```
from google.colab import files
uploaded = files.upload()
```

Browse... corrected.gz
corrected.gz(application/x-gzip) - 1409035 bytes, last modified: n/a - 100% done

```
Saving corrected.gz to corrected.gz
```

```
# Show plots inline
%matplotlib inline

# Import required libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.metrics import classification_report, confusion_matrix
import tensorflow as tf
from tensorflow.keras import layers
import time

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Upload the 'corrected.gz' dataset
from google.colab import files
uploaded = files.upload()

# Load the .gz file into a DataFrame (assuming it’s a CSV)
df = pd.read_csv("corrected.gz", header=None)

# Display first 5 rows to verify
df.head()
```

Browse... corrected.gz
corrected.gz(application/x-gzip) - 1409035 bytes, last modified: n/a - 100% done

```
Saving corrected.gz to corrected (1).gz
```

	0	1	2	3	4	5	6	7	8	9	...	32	33	34	35	36	37	38	39
0	0	udp	private	SF	105	146	0	0	0	0	...	254	1.0	0.01	0.00	0.0	0.0	0.0	0.0
1	0	udp	private	SF	105	146	0	0	0	0	...	254	1.0	0.01	0.00	0.0	0.0	0.0	0.0
2	0	udp	private	SF	105	146	0	0	0	0	...	254	1.0	0.01	0.00	0.0	0.0	0.0	0.0
3	0	udp	private	SF	105	146	0	0	0	0	...	254	1.0	0.01	0.00	0.0	0.0	0.0	0.0
4	0	udp	private	SF	105	146	0	0	0	0	...	254	1.0	0.01	0.01	0.0	0.0	0.0	0.0

5 rows × 42 columns

```
# Upload the corrected.gz file from local system
from google.colab import files
uploaded = files.upload()

# Set up dtype dictionary
import numpy as np
import pandas as pd

dtypes = {
    "duration": np.int8,
    "protocol_type": object,
    "service": object,
    "flag": object,
    "src_bytes": np.int8,
    "dst_bytes": np.int8,
    "land": np.int8,
    "wrong_fragment": np.int8,
    "urgent": np.int8,
    "hot": np.int8,
    "m_failed_logins": np.int8,
    "logged_in": np.int8,
    "num_compromised": np.int8,
    "root_shell": np.int8,
    "su_attempted": np.int8,
    "num_root": np.int8,
    "num_file_creations": np.int8,
    "num_shells": np.int8,
    "num_access_files": np.int8,
    "num_outbound_cmds": np.int8,
    "is_host_login": np.int8,
    "is_guest_login": np.int8,
    "count": np.int8,
    "srv_count": np.int8,
    "serror_rate": np.float16,
    "srv_serror_rate": np.float16,
    "rerror_rate": np.float16,
    "srv_rerror_rate": np.float16,
    "same_srv_rate": np.float16,
    "diff_srv_rate": np.float16,
    "srv_diff_host_rate": np.float16,
    "dst_host_count": np.int8,
    "dst_host_srv_count": np.int8,
    "dst_host_same_srv_rate": np.float16,
    "dst_host_diff_srv_rate": np.float16,
    "dst_host_same_src_port_rate": np.float16,
    "dst_host_srv_diff_host_rate": np.float16,
    "dst_host_serror_rate": np.float16,
    "dst_host_srv_serror_rate": np.float16,
    "dst_host_rerror_rate": np.float16,
    "dst_host_srv_rerror_rate": np.float16,
    "label": object
}

# Column names for the dataset
columns = [
    "duration","protocol_type","service","flag","src_bytes","dst_bytes","land",
    "wrong_fragment","urgent","hot","m_failed_logins","logged_in", "num_compromised",
    "root_shell","su_attempted","num_root","num_file_creations","num_shells",
    "num_access_files","num_outbound_cmds","is_host_login","is_guest_login",
    "count","srv_count","serror_rate","srv_serror_rate","rerror_rate","srv_rerror_rate",
    "same_srv_rate","diff_srv_rate","srv_diff_host_rate","dst_host_count",
    "dst_host_srv_count","dst_host_same_srv_rate","dst_host_diff_srv_rate",
    "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate","dst_host_serror_rate",
    "dst_host_srv_serror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate","label"
]

# Read the uploaded corrected.gz file
df = pd.read_csv("corrected.gz", compression="gzip", names=columns, dtype=dtypes)

# Display first few rows
df.head()
```

Browse... corrected.gz

corrected.gz(application/x-gzip) - 1409035 bytes, last modified: n/a - 100% done

```
Saving corrected.gz to corrected (2).gz
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarn
  has_large_values = (abs_vals > 1e6).any()
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarn
  has_large_values = (abs_vals > 1e6).any()
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment
0	0	udp	private	SF	105	-110	0	0
1	0	udp	private	SF	105	-110	0	0
2	0	udp	private	SF	105	-110	0	0
3	0	udp	private	SF	105	-110	0	0
4	0	udp	private	SF	105	-110	0	0

5 rows × 42 columns

```
df.label.value_counts()
```

count

label

label	
smurf.	164091
normal.	60593
neptune.	58001
snmpgetattack.	7741
mailbomb.	5000
guess_passwd.	4367
snmpguess.	2406
satan.	1633
warezmaster.	1602
back.	1098
mscan.	1053
apache2.	794
processtable.	759
saint.	736
portsweep.	354
ipsweep.	306
httptunnel.	158
pod.	87
nmap.	84
buffer_overflow.	22
multihop.	18
named.	17
sendmail.	17
ps.	16
xterm.	13
rootkit.	13
teardrop.	12
xlock.	9
land.	9
xsnoop.	4

ftp_write.	3
perl.	2
phf.	2
udpstorm.	2
worm.	2
loadmodule.	2
sqlattack.	2
imap.	1

dtype: int64

The last executed cell displays the count of each unique value in the 'label' column of the df DataFrame. This shows the distribution of different types of network traffic or attacks within the dataset.

Double-click (or enter) to edit

```
new_features = [
    'dst_bytes',
    'logged_in',
    'count',
    'srv_count',
    'dst_host_count',
    'dst_host_srv_count',
    'dst_host_same_srv_rate',
    'dst_host_same_src_port_rate',
    'label'
]

df_new = df[new_features]

# Display first 5 rows
df_new.head()
```

```
/usr/local/lib/python3.11/dist-packages/pandas/io/formats/format.py:1458: RuntimeWarn
  has_large_values = (abs_vals > 1e6).any()
```

	dst_bytes	logged_in	count	srv_count	dst_host_count	dst_host_srv_count	dst_ho
0	-110	0	1	1	-1	-2	
1	-110	0	1	1	-1	-2	
2	-110	0	1	1	-1	-2	

2	-110	0	1	1	-1	-2
3	-110	0	2	2	-1	-2
4	-110	0	2	2	-1	-2

The last executed cell selects a subset of columns from the df DataFrame and creates a new DataFrame called df_new with these selected columns. The selected columns are: 'dst_bytes', 'logged_in', 'count', 'srv_count', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_same_src_port_rate', and 'label'. Finally, it displays the first 5 rows of the new df_new DataFrame.

```
# 1. Convert 'normal.' to 0 (benign), everything else to 1 (attack)
df.label = df.label.apply(lambda x: 0 if x == 'normal.' else 1)

# 2. Filter only selected features (including label)
df = df[new_features]

# 3. Encode object-type columns (categorical)
from sklearn.preprocessing import LabelEncoder
for column in df.columns:
    if df[column].dtype == object:
        encoded = LabelEncoder()
        df[column] = encoded.fit_transform(df[column])

# 4. Sample 500 benign (label == 0) records for training
df_train = df[df.label == 0].sample(500, random_state=42)

# 5. Remove sampled data from main dataset
df = df.drop(df_train.index)

# 6. Drop the label column from training set
df_train = df_train.drop('label', axis=1)

# 7. Final shape of training set
df_train.shape
```

```
(500, 8)
```

```
Converts the 'label' column: It transforms the 'label' column into a binary classificati
Filters selected features: It keeps only the columns specified in the new_features list
Encodes categorical columns: It iterates through the columns of the modified df DataFram
Samples training data: It selects 500 rows from the df DataFrame where the 'label' is 0
Removes sampled data: It removes the rows that were sampled for df_train from the origir
Drops the label column from training set: It removes the 'label' column from the df_trai
```

```
Displays training set shape: It prints the shape (number of rows and columns) of the fir
```

```
import tensorflow as tf
from tensorflow.keras import layers

# Generator model
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(16, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(16))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(32))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(8))  # output matches df_train features
    return model

# Discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(16, use_bias=False, input_shape=[8]))  # corrected input shape
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Dense(32))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Dense(1))  # Output: real (1) or fake (0)
    return model

# Instantiate models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Loss functions
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Test binary cross-entropy
bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)
loss = bce(tf.ones(4), tf.zeros(4))  # max mismatch, should return high loss
print('Sample BCE loss:', loss.numpy())
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarnin
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Sample BCE loss: 0.6931472
```

```
import tensorflow as tf

bce = tf.keras.losses.BinaryCrossentropy()
loss = bce([1., 1., 1., 1.], [1., 1., 1., 1.])
print('Loss: ', loss.numpy())
```

```
Loss:  1.1920929e-07
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

The last two executed cells are related to setting up the loss functions for the Generative Adversarial Network (GAN) models.

The second to last cell (cell 8U2fIAVu6OCA) calculates and prints a sample Binary Crossentropy (BCE) loss using tf.keras.losses.BinaryCrossentropy(). It provides inputs where the true labels are all 1.0 and the predicted values are also all 1.0. This is a test to show that when the predictions perfectly match the true labels, the BCE loss is very close to zero, as expected.

The last cell (cell G8cWHdJ06fEj) redefines the generator_loss function. This function calculates the loss for the generator model in the GAN. It takes the fake_output (the discriminator's predictions on the generated fake data) as input and uses the cross_entropy loss function (which was defined in cell 1tKbcXVK5oHk as tf.keras.losses.BinaryCrossentropy(from_logits=True)) to compare these predictions to a tensor of ones (tf.ones_like(fake_output)). The goal of the generator is to produce fake data that is so realistic that the discriminator predicts it as real (close to 1), so the generator's loss is minimized when the discriminator's output for fake data is close to 1.

```
@tf.function
def train_step(real_images):
    # real_images: shape (BATCH_SIZE, 8)

    noise = tf.random.normal([BATCH_SIZE, noise_dim])  # input to generator

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_va

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_v
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator

    return gen_loss, disc_loss
    # Convert df_train to tf.data.Dataset
import tensorflow as tf

BATCH_SIZE = 64
noise_dim = 100
EPOCHS = 1000

train_dataset = tf.data.Dataset.from_tensor_slices(df_train.values.astype('float32'))
train_dataset = train_dataset.shuffle(buffer_size=500).batch(BATCH_SIZE, drop_remainder=T
```

```
# Convert df_train to tf.data.Dataset
import tensorflow as tf

BATCH_SIZE = 64
noise_dim = 100
EPOCHS = 1000

train_dataset = tf.data.Dataset.from_tensor_slices(df_train.values.astype('float32'))
train_dataset = train_dataset.shuffle(buffer_size=500).batch(BATCH_SIZE, drop_remainder=Tru
```

The last two executed cells are related to setting up the loss functions for the Generative Adversarial Network (GAN) models.

The second to last cell (cell 8U2fIAVu6OCA) calculates and prints a sample Binary Crossentropy (BCE) loss using tf.keras.losses.BinaryCrossentropy(). It provides inputs where the true labels are all 1.0 and the predicted values are also all 1.0. This is a test to show that when the predictions perfectly match the true labels, the BCE loss is very close to zero, as expected.

The last cell (cell G8cWHdJ06fEj) redefines the generator_loss function. This function calculates the loss for the generator model in the GAN. It takes the fake_output (the discriminator's predictions on the generated fake data) as input and uses the cross_entropy loss function (which was defined in cell 1tKbcXVK5oHk as tf.keras.losses.BinaryCrossentropy(from_logits=True)) to compare these predictions to a tensor of ones (tf.ones_like(fake_output)). The goal of the generator is to produce fake data that is so realistic that the discriminator predicts it as real (close to 1), so the generator's loss is minimized when the discriminator's output for fake data is close to 1.

```
Start coding or generate with AI.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-18-2845037084.py in <cell line: 0>()
      1 for epoch in range(EPOCHS):
      2     for batch in train_dataset:
----> 3         gen_loss, disc_loss = train_step(batch)
      4 
      5     if epoch % 100 == 0:

                              1 frames
/tmp/__autograph_generated_filev9h_xfvc.py in tf__train_step(real_images)
     17                 gradients_of_generator =
ag__.converted_call(ag__.ld(gen_tape).gradient, (ag__.ld(gen_loss),
ag__.ld(generator).trainable_variables), None, fscope)
     18                 gradients_of_discriminator =
ag__.converted_call(ag__.ld(disc_tape).gradient, (ag__.ld(disc_loss),
ag__.ld(discriminator).trainable_variables), None, fscope)
---> 19
ag__.converted_call(ag__.ld(generator_optimizer).apply_gradients,
(ag__.converted_call(ag__.ld(zip), (ag__.ld(gradients_of_generator),
ag__.ld(generator).trainable_variables), None, fscope),), None, fscope)
     20
ag__.converted_call(ag__.ld(discriminator_optimizer).apply_gradients,
(ag__.converted_call(ag__.ld(zip), (ag__.ld(gradients_of_discriminator),
ag__.ld(discriminator).trainable_variables), None, fscope),), None, fscope)
     21                 try:

NameError: in user code:
```

Next steps: Explain error

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import numpy as np

# Generator model
def make_generator_model():
    model = tf.keras.Sequential([
        layers.Dense(16, use_bias=False, input_shape=(100,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Dense(16),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Dense(32),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Dense(8)  # Output shape matches df_train (8 features)
    ])
    return model

# Discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential([
        layers.Dense(16, use_bias=False, input_shape=(8,)),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
        layers.Dense(32),
        layers.LeakyReLU(),
        layers.Dropout(0.3),
        layers.Dense(1)  # Real (1) or Fake (0)
    ])
    return model

# Loss functions
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Training parameters
```

```
EPOCHS = 1000
BATCH_SIZE = 64
noise_dim = 100

# Convert df_train to TensorFlow Dataset
train_dataset = tf.data.Dataset.from_tensor_slices(df_train.values.astype('float32'))
train_dataset = train_dataset.shuffle(buffer_size=500).batch(BATCH_SIZE, drop_remainder=T

# Training step
@tf.function
def train_step(real_images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_va

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_v
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator

    return gen_loss, disc_loss

# Training loop
for epoch in range(EPOCHS):
    for batch in train_dataset:
        gen_loss, disc_loss = train_step(batch)

    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Gen Loss: {gen_loss.numpy():.4f}, Disc Loss: {disc_loss.nu
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarnin
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 0, Gen Loss: 0.6000, Disc Loss: 3.7832
Epoch 100, Gen Loss: 0.5142, Disc Loss: 1.5397
Epoch 200, Gen Loss: 0.6550, Disc Loss: 1.1556
Epoch 300, Gen Loss: 0.6793, Disc Loss: 1.2715
Epoch 400, Gen Loss: 0.8663, Disc Loss: 0.9204
Epoch 500, Gen Loss: 0.9788, Disc Loss: 1.0949
Epoch 600, Gen Loss: 1.1003, Disc Loss: 1.0378
Epoch 700, Gen Loss: 1.1123, Disc Loss: 0.7693
Epoch 800, Gen Loss: 1.0568, Disc Loss: 0.9530
Epoch 900, Gen Loss: 1.1212, Disc Loss: 0.9956
```

```
Imports Libraries: Imports tensorflow and numpy.
Defines Generator Model: The make_generator_model function creates a sequential Keras mo
Defines Discriminator Model: The make_discriminator_model function creates another seque
Defines Loss Functions:
    cross_entropy: Uses tf.keras.losses.BinaryCrossentropy with from_logits=True for cal
    discriminator_loss: Calculates the total loss for the discriminator by summing the b
    generator_loss: Calculates the loss for the generator. It uses the binary cross-entr
Defines Optimizers: Initializes Adam optimizers for both the generator and the discrimin
Instantiates Models: Creates instances of the generator and discriminator models.
Sets Training Parameters: Defines EPOCHS, BATCH_SIZE, and noise_dim.
Prepares Training Data: Converts the df_train DataFrame into a TensorFlow Dataset, shuff
Defines train_step Function: This function is decorated with @tf.function for performanc
    It generates random noise for the generator.
    It uses tf.GradientTape to record operations for automatic differentiation.
    It generates fake images using the generator.
    It gets the discriminator's output for both real and fake images.
    It calculates the generator and discriminator losses.
    It computes the gradients of the losses with respect to the trainable variables of e
    It applies the gradients to update the model weights using the respective optimizers
    It returns the generator and discriminator losses.
Training Loop:
    It iterates through the specified number of EPOCHS.
    Inside the epoch loop, it iterates through batches in the train_dataset.
    For each batch, it calls the train_step function to perform one training step for th
    It prints the generator and discriminator losses every 100 epochs to track training 
```

This cell essentially sets up and runs the training process for the GAN to generate synthetic data similar to the benign traffic in your dataset.

```
import time

# Initialize history to track loss
history = {
    'gen': [],
    'dis': []
}

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for batch in dataset:
            gen_loss, dis_loss = train_step(batch)
            gen_loss, dis_loss = train_step(batch)

        # Store the last batch's loss (can modify to average if needed)
        history['gen'].append(gen_loss.numpy())
        history['dis'].append(dis_loss.numpy())

        print(f"Epoch {epoch+1}, Gen Loss: {gen_loss:.4f}, Disc Loss: {dis_loss:.4f}, Tim
# Convert df_train to tf.data.Dataset with batching
x_train = df_train.values.astype('float32')

train_dataset = tf.data.Dataset.from_tensor_slices(x_train)
train_dataset = train_dataset.shuffle(500).batch(BATCH_SIZE, drop_remainder=True)

# Train
train(train_dataset, EPOCHS)
```

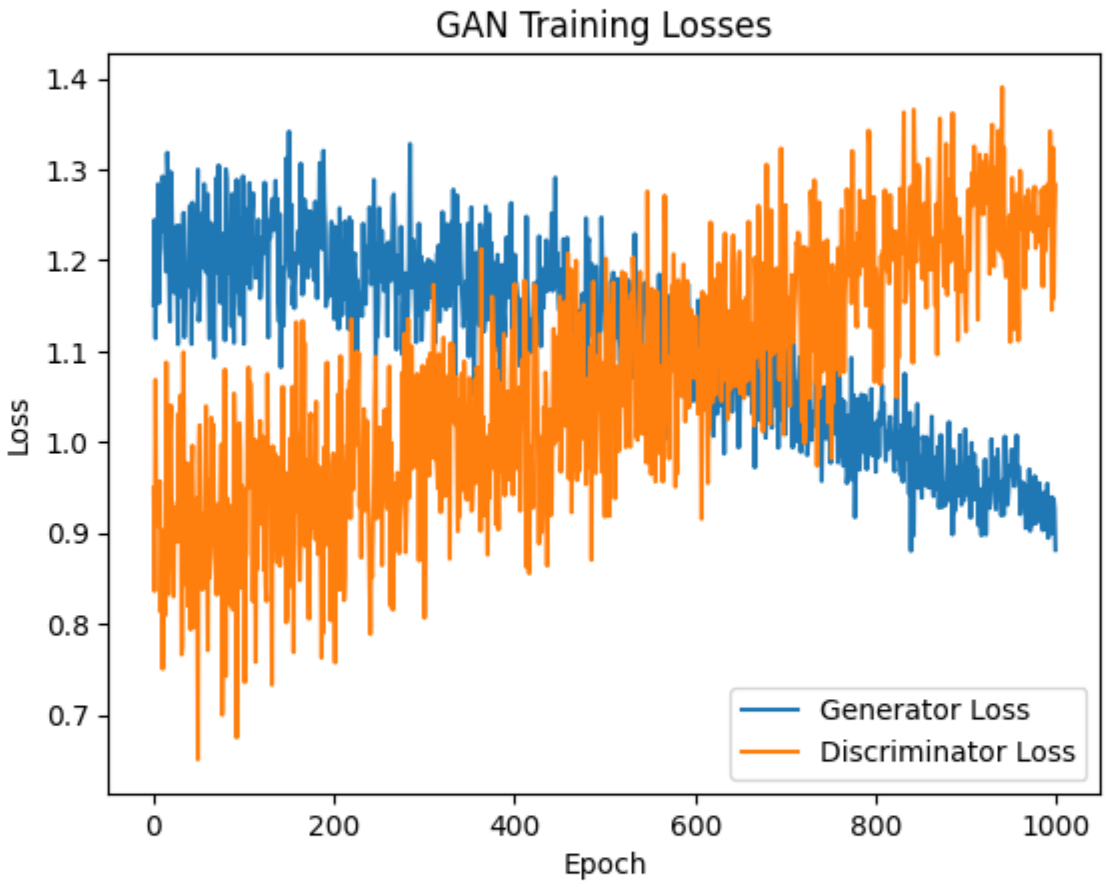
```
Epoch 1, Gen Loss: 1.1507, Disc Loss: 0.9493, Time: 0.02s
Epoch 2, Gen Loss: 1.2445, Disc Loss: 0.8360, Time: 0.02s
Epoch 3, Gen Loss: 1.1141, Disc Loss: 1.0685, Time: 0.02s
Epoch 4, Gen Loss: 1.2112, Disc Loss: 0.9081, Time: 0.02s
Epoch 5, Gen Loss: 1.1905, Disc Loss: 0.9575, Time: 0.02s
Epoch 6, Gen Loss: 1.2838, Disc Loss: 0.9171, Time: 0.02s
Epoch 7, Gen Loss: 1.1528, Disc Loss: 0.9070, Time: 0.02s
Epoch 8, Gen Loss: 1.2037, Disc Loss: 0.9568, Time: 0.02s
Epoch 9, Gen Loss: 1.2136, Disc Loss: 0.8128, Time: 0.02s
Epoch 10, Gen Loss: 1.2498, Disc Loss: 0.9031, Time: 0.02s
Epoch 11, Gen Loss: 1.2923, Disc Loss: 0.7505, Time: 0.06s
Epoch 12, Gen Loss: 1.2851, Disc Loss: 0.8551, Time: 0.02s
Epoch 13, Gen Loss: 1.2316, Disc Loss: 0.8566, Time: 0.02s
Epoch 14, Gen Loss: 1.2821, Disc Loss: 0.8094, Time: 0.02s
Epoch 15, Gen Loss: 1.1874, Disc Loss: 1.0876, Time: 0.02s
Epoch 16, Gen Loss: 1.3183, Disc Loss: 0.8799, Time: 0.02s
Epoch 17, Gen Loss: 1.2913, Disc Loss: 0.8323, Time: 0.02s
Epoch 18, Gen Loss: 1.1902, Disc Loss: 0.8608, Time: 0.02s
Epoch 19, Gen Loss: 1.1326, Disc Loss: 1.0113, Time: 0.02s
Epoch 20, Gen Loss: 1.2971, Disc Loss: 1.0405, Time: 0.06s
Epoch 21, Gen Loss: 1.2670, Disc Loss: 0.9337, Time: 0.03s
Epoch 22, Gen Loss: 1.1994, Disc Loss: 0.9330, Time: 0.06s
Epoch 23, Gen Loss: 1.2282, Disc Loss: 0.8300, Time: 0.06s
Epoch 24, Gen Loss: 1.1644, Disc Loss: 0.9142, Time: 0.06s
Epoch 25, Gen Loss: 1.2032, Disc Loss: 0.9270, Time: 0.03s
Epoch 26, Gen Loss: 1.2377, Disc Loss: 0.8902, Time: 0.06s
Epoch 27, Gen Loss: 1.1653, Disc Loss: 0.9091, Time: 0.03s
Epoch 28, Gen Loss: 1.1076, Disc Loss: 0.9444, Time: 0.06s
Epoch 29, Gen Loss: 1.2382, Disc Loss: 1.0318, Time: 0.06s
Epoch 30, Gen Loss: 1.1642, Disc Loss: 0.8903, Time: 0.06s
Epoch 31, Gen Loss: 1.1694, Disc Loss: 1.0517, Time: 0.06s
Epoch 32, Gen Loss: 1.1599, Disc Loss: 0.7662, Time: 0.03s
Epoch 33, Gen Loss: 1.2009, Disc Loss: 0.7747, Time: 0.03s
Epoch 34, Gen Loss: 1.2521, Disc Loss: 1.0988, Time: 0.06s
Epoch 35, Gen Loss: 1.1151, Disc Loss: 0.9388, Time: 0.06s
Epoch 36, Gen Loss: 1.1560, Disc Loss: 0.9062, Time: 0.06s
Epoch 37, Gen Loss: 1.1864, Disc Loss: 0.8429, Time: 0.03s
Epoch 38, Gen Loss: 1.1796, Disc Loss: 0.9760, Time: 0.06s
```

```
    Epoch 38, Gen Loss: 1.1706, Disc Loss: 0.9769, Time: 0.06s
    Epoch 39, Gen Loss: 1.1937, Disc Loss: 0.8197, Time: 0.03s
    Epoch 40, Gen Loss: 1.2325, Disc Loss: 0.9360, Time: 0.06s
    Epoch 41, Gen Loss: 1.2597, Disc Loss: 0.8790, Time: 0.06s
    Epoch 42, Gen Loss: 1.2004, Disc Loss: 0.7937, Time: 0.06s
    Epoch 43, Gen Loss: 1.1078, Disc Loss: 0.9737, Time: 0.03s
    Epoch 44, Gen Loss: 1.2633, Disc Loss: 0.9962, Time: 0.06s
    Epoch 45, Gen Loss: 1.1553, Disc Loss: 0.9244, Time: 0.06s
    Epoch 46, Gen Loss: 1.1622, Disc Loss: 0.7965, Time: 0.06s
    Epoch 47, Gen Loss: 1.2504, Disc Loss: 0.9026, Time: 0.04s
    Epoch 48, Gen Loss: 1.2179, Disc Loss: 0.9006, Time: 0.06s
    Epoch 49, Gen Loss: 1.1876, Disc Loss: 0.9381, Time: 0.04s
    Epoch 50, Gen Loss: 1.2998, Disc Loss: 0.6508, Time: 0.06s
    Epoch 51, Gen Loss: 1.1332, Disc Loss: 0.9875, Time: 0.06s
    Epoch 52, Gen Loss: 1.1434, Disc Loss: 1.0185, Time: 0.03s
    Epoch 53, Gen Loss: 1.2504, Disc Loss: 0.8426, Time: 0.04s
    Epoch 54, Gen Loss: 1.1912, Disc Loss: 0.9214, Time: 0.06s
    Epoch 55, Gen Loss: 1.2090, Disc Loss: 0.8377, Time: 0.06s
    Epoch 56, Gen Loss: 1.2478, Disc Loss: 0.8410, Time: 0.04s
    Epoch 57, Gen Loss: 1.2838, Disc Loss: 0.8655, Time: 0.06s
    Epoch 58, Gen Loss: 1.1909, Disc Loss: 0.9604, Time: 0.06s
```

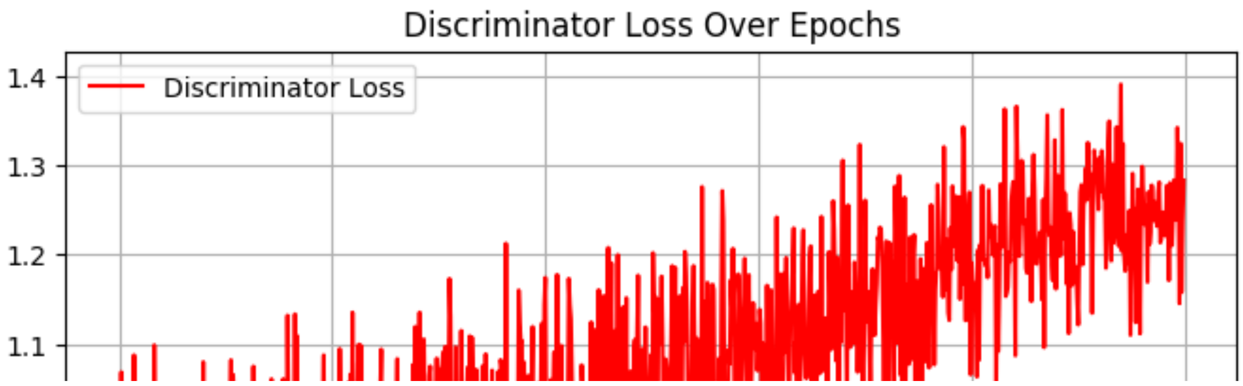
```
Imports time: This is used to measure the time taken for each epoch.
Initializes history: A dictionary called history is created to store the generator and d
Defines the train function: This function takes the dataset (the TensorFlow dataset crea
    It iterates through the specified number of epochs.
    At the beginning of each epoch, it records the start time.
    It then iterates through each batch in the dataset.
    For each batch, it calls the train_step function (defined in the previous cell) to p
    After processing all batches in an epoch, it appends the generator and discriminator
    Finally, it prints the epoch number, the generator and discriminator losses for the
Prepares Training Data: This part is repeated from the previous cell. It converts the df
Calls the train function: It starts the training process by calling the train function w
```

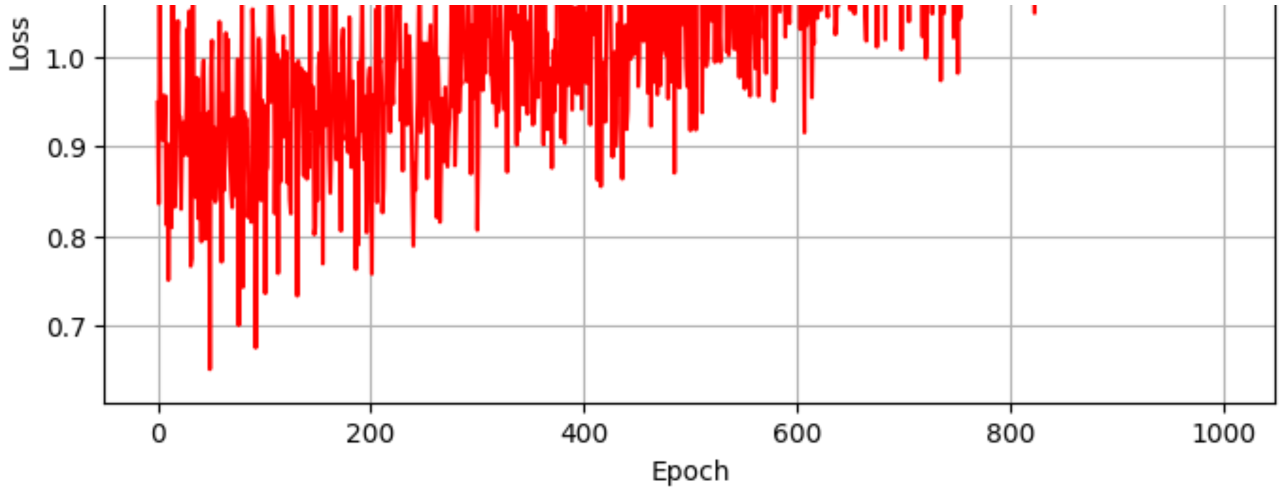
In essence, this cell orchestrates the training of the GAN by repeatedly feeding batches of real data to the train_step function, which updates the generator and discriminator based on their performance in distinguishing between real and generated data. The printed output shows the progress of the training process by displaying the losses and time for each epoch.

```
import matplotlib.pyplot as plt

plt.plot(history['gen'], label='Generator Loss')
plt.plot(history['dis'], label='Discriminator Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('GAN Training Losses')
plt.show()
```



```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
plt.plot(history['dis'], label='Discriminator Loss', color='red')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Discriminator Loss Over Epochs')
plt.legend()
plt.grid(True)
plt.show()
```



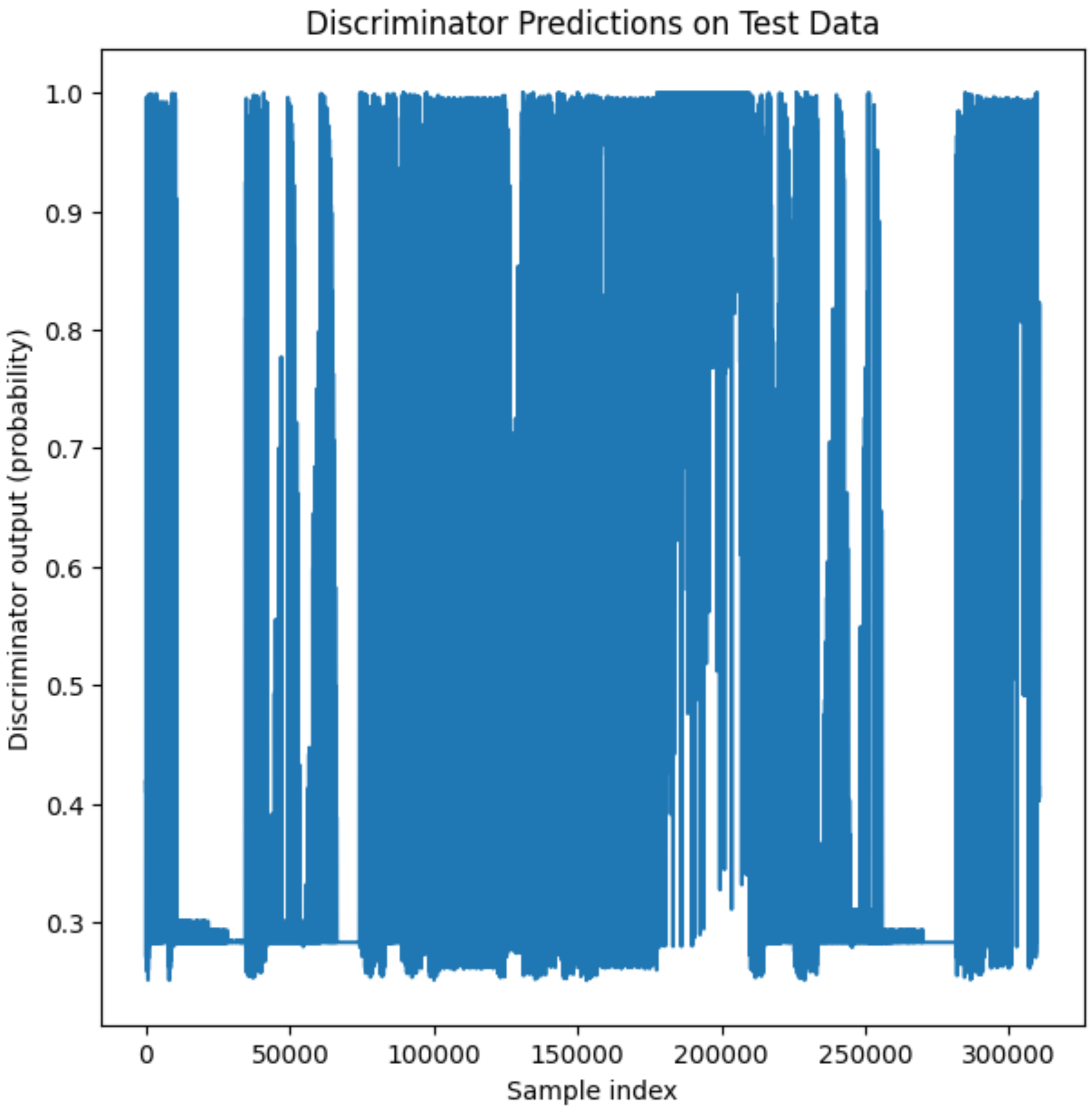


```
Imports matplotlib.pyplot: This line imports the necessary library for plotting and give
Creates a figure: plt.figure(figsize=(8, 5)) creates a new figure for the plot with a sp
Plots Discriminator Loss: plt.plot(history['dis'], label='Discriminator Loss', color='re
Sets X-axis Label: plt.xlabel('Epoch') sets the label for the x-axis to 'Epoch'.
Sets Y-axis Label: plt.ylabel('Loss') sets the label for the y-axis to 'Loss'.
Sets Plot Title: plt.title('Discriminator Loss Over Epochs') sets the title of the plot.
Displays Legend: plt.legend() displays the legend on the plot, which shows the label for
Adds Grid: plt.grid(True) adds a grid to the plot for better readability.
Shows Plot: plt.show() displays the generated plot.
```

This plot helps in visualizing how the discriminator's ability to distinguish between real and fake data changes over the training process. A fluctuating loss or a loss that settles around a certain value might indicate the training dynamics of the GAN.

```
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import average_precision_score, accuracy_score, recall_score, f1_sco

# Prepare test data (no extra reshape, just numpy float32 array)
x_test = df.drop('label', axis=1).values.astype('float32')
y_test = df['label'].values

# Get raw discriminator output (logits)
y_pred_logits = discriminator.predict(x_test)

# Convert logits to probabilities with sigmoid
y_pred_prob = tf.sigmoid(y_pred_logits).numpy().reshape(-1)

# Plot discriminator output probabilities for each test sample
plt.figure(figsize=(7,7))
plt.plot(range(len(y_pred_prob)), y_pred_prob)
plt.ylabel('Discriminator output (probability)')
plt.xlabel('Sample index')
plt.title('Discriminator Predictions on Test Data')
plt.show()

# Threshold at 0.5 to get predicted labels
y_pred_label = (y_pred_prob > 0.5).astype(int)

# Calculate and print metrics
print("Accuracy:", accuracy_score(y_test, y_pred_label))
print("Recall:", recall_score(y_test, y_pred_label))
print("F1 Score:", f1_score(y_test, y_pred_label))
print("Average Precision:", average_precision_score(y_test, y_pred_prob))
```

```
9705/9705 ━━━━━━━━━━━━━━━━━━━━ 14s 1ms/step
```



```
Accuracy: 0.3925816912430079
```

```
Recall: 0.33335462952610645
F1 Score: 0.4695533326396486
Average Precision: 0.8307500547710293
```

```
import numpy as np
from sklearn.metrics import f1_score, accuracy_score, average_precision_score, recall_sco

# Convert logits to probabilities
y_pred_prob = tf.sigmoid(y_pred_logits).numpy().reshape(-1)

prob_thresholds = [-0.2, -0.1, -0.05, 0, 0.005, 0.1]

for p in prob_thresholds:
    # Binary predictions using threshold p on probabilities
    pred_value = [1 if prob > p else 0 for prob in y_pred_prob]

    f1 = f1_score(y_test, pred_value)
    acc = accuracy_score(y_test, pred_value)
    precision = average_precision_score(y_test, y_pred_prob)  # Use probs, not hard preds
    recall = recall_score(y_test, pred_value)

    print(f'prob = {p:.3f} | f1 score = {f1:.4f} | accuracy = {acc:.4f} | precision = {pr
    print(f'Counts -> 0: {pred_value.count(0)}, 1: {pred_value.count(1)}\n')
```

```
prob = -0.200 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529

prob = -0.100 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529

prob = -0.050 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529

prob = 0.000 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529

prob = 0.005 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529

prob = 0.100 | f1 score = 0.8929 | accuracy = 0.8065 | precision = 0.8308 | recall =
Counts -> 0: 0, 1: 310529
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Step 1: Prepare X, y
X = df.drop('label', axis=1)
y = df['label']

# Step 2: Encode categorical columns if any (example if columns are object dtype)
for col in X.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    X[col] = le.fit_transform(X[col])

# Step 3: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42,

# Step 4: Feature scaling (important for KNN and Logistic Regression)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 5: Initialize models
rf = RandomForestClassifier(random_state=42)
knn = KNeighborsClassifier()
lr = LogisticRegression(max_iter=1000, random_state=42)

# Step 6: Train models
rf.fit(X_train, y_train)             # Random Forest on original scale (tree-based, no sc
knn.fit(X_train_scaled, y_train)    # KNN on scaled data
lr.fit(X_train_scaled, y_train)     # Logistic Regression on scaled data

# Step 7: Predict
rf_preds = rf.predict(X_test)
knn_preds = knn.predict(X_test_scaled)
lr_preds = lr.predict(X_test_scaled)

# Step 8: Calculate accuracy
print("Random Forest Accuracy:", accuracy_score(y_test, rf_preds))
print("KNN Accuracy:", accuracy_score(y_test, knn_preds))
print("Logistic Regression Accuracy:", accuracy_score(y_test, lr_preds))
```

```
Random Forest Accuracy: 0.9785313281593834
KNN Accuracy: 0.972423491020728
Logistic Regression Accuracy: 0.8996983651606394
```

```
Imports Libraries: Imports numpy, pandas,
LabelEncoder, StandardScaler, classificat
accuracy_score from scikit-learn.
Prepares X, y: Separates the features (X)
```

```
Imports Libraries: Imports numpy, pandas,
Prepares X, y: Separates the features (X)
```

```
    which is the 'label' column) from the df
    Encodes categorical columns: Although you
    'object' type columns at this stage (as t
    this step is included as a general practi
    categorical features in X.
    Train-test split: Splits the X and y data
    sets (X_train, X_test, y_train, y_test) w
    random_state=42 ensures reproducibility,
    the proportion of classes in y is maintai
    testing sets.
    Feature scaling: Initializes a StandardSc
    training data (X_train) to learn the mear
    then transforms both the training and tes
    X_test_scaled) by centering and scaling t
    important for algorithms like KNN and Log
    sensitive to feature scales.
    Initialize models: Initializes instances
    KNeighborsClassifier, and LogisticRegress
    states for reproducibility.
    Train models: Trains each of the initiali
    training data (X_train for Random Forest,
    and Logistic Regression).
    Predict: Makes predictions on the test da
    (rf_preds, knn_preds, lr_preds). Note tha
    Logistic Regression are made on the scale
    Calculate accuracy: Calculates and prints
    model by comparing the predicted labels t
    (y_test).

This cell trains and evaluates three differer
learning models on the dataset to provide a b
comparison, separate from the GAN anomaly det
```

```
Encodes categorical columns: Although your
Train-test split: Splits the X and y data
Feature scaling: Initializes a StandardSca
Initialize models: Initializes instances o
Train models: Trains each of the initializ
Predict: Makes predictions on the test data
Calculate accuracy: Calculates and prints
```

This cell trains and evaluates three different traditional machine learning models on the dataset to provide a baseline performance comparison, separate from the GAN anomaly detection approach.