⏱️ Training time: 0.62s
  📊 RMSE: 303.3880
  📈 R²: -0.0006

🌲 Training Random Forest Regression...
  ⏱️ Training time: 19.59s
  📊 RMSE: 303.7761
  📈 R²: -0.0031

🎯 STEP 6: Clustering Analysis

==================================================
🎯 CLUSTERING ANALYSIS
==================================================

🔍 K-Means clustering with k=3...
  ⏱️ Training time: 3.31s
  📊 WSSSE: 376547.4455
  🎯 Cluster distribution:
+-------+-----+
|cluster|count|
+-------+-----+
|      0| 8340|
|      1|19911|
|      2|21749|
+-------+-----+


🔍 K-Means clustering with k=5...
  ⏱️ Training time: 2.94s
  📊 WSSSE: 294193.5489
  🎯 Cluster distribution:
+-------+-----+
|cluster|count|
+-------+-----+
|      0| 5848|
|      1| 8381|
|      2| 8106|
|      3|15070|
|      4|12595|
+-------+-----+
```

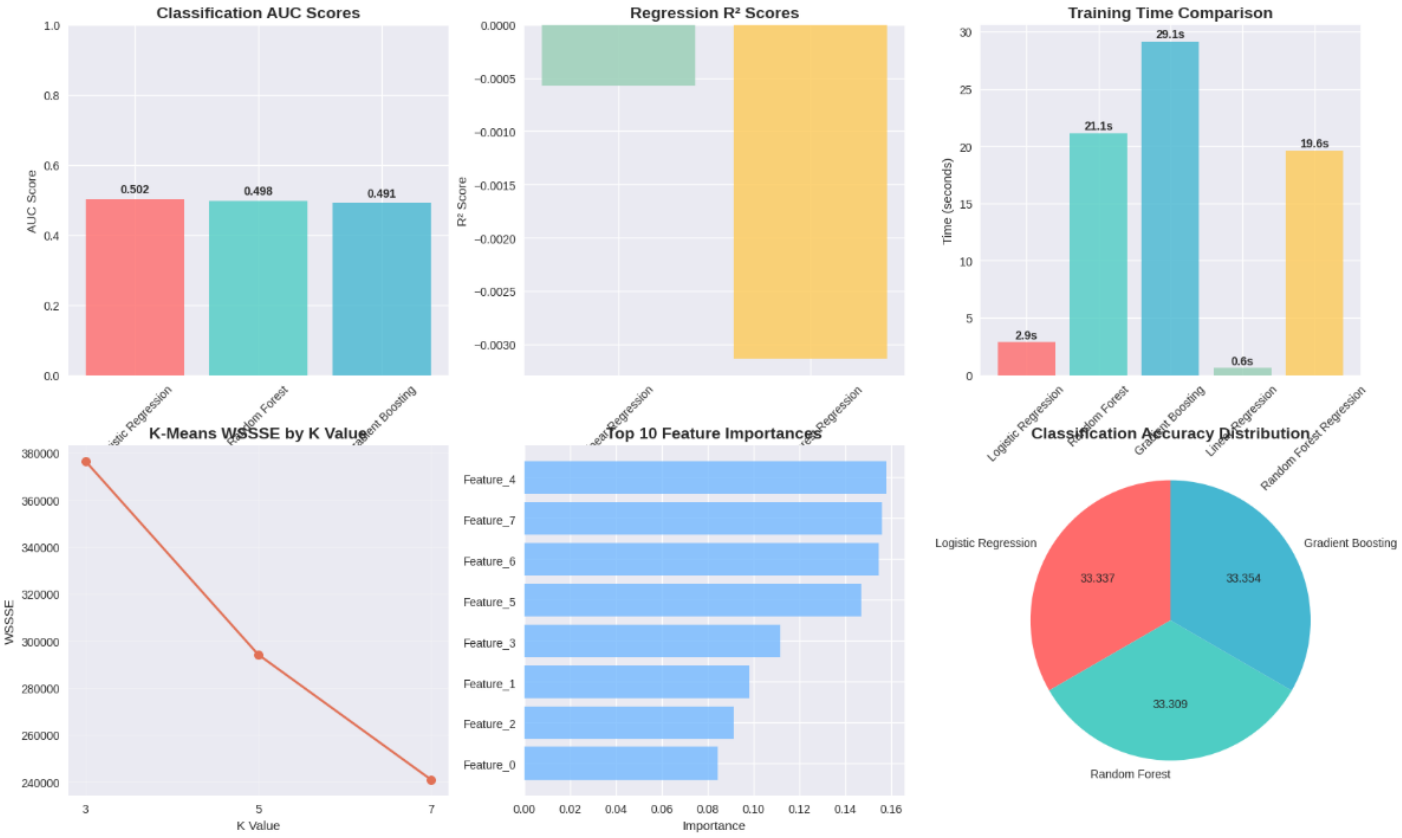
```
🔍 K-Means clustering with k=7...
  ⏱️ Training time: 2.27s
  📊 WSSSE: 240958.8889
  🎯 Cluster distribution:
+-------+-----+
|cluster|count|
+-------+-----+
|      0| 3801|
|      1| 9154|
|      2| 8794|
|      3| 5848|
|      4| 8381|
|      5| 8106|
|      6| 5916|
+-------+-----+


🔍 STEP 7: Feature Importance

==================================================
🔍 FEATURE IMPORTANCE ANALYSIS
==================================================
🏆 Top 10 Most Important Features:
   Feature  Importance
Feature_4    0.157762
Feature_7    0.155800
Feature_6    0.154551
Feature_5    0.146712
Feature_3    0.111448
Feature_1    0.098103
Feature_2    0.091304
Feature_0    0.084320
```

```
🎨 Creating results visualization...
```






```
🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉
PIPELINE COMPLETED SUCCESSFULLY!
🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉
⏱️ Total execution time: 94.73 seconds
📊 Dataset processed: 50,000 rows
🤖 Models trained: 5 ML models
🎯 Clustering algorithms: 3

==================================================
📊 MODEL PERFORMANCE DASHBOARD
==================================================
```




```
📈 PERFORMANCE SUMMARY:
----------------------------------------
🏆 Best Classifier: Logistic Regression (AUC: 0.5024)
📈 Best Regressor: Linear Regression (R²: -0.0006)
⚡ Fastest Training: Logistic Regression (2.86s)
⏱️ Total Pipeline Time: 94.73s
📁 Google Drive not mounted. To save results:
 1. Run: from google.colab import drive
 2. Run: drive.mount('/content/drive')

🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉
COLAB SPARK ML PIPELINE COMPLETED!
🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉
📊 Processed 50,000 rows
🤖 Trained 5 models
⏱️ Total time: 94.73 seconds
🔗 Spark UI: http://5a7456b12781:4040
```

Learning Outcomes:

1. Understand the architecture and core components of Apache Spark and its capabilities in distributed data processing.
2. Gain practical experience in preprocessing large datasets, feature engineering, and building scalable machine learning models using Spark MLlib.
3. Develop the ability to evaluate model performance, interpret results, and visualize insights from big data efficiently.

EXPERIMENT-5

Aim: To implement and train sequence-to-sequence models for language translation tasks using attention mechanisms.

Objectives:

- To implement and enhance Seq2Seq models with attention for improved neural machine translation.
- To train and evaluate translation models on bilingual datasets for better translation quality..

Theory:

Sequence-to-Sequence Models

Sequence-to-sequence (Seq2Seq) models are a class of neural network architectures developed to handle tasks where both input and output are sequences of variable lengths. These tasks include:

- Machine translation (e.g., English to French sentence translation)
- Text summarization (e.g., converting long documents into concise summaries)
- Speech recognition (e.g., converting audio signals into text)

A basic Seq2Seq model consists of two main components:

- **Encoder:** Reads and processes the input sequence and encodes it into a fixed-length context vector.
- **Decoder:** Generates the output sequence step by step, using the context vector as input.

While effective for short sequences, this approach struggles with **long or complex sentences** because compressing all information into a single fixed-length vector often causes loss of important context and semantic details.

Need for Attention Mechanism

To overcome the limitations of the fixed-length context, attention mechanisms were introduced. Attention enables the decoder to focus on specific parts of the input sequence during each step of output generation. By calculating alignment scores between the encoder outputs and the decoder states, the model creates a dynamic context vector that captures relevant information, improving accuracy for longer and more complex inputs.

Impact on Neural Machine Translation

In neural machine translation, attention-based Seq2Seq models achieve superior results compared to traditional statistical models and basic Seq2Seq frameworks. They produce more fluent, contextually accurate translations and serve as the foundation for advanced architectures like the Transformer. Consequently, attention-integrated Seq2Seq models mark a significant advancement in natural language processing, combining effectiveness and quality in sequence modeling.

Code and Output:

```
import os
import math
import time
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from datasets import load_dataset
import sentencepiece as spm
import numpy as np
from pathlib import Path
import sacrebleu

DEVICE = torch.device("cuda" if torch.cuda.is  available() else "cpu")
SEED = 42
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED);
if DEVICE.type == "cuda": torch.cuda.manual  seed  all(SEED)
print("Device:", DEVICE)
```

```
•••  Device: cuda
```

```
lang  pair = "en-fr"
src  lang, tgt  lang = lang  pair.split("-")

ds = load_dataset("opus_books", lang_pair, split={"train":"train[:95%]", "validation":"train[95%:]"})
print(ds)
print("Samples:", len(ds["train"]), "/", len(ds["validation"]))
```

```
DatasetDict({
    train: Dataset({
        features: ['id', 'translation'],
        num_rows: 120731
    })
    validation: Dataset({
        features: ['id', 'translation'],
        num_rows: 6354
    })
})
Samples: 120731 / 6354
```

```
for k in range(2):
  ex = ds["train"][k]
  print(f"\nSRC: {ex['translation'][src  lang]}\nTGT: {ex['translation'][tgt  lang]}")
```

```
SRC: The Wanderer
TGT: Le grand Meaulnes

SRC: Alain-Fournier
TGT: Alain-Fournier
```

```
sp_dir = Path("spm")
sp_dir.mkdir(exist_ok=True)

src_corpus_file = sp_dir / f"train.{src_lang}.txt"
tgt_corpus_file = sp_dir / f"train.{tgt_lang}.txt"

with open(src_corpus_file, "w", encoding="utf-8") as fs, open(tgt_corpus_file, "w", encoding="utf-8") as ft:
    for r in ds["train"]:
        fs.write(r["translation"][src_lang].strip().replace("\n"," ") + "\n")
        ft.write(r["translation"][tgt_lang].strip().replace("\n"," ") + "\n")

PAD, BOS, EOS, UNK = "<pad>", "<s>", "</s>", "<unk>"

def train_spm(input_path, model_prefix, vocab_size=8000):
    spm.SentencePieceTrainer.Train(
        input=str(input_path),
        model_prefix=str(model_prefix),
        vocab_size=vocab_size,
        character_coverage=1.0,
        model_type="bpe",
        input_sentence_size=200000,
        shuffle_input_sentence=True,
        bos_id=1, eos_id=2, pad_id=0, unk_id=3,
        bos_piece=BOS, eos_piece=EOS, pad_piece=PAD, unk_piece=UNK
    )

src_model_prefix = sp_dir / f"spm_{src_lang}"
tgt_model_prefix = sp_dir / f"spm_{tgt_lang}"

if not (sp_dir / f"spm_{src_lang}.model").exists():
    train_spm(src_corpus_file, src_model_prefix, vocab_size=8000)
if not (sp_dir / f"spm_{tgt_lang}.model").exists():
    train_spm(tgt_corpus_file, tgt_model_prefix, vocab_size=8000)

sp_src = spm.SentencePieceProcessor(model_file=str(sp_dir / f"spm_{src_lang}.model"))
sp_tgt = spm.SentencePieceProcessor(model_file=str(sp_dir / f"spm_{tgt_lang}.model"))

SRC_PAD_ID = sp_src.pad_id()
SRC_BOS_ID = sp_src.bos_id()
SRC_EOS_ID = sp_src.eos_id()
TGT_PAD_ID = sp_tgt.pad_id()
TGT_BOS_ID = sp_tgt.bos_id()
TGT_EOS_ID = sp_tgt.eos_id()
len_src_vocab = sp_src.vocab_size()
```

```
len_tgt_vocab = sp_tgt.vocab_size()
print("Vocab sizes:", len_src_vocab, len_tgt_vocab)
```

```
Vocab sizes: 8000 8000
```

```
MAX_LEN = 128
def encode_src(text):
    ids = sp_src.encode(text, out_type=int, add_bos=True, add_eos=True)
    return ids[:MAX_LEN]

def encode_tgt(text):
    ids = sp_tgt.encode(text, out_type=int, add_bos=True, add_eos=True)
    return ids[:MAX_LEN]

class MTDataset(Dataset):
    def __init__(self, hf_split):
        self.data = hf_split
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        pair = self.data[idx]["translation"]
        src_ids = encode_src(pair[src_lang])

 tgt_ids = encode_tgt(pair[tgt_lang])
        return torch.tensor(src_ids, dtype=torch.long), torch.tensor(tgt_ids, dtype=torch.long)

def pad_batch(batch, pad_id_src=SRC_PAD_ID, pad_id_tgt=TGT_PAD_ID):
    src_seqs, tgt_seqs = zip(*batch)
    src_lens = [len(x) for x in src_seqs]
    tgt_lens = [len(x) for x in tgt_seqs]
    max_src = max(src_lens)
    max_tgt = max(tgt_lens)
    src_padded = torch.full((len(batch), max_src), pad_id_src, dtype=torch.long)
    tgt_padded = torch.full((len(batch), max_tgt), pad_id_tgt, dtype=torch.long)
    for i,(s,t) in enumerate(zip(src_seqs,tgt_seqs)):
        src_padded[i,:len(s)] = s
        tgt_padded[i,:len(t)] = t
    return src_padded, tgt_padded

train_data = MTDataset(ds["train"])
val_data   = MTDataset(ds["validation"])
BATCH_SIZE = 64
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True, collate_fn=pad_batch, drop_last=False)
val_loader   = DataLoader(val_data, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_batch, drop_last=False)
print("Train/Val sizes:", len(train_data), len(val_data))
```

```
Train/Val sizes: 120731 6354
```

```
class Encoder(nn.Module):
```

```
    def __init__(self, vocab_size, emb_dim, hid_dim, num_layers=1, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_dim, padding_idx=SRC_PAD_ID)
        self.rnn = nn.GRU(emb_dim, hid_dim, num_layers=num_layers, batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hid_dim*2, hid_dim)  # project biGRU -> decoder hidden dim
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask=None):
        emb = self.dropout(self.embedding(src))
        outputs, hidden = self.rnn(emb)
        h_cat = torch.cat([hidden[-2], hidden[-1]], dim=1)
        h0 = torch.tanh(self.fc(h_cat)).unsqueeze(0)
        return outputs, h0

class BahdanauAttention(nn.Module):
    def    init    (self, enc  hid  dim, dec  hid  dim):
        super().    init    ()

        self.W1 = nn.Linear(enc_hid_dim*2, dec_hid_dim)
        self.W2 = nn.Linear(dec  hid  dim, dec  hid  dim)
        self.v  = nn.Linear(dec  hid  dim, 1, bias=False)

    def forward(self, enc  outputs, dec  hidden, src  mask=None):
        dec  hidden = dec  hidden.transpose(0,1)

 scores = self.v(torch.tanh(self.W1(enc_outputs) + self.W2(dec_hidden)))
        scores = scores.squeeze(-1)  # (B,S)
        if src  mask is not None:
            scores = scores.masked  fill(~src  mask, -1e4)
        attn = torch.softmax(scores, dim=-1)    # (B,S)
        context = torch.bmm(attn.unsqueeze(1), enc  outputs).squeeze(1)
        return context, attn

class Decoder(nn.Module):
    def    init    (self, vocab  size, emb  dim, enc  hid  dim, dec  hid  dim, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, emb_dim, padding_idx=TGT_PAD_ID)
        self.attn = BahdanauAttention(enc_hid_dim, dec_hid_dim)
        self.rnn = nn.GRU(emb_dim + enc_hid_dim*2, dec_hid_dim, batch_first=True)
        self.fc_out = nn.Linear(dec_hid_dim + enc_hid_dim*2 + emb_dim, vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input_tok, hidden, enc_outputs, src_mask=None):
        emb = self.dropout(self.embedding(input_tok)).unsqueeze(1)
        context, attn = self.attn(enc_outputs, hidden, src_mask)
        rnn_input = torch.cat([emb, context.unsqueeze(1)], dim=-1)
        output, hidden = self.rnn(rnn_input, hidden)
 logits = self.fc_out(torch.cat([output.squeeze(1), context, emb.squeeze(1)], dim=-1))
        return logits, hidden, attn
```

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, src_pad_id=SRC_PAD_ID, tgt_pad_id=TGT_PAD_ID):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_id = src_pad_id
        self.tgt_pad_id = tgt_pad_id

    def make_src_mask(self, src):
        return (src != self.src  pad  id)
    def forward(self, src, tgt, teacher_forcing_ratio=0.5):
        batch_size, tgt_len = tgt.size()
        src_mask = self.make_src_mask(src)
        enc_outputs, hidden = self.encoder(src, src_mask)
        outputs = []
        inp = tgt[:,0]
        for t in range(1, tgt  len):
            logits, hidden, _ = self.decoder(inp, hidden, enc_outputs, src_mask)
            outputs.append(logits.unsqueeze(1))
            teacher = (random.random() < teacher  forcing  ratio)

     next_tok = tgt[:,t] if teacher else torch.argmax(logits, dim=-1)
            inp = next  tok
        return torch.cat(outputs, dim=1)

EMB_DIM = 256
HID  DIM = 512
ENC  LAYERS = 1
LR = 3e-4
EPOCHS = 8
CLIP = 1.0
USE  AMP = True

encoder = Encoder(len_src_vocab, EMB_DIM, HID_DIM, num_layers=ENC_LAYERS, dropout=0.2)
decoder = Decoder(len  tgt  vocab, EMB  DIM, HID  DIM, HID  DIM, dropout=0.2)
model = Seq2Seq(encoder, decoder).to(DEVICE)
optimizer = torch.optim.AdamW(model.parameters(), lr=LR)
criterion = nn.CrossEntropyLoss(ignore_index=TGT_PAD_ID)
scaler = torch.cuda.amp.GradScaler(enabled=(USE_AMP and DEVICE.type=="cuda"))

def train_epoch(model, loader):
    model.train()
    total_loss, total_tok = 0.0, 0
    for src, tgt in loader:
        src, tgt = src.to(DEVICE), tgt.to(DEVICE)
        optimizer.zero_grad(set_to_none=True)
        with torch.cuda.amp.autocast(enabled=(USE_AMP and DEVICE.type=="cuda")):
            logits = model(src, tgt, teacher_forcing_ratio=0.5)
            gold = tgt[:,1:].contiguous()
```

```
        loss = criterion(logits.reshape(-1, logits.size(-1)), gold.reshape(-1))
        scaler.scale(loss).backward()
        if (USE_AMP and DEVICE.type=="cuda"):
            scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), CLIP)
        scaler.step(optimizer)
        scaler.update()
        ntokens = (gold != TGT_PAD_ID).sum().item()
        total_loss += loss.item() * ntokens
        total  tok += ntokens
    return total_loss / max(1,total_tok)

@torch.no_grad()
def greedy_decode(model, src, max_len=100):

    src  mask = model.make  src  mask(src)
    enc  outputs, hidden = model.encoder(src, src  mask)
    B = src.size(0)
    inp = torch.full((B,), TGT_BOS_ID, dtype=torch.long, device=src.device)
    finished = torch.zeros(B, dtype=torch.bool, device=src.device)

   outs = [[] for _ in range(B)]
    for    in range(max  len):
        logits, hidden,    = model.decoder(inp, hidden, enc  outputs, src  mask)
        next  tok = torch.argmax(logits, dim=-1)
        inp = next_tok
        for i, tok in enumerate(next  tok.tolist()):
            if not finished[i]:
                if tok == TGT  EOS  ID:
                    finished[i] = True
                else:
                    outs[i].append(tok)
        if finished.all(): break
    return outs

@torch.no_grad()
def evaluate_bleu(model, loader, max_len=100):
    model.eval()
    sys_outputs, refs = [], []
    for src, tgt in loader:
        src = src.to(DEVICE)
        preds = greedy_decode(model, src, max_len=max_len)
        for i in range(len(preds)):
            hyp = sp_tgt.decode(preds[i])
            tgt_i = tgt[i].tolist()

            if TGT_BOS_ID in tgt_i:
                tgt_i = tgt_i[1:]
            if TGT_EOS_ID in tgt_i:
```

```
            eos_pos = tgt_i.index(TGT_EOS_ID)
            tgt_i = tgt_i[:eos_pos]
        ref = sp_tgt.decode(tgt_i)
        sys_outputs.append(hyp)
        refs.append(ref)
    bleu = sacrebleu.corpus_bleu(sys_outputs, [refs]).score
    return bleu

ckpt_dir = Path("checkpoints"); ckpt_dir.mkdir(exist_ok=True)

def load_checkpoint(path="checkpoints/best_seq2seq_attn.pt", model=model):
    state = torch.load(path, map_location=DEVICE)
    model.load_state_dict(state["model"])
    model.to(DEVICE).eval()
    print("Checkpoint loaded. Ready to translate.")
    return model

@torch.no_grad()
def translate(sentences, max_len=80):
    if isinstance(sentences, str):
        sentences = [sentences]
    src_batch = []
    for s in sentences:
        ids = encode  src(s)
        src  batch.append(torch.tensor(ids, dtype=torch.long))
    maxS = max(len(x) for x in src_batch)
    pad  src = torch.full((len(src  batch), maxS), SRC  PAD  ID, dtype=torch.long)
    for i, s in enumerate(src  batch):
        pad  src[i,:len(s)] = s
    pad_src = pad_src.to(DEVICE)
    pred  ids = greedy  decode(model, pad  src, max  len=max  len)
    return [sp  tgt.decode(p) for p in pred  ids]

best_bleu = -1.0
try:
    for epoch in range(1, EPOCHS+1):
        t0 = time.time()
        train_loss = train_epoch(model, train_loader)
        val_bleu = evaluate_bleu(model, val_loader, max_len=80)
        dt = time.time() - t0
        print(f"Epoch {epoch:02d} | train NLL/token: {train_loss:.6f} | val BLEU: {val_bleu:5.2f} | {dt:.1f}s")
        if val_bleu > best_bleu:
            best_bleu = val_bleu
            torch.save({
                "model": model.state_dict(),
                "src_spm": str(sp_dir / f"spm_{src_lang}.model"),
                "tgt_spm": str(sp_dir / f"spm_{tgt_lang}.model"),
                "cfg": {
                    "EMB_DIM": EMB_DIM, "HID_DIM": HID_DIM
                }
```

```
        }, ckpt_dir / "best_seq2seq_attn.pt")
        print("  ✓ Saved new best checkpoint.")
```

```
Epoch 01 | train NLL/token: 5.591140 | val BLEU:  4.05 | 1007.5s
  ✓ Saved new best checkpoint.
Epoch 02 | train NLL/token: 4.733817 | val BLEU:  6.08 | 1012.3s
  ✓ Saved new best checkpoint.
Epoch 03 | train NLL/token: 4.355662 | val BLEU:  8.09 | 1009.1s
  ✓ Saved new best checkpoint.
Epoch 04 | train NLL/token: 4.139473 | val BLEU:  8.58 | 1007.4s
  ✓ Saved new best checkpoint.
Epoch 05 | train NLL/token: 3.969395 | val BLEU:  9.80 | 1006.7s
  ✓ Saved new best checkpoint.
Epoch 06 | train NLL/token: 3.836784 | val BLEU: 10.21 | 1009.5s
  ✓ Saved new best checkpoint.
Epoch 07 | train NLL/token: 3.715976 | val BLEU: 10.54 | 1005.1s
  ✓ Saved new best checkpoint.
Epoch 08 | train NLL/token: 3.633393 | val BLEU: 11.04 | 1005.1s
  ✓ Saved new best checkpoint.
```

```
except RuntimeError as e:
    print("RuntimeError during training:", e)
    print("Possible OOM. Try reducing BATCH_SIZE or MAX_LEN, or set USE_AMP=False if on CPU.")
except Exception as e:
    print("Unexpected error:", e)

samples = [
    "I will meet you tomorrow at the library.",
    "This book is very interesting and easy to read."
]
print("\nTranslations:")
for s, t in zip(samples, translate(samples)):
    print(f"{s}\n -> {t}\n")
```

```
Translations:
I will meet you tomorrow at the library.
 -> Je vousrai demain la bibliothèque.

This book is very interesting and easy to read.
 -> livre livre est intéressante et et facile.
```

Learning Outcomes:

1. To understand the architecture and working of sequence-to-sequence (Seq2Seq) models used for handling sequential data like translation and summarization.
2. To explore and implement attention mechanisms that allow models to focus on relevant parts of input sequences for improved translation accuracy.
3. To train, test, and evaluate attention-based Seq2Seq models on bilingual datasets to analyze their performance and translation quality improvements.

EXPERIMENT-6

Aim: To apply various anomaly detection techniques on time series data and evaluate their effectiveness.

Objectives:

- To understand and implement various anomaly detection techniques in time series data, including Z-score, Isolation Forest, and One-Class SVM.
- To evaluate and compare the performance of these techniques using visualization and standard metrics such as Precision, Recall, and F1-score.

Theory:

Anomaly Detection

Anomaly detection, or outlier detection, is the process of identifying data points or patterns that deviate from expected behavior within a dataset. In time series data, anomalies often represent unusual events like equipment failures, network intrusions, or medical emergencies.

Types of Anomalies

- **Point Anomalies:** A single data point differs significantly from the rest.
- **Contextual Anomalies:** A point is abnormal only under certain conditions (e.g., high temperature in winter).
- **Collective Anomalies:** A group of data points collectively shows abnormal behavior.

Techniques Used

- **Z-Score Method:** A statistical technique based on mean and standard deviation.

$$Z = (x - \mu)$$

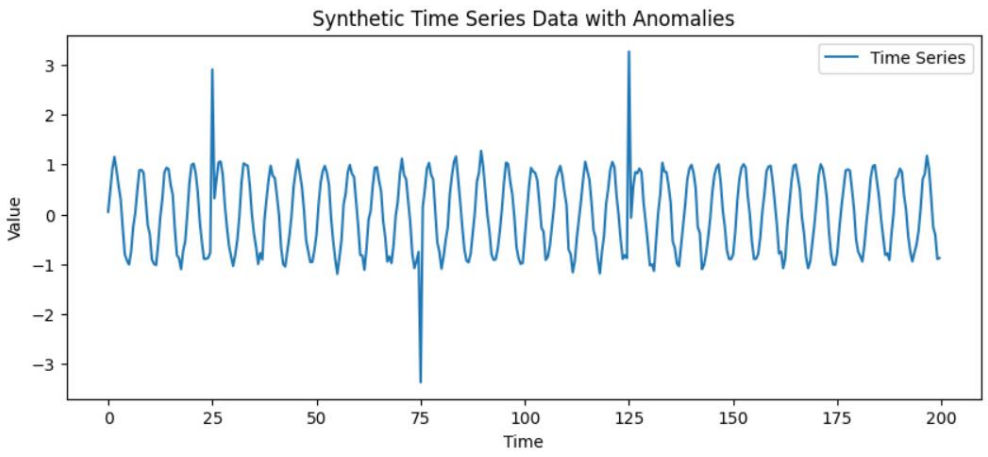
  If |Z| exceeds a threshold (e.g., 3), the point is identified as an anomaly.
- **Isolation Forest:** A tree-based ensemble method that isolates anomalies by random partitioning.
  Since anomalies are rare and distinct, they require fewer splits to be isolated.
- **One-Class SVM:** A boundary-based machine learning method that defines the region containing most data points and flags those outside it as anomalies.

Evaluation Metrics

- **Precision:** Proportion of correctly identified anomalies among predicted ones.
- **Recall:** Proportion of correctly detected anomalies among actual anomalies.
- **F1-Score:** Harmonic mean of precision and recall.

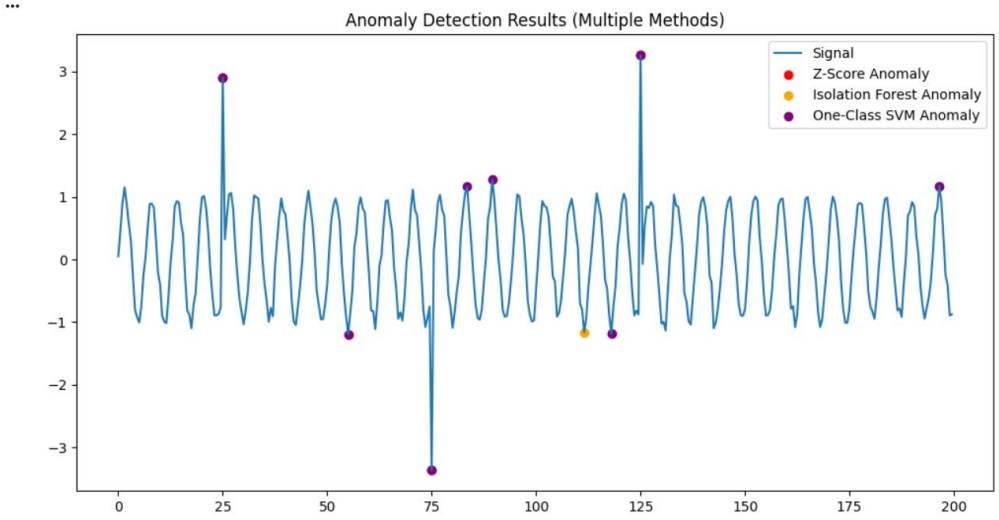
Code and Output:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import precision_score, recall_score, f1_score
np.random.seed(42)
time = np.arange(0, 200, 0.5)
signal = np.sin(time) + np.random.normal(0, 0.1, len(time))

signal[50] += 3
signal[150] -= 3
signal[250] += 4

data = pd.DataFrame({"time": time, "value": signal})
plt.figure(figsize=(10,4))
plt.plot(data["time"], data["value"], label="Time Series")
plt.title("Synthetic Time Series Data with Anomalies")
plt.xlabel("Time")
plt.ylabel("Value")
plt.legend()
plt.show()
```




```
true_anomalies = np.zeros(len(signal))
true_anomalies[[50,150,250]] = 1

z_scores = np.abs((signal - np.mean(signal)) / np.std(signal))
threshold = 3
z_pred = (z_scores > threshold).astype(int)

scaler = StandardScaler()
scaled_signal = scaler.fit_transform(signal.reshape(-1,1))
iso_forest = IsolationForest(contamination=0.02, random_state=42)
iso_pred = (iso_forest.fit_predict(scaled_signal) == -1).astype(int)
```

```
ocsvm = OneClassSVM(kernel='rbf', nu=0.02, gamma=0.1)
svm_pred = (ocsvm.fit_predict(scaled_signal) == -1).astype(int)
plt.figure(figsize=(12,6))
plt.plot(time, signal, label="Signal")
plt.scatter(time[z_pred==1], signal[z_pred==1], color='red', label='Z-Score
Anomaly')
plt.scatter(time[iso_pred==1], signal[iso_pred==1], color='orange',
label='Isolation Forest Anomaly')
plt.scatter(time[svm_pred==1], signal[svm_pred==1], color='purple', label='One-
Class SVM Anomaly')
plt.legend()
plt.title("Anomaly Detection Results (Multiple Methods)")
plt.show()
```




```
def evaluate(y_true, y_pred, name):
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    print(f"{name} -> Precision: {precision:.2f}, Recall: {recall:.2f}, F1-Score:
{f1:.2f}")


print("\n--- Evaluation Metrics ---")
evaluate(true_anomalies, z_pred, "Z-Score")
evaluate(true_anomalies, iso_pred, "Isolation Forest")
evaluate(true_anomalies, svm_pred, "One-Class SVM")
```

```
--- Evaluation Metrics ---
Z-Score -> Precision: 1.00, Recall: 1.00, F1-Score: 1.00
Isolation Forest -> Precision: 0.38, Recall: 1.00, F1-Score: 0.55
One-Class SVM -> Precision: 0.38, Recall: 1.00, F1-Score: 0.55
```

Learning Outcomes:

1. To understand the concept and types of anomalies in time series data and their real-world significance.
2. To apply different anomaly detection techniques such as Z-score, Isolation Forest, and One-Class SVM.
3. To evaluate and compare the performance of these techniques using metrics like Precision, Recall, and F1-score.

EXPERIMENT-7

Aim: To assess and mitigate bias in machine learning models using Fairness Indicators and AIF360.

Objectives:

- To understand and evaluate bias in machine learning models using fairness metrics and toolkits like AIF360 and Fairness Indicators.
- To apply and compare the effectiveness of the Reweighing technique for bias mitigation by analyzing model fairness before and after its application.

Theory:

Bias in Machine Learning

Bias in machine learning refers to systematic errors in model predictions that unfairly favor or disadvantage specific groups. It often results from imbalanced datasets, biased labeling, or historical inequalities present in the data. For instance, a hiring model trained on past biased data may prefer one gender over another due to unequal representation.

Fairness in Machine Learning

Fairness ensures that a model's predictions are equitable across groups defined by sensitive attributes like gender, race, or age. A fair model should generate consistent predictions for individuals from different demographic groups under similar circumstances.

Fairness Indicators

Fairness Indicators are quantitative metrics used to assess how fair a machine learning model is. Common fairness metrics include:

Metric	Description	Ideal Value
Disparate Impact (DI)	Ratio of favorable outcomes between unprivileged and privileged groups.	≈ 1.0
Statistical Parity Difference (SPD)	Difference in favorable outcome rates between groups.	≈ 0
Equal Opportunity Difference (EOD)	Difference in true positive rates between groups.	≈ 0
Average Odds Difference (AOD)	Average of differences in true and false positive rates between groups.	≈ 0

AIF360 (AI Fairness 360)

AIF360, developed by IBM, is an open-source toolkit designed to detect, measure, and mitigate bias in machine learning models. It allows users to:

- Detect bias using various fairness metrics
- Apply bias mitigation algorithms
- Evaluate fairness improvements after mitigation

It supports preprocessing (e.g., Reweighing), in-processing, and post-processing bias mitigation techniques.

Bias Mitigation Technique: Reweighing

The Reweighing algorithm minimizes bias by adjusting the weights of training samples so that privileged and unprivileged groups influence the model equally. The weight for each sample is computed as:

$$w_{ij} = P(Y = y_j) / P(A = a_i, Y = y_j)$$

Where:

- **A** = protected attribute (e.g., gender)
- **Y** = target variable
- **wij** = reweighting factor for group *i* and class *j*

This approach helps balance the dataset and ensures fairer model training by reducing the impact of biased data distributions.

Code and Output:

```
import os
target_dir = '/usr/local/lib/python3.12/dist-packages/aif360/data/raw/adult'
os.makedirs(target_dir, exist_ok=True)
!wget -P {target_dir} https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data
!wget -P {target_dir} https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.test
!wget -P {target_dir} https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names

print(f"Downloaded files to {target_dir}")
```

```
...  Downloaded files to /usr/local/lib/python3.12/dist-packages/aif360/data/raw/adult
```

```
!ls {target_dir}
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.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
from aif360.datasets import AdultDataset
from aif360.metrics import BinaryLabelDatasetMetric, ClassificationMetric
from aif360.algorithms.preprocessing import Reweighing

dataset = AdultDataset()
```

```
privileged_groups = [{'sex': 1}]
unprivileged_groups = [{'sex': 0}]

train, test = dataset.split([0.7], shuffle=True)
X_train, y_train = train.features, train.labels.ravel()
X_test, y_test = test.features, test.labels.ravel()

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X  test  scaled = scaler.transform(X  test)
model = LogisticRegression(max_iter=1000)
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)

acc = accuracy_score(y_test, y_pred)
print(f"Baseline Accuracy: {acc:.2f}")
```

```
Baseline Accuracy: 0.85
```

```
test  pred = test.copy()
test  pred.labels = y  pred.reshape(-1,1)

metric = ClassificationMetric(

test, test  pred,
   privileged  groups=privileged  groups,
   unprivileged_groups=unprivileged_groups
)
print("\n--- Fairness Metrics (Before Mitigation) ---")
print(f"Disparate Impact: {metric.disparate  impact():.3f}")
print(f"Statistical Parity Difference: {metric.statistical_parity_difference():.3f}")
print(f"Equal Opportunity Difference: {metric.equal  opportunity  difference():.3f}")
print(f"Average Odds Difference: {metric.average  odds  difference():.3f}")
```

```
--- Fairness Metrics (Before Mitigation) ---
Disparate Impact: 0.301
Statistical Parity Difference: -0.186
Equal Opportunity Difference: -0.084
Average Odds Difference: -0.082
```

```
RW = Reweighing(unprivileged_groups=unprivileged_groups, privileged_groups=privileged_groups)
train_transf = RW.fit_transform(train)
model_rw = LogisticRegression(max_iter=1000)
model_rw.fit(X_train_scaled, y_train, sample_weight=train_transf.instance_weights)
y_pred_rw = model_rw.predict(X_test_scaled)
acc_rw = accuracy_score(y_test, y_pred_rw)
print(f"\nAccuracy After Reweighing: {acc_rw:.2f}")
```

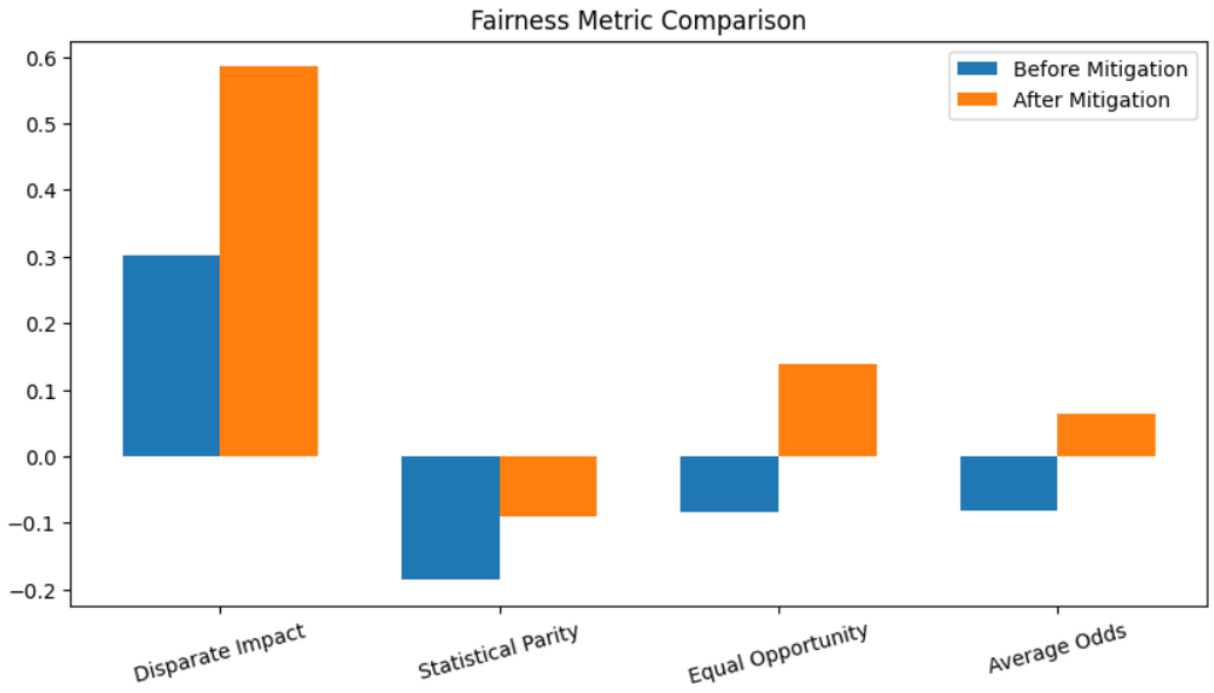
```
Accuracy After Reweighing: 0.84
```

```
test_pred_rw = test.copy()
test_pred_rw.labels = y_pred_rw.reshape(-1,1)
metric_rw = ClassificationMetric(
    test, test_pred_rw,
    privileged_groups=privileged_groups,
    unprivileged_groups=unprivileged_groups
)
print("\n--- Fairness Metrics (After Reweighing) ---")
print(f"Disparate Impact: {metric_rw.disparate_impact():.3f}")
print(f"Statistical Parity Difference: {metric_rw.statistical_parity_difference():.3f}")
print(f"Equal Opportunity Difference: {metric_rw.equal_opportunity_difference():.3f}")
print(f"Average Odds Difference: {metric_rw.average_odds_difference():.3f}")
```

```
--- Fairness Metrics (After Reweighing) ---
Disparate Impact: 0.585
Statistical Parity Difference: -0.091
Equal Opportunity Difference: 0.140
Average Odds Difference: 0.064
```

```
metrics_before = [
    metric.disparate_impact(),
    metric.statistical_parity_difference(),
    metric.equal_opportunity_difference(),
    metric.average_odds_difference()
]
metrics_after = [
    metric_rw.disparate_impact(),
    metric_rw.statistical_parity_difference(),
    metric_rw.equal_opportunity_difference(),
    metric_rw.average_odds_difference()
]
labels = ['Disparate Impact', 'Statistical Parity', 'Equal Opportunity', 'Average Odds']
x = np.arange(len(labels))
width = 0.35

plt.figure(figsize=(10,5))
plt.bar(x - width/2, metrics_before, width, label='Before Mitigation')
plt.bar(x + width/2, metrics_after, width, label='After Mitigation')
plt.xticks(x, labels, rotation=15)
plt.title('Fairness Metric Comparison')
plt.legend()
plt.show()
```

...



Learning Outcomes:

1. To understand the concept of bias and fairness in machine learning models and their impact on decision-making.

2. To evaluate model fairness using metrics such as Disparate Impact, Statistical Parity Difference, Equal Opportunity Difference, and Average Odds Difference.

3. To apply the AIF360 toolkit and implement the Reweighing technique to detect and mitigate bias in machine learning models.

EXPERIMENT-8

Aim: To interpret and explain the predictions of complex machine learning models using LIME and SHAP techniques.

Objectives:

- To learn and implement model-agnostic interpretation methods such as LIME and SHAP for explaining predictions.
- To analyze and visualize the contribution of individual features toward overall model outputs..

Theory:

Machine learning models like Random Forests, Gradient Boosting, and Deep Neural Networks often function as black boxes, making it difficult to understand how they reach predictions. Explainable AI (XAI) techniques, such as LIME (Local Interpretable Model-agnostic Explanations) and SHAP (SHapley Additive exPlanations), are used to make these models more transparent, trustworthy, and fair.

LIME (Local Interpretable Model-agnostic Explanations):

- Explains predictions for individual instances by approximating the complex model locally with a simple, interpretable model like linear regression.
- Perturbs the input data slightly and observes changes in predictions to determine the importance of each feature for that instance.
- Useful for understanding local behavior of the model and identifying potential biases or errors in specific predictions.

SHAP (SHapley Additive exPlanations):

- Based on cooperative game theory, treating each feature as a "player" contributing to the model's output.
- Computes the average marginal contribution of each feature across all possible feature combinations, providing a fair and consistent measure of importance.
- Helps explain both individual predictions and overall model behavior, making it easier to debug models and detect bias.

By using LIME and SHAP, practitioners can visualize how features impact predictions, gain insights into model decision-making, and increase confidence in AI systems, especially for high-stakes applications like healthcare, finance, and hiring.

Code and Output:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import shap
import lime
import lime.lime_tabular
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, classification_report
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target)
print("Features shape:", X.shape)
print("Target distribution:\n", y.value_counts())
```

```
Features shape: (569, 30)
Target distribution:
 1    357
0    212
Name: count, dtype: int64
```

```
RND = 42
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=RND, stratify=y
)
pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("rf", RandomForestClassifier(n_estimators=200, random_state=RND))
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
print("\nAccuracy on test set:", accuracy_score(y_test, y_pred))
print("\nClassification report:\n", classification_report(y_test, y_pred, digits=3))
```

```
Accuracy on test set: 0.958041958041958

Classification report:
              precision    recall  f1-score   support

           0      0.961     0.925     0.942        53
           1      0.957     0.978     0.967        90

    accuracy                          0.958       143
   macro avg      0.959     0.951     0.955       143
weighted avg      0.958     0.958     0.958       143
```

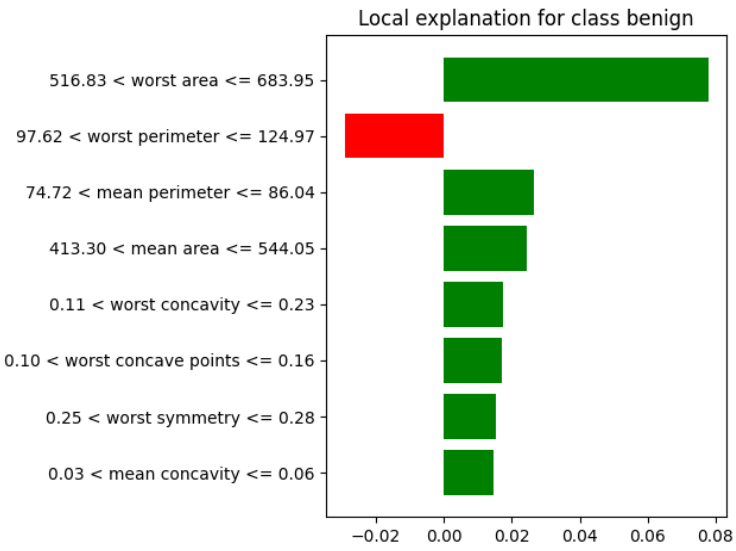
```
explainer_lime = lime.lime_tabular.LimeTabularExplainer(
    training_data=np.array(X_train),
```

```
 feature_names=X_train.columns.tolist(),
    class_names=[str(c) for c in data.target_names],
    discretize_continuous=True,
    random_state=RND
)

instance_idx = 3
instance = X_test.iloc[instance_idx]
instance_array = instance.values.reshape(1, -1)

lime_exp = explainer_lime.explain_instance(
    data_row=instance.values,
    predict_fn=pipe.predict_proba,
    num_features=8
)
print("\n--- LIME explanation (text) ---")
print(lime  exp.as  list(label=1))
```

```
--- LIME explanation (text) ---
[('516.83 < worst area <= 683.95', 0.07785819353633869), ('97.62 < worst perimeter <= 124.97', -0.029296337921043882),
('74.72 < mean perimeter <= 86.04', 0.026426967618676606), ('413.30 < mean area <= 544.05', 0.02439429858609457),
('0.11 < worst concavity <= 0.23', 0.017239650874613342), ('0.10 < worst concave points <= 0.16',
0.017113716622345745), ('0.25 < worst symmetry <= 0.28', 0.01525494740482978), ('0.03 < mean concavity <= 0.06',
0.014740511617709737)]
```

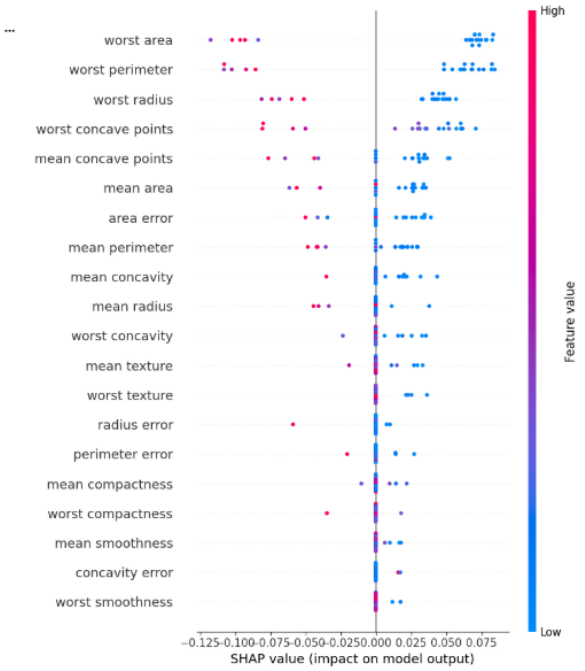
```
lime_fig = lime_exp.as_pyplot_figure(label=1)
plt.tight  layout()
plt.show()
```

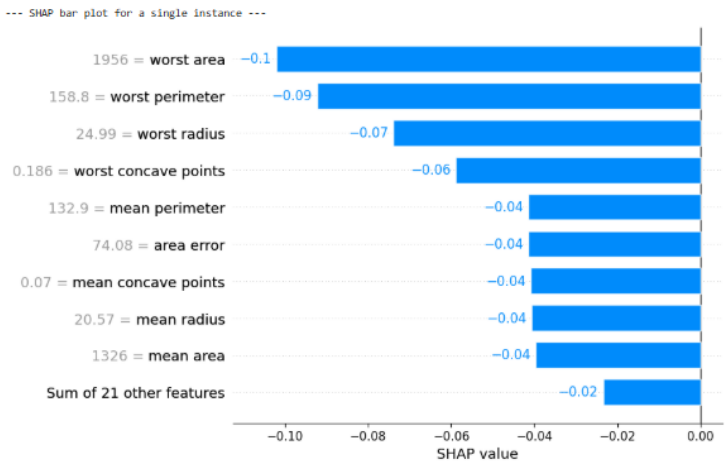
...




```
import shap
import matplotlib.pyplot as plt
import numpy as np

shap.initjs()
```

```
def model_predict(X):
    return pipe.predict_proba(X)[:, 1]
background = X_train.sample(n=min(100, len(X_train)), random_state=RND)
explainer = shap.KernelExplainer(model_predict, background)
X_sample = X_test.sample(20, random_state=RND)
shap_values = explainer.shap_values(X_sample, nsamples=100)
plt.figure(figsize=(8,6))
shap.summary_plot(shap_values, X_sample, feature_names=X.columns, show=True)
instance_idx = 5
print("\n--- SHAP bar plot for a single instance ---")
shap.plots.bar(
    shap.Explanation(
        values=shap_values[instance_idx],
        base_values=explainer.expected_value,
        data=X_sample.iloc[instance_idx],
        feature  names=X.columns
    )
)
```





Learning Outcomes:

1. To understand the importance of explainable AI (XAI) and its role in interpreting complex machine learning models.
2. To apply LIME and SHAP techniques for analyzing and visualizing feature contributions in individual predictions.
3. To use interpretability methods for model debugging, bias detection, and building trust in AI systems.

EXPERIMENT-9

Aim: To design and train reinforcement learning agents to play games and achieve high scores.

Objectives:

- To understand reinforcement learning concepts and the interaction between an agent and its environment.
- To design, train, and evaluate RL agents using techniques like Q-learning or Deep Q-Networks for maximizing cumulative rewards.

Theory:

Reinforcement Learning (RL)

Reinforcement Learning is a branch of machine learning in which an agent learns to make decisions by interacting with an environment. Unlike supervised learning, RL does not require labeled data. Instead, the agent learns through a system of rewards and penalties, reinforcing behaviors that lead to desirable outcomes.

1. RL Framework

An RL problem is generally modeled as a Markov Decision Process (MDP) defined by:

- **S** → Set of states
- **A** → Set of actions
- **R** → Reward function
- **P** → Transition probabilities (defines next state based on current state and action)
- **γ (gamma)** → Discount factor for future rewards

At each time step *t*:

- The agent observes the current state s_t.
- It selects an action $a_t$according to its policy $\pi(a \mid s)$.
- The environment returns a reward $r_t$and transitions to the next state s_{t+1}.

The objective is to learn an optimal policy π^*that maximizes the expected cumulative reward (return).

2. Value and Q-Functions

- **State Value Function:** $V(s) = E[G_t \mid s_t = s]$
- **Action Value Function (Q-Function):** $Q(s,a) = E[G_t \mid s_t = s, a_t = a]$

The Q-function evaluates the expected reward of taking action a in state s.

3. Bellman Equation

The optimal Q-value satisfies the Bellman Optimality Equation:

$$Q^*(s,a) = E[r + \gamma \max_{a'} Q^*(s',a')]$$

4. Deep Q-Network (DQN)

In complex environments with large state spaces, a deep neural network is used to approximate the Q-function:

$$Q(s,a;\theta) \approx Q^*(s,a)$$

DQN employs **experience replay** and a **target network** to stabilize training. The loss function is:

$$L(\theta) = E[(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta))^2]$$

where θ^-represents the parameters of the periodically updated target network.

5. Exploration vs Exploitation

The agent must balance:

- **Exploration:** Trying new actions to discover potentially better rewards.
- **Exploitation:** Selecting the best-known action to maximize reward.

This balance is often achieved using an **ε-greedy strategy**, where with probability ε the agent explores randomly, and with probability $1 - \epsilon$it exploits the best-known action.

.

Code and Output:

```
import gymnasium as gym
import math
import random
import numpy as np
from collections import deque, namedtuple
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import os
from typing import Tuple
ENV  NAME = "CartPole-v1"
SEED = 42
DEVICE = torch.device("cuda" if torch.cuda.is  available() else "cpu")
NUM  EPISODES = 500
MAX  STEPS = 500
BATCH_SIZE = 64
GAMMA = 0.99
LR = 1e-3
```

```
BUFFER_SIZE = 100000
MIN_REPLAY_SIZE = 1000
TARGET_UPDATE_FREQ = 1000
EPS_START = 1.0
EPS_END = 0.01
EPS_DECAY = 0.995
MODEL_DIR = "./dqn_cartpole_model"
os.makedirs(MODEL_DIR, exist_ok=True)
Transition = namedtuple('Transition', ('state', 'action', 'reward', 'next_state', 'done'))
class ReplayBuffer:
    def __init__(self, capacity:int):
        self.buffer = deque(maxlen=capacity)
    def push(self, *args):
        self.buffer.append(Transition(*args))
    def sample(self, batch_size:int) -> Transition:
        batch = random.sample(self.buffer, batch  size)
        return Transition(*zip(*batch))
    def __len__(self):
        return len(self.buffer)
class QNetwork(nn.Module):
    def    init    (self, state  dim:int, action  dim:int):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state  dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action  dim)
        )
    def forward(self, x):
        return self.net(x)
def set  seed(env, seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual  seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

def select_action_epsilon(net: nn.Module, state: np.ndarray, eps: float) -> int:
    if random.random() < eps:
        return env.action_space.sample()
    else:
        state_t = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
        with torch.no_grad():
            qvals = net(state_t)
        return int(qvals.argmax().item())

def compute_td_loss(policy_net: nn.Module, target_net: nn.Module, batch: Transition, optimizer) -> float:
    states = torch.FloatTensor(np.vstack(batch.state)).to(DEVICE)
    actions = torch.LongTensor(batch.action).unsqueeze(1).to(DEVICE)
```

```
    rewards = torch.FloatTensor(batch.reward).unsqueeze(1).to(DEVICE)
    next_states = torch.FloatTensor(np.vstack(batch.next_state)).to(DEVICE)
    dones = torch.FloatTensor(batch.done).unsqueeze(1).to(DEVICE)
    q_values = policy_net(states).gather(1, actions)
    with torch.no_grad():
        next_q = target_net(next_states).max(1)[0].unsqueeze(1)
        expected_q = rewards + GAMMA * next_q * (1 - dones)
    loss = nn.MSELoss()(q_values, expected_q)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.item()
env = gym.make(ENV_NAME)
set_seed(env, SEED)
state_dim = env.observation_space.shape[0]
action  dim = env.action  space.n
policy  net = QNetwork(state  dim, action  dim).to(DEVICE)
target_net = QNetwork(state_dim, action_dim).to(DEVICE)
target_net.load_state_dict(policy_net.state_dict())
optimizer = optim.Adam(policy  net.parameters(), lr=LR)
replay  buffer = ReplayBuffer(BUFFER  SIZE)
print("Populating replay buffer with random transitions...")
```

```
... Populating replay buffer with random transitions...
```

```
state, _ = env.reset(seed=SEED)
for   in range(MIN  REPLAY  SIZE):
    action = env.action  space.sample()

 next_state, reward, terminated, truncated, _ = env.step(action)
    done = terminated or truncated
    replay  buffer.push(state, action, reward, next  state, done)
    state = next_state if not done else env.reset()[0]
print("Replay buffer size:", len(replay_buffer))
```

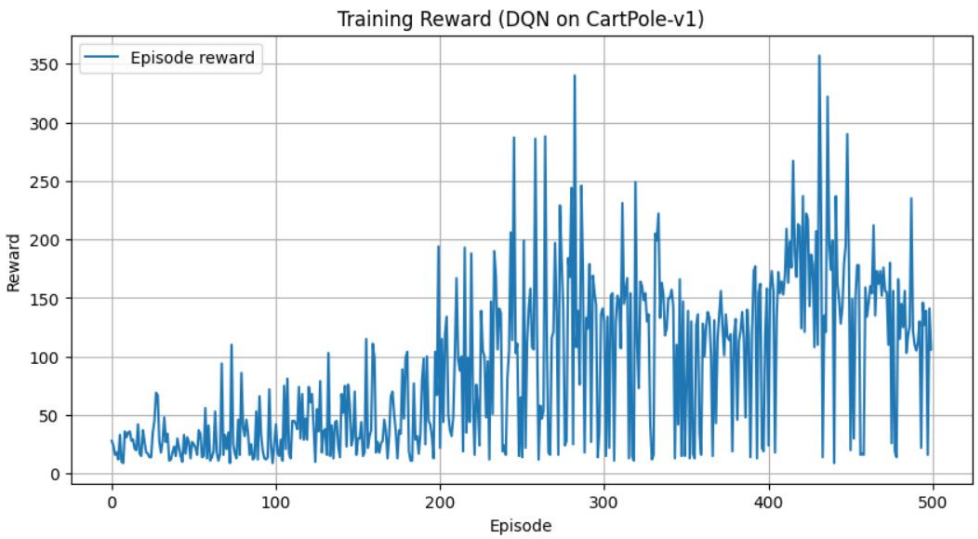
```
... Replay buffer size: 1000
```

```
episode_rewards = []
total_steps = 0
eps = EPS_START
for episode in range(1, NUM_EPISODES + 1):
    state, _ = env.reset()
    episode_reward = 0.0
    for step in range(MAX_STEPS):
        action = select_action_epsilon(policy_net, state, eps)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        replay_buffer.push(state, action, reward, next_state, done)
        episode_reward += reward
        state = next_state
```

```
        total_steps += 1
        if len(replay_buffer) >= BATCH_SIZE:
            batch = replay_buffer.sample(BATCH_SIZE)
            loss = compute_td_loss(policy_net, target_net, batch, optimizer)
        if total_steps % TARGET_UPDATE_FREQ == 0:
            target_net.load_state_dict(policy_net.state_dict())
        if done:
            break
    eps = max(EPS_END, eps * EPS_DECAY)
    episode_rewards.append(episode_reward)
    if episode % 10 == 0:
        avg_reward = np.mean(episode_rewards[-50:])
        print(f"Episode {episode}\tReward: {episode_reward:.2f}\tAvg(50): {avg_reward:.2f}\tEps: {eps:.3f}")
    if episode % 100 == 0:
        torch.save(policy_net.state_dict(), os.path.join(MODEL_DIR, f"policy_ep{episode}.pth"))
torch.save(policy_net.state_dict(), os.path.join(MODEL_DIR, "policy_final.pth"))
print("Training finished. Model saved to", MODEL_DIR)
```

```
Episode 400 Reward: 158.00  Avg(50): 99.40  Eps: 0.135
Episode 410 Reward: 153.00  Avg(50): 111.50 Eps: 0.128
Episode 420 Reward: 211.00  Avg(50): 130.20 Eps: 0.122
Episode 430 Reward: 207.00  Avg(50): 142.26 Eps: 0.116
Episode 440 Reward: 199.00  Avg(50): 159.76 Eps: 0.110
Episode 450 Reward: 179.00  Avg(50): 170.82 Eps: 0.105
Episode 460 Reward: 159.00  Avg(50): 163.00 Eps: 0.100
Episode 470 Reward: 152.00  Avg(50): 155.76 Eps: 0.095
Episode 480 Reward: 166.00  Avg(50): 144.04 Eps: 0.090
Episode 490 Reward: 110.00  Avg(50): 133.96 Eps: 0.086
Episode 500 Reward: 106.00  Avg(50): 121.44 Eps: 0.082
Training finished. Model saved to ./dqn_cartpole_model
```

```
plt.figure(figsize=(10,5))
plt.plot(episode_rewards, label="Episode reward")
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.title("Training Reward (DQN on CartPole-v1)")
plt.legend()
plt.grid(True)
plt.show()
```



```
def evaluate_policy(net: nn.Module, env, episodes=20, render=False) -> float:
    returns = []
    for _ in range(episodes):
        s, _ = env.reset()
        total_r = 0.0
        done = False
        while not done:
            a = select_action_epsilon(net, s, eps=0.0)
            s, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            total_r += r
            if render:
                env.render()
        returns.append(total_r)
    return float(np.mean(returns)), float(np.std(returns))
mean_ret, std_ret = evaluate_policy(policy_net, env, episodes=20, render=False)
print(f"Evaluation over 20 episodes: mean reward = {mean_ret:.2f}, std = {std_ret:.2f}")
env.close()
```

```
Evaluation over 20 episodes: mean reward = 107.35, std = 28.65
```

Learning Outcomes:

1. To understand the fundamentals of reinforcement learning, including the agent–environment framework, value functions, and the Bellman equation.
2. To design and train RL agents using techniques like Q-learning and Deep Q-Networks (DQN) for decision-making in complex environments.
3. To evaluate and improve agent performance by balancing exploration and exploitation to maximize cumulative rewards.

EXPERIMENT-10

Aim: To apply differential privacy techniques to protect sensitive information while performing data analysis.

Objectives:

- To understand the concept of Differential Privacy (DP) and its importance in safeguarding individual data during model training.
- To implement DP-SGD using the Opacus library and evaluate the trade-off between model accuracy and privacy levels (ε value).

Theory:

Differential Privacy (DP)

Differential Privacy is a formal mathematical framework that ensures an algorithm's output does not reveal sensitive information about any individual in a dataset. It is widely used in machine learning to protect personal data during model training.

Key Concepts:

- **DP-SGD (Differentially Private Stochastic Gradient Descent):** Introduces controlled random noise to gradients during training to limit the influence of any single data point.
- **Privacy Parameters:**
  - **ε (epsilon):** Controls the strength of privacy; smaller ε means stronger privacy but may reduce model accuracy.
  - **δ (delta):** Allows a small probability of the privacy guarantee being violated.
- **Trade-off:** There is a balance between privacy and model performance. Stronger privacy usually comes at the cost of slightly lower accuracy.

An algorithm $A$is (ϵ, δ)-differentially private if, for any two neighboring datasets $D_1$and D_2, and for all possible outputs S:

$$P[A(D_1) \in S] \leq e^{\epsilon} \cdot P[A(D_2) \in S] + \delta$$

Importance:

- Protects individuals' data from being inferred by attackers.
- Enables safe use of sensitive datasets in machine learning.
- Provides a measurable and controllable level of privacy, making it suitable for real-world applications like healthcare, finance, and social networks.

Code and Output:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy  score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from opacus import PrivacyEngine
RND = 42
np.random.seed(RND)
torch.manual_seed(RND)
def laplace_mechanism(value, sensitivity, epsilon):
    scale = sensitivity / epsilon
    return value + np.random.laplace(0.0, scale)

def gaussian_mechanism(value, sensitivity, epsilon, delta):
    sigma = sensitivity * np.sqrt(2 * np.log(1.25 / delta)) / epsilon
    return value + np.random.normal(0.0, sigma)
N = 1000
ages = np.random.normal(loc=40, scale=12, size=N)
ages = np.clip(ages, 18, 90)
true  mean = ages.mean()
true_count = len(ages)
print("True mean age:", round(true_mean, 3), "Count:", true_count)
```

```
True mean age: 40.338 Count: 1000
```

```
val  min, val  max = 18.0, 90.0
sensitivity  mean = (val  max - val  min) / N
eps  list = [0.1, 0.5, 1.0, 2.0]
laplace_means = []
for eps in eps  list:
    noisy = laplace_mechanism(true_mean, sensitivity_mean, eps)
    laplace_means.append(noisy)
    print(f"Laplace (eps={eps}): noisy mean = {noisy:.4f}")
```

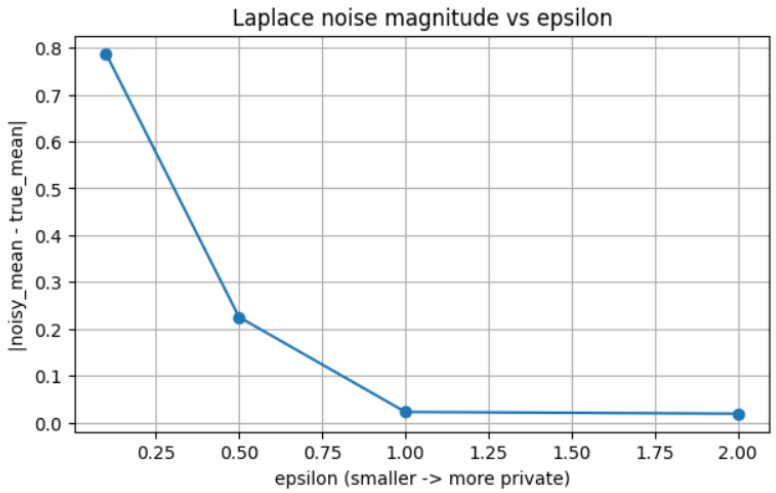
```
Laplace (eps=0.1): noisy mean = 39.5506
Laplace (eps=0.5): noisy mean = 40.1127
Laplace (eps=1.0): noisy mean = 40.3610
Laplace (eps=2.0): noisy mean = 40.3572
```

```
delta = 1e-5
sensitivity_count = 1.0
```

```
eps = 1.0
noisy_count = gaussian_mechanism(true_count, sensitivity_count, eps, delta)
print(f"\nGaussian (eps={eps}, delta={delta}): noisy count = {noisy_count:.2f}")
```

```
Gaussian (eps=1.0, delta=1e-05): noisy count = 1000.29
```

```
plt.figure(figsize=(7,4))
plt.plot(eps_list, np.abs(np.array(laplace_means) - true_mean), marker='o')
plt.xlabel("epsilon (smaller -> more private)")
plt.ylabel("|noisy_mean - true_mean|")
plt.title("Laplace noise magnitude vs epsilon")
plt.grid(True)
plt.show()
```




```
data = load_breast_cancer()
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=RND, stratify=y)
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
tensor_x = torch.tensor(X_train, dtype=torch.float32)
tensor_y = torch.tensor(y_train, dtype=torch.long)
train_ds = TensorDataset(tensor_x, tensor_y)
tensor_x_test = torch.tensor(X_test, dtype=torch.float32)
tensor_y_test = torch.tensor(y_test, dtype=torch.long)
test_ds = TensorDataset(tensor_x_test, tensor_y_test)
BATCH_SIZE = 64
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
test_loader = DataLoader(test_ds, batch_size=256, shuffle=False)

class SimpleLogistic(nn.Module):
    def __init__(self, input_dim, hidden=64):
        super().__init__()
        self.net = nn.Sequential(
```

```
            nn.Linear(input_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 2)
        )
    def forward(self, x):
        return self.net(x)

input_dim = X_train.shape[1]
model = SimpleLogistic(input_dim).to('cpu')

def train_standard(model, loader, epochs=10, lr=1e-3):
    m = model
    opt = optim.Adam(m.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    m.train()
    for epoch in range(epochs):
        for xb, yb in loader:
            opt.zero_grad()
            logits = m(xb)
            loss = loss  fn(logits, yb)
            loss.backward()
            opt.step()

def evaluate(model, loader):
    model.eval()
    preds, trues = [], []
    with torch.no  grad():
        for xb, yb in loader:
            logits = model(xb)
            p = logits.argmax(dim=1).cpu().numpy()
            preds.append(p)
            trues.append(yb.cpu().numpy())
    preds = np.concatenate(preds)
    trues = np.concatenate(trues)
    return accuracy  score(trues, preds)
baseline_model = SimpleLogistic(input_dim)
train_standard(baseline_model, DataLoader(train_ds, batch_size=64, shuffle=True), epochs=20, lr=1e-3)
baseline_acc = evaluate(baseline_model, test_loader)
print("\nBaseline (non-private) test accuracy:", round(baseline_acc, 4))
```

```
Baseline (non-private) test accuracy: 0.965
```

```
from opacus import PrivacyEngine
EPOCHS = 10
LR = 1e-3
MAX_GRAD_NORM = 1.0
NOISE_MULTIPLIER = 1.1
DELTA = 1e-5
BATCH_SIZE = 64
```

```
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
test_loader = DataLoader(test_ds, batch_size=256, shuffle=False)
dp_model = SimpleLogistic(input_dim)
optimizer = optim.Adam(dp_model.parameters(), lr=LR)
criterion = nn.CrossEntropyLoss()
privacy_engine = PrivacyEngine()
dp_model, optimizer, train_loader = privacy_engine.make_private(
    module=dp_model,
    optimizer=optimizer,
    data_loader=train_loader,
    noise_multiplier=NOISE_MULTIPLIER,
    max_grad_norm=MAX_GRAD_NORM,
)
print("✅ Model and optimizer successfully made private with DP-SGD")
```

```
✅ Model and optimizer successfully made private with DP-SGD
```

```
for epoch in range(EPOCHS):
    dp_model.train()
    total_loss = 0
    for xb, yb in train_loader:
        optimizer.zero_grad()
        outputs = dp_model(xb)
        loss = criterion(outputs, yb)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    epsilon = privacy_engine.get_epsilon(delta=DELTA)
    test_acc = evaluate(dp_model, test_loader)
    print(f"Epoch {epoch+1}/{EPOCHS} | Loss={total_loss/len(train_loader):.4f} | Acc={test_acc:.4f} | ε={epsilon:.2f}")
```

```
Epoch 1/10 | Loss=0.7603 | Acc=0.3776 | ε=3.57
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 2/10 | Loss=0.6693 | Acc=0.6154 | ε=4.49
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 3/10 | Loss=0.5967 | Acc=0.7972 | ε=5.21
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 4/10 | Loss=0.5330 | Acc=0.8741 | ε=5.83
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 5/10 | Loss=0.4661 | Acc=0.8951 | ε=6.40
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 6/10 | Loss=0.4329 | Acc=0.9021 | ε=6.92
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 7/10 | Loss=0.3595 | Acc=0.9161 | ε=7.40
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 8/10 | Loss=0.3286 | Acc=0.9231 | ε=7.86
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 9/10 | Loss=0.2927 | Acc=0.9231 | ε=8.30
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py:1864: FutureWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Epoch 10/10 | Loss=0.2668 | Acc=0.9231 | ε=8.72
```

```
print("\n✅ Differentially Private Training Complete")
final_acc = evaluate(dp_model, test_loader)
final_eps = privacy_engine.get_epsilon(delta=DELTA)
print(f"Final Test Accuracy: {final_acc:.4f}")
print(f"Final Privacy Budget: ε={final_eps:.2f}, δ={DELTA}")
```

```
✅ Differentially Private Training Complete
Final Test Accuracy: 0.9231
Final Privacy Budget: ε=8.72, δ=1e-05
```

Learning Outcomes:

1. To understand the concept of Differential Privacy and its role in protecting individual data during machine learning.
2. To implement DP-SGD for training neural networks with privacy-preserving mechanisms using tools like Opacus.
3. To analyze the trade-off between model accuracy and privacy levels (ε and δ) and make informed decisions for practical applications.