

**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.



**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

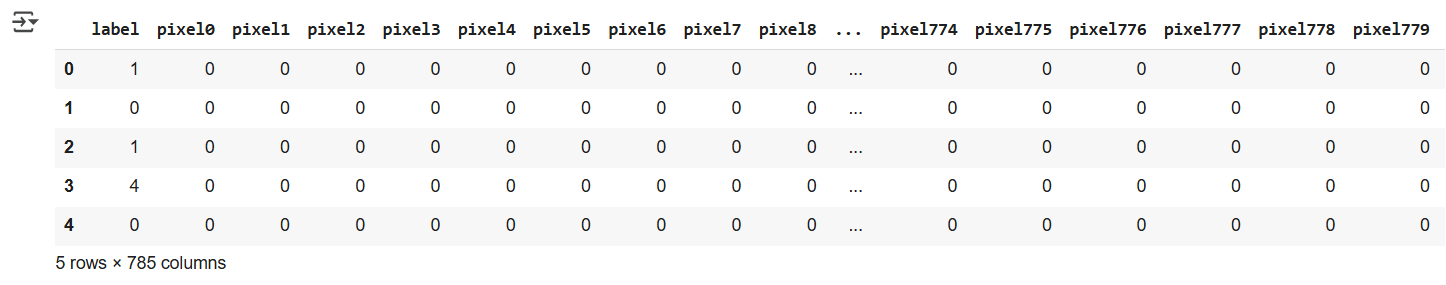
from tensorflow.keras.utils import to\_categorical

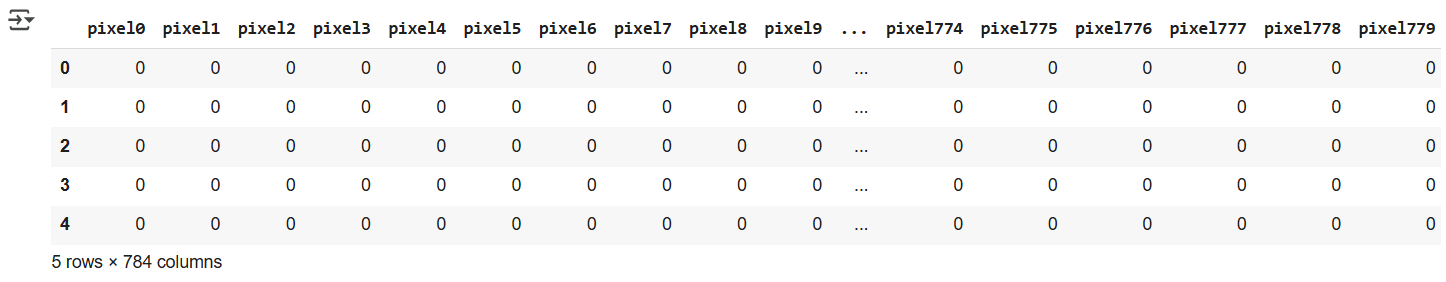
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout

train = pd.read\_csv("train.csv")

test = pd.read\_csv("test.csv")

train.head()

test.head()

print(train.isna().sum().sum())

print(test.isna().sum().sum())



train['label'].value\_counts().sort\_index()



fig, ax = plt.subplots(figsize=(18, 8))

for ind, row in train.iloc[:8, :].iterrows():

    plt.subplot(2, 4, ind+1)

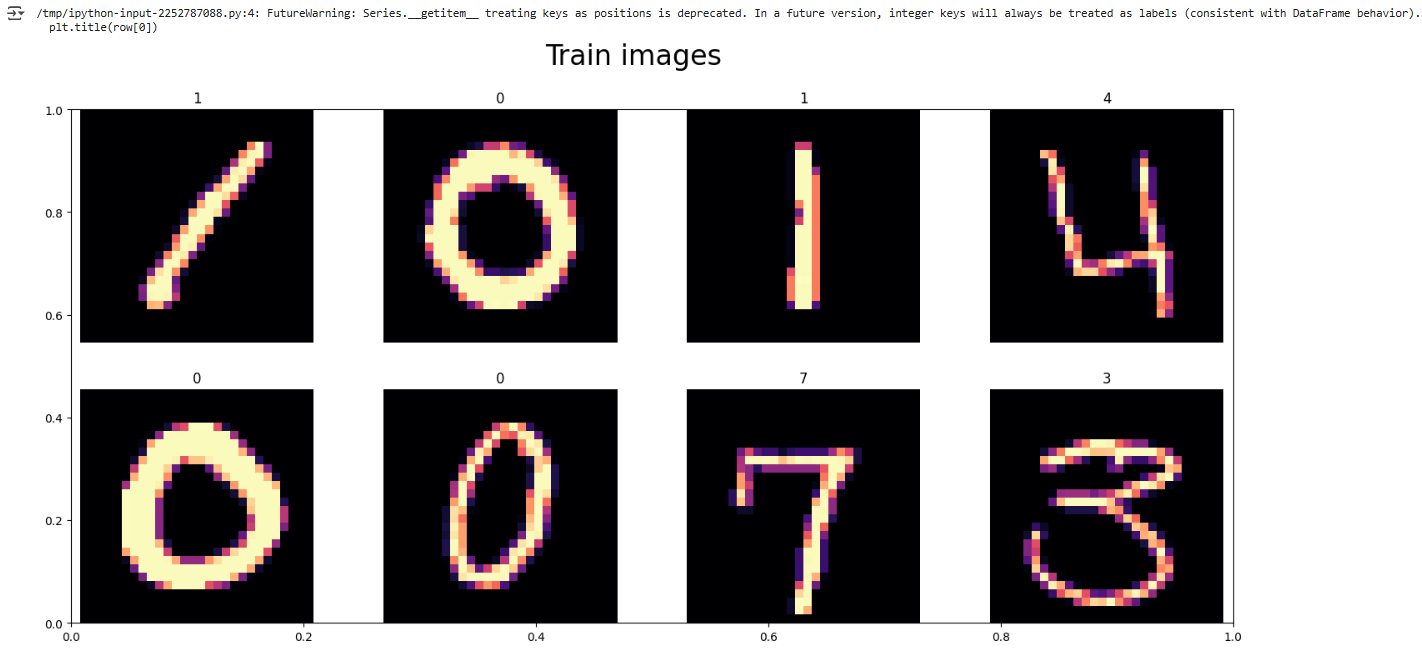
    plt.title(row[0])

    img = row.to\_numpy()[1:].reshape(28, 28)

    fig.suptitle('Train images', fontsize=24)

    plt.axis('off')

    plt.imshow(img, cmap='magma')



fig, ax = plt.subplots(figsize=(18, 8))

for ind, row in test.iloc[:8, :].iterrows():

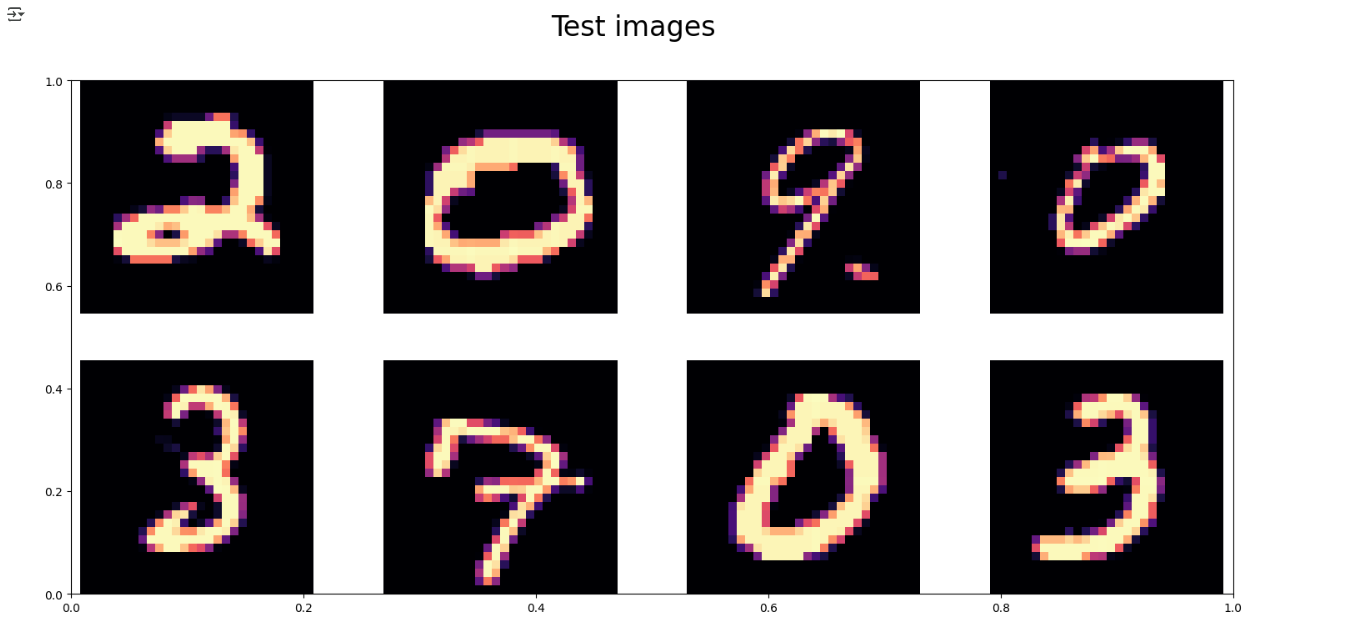
    plt.subplot(2, 4, ind+1)

    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)



X = X / 255.0

test = test / 255.0

print(X.shape)

print(test.shape)



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

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

print(X.shape)

print(test.shape)



print(y.shape)

print(y[0])



y\_enc = to\_categorical(y, num\_classes = 10)

print(y\_enc.shape)

print(y\_enc[0])



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)

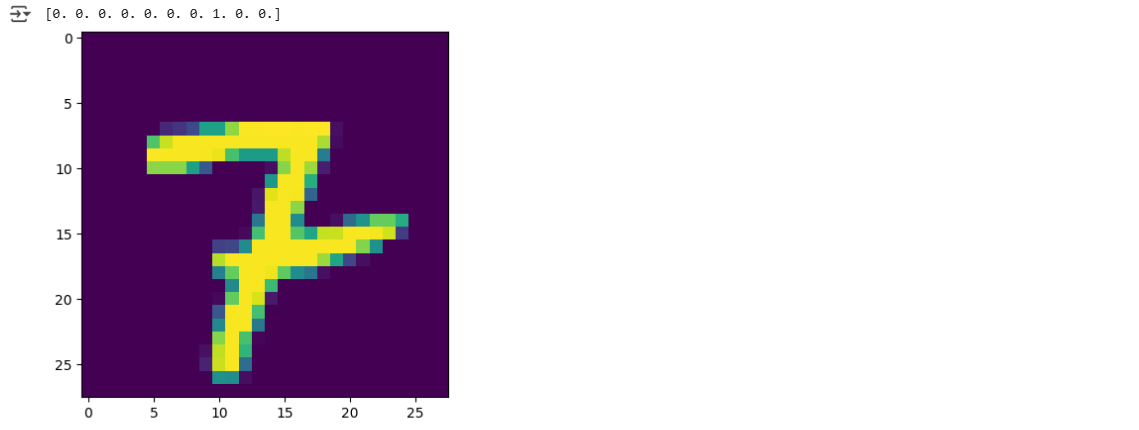
g = plt.imshow(X\_train[0][:,:,0])

print(y\_train\_enc[0])



g = plt.imshow(X\_train[9][:,:,0])

print(y\_train\_enc[9])



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))

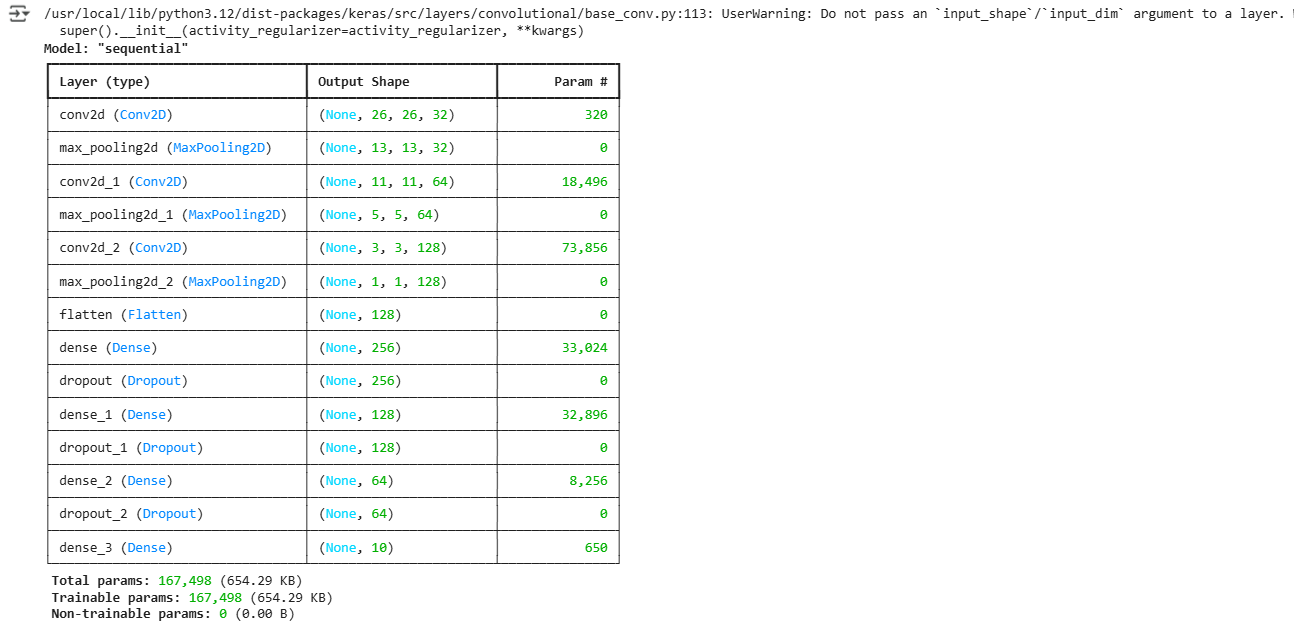
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

model.summary()



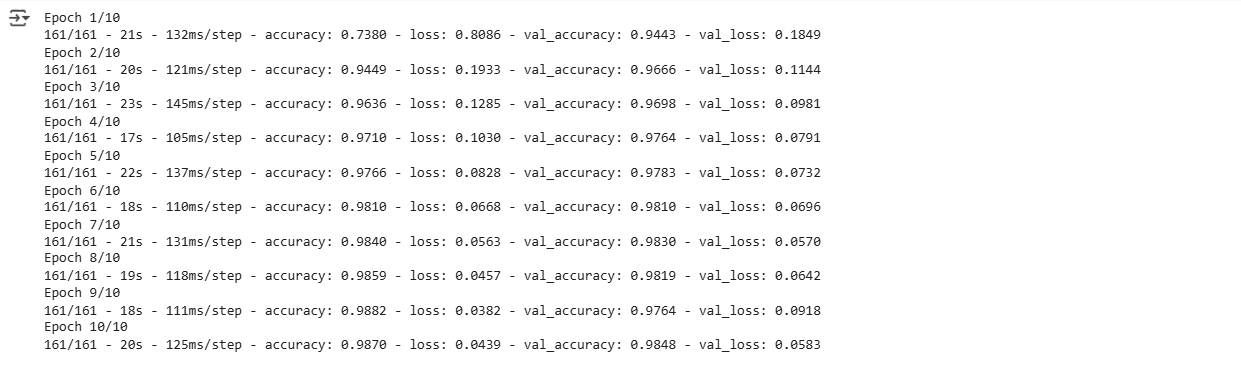
history = model.fit(X\_train, y\_train\_enc,

                    epochs=EPOCHS,

                    batch\_size=BATCH\_SIZE,

                    verbose=VERBOSE,

                    validation\_split=0.3)



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')

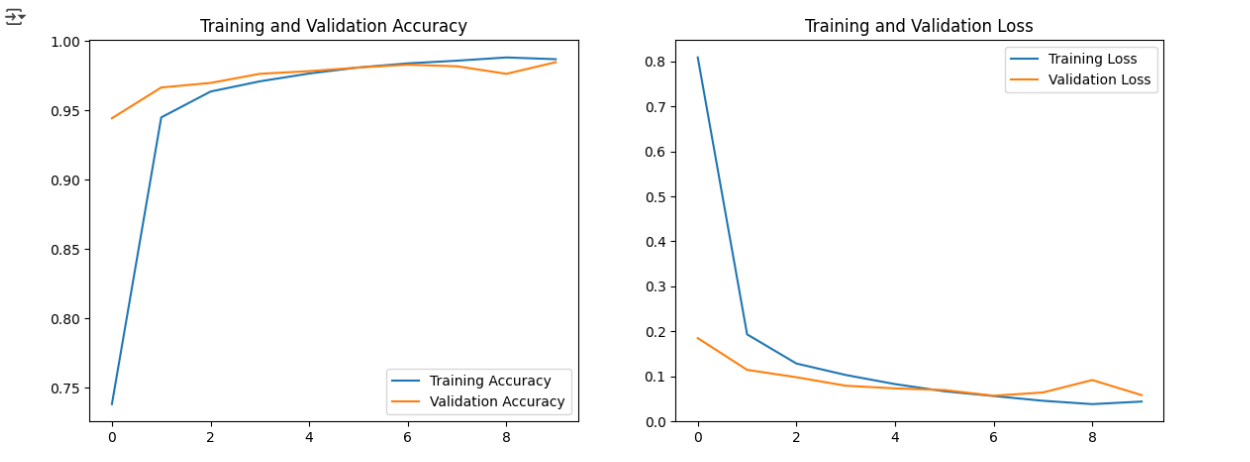
plt.plot(history.history['val\_loss'], label='Validation Loss')

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)



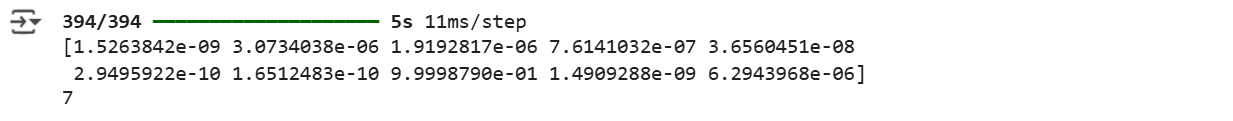
y\_pred\_enc = model.predict(X\_val)

y\_act = [np.argmax(i) for i in y\_val\_enc]

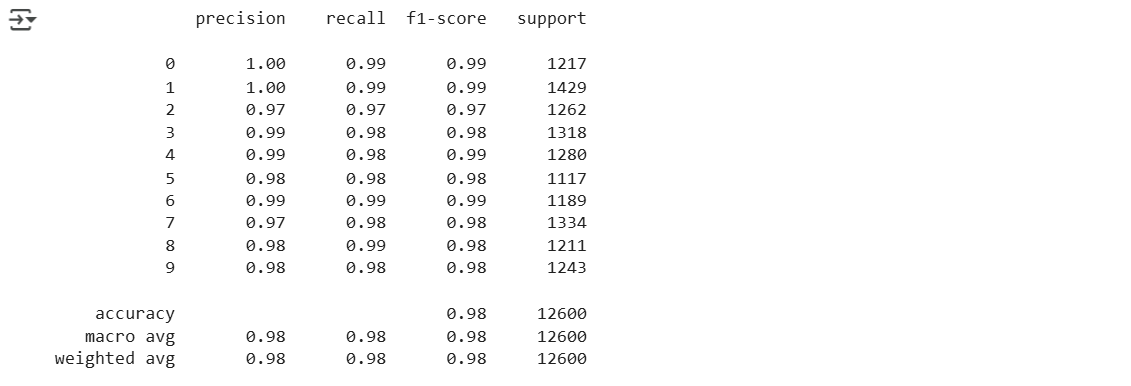
y\_pred = [np.argmax(i) for i in y\_pred\_enc]

print(y\_pred\_enc[0])

print(y\_pred[0])



print(classification\_report(y\_act, y\_pred))



fig, ax = plt.subplots(figsize=(7, 7))

sns.heatmap(confusion\_matrix(y\_act, y\_pred), annot=True,

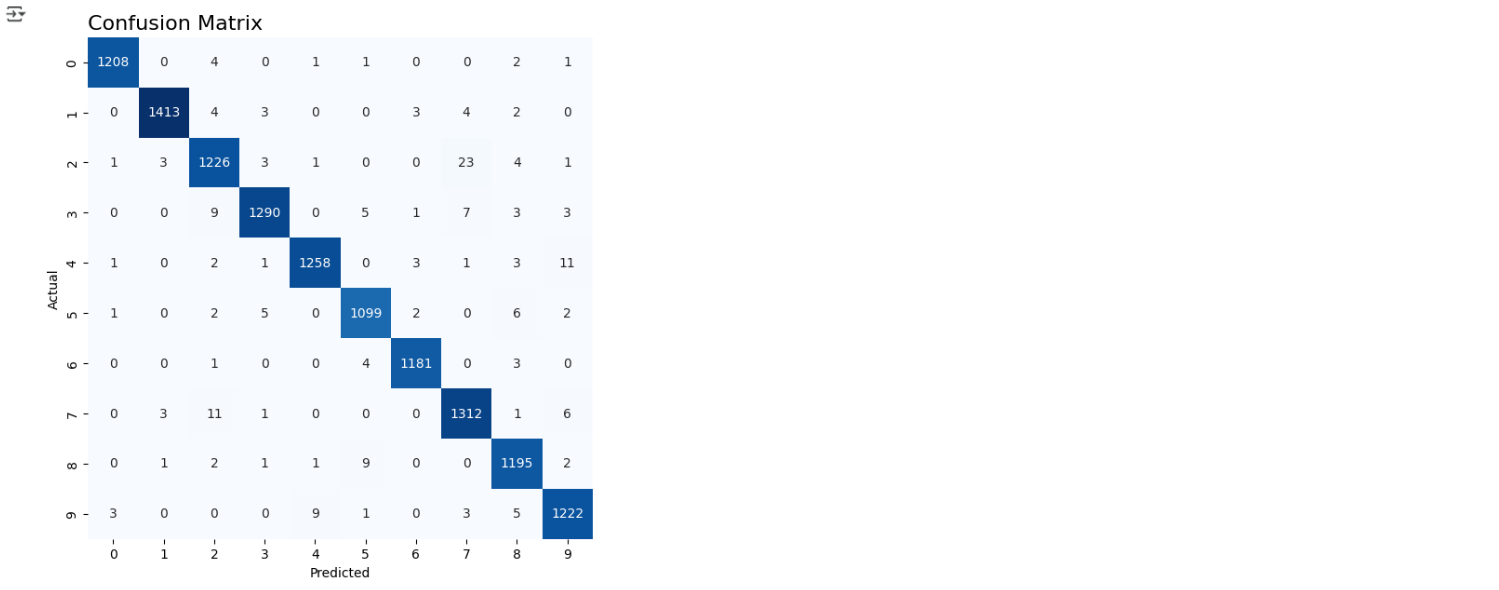
            cbar=False, fmt='1d', cmap='Blues', ax=ax)

ax.set\_title('Confusion Matrix', loc='left', fontsize=16)

ax.set\_xlabel('Predicted')

ax.set\_ylabel('Actual')

plt.show()

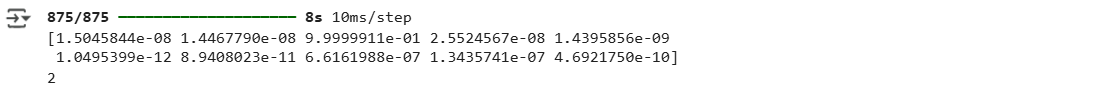


y\_pred\_enc = model.predict(test)

y\_pred = [np.argmax(i) for i in y\_pred\_enc]

print(y\_pred\_enc[0])

print(y\_pred[0])



fig, ax = plt.subplots(figsize=(18, 12))

for ind, row in enumerate(test[:15]):

    plt.subplot(3, 5, ind+1)

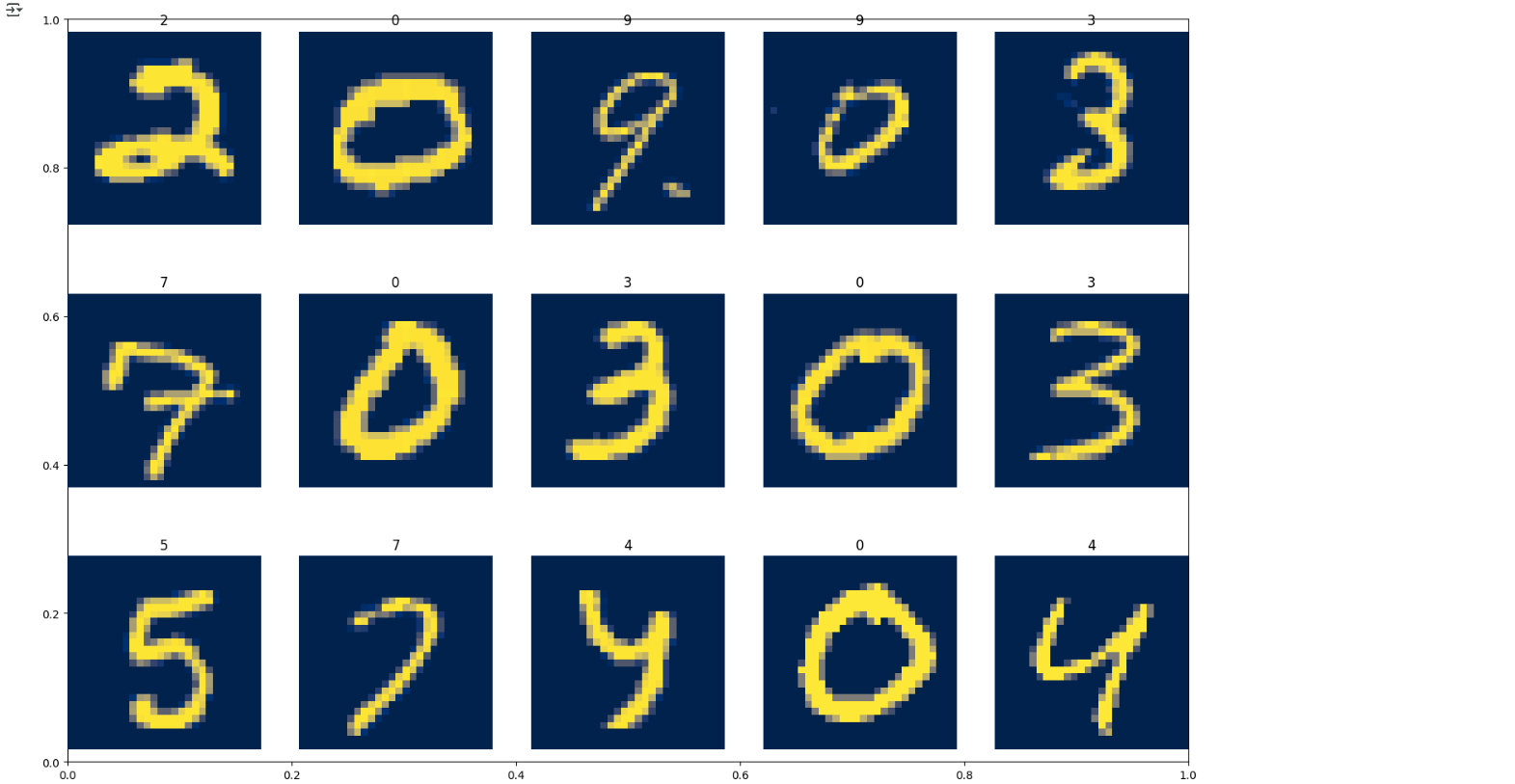
    plt.title(y\_pred[ind])

    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)



def imshow(img):

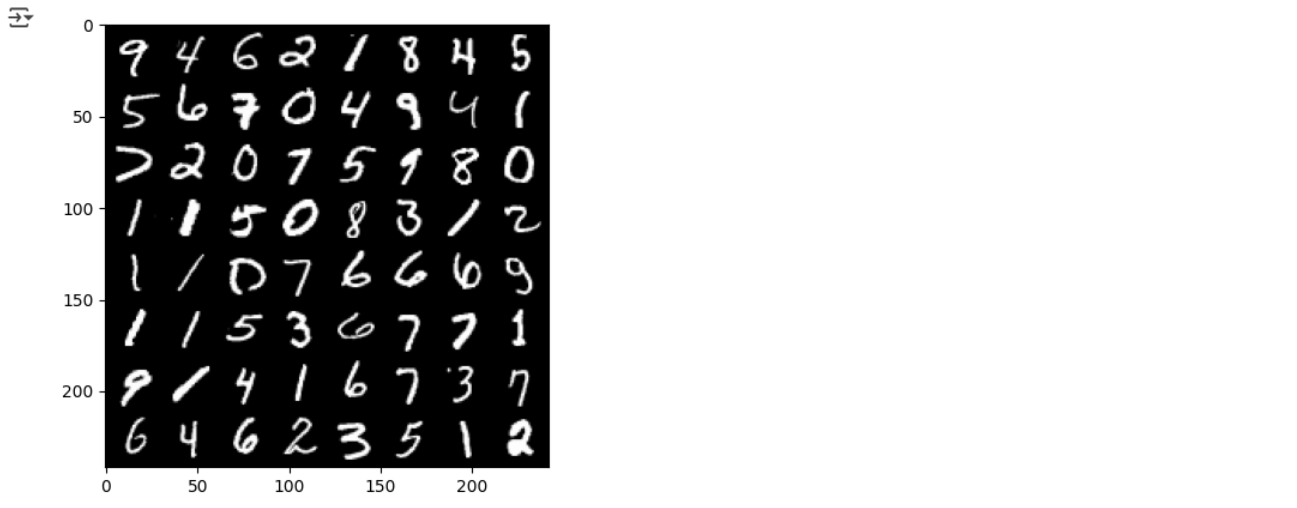
    npimg = img.numpy()

    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)

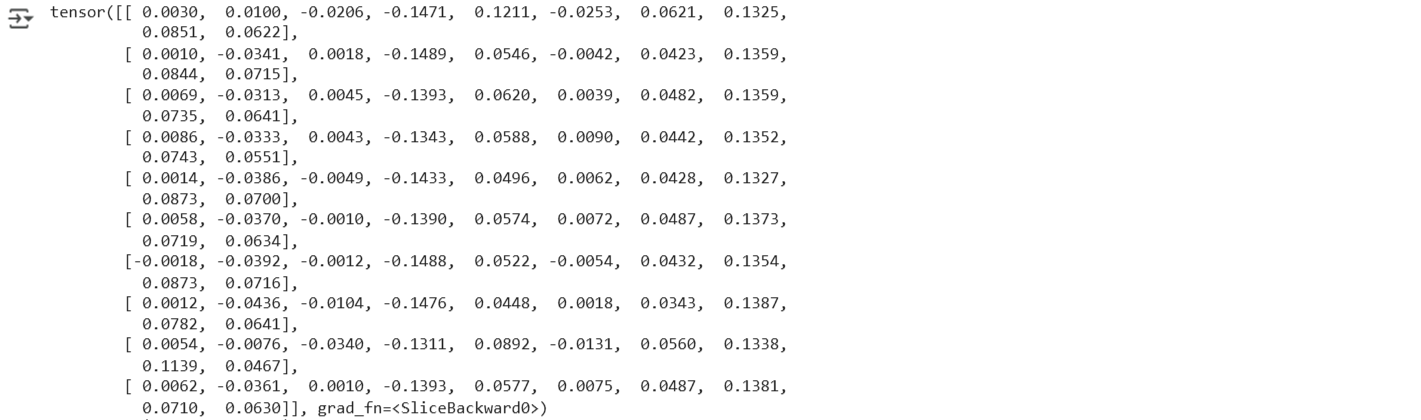
dataiter = iter(trainloader)

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])



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()

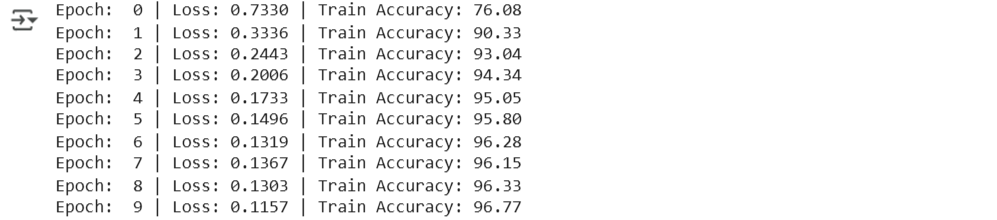
        train\_running\_loss += loss.detach().item()

        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))



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)

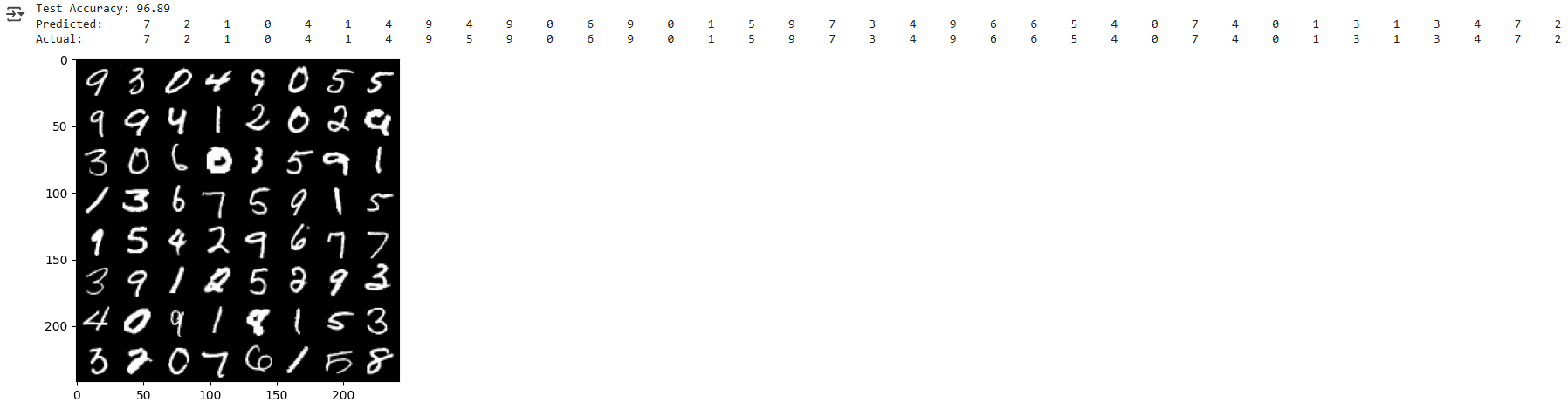
images = images.view(-1, 28, 28)

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)))



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

axes = axes.flatten()

for i in range(BATCH\_SIZE):

    img = images[i].squeeze().numpy()

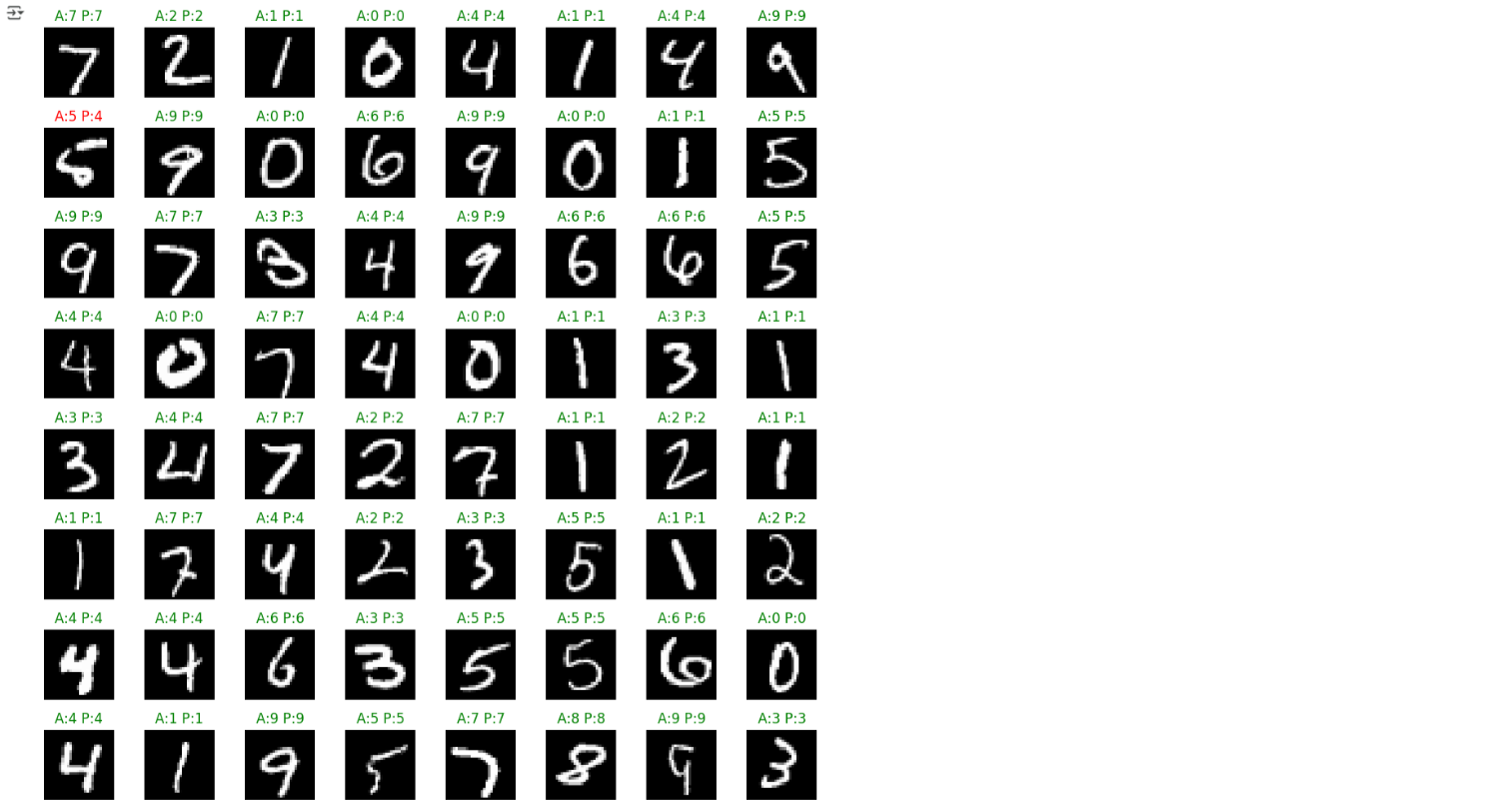
    axes[i].imshow(img, cmap='gray')

    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!")

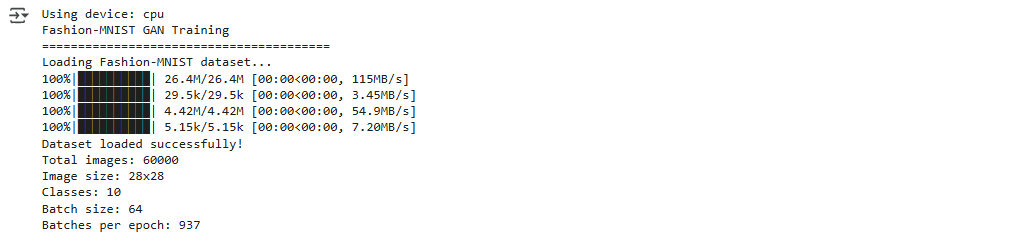
  print("Check 'fashion\_samples' folder for generated images during training.")

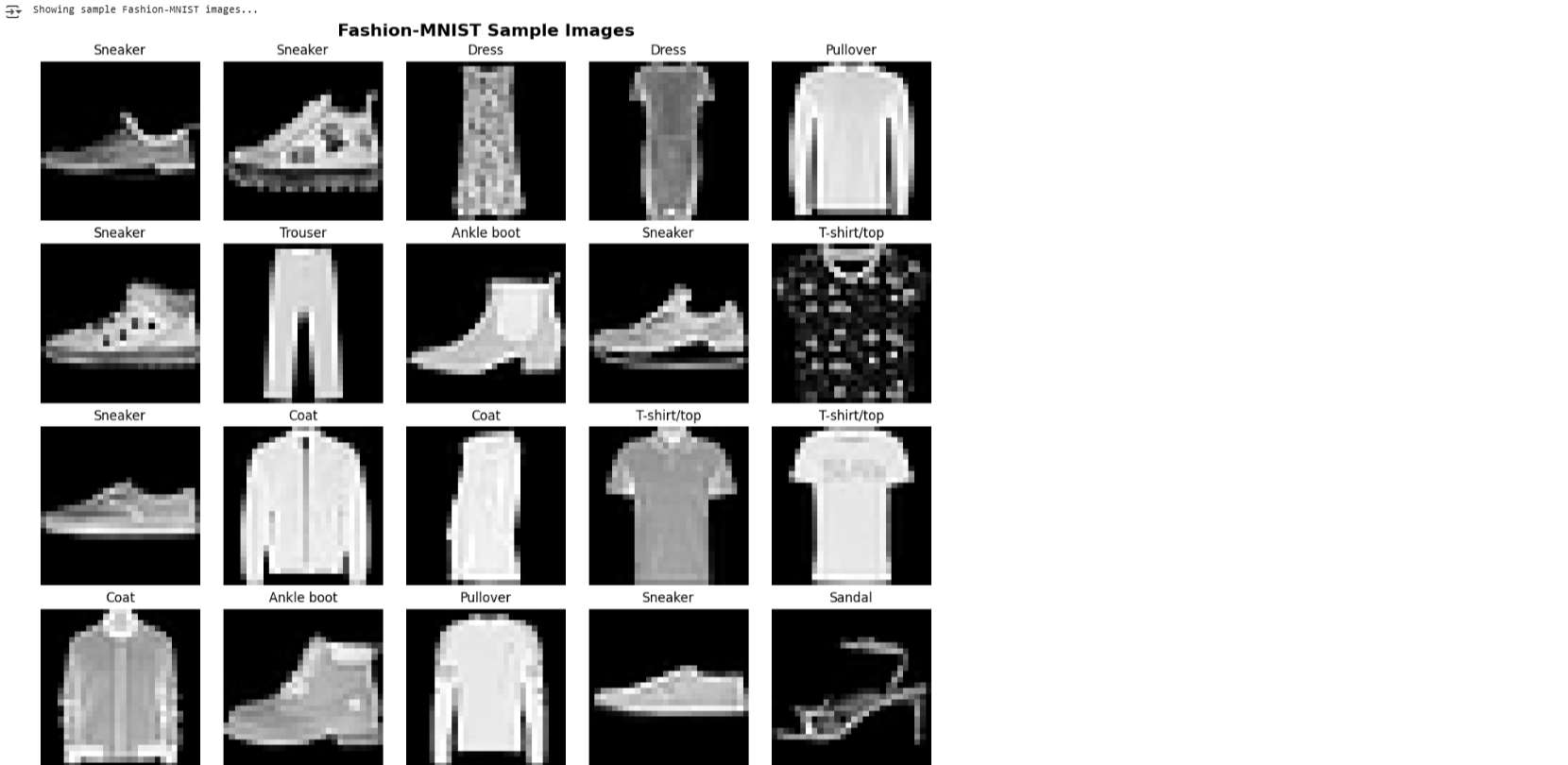
    print("Models saved in 'fashion\_gan\_models' folder.")

if \_\_name\_\_ == "\_\_main\_\_":

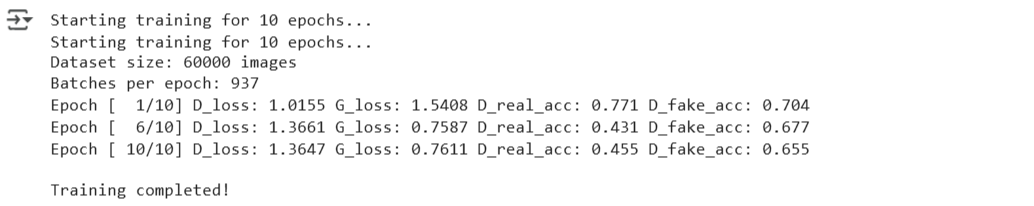
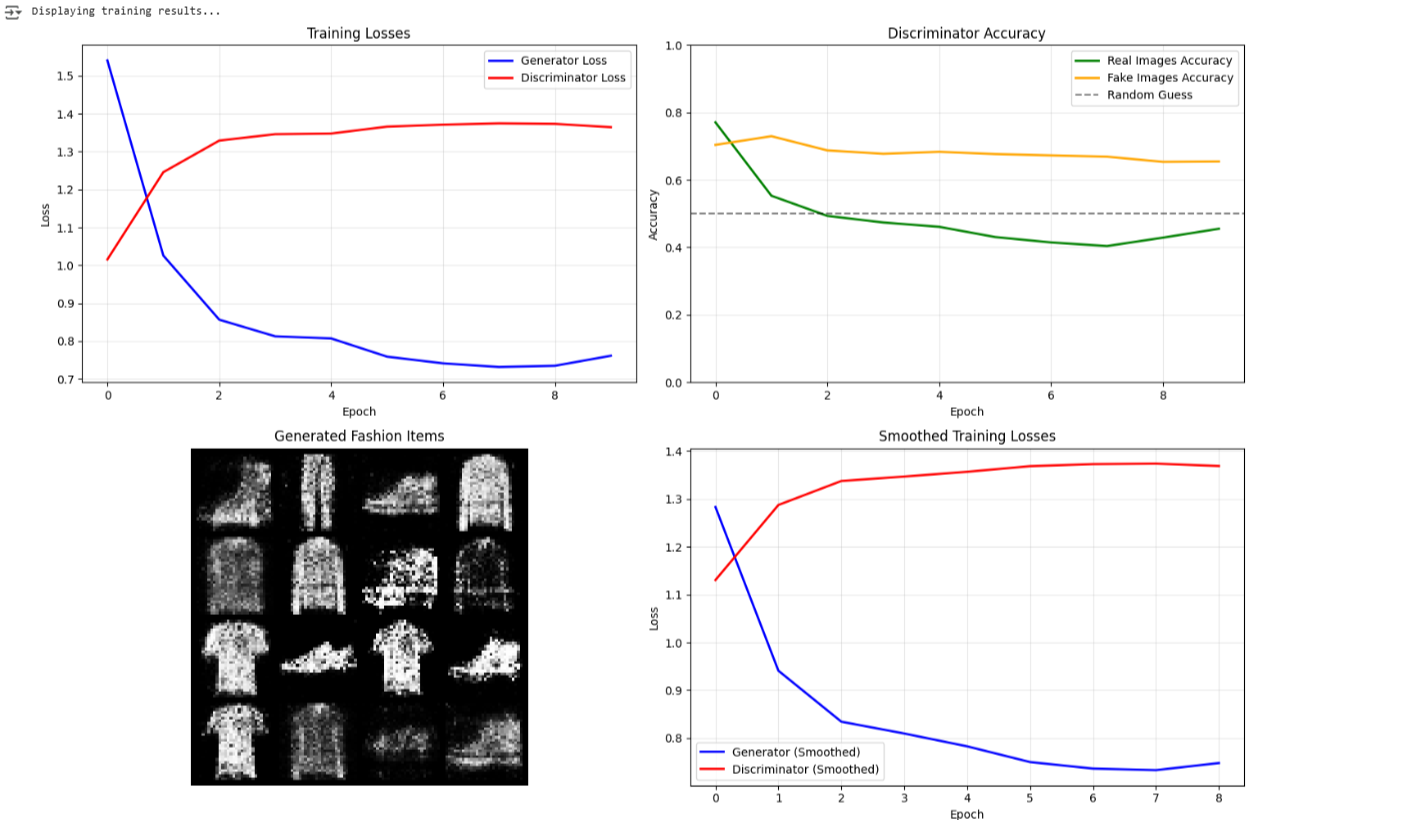
    main()

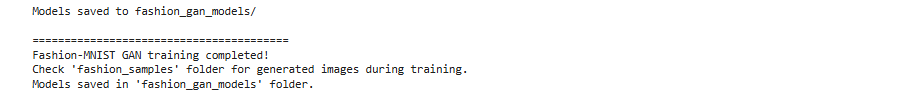
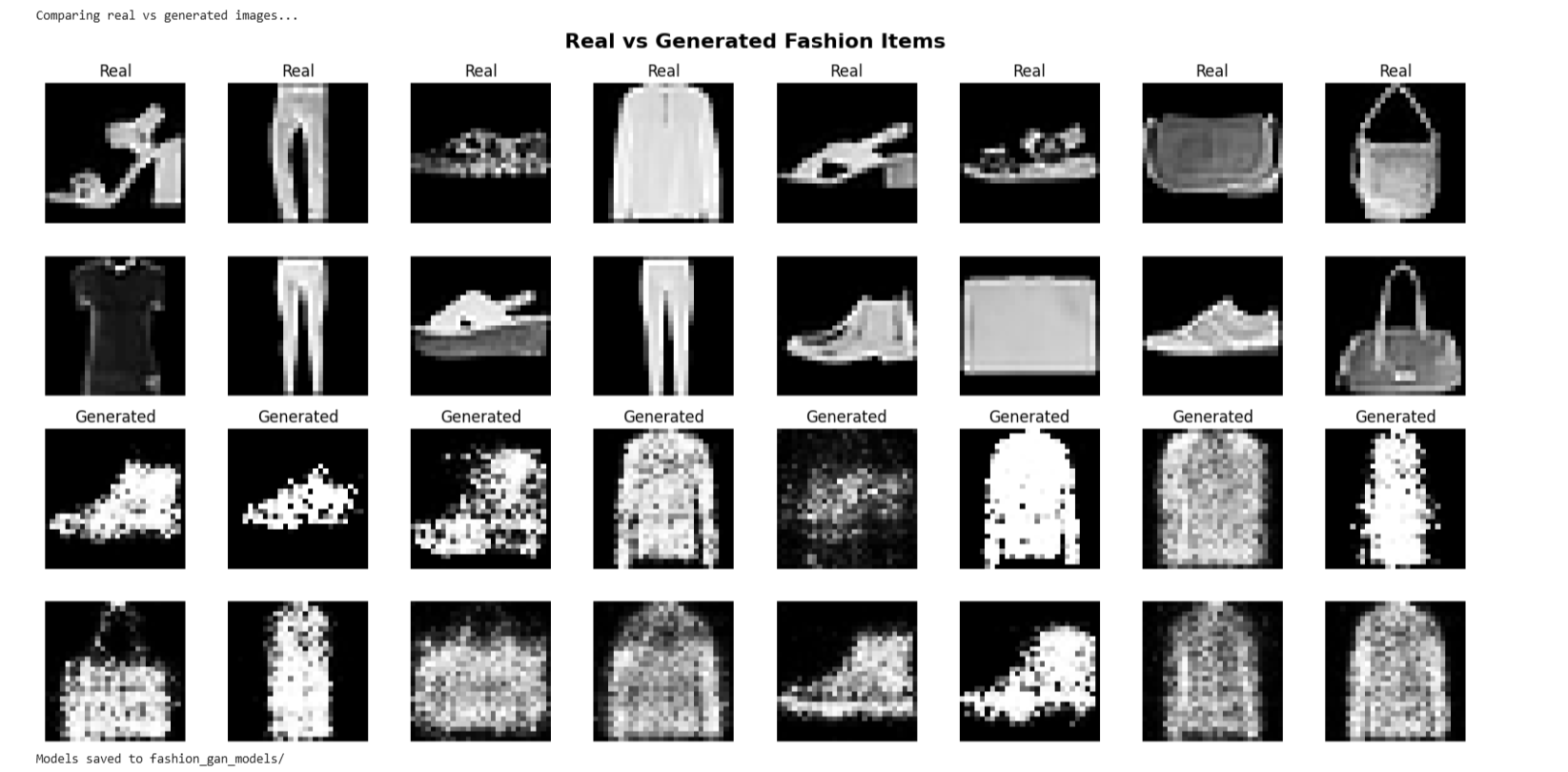
**Output:**





****





**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])



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

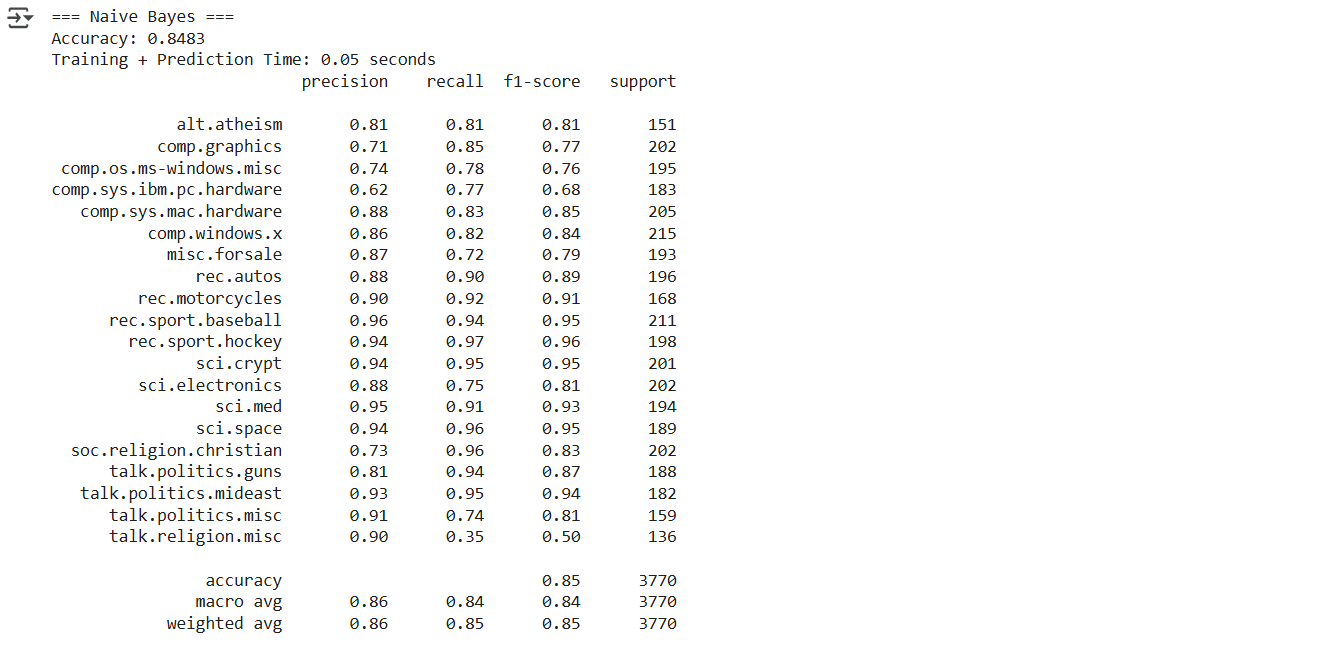
results[name] = {"accuracy": acc, "runtime": runtime}

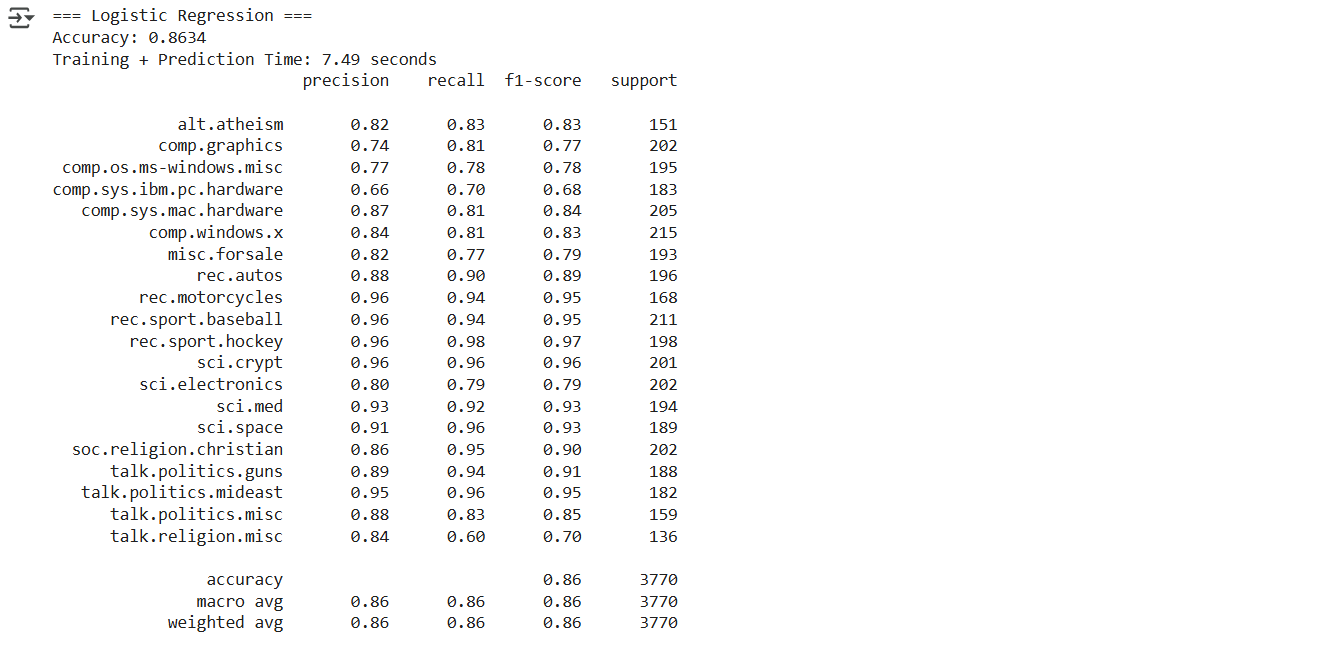
    print(f"\n=== {name} ===")

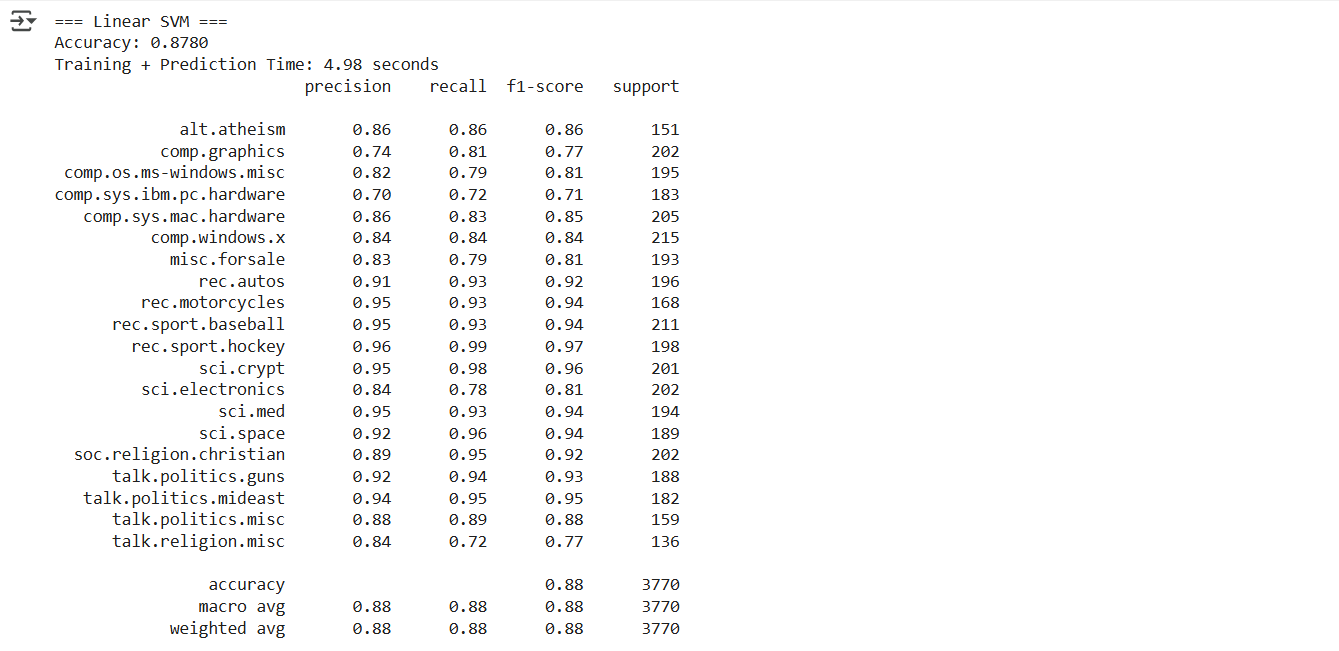
    print(f"Accuracy: {acc:.4f}")

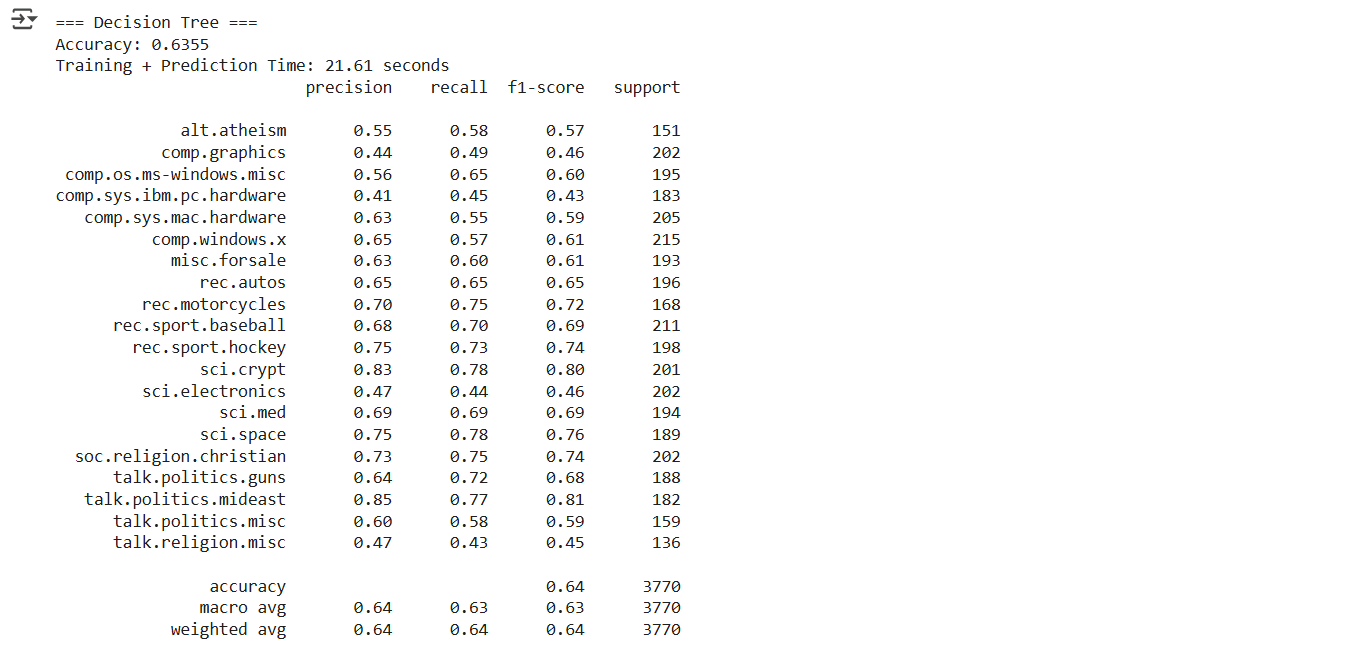
    print(f"Training + Prediction Time: {runtime:.2f} seconds")

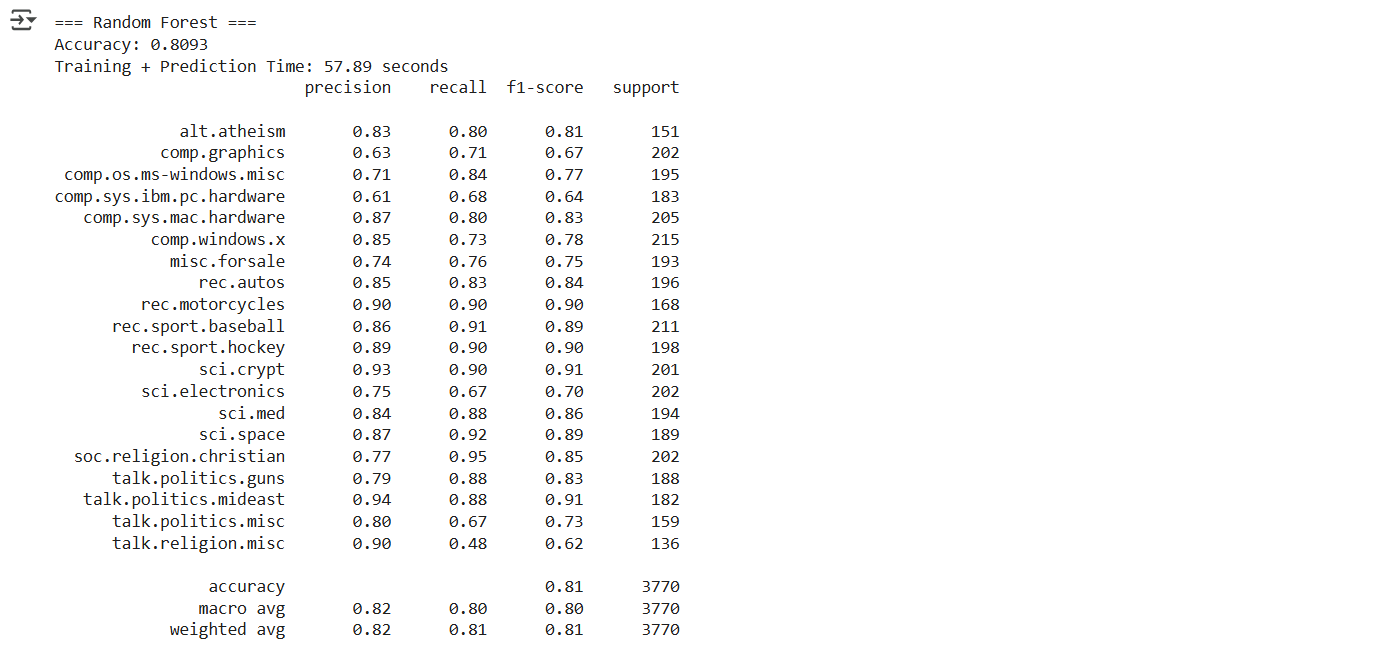
    print(classification\_report(y\_test, y\_pred, target\_names=newsgroups.target\_names))

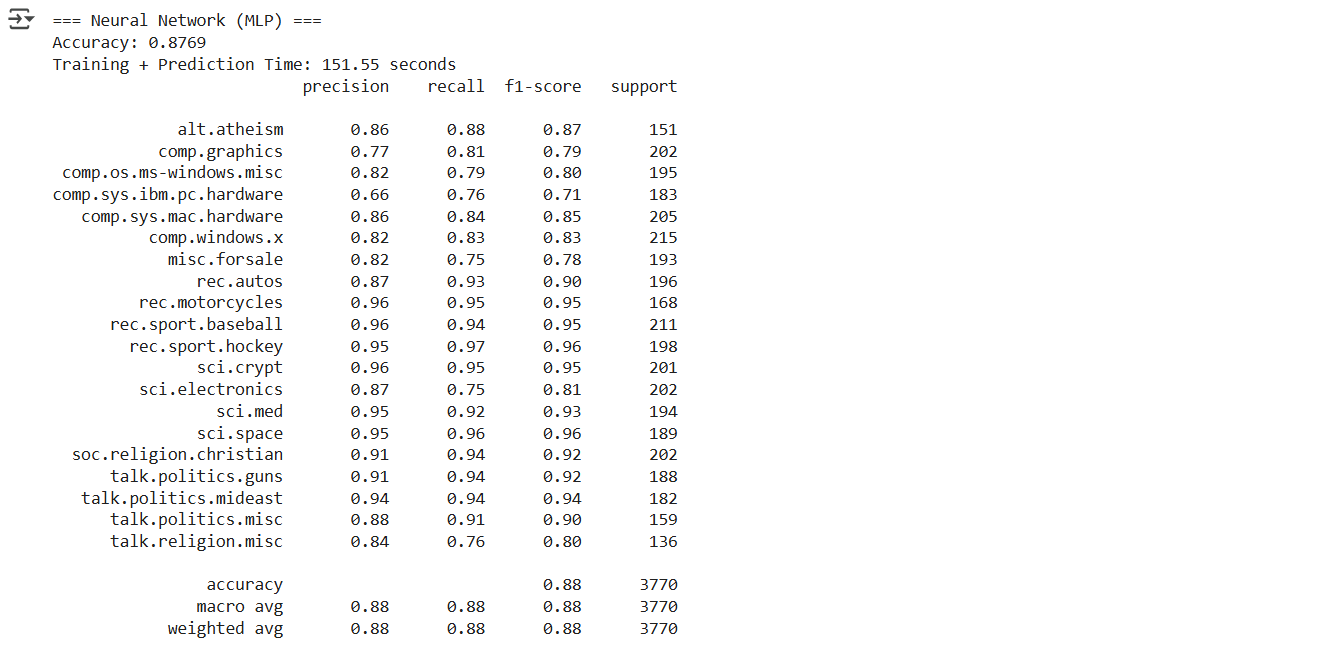












model\_names = list(results.keys())

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)

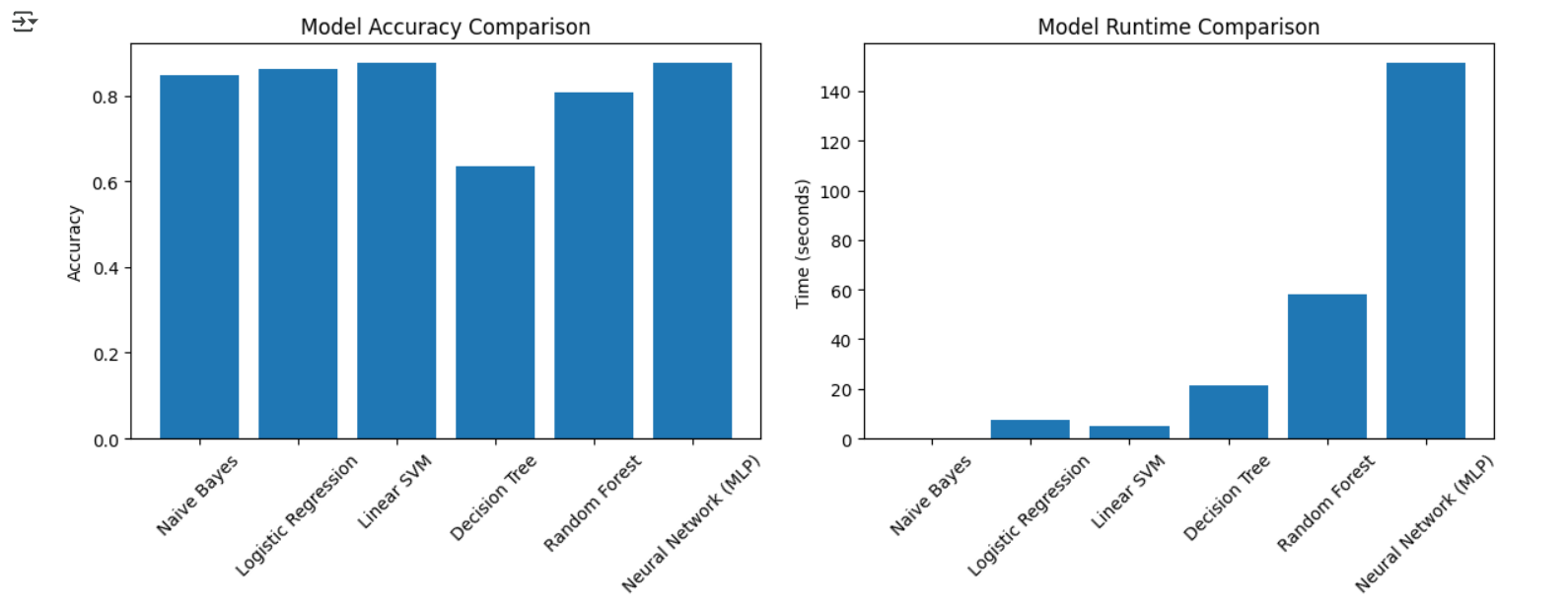
plt.title("Model Runtime Comparison")

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 R². 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"

# Initialize findspark

import findspark

findspark.init()

print("✅ Spark installation completed!")

# STEP 2: IMPORT LIBRARIES

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")

        # Get available cores

        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()

        # Create dataset with realistic features

        df = self.spark.range(num\_rows) \

            .select(

                col("id"),

                # Numeric features

                (rand() \* 100).alias("feature1"),

                (randn() \* 10 + 50).alias("feature2"),

                (rand() \* 1000).alias("feature3"),

                (randn() \* 5 + 25).alias("feature4"),

                # Categorical features

                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"),

                # Age and income

                (rand() \* 60 + 18).cast("int").alias("age"),

                (rand() \* 80000 + 20000).alias("income"),

                # Binary target

                (rand() > 0.4).cast("int").alias("target\_binary"),

                # Continuous target

                (rand() \* 1000 + randn() \* 100).alias("target\_continuous")

            )

        # Optimize for Colab

        df = df.repartition(4)

        df.cache()

        # Force computation

        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)

        # Basic info

        print("\n🔍 Dataset Schema:")

        df.printSchema()

        print(f"\n📏 Dataset size: {df.count():,} rows")

        # Statistical summary

        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()

        # Convert to Pandas for visualization

        stats\_pandas = stats\_df.toPandas()

        # Null analysis

        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 analysis

        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)

        # Create visualizations

        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 data for plotting (to avoid memory issues)

        sample\_df = df.sample(0.1, seed=42).toPandas()

        # Set up the plotting style

        plt.style.use('seaborn-v0\_8')

        fig = plt.figure(figsize=(20, 15))

        # Numeric features distribution

        if len(numeric\_cols) > 0:

            for i, col in enumerate(numeric\_cols[:6]): # Limit to 6 plots

                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')

        # Categorical features

        plot\_idx = 7

        for col in categorical\_cols[:2]: # Limit to 2 categorical plots

            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

        # Correlation heatmap

        if len(numeric\_cols) > 1:

            plt.subplot(3, 4, 9)

            numeric\_sample = sample\_df[numeric\_cols[:6]] # Limit columns

            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()

        # Identify column types

        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()

        # Force computation and show results

        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)

        # 1. Logistic Regression

        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}")

        # 2. Random Forest

        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}")

        # 3. Gradient Boosting

        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}")

        # Results summary

        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)

        # Split data

        train\_df, test\_df = df.randomSplit([0.8, 0.2], seed=42)

        train\_df.cache()

        test\_df.cache()

        models = {}

        evaluator = RegressionEvaluator(labelCol=target\_col)

        # Linear Regression

        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}")

        # Random Forest Regression

        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()

            # Calculate metrics

            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}")

            # Show cluster distribution

            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...")

        # Extract metrics

        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]

        # Create plots

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

        # AUC comparison

        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)

        # Add value labels on bars

        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')

        # Accuracy comparison

        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')

        # Training time comparison

        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)

        # Get feature importances

        importances = rf\_model.featureImportances.toArray()

        if feature\_names is None:

            feature\_names = [f"Feature\_{i}" for i in range(len(importances))]

        # Create importance DataFrame

        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))

        # Visualization

        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')

        # Add value labels

        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:

            # Step 1: Create dataset

            print("\n📁 STEP 1: Creating Dataset")

            df = self.create\_sample\_dataset(dataset\_size)

            # Step 2: Explore dataset

            print("\n🔍 STEP 2: Dataset Exploration")

            exploration\_results = self.explore\_dataset(df)

            # Step 3: Preprocess data

            print("\n🔧 STEP 3: Data Preprocessing")

            processed\_df, preprocessing\_model = self.preprocess\_data(df)

            # Step 4: Classification

            print("\n🤖 STEP 4: Classification Models")

            classification\_models = self.train\_classification\_models(processed\_df)

            # Step 5: Regression

            print("\n📈 STEP 5: Regression Models")

            regression\_models = self.train\_regression\_models(processed\_df)

            # Step 6: Clustering

            print("\n🎯 STEP 6: Clustering Analysis")

            clustering\_results = self.perform\_clustering(processed\_df)

            # Step 7: Feature importance

            print("\n🔍 STEP 7: Feature Importance")

            rf\_model = classification\_models['Random Forest']['model']

            importance\_df = self.feature\_importance\_analysis(rf\_model)

            # Pipeline summary

            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)

        # Extract data from results

        classification\_models = results['classification\_models']

        regression\_models = results['regression\_models']

        clustering\_results = results['clustering\_results']

        # Create comprehensive dashboard

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

        # 1. Classification Performance

        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')

        # 2. Regression Performance

        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')

        # 3. Training Time Comparison

        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')

        # 4. Clustering Results

        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)

        # 5. Feature Importance (Top 10)

        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()

        # 6. Model Accuracy Comparison

        ax6 = plt.subplot(2, 3, 6)

        accuracies = [classification\_models[name]['accuracy'] for name in model\_names]

        # Create a pie chart for accuracy

        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 summary statistics

        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)

            # Save model performance summary

            summary\_data = []

            # Classification models

            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']

                })

            # Regression models

            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)

            # Save execution summary

            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)

        # print("\n🎛️ Running hyperparameter tuning demo...")

        # cv\_model, best\_params, test\_auc = processor.hyperparameter\_tuning\_demo(

        #     results['dataset'], target\_col="target\_binary"

        # )

        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)}")

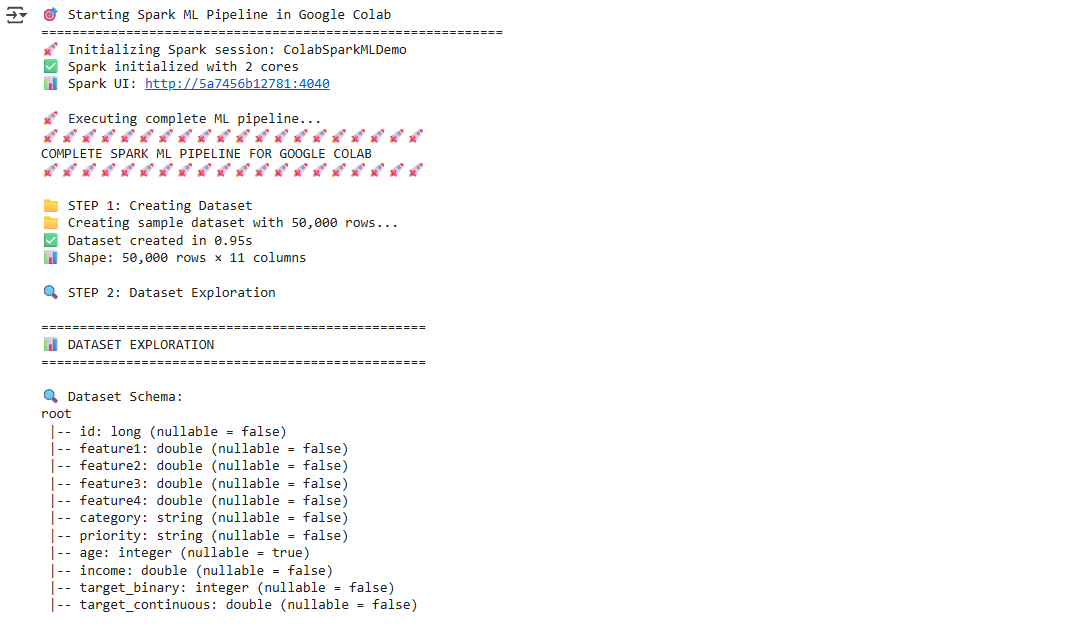
        processor.close()

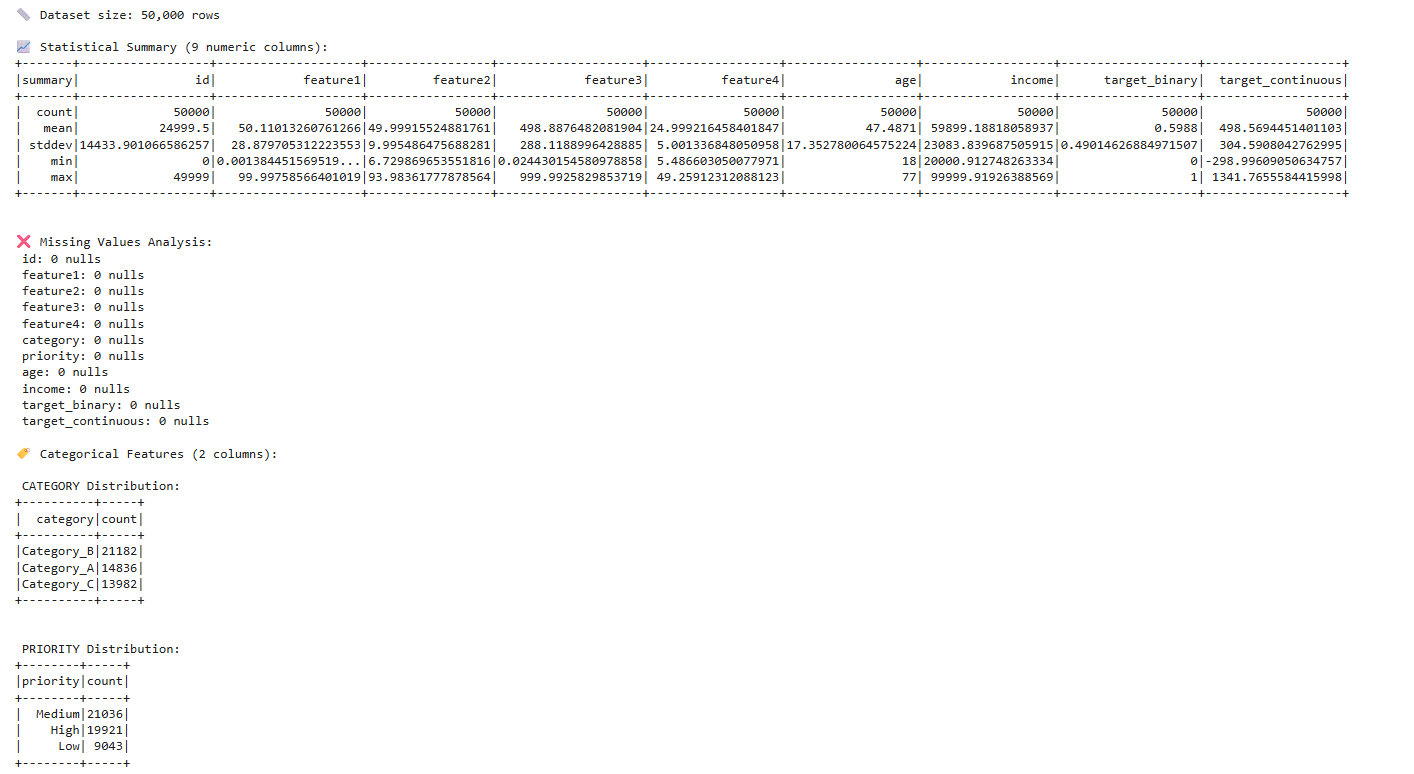
        raise e

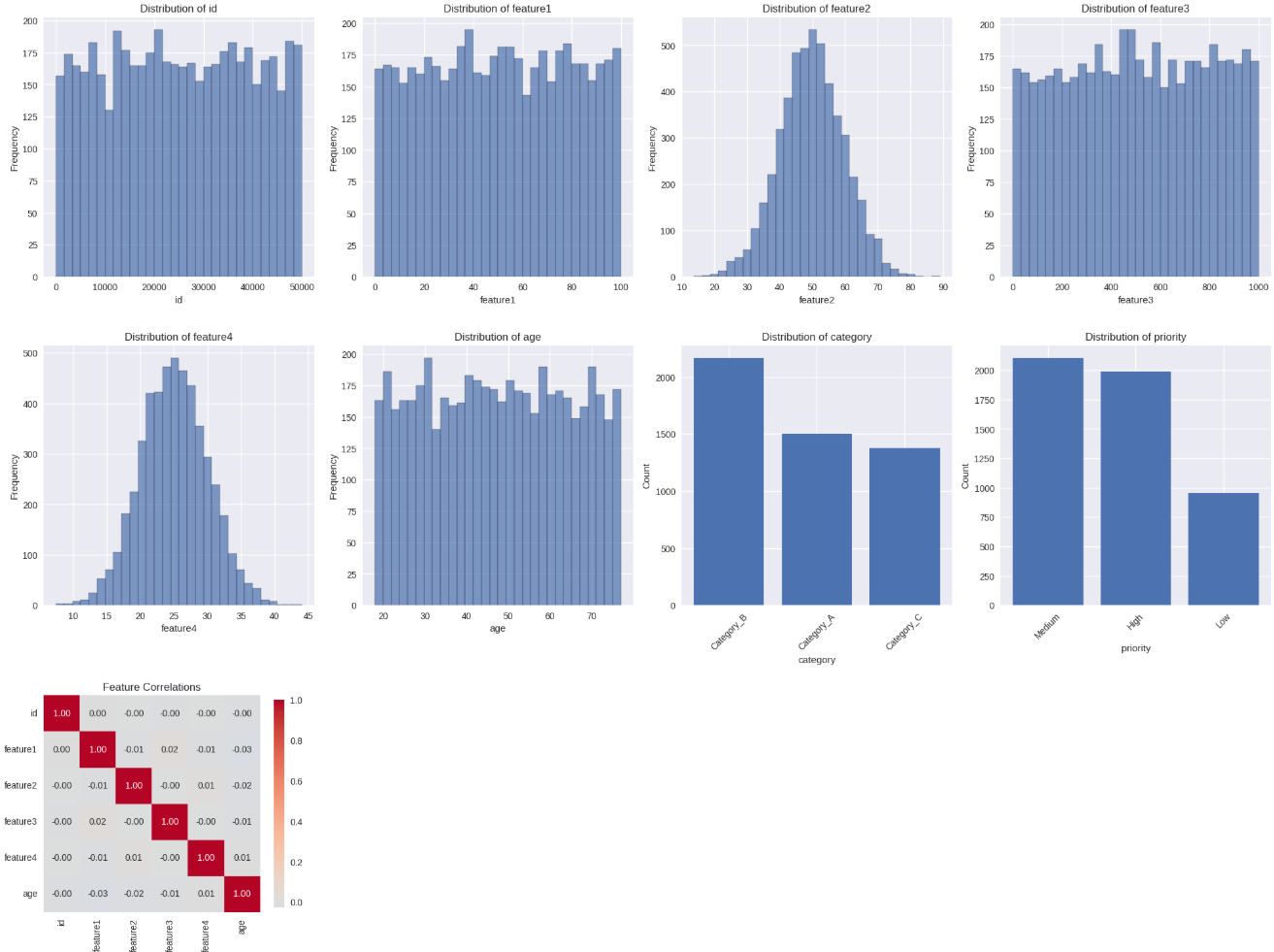
if \_\_name\_\_ == "\_\_main\_\_":

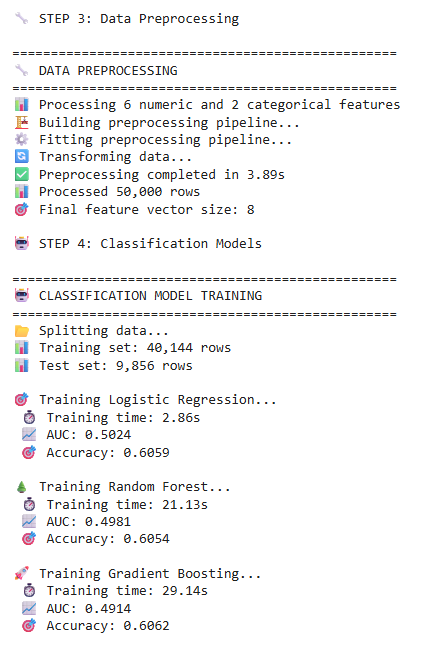
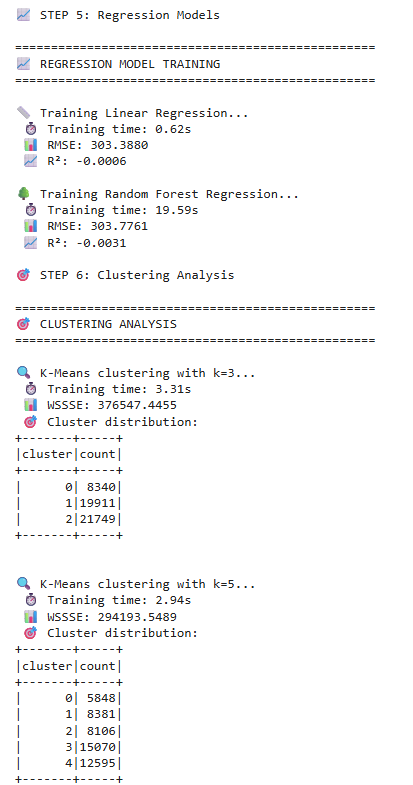
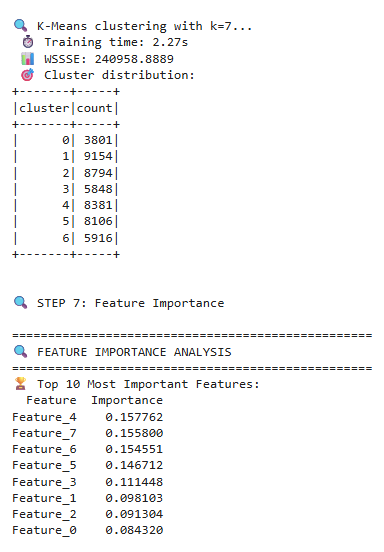
    results, processor = main()

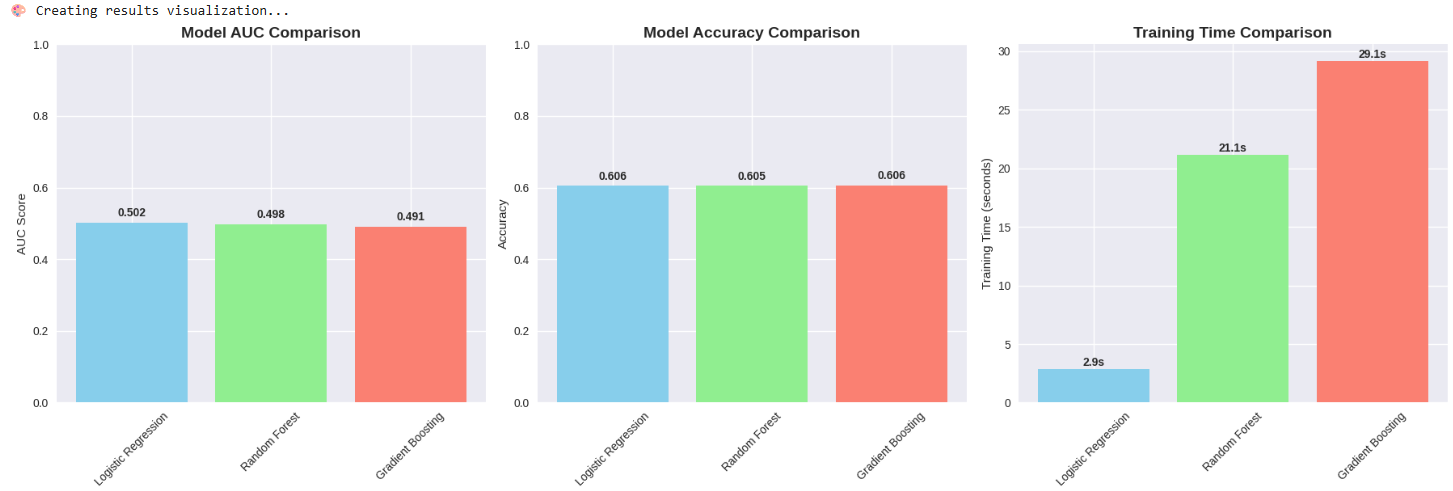
**Output:**

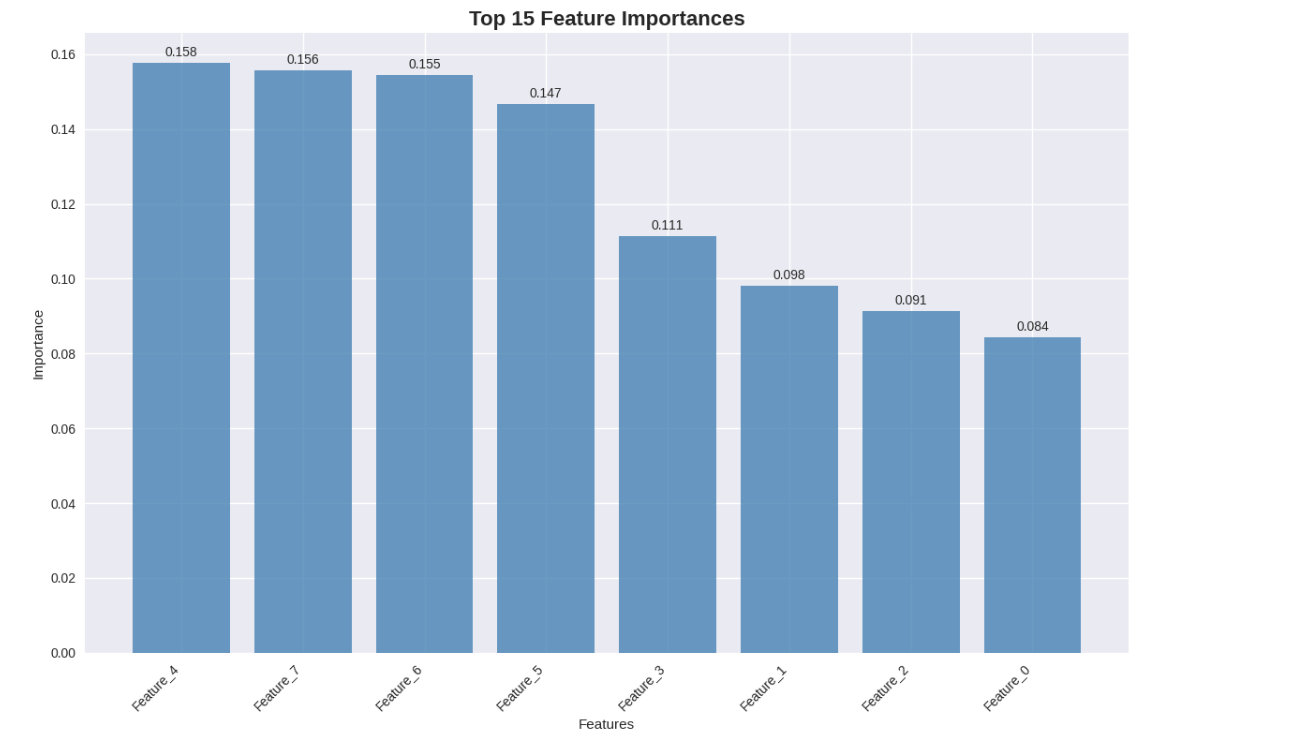


****

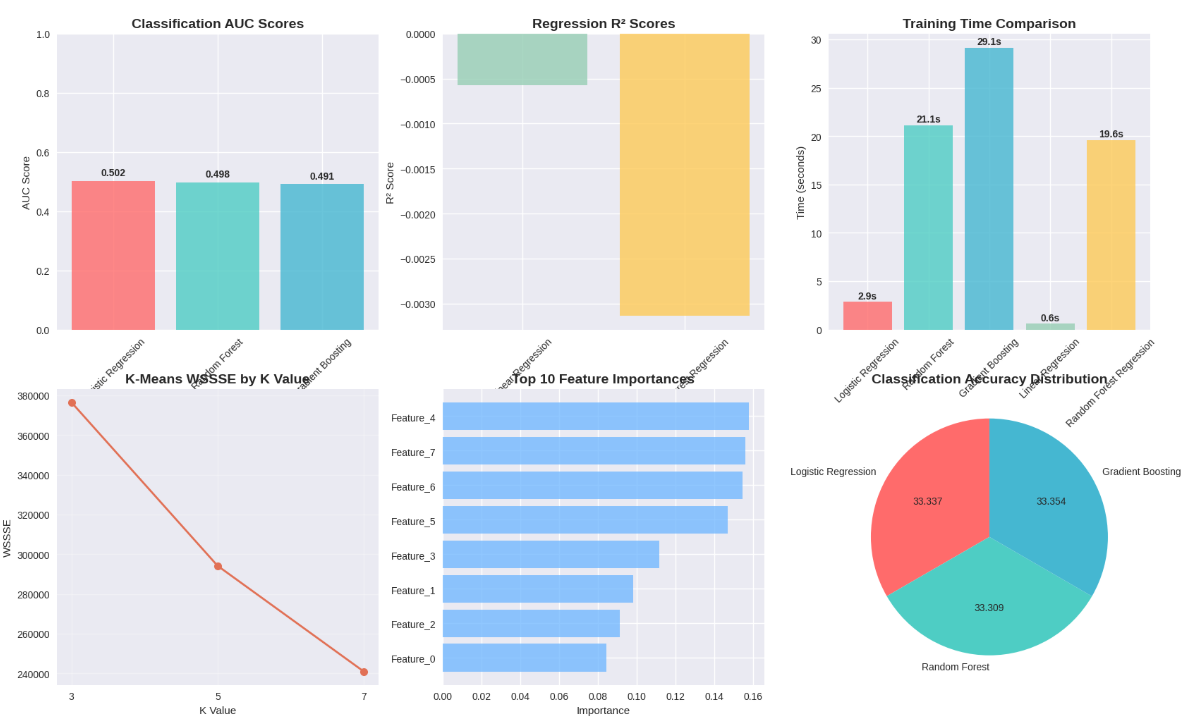
****

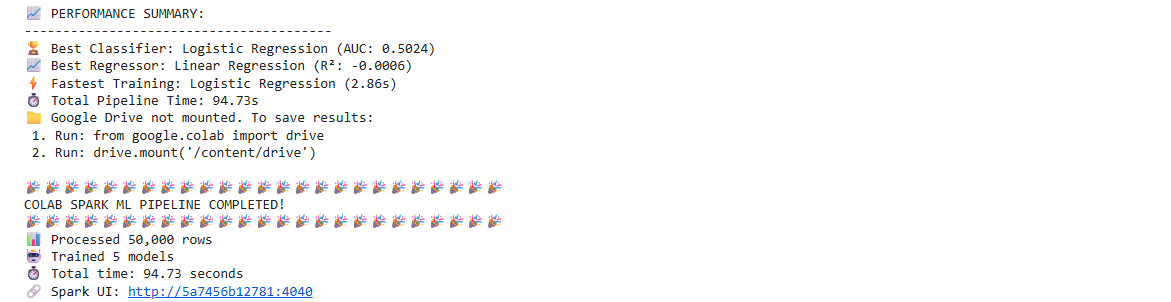
****

****

****

****

****

****

**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)



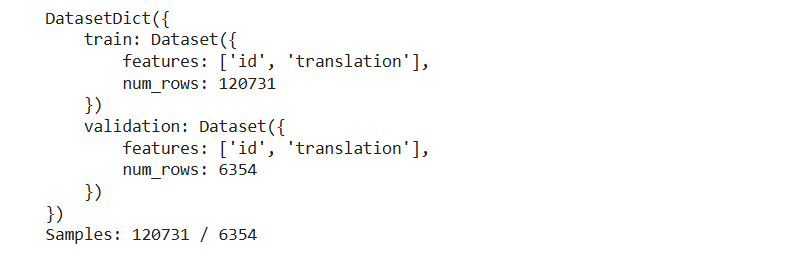
lang\_pair = "en-fr"

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"]))



for k in range(2):

    ex = ds["train"][k]

    print(f"\nSRC: {ex['translation'][src\_lang]}\nTGT: {ex['translation'][tgt\_lang]}")



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)



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))



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"),

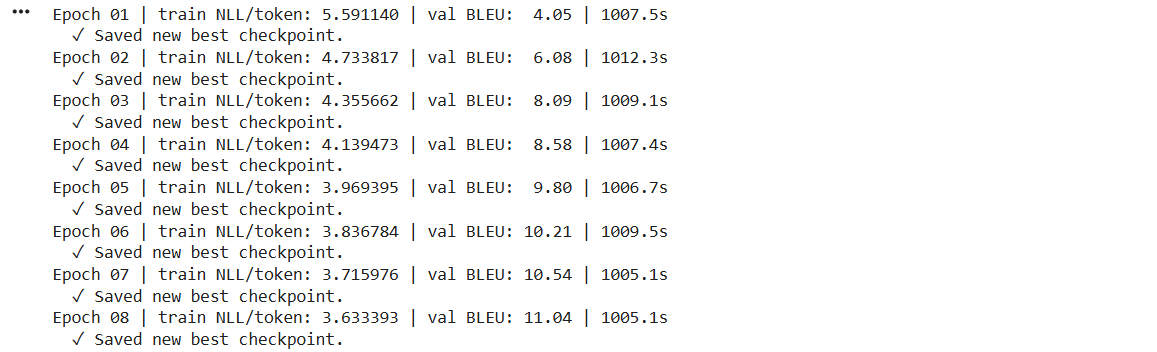
                "cfg": {

                    "EMB\_DIM": EMB\_DIM, "HID\_DIM": HID\_DIM

                }

            }, ckpt\_dir / "best\_seq2seq\_attn.pt")

            print("  ✓ Saved new best checkpoint.")



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")



**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 − μ)**

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")

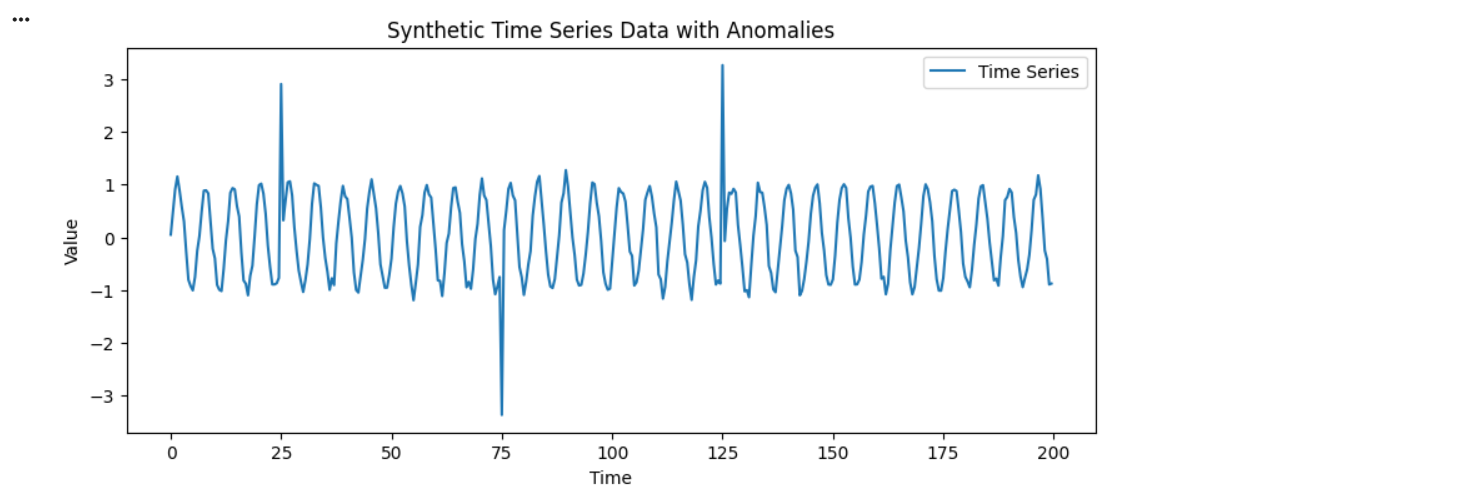
plt.title("Synthetic Time Series Data with Anomalies")

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')

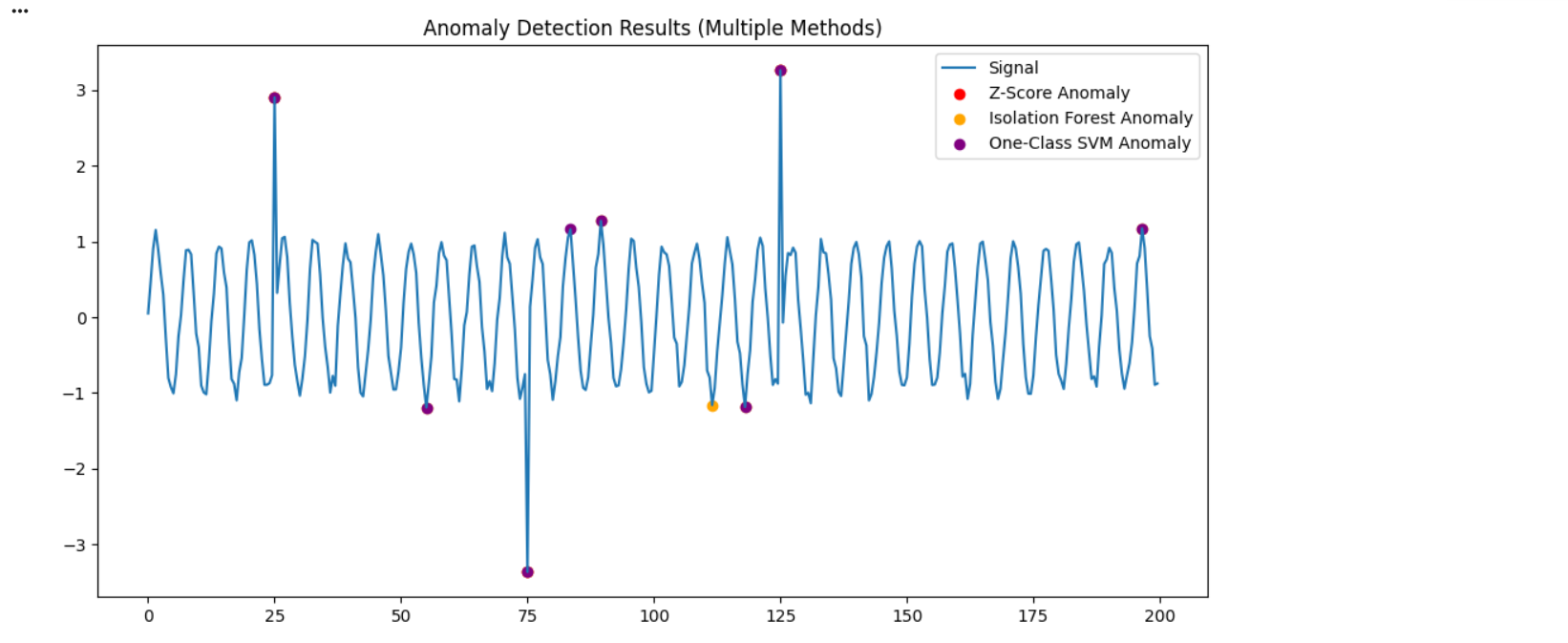
plt.scatter(time[iso\_pred==1], signal[iso\_pred==1], color='orange', label='Isolation Forest Anomaly')

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)

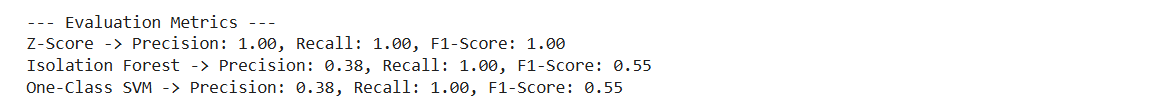
    print(f"{name} -> Precision: {precision:.2f}, Recall: {recall:.2f}, F1-Score: {f1:.2f}")

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")



**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:

**wij = P(Y = yj) / P(A = ai, Y = yj)**

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}")



!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}")



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

)

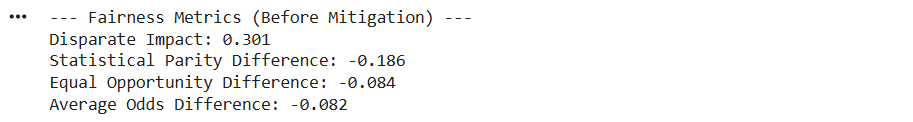
print("\n--- Fairness Metrics (Before Mitigation) ---")

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}")



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}")



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

)

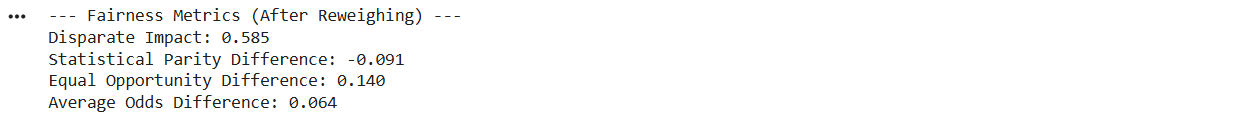
print("\n--- Fairness Metrics (After Reweighing) ---")

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}")



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')

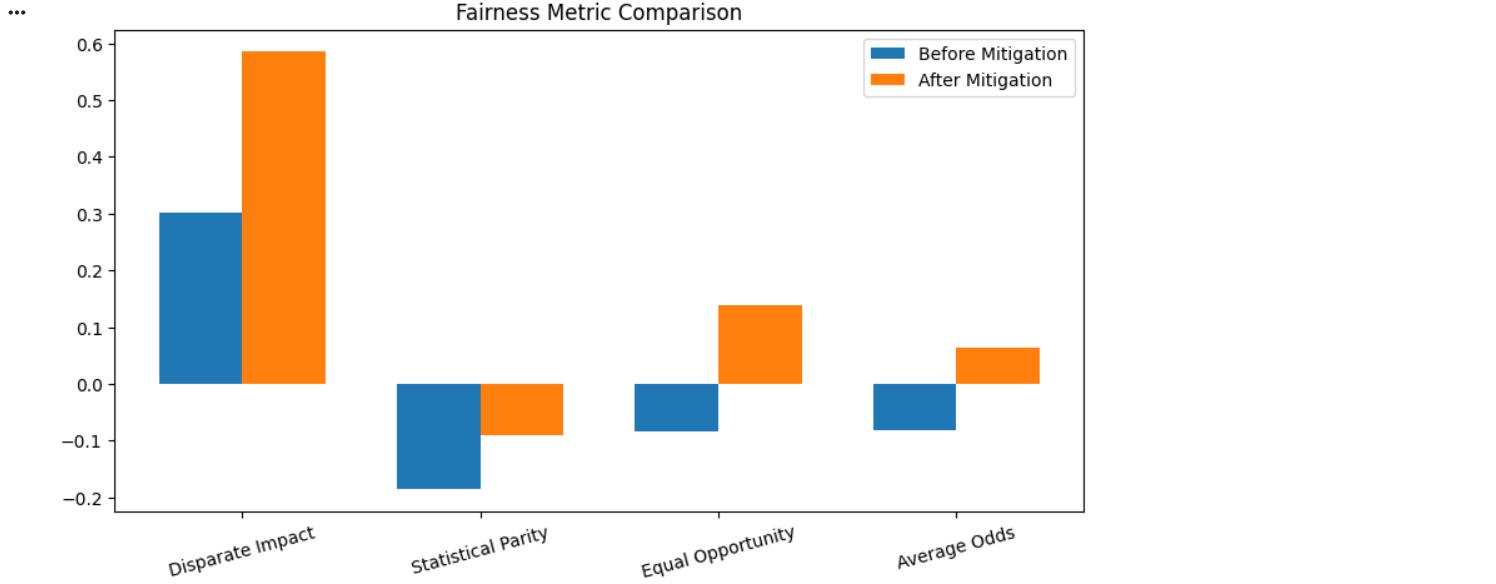
plt.bar(x + width/2, metrics\_after, width, label='After Mitigation')

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())



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))

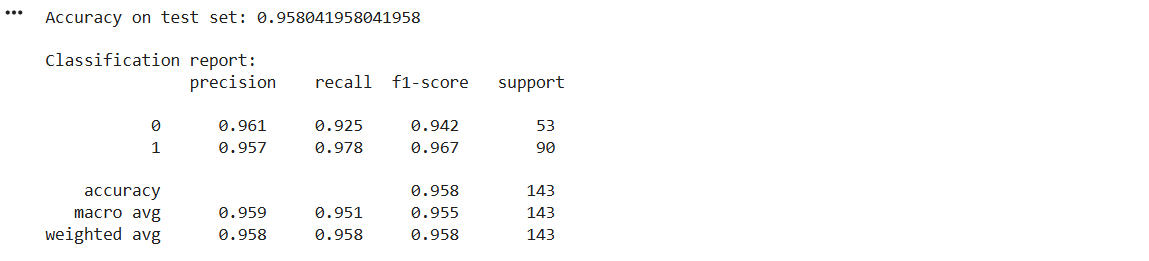
])

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))



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))

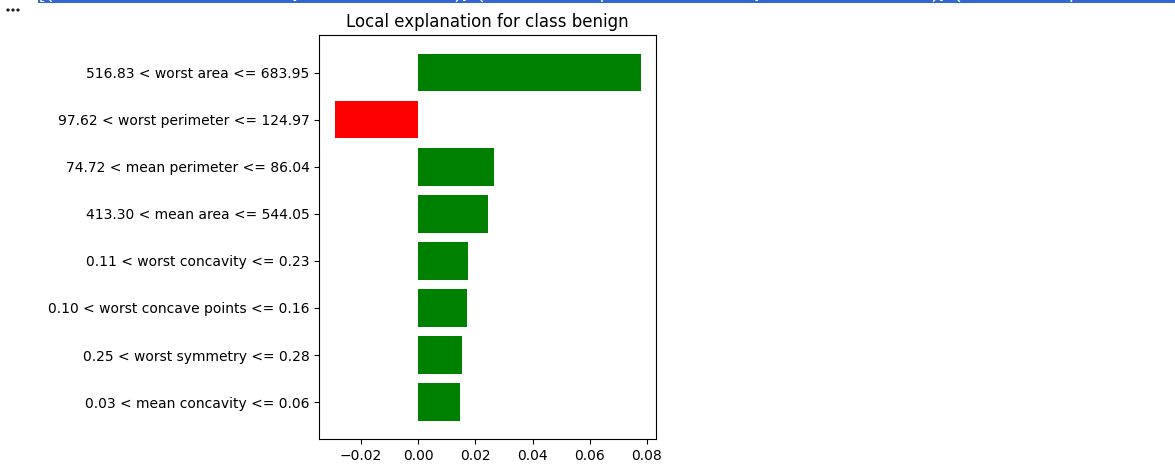
--- LIME explanation (text) ---

[('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],

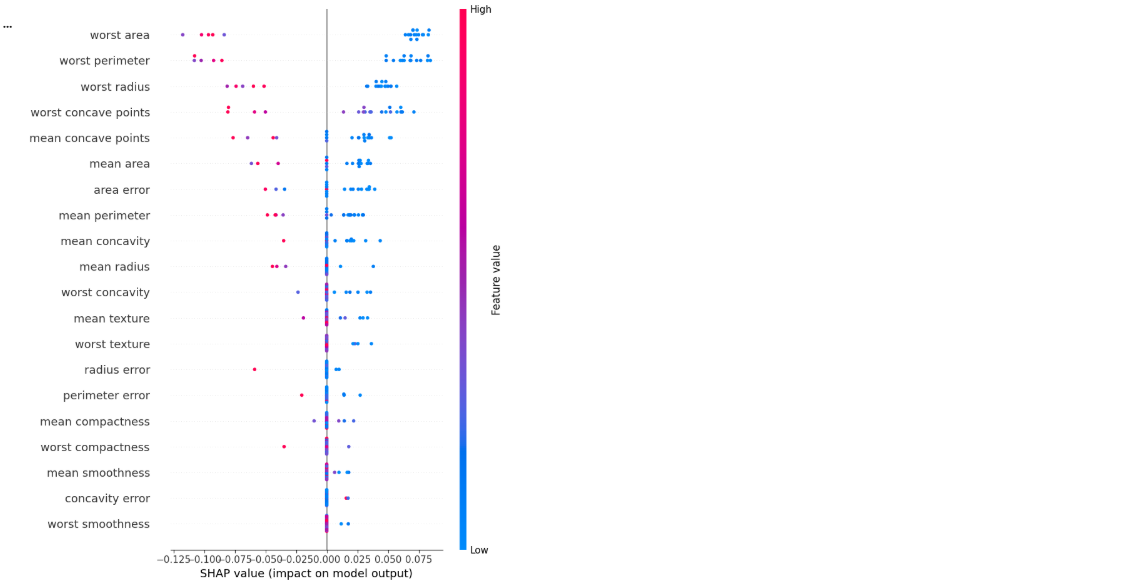
        base\_values=explainer.expected\_value,

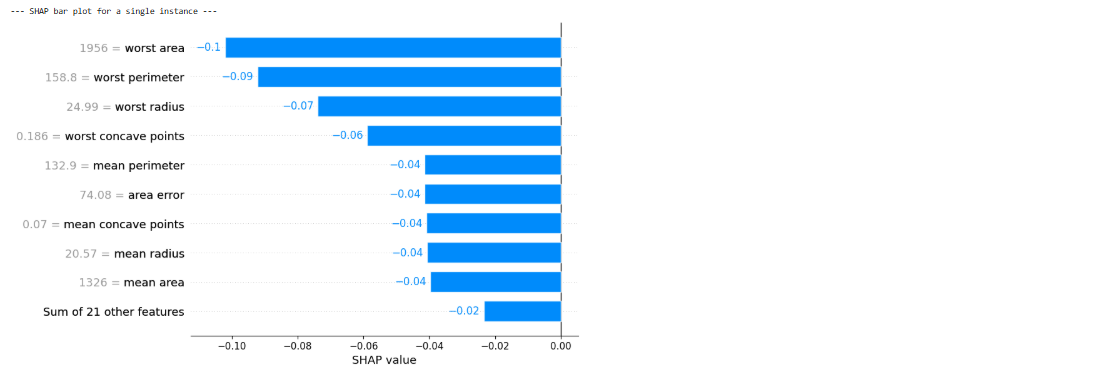
        data=X\_sample.iloc[instance\_idx],

        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 .
* It selects an action according to its policy .
* The environment returns a reward and transitions to the next state .

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

**2. Value and Q-Functions**

* **State Value Function:**
* **Action Value Function (Q-Function):**

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:

**4. Deep Q-Network (DQN)**

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

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

where 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 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...")



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))



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:])

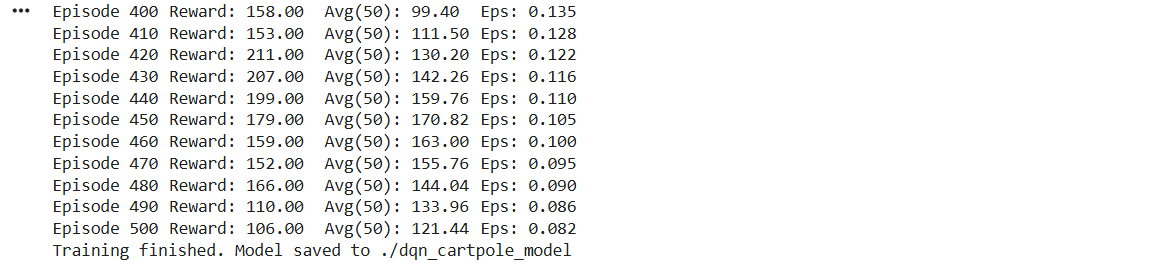
        print(f"Episode {episode}\tReward: {episode\_reward:.2f}\tAvg(50): {avg\_reward:.2f}\tEps: {eps:.3f}")

    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)



plt.figure(figsize=(10,5))

plt.plot(episode\_rewards, label="Episode reward")

plt.xlabel("Episode")

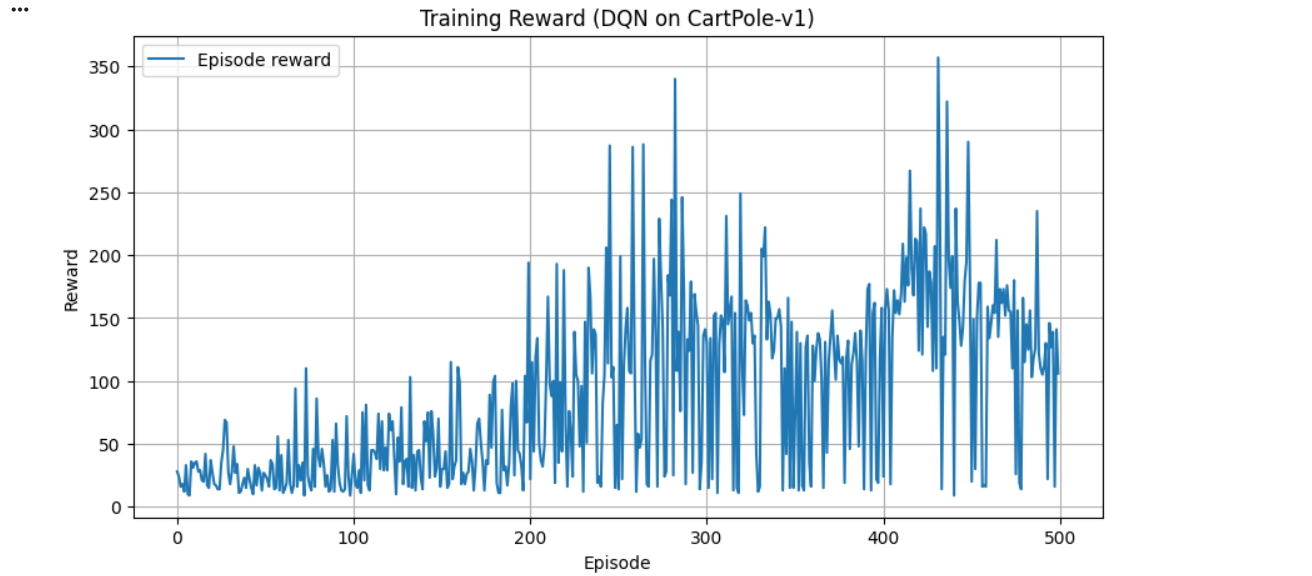
plt.ylabel("Reward")

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()



**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 is -differentially private if, for any two neighboring datasets and , and for all possible outputs :

**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)



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}")



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}")



plt.figure(figsize=(7,4))

plt.plot(eps\_list, np.abs(np.array(laplace\_means) - true\_mean), marker='o')

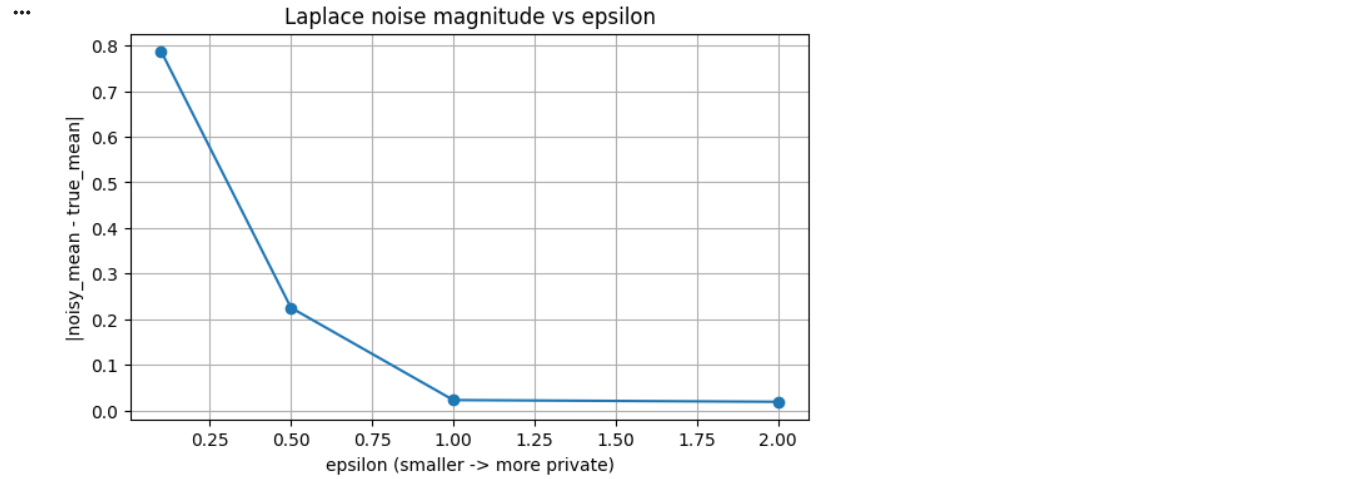
plt.xlabel("epsilon (smaller -> more private)")

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))



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")



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()

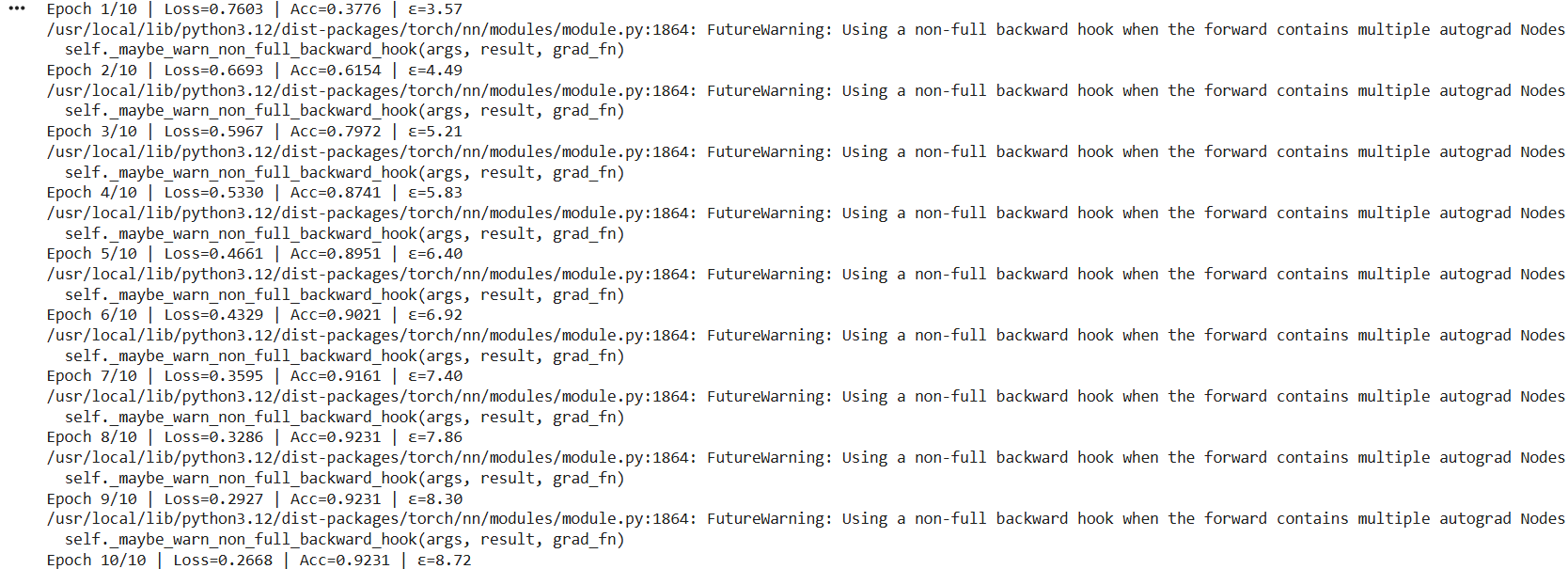
        optimizer.step()

        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}")



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}")



**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.