

```

1  #%%
2  # Cell 0: This Cell will serve to load any lybraries I will need throughout my
   project. This Helps me keep everything neat.
3
4  # Basic data handling and computation
5  import pandas as pd
6  import numpy as np
7
8  # Data visualization
9  import matplotlib.pyplot as plt
10 import seaborn as sns
11
12 # Preprocessing
13 from sklearn.model_selection import train_test_split
14 from sklearn.preprocessing import StandardScaler, LabelEncoder
15 from sklearn.metrics import classification_report, confusion_matrix,
   accuracy_score
16 from sklearn.ensemble import RandomForestClassifier
17
18
19 # PyTorch for model building and training
20 import torch
21 import torch.nn as nn
22 import torch.nn.functional as F
23 from torch.utils.data import DataLoader, TensorDataset
24 import torch.optim as optim
25
26 # Additional tools
27 import os # For directory and file operations
28 import sys # For system-specific parameters and functions
29 #%%
30 #Cell 1: Import new cleaned CSV
31
32 # Replace the file path with your specific file location
33 file_path = r'C:\Users\gsmit\OneDrive\Desktop\CS691 Project Codename
   Prayer\cleaned_datasetV2.csv'
34 df = pd.read_csv(file_path)
35
36 # Display the first few rows to ensure it's loaded correctly
37 print(df.head())
38
39 #%%
40 # Cell 2: Check if CUDA is available and set the device accordingly
41 # Specify the GPU device
42 if torch.cuda.is_available():

```

```

43 print("Available CUDA devices:")
44 for i in range(torch.cuda.device_count()):
45     print(f"Device {i}: {torch.cuda.get_device_name(i)} with {torch.cuda.
get_device_properties(i).total_memory / 1e9} GB")
46 else:
47     print("No CUDA devices available.")
48
49 print(torch.cuda.device_count())
50 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
51 print(f"Using {device} device")
52 ###
53 # Cell 3: Print the 'Label' feature
54
55 print("Unique labels in the dataset:", df[' Label'].unique()) # Note the leading
space in ' Label'
56 print("Value counts of each label:\n", df[' Label'].value_counts()) # Note the
leading space in ' Label'
57
58 class_distribution = df[' Label'].value_counts()
59
60 plt.figure(figsize=(12, 8))
61 sns.barplot(x=class_distribution.values, y=class_distribution.index, palette='
viridis')
62 plt.title('Distribution of Network Traffic Types', fontsize=16)
63 plt.xlabel('Number of Instances', fontsize=14)
64 plt.ylabel('Traffic Type', fontsize=14)
65 plt.xticks(fontsize=12)
66 plt.yticks(fontsize=12)
67 plt.grid(axis='x')
68
69 plt.show()
70
71 benign_count = df[' Label'].value_counts()['BENIGN']
72
73 # Calculate the count of all network anomalies by subtracting benign traffic from
total
74 total_traffic_count = df[' Label'].value_counts().sum()
75 anomalies_count = total_traffic_count - benign_count
76
77 # Data to plot
78 labels = ['Network Anomalies', 'Benign Traffic']
79 sizes = [anomalies_count, benign_count]
80 colors = ['#ff9999', '#66b3ff']
81 explode = (0.1, 0) # explode 1st slice
82

```

```

83 # Plot
84 plt.figure(figsize=(8, 6))
85 plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f
    %%', shadow=True, startangle=140)
86 plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
87 plt.title('Proportion of Network Anomalies vs Benign Traffic')
88 plt.show()
89
90 %%%
91 # Cell 4: Addressing class imbalance with SMOTE
92
93 # Import necessary library for SMOTE
94 #from imblearn.over_sampling import SMOTE
95 #from sklearn.preprocessing import LabelEncoder
96
97 # Ensure all categorical data is encoded
98 #label_encoder = LabelEncoder()
99 #df['Label'] = label_encoder.fit_transform(df['Label'])
100
101 # Define your features and target variable
102 #X = df.drop('Label', axis=1) # Features
103 #y = df['Label'] # Target variable
104
105 #Initializing SMOTE
106 #smote = SMOTE()
107
108 # Applying SMOTE to your data and creating a new balanced dataset
109 #X_smote, y_smote = smote.fit_resample(X, y)
110
111 # Check the balanced dataset
112 #print("After SMOTE, counts of label '1': {}".format(sum(y_smote == 1)))
113 #print("After SMOTE, counts of label '0': {}".format(sum(y_smote == 0)))
114
115 # Proceed to split your dataset into training and testing sets
116 #from sklearn.model_selection import train_test_split
117
118 # Splitting the dataset into the Training set and Test set
119 #X_train, X_test, y_train, y_test = train_test_split(X_smote, y_smote, test_size=0
    .2, random_state=42)
120
121 #print("Training set shape: ", X_train.shape, y_train.shape)
122 #print("Testing set shape: ", X_test.shape, y_test.shape)
123
124 %%%
125 #Cell 5: Splitting Data into Training and Test

```

```

126
127 X = df.drop(' Label', axis=1) # Features
128 y = df[' Label'] # Target variable
129
130 # Encoding the categorical target variable to numeric
131 y_encoded = LabelEncoder().fit_transform(y)
132
133 # Splitting the dataset into the Training set and Test set
134 X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2,
    random_state=42, stratify=y_encoded)
135
136 print(f"Training set shape: {X_train.shape}, {y_train.shape}")
137 print(f"Test set shape: {X_test.shape}, {y_test.shape}")
138
139 ###
140 # Cell 6: Define the Neural Network Model
141 class BasicNN(nn.Module):
142     def __init__(self, input_size, output_size):
143         super(BasicNN, self).__init__()
144         self.layer1 = nn.Linear(input_size, 64) # Adjust input layer to hidden layer
145         self.relu = nn.ReLU() # Activation function
146         self.layer2 = nn.Linear(64, output_size) # Adjust hidden layer to output
    layer
147
148     def forward(self, x):
149         x = self.relu(self.layer1(x))
150         x = self.layer2(x)
151         return x
152
153 # Specify the input and output dimensions based on your dataset
154 input_size = 44 # Adjust this based on the number of features in your dataset
155 output_size = len(np.unique(y_train)) # Adjust this based on the number of
    unique labels/classes
156
157 # Instantiate the model
158 model = BasicNN(input_size, output_size)
159 print("Model defined.")
160 ###
161 # Cell 7: Move the Model to the Appropriate Device
162 model = model.to(device)
163 print(f"Model moved to {device}.")
164
165 # Convert the training dataset to PyTorch tensors
166 X_train_tensor = torch.tensor(X_train.values, dtype=torch.float).to(device)
167 y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)

```

```

168 ###
169 # Cell 8: Train the Model
170
171 from torch.utils.data import DataLoader, TensorDataset
172
173 # Assuming X_train_tensor and y_train_tensor have already been moved to the
appropriate device
174 # Create a TensorDataset and DataLoader for batching
175 train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
176 train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
# Adjust batch_size as needed
177
178 # Define the loss function and optimizer
179 criterion = nn.CrossEntropyLoss()
180 optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # Learning rate
can be adjusted
181
182 # Define the training function
183 def train_model(model, criterion, optimizer, train_loader, epochs=10):
184     model.train()
185     for epoch in range(epochs):
186         for i, (inputs, labels) in enumerate(train_loader):
187             optimizer.zero_grad()
188             outputs = model(inputs)
189             loss = criterion(outputs, labels)
190             loss.backward()
191             optimizer.step()
192
193         # Log the loss
194         if (epoch+1) % 1 == 0: # Adjust the logging frequency as needed
195             print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
196
197 # Train the model
198 train_model(model, criterion, optimizer, train_loader, epochs=10)
199
200 ###
201 # Cell 9: Move Model to CPU
202 model = model.to('cpu')
203 print("Model moved to CPU.")
204
205 ###
206 # Cell 10: Evaluate Model on CPU
207
208 # Ensure data is in CPU for evaluation
209 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')

```

```

210 y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
211
212 # Evaluation mode
213 model.eval()
214
215 # Disable gradient calculation
216 with torch.no_grad():
217     # Forward pass
218     outputs = model(X_test_tensor)
219     _, predictions = torch.max(outputs, 1)
220
221     # Calculate accuracy
222     correct_predictions = (predictions == y_test_tensor).sum().item()
223     total_predictions = y_test_tensor.size(0)
224     accuracy = 100 * correct_predictions / total_predictions
225     print(f'Accuracy on test set: {accuracy:.2f}%')
226
227 ###
228 # Enhanced Cell 11: Evaluate Model with Additional Performance Measures
229
230 from sklearn.metrics import precision_score, recall_score, f1_score,
    confusion_matrix
231 import seaborn as sns
232
233 # Ensure data is in CPU for evaluation
234 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')
235 y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
236
237 # Evaluation mode
238 model.eval()
239
240 # Disable gradient calculation
241 with torch.no_grad():
242     # Forward pass
243     outputs = model(X_test_tensor)
244     _, predictions = torch.max(outputs, 1)
245
246     # Convert predictions and actuals to NumPy arrays for sklearn metrics
247     predictions_np = predictions.numpy()
248     y_test_np = y_test_tensor.numpy()
249
250     # Calculate accuracy
251     accuracy = accuracy_score(y_test_np, predictions_np)
252     precision = precision_score(y_test_np, predictions_np, average='weighted')
253     recall = recall_score(y_test_np, predictions_np, average='weighted')

```

```
254     f1 = f1_score(y_test_np, predictions_np, average='weighted')
255
256     # Display metrics
257     print(f'Accuracy: {accuracy:.4f}')
258     print(f'Precision: {precision:.4f}')
259     print(f'Recall: {recall:.4f}')
260     print(f'F1 Score: {f1:.4f}')
261
262     # Confusion Matrix
263     cm = confusion_matrix(y_test_np, predictions_np)
264     plt.figure(figsize=(10, 7))
265     sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
266     plt.title('Confusion Matrix')
267     plt.ylabel('Actual label')
268     plt.xlabel('Predicted label')
269     plt.show()
270
271     ###
272     # Cell 12: Create a new DataFrame df2 by copying df
273
274     df2 = df.copy()
275
276     # Display the first few rows to ensure it's copied correctly
277     print(df2.head())
278
279     ###
280     # Cell 13: Clearing DataFrame 'df' from memory
281
282     # Delete the DataFrame
283     del df
284
285     # Import the garbage collector module
286     import gc
287
288     # Manually trigger garbage collection
289     gc.collect()
290
291     print("DataFrame 'df' has been deleted and memory cleared.")
292
293     ###
294     # Cell 14: Analyzing the new DataFrame df2
295
296     # Print the shape of df2
297     print("Shape of df2:", df2.shape)
298
```

```

299 # Display descriptive statistics for df2
300 print("\nDescriptive Statistics of df2:")
301 print(df2.describe())
302
303 # Count the number of unique labels and their occurrence
304 label_counts = df2[' Label'].value_counts() # Corrected 'Label' to 'Label' to
match the actual column name
305 print("\nNumber of unique labels:", df2[' Label'].nunique()) # Same
adjustment as above
306 print("\nCounts of each label:")
307 print(label_counts)
308
309 ###
310 # Cell 15: Create a new DataFrame df3 with only TCP-based attacks and Benign
traffic
311
312 # Define the labels for TCP based attacks from the image provided
313 tcp_based_attacks = ['MSSQL', 'DrDoS_SSDP']
314
315 # Include benign traffic
316 tcp_based_attacks.append('BENIGN')
317
318 # Filter df2 for these specific attacks and benign traffic
319 df3 = df2[df2[' Label'].isin(tcp_based_attacks)]
320
321 # Check the new shape and the balance of the labels
322 print("Shape of df3:", df3.shape)
323 print("\nCounts of each label in df3:")
324 print(df3[' Label'].value_counts())
325
326 ###
327 # Cell 16: Visualizing the distribution of the target variable in df3
328
329
330 # Assuming 'Label' is the target variable and it has leading space as before
331 label_counts = df3[' Label'].value_counts()
332
333 # Scatter Plot
334 plt.figure(figsize=(10, 6))
335 plt.scatter(label_counts.index, label_counts.values, color='blue')
336 plt.title('Scatter Plot of Label Distribution')
337 plt.xlabel('Labels')
338 plt.ylabel('Frequency')
339 plt.grid(True)
340 plt.show()

```



```

341
342 # Pie Chart
343 plt.figure(figsize=(8, 8))
344 plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle
    =140, colors=['skyblue', 'orange', 'green'])
345 plt.title('Pie Chart of Label Distribution')
346 plt.axis('equal') # Equal aspect ratio ensures that pie chart is drawn as a circle.
347 plt.show()
348
349 # Bar Graph
350 plt.figure(figsize=(12, 8))
351 sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')
352 plt.title('Bar Graph of Label Distribution')
353 plt.xlabel('Labels')
354 plt.ylabel('Frequency')
355 plt.xticks(rotation=45)
356 plt.show()
357
358 ###
359 # Cell 17: Splitting df3 Data into Training and Testing Sets
360
361 from sklearn.preprocessing import LabelEncoder
362
363 # Features and Labels
364 X = df3.drop('Label', axis=1)
365 y = df3['Label']
366
367 # Encoding the Labels
368 encoder = LabelEncoder()
369 y_encoded = encoder.fit_transform(y)
370
371 # Splitting the data
372 X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2,
    random_state=42)
373
374 # Printing shapes of the splits
375 print(f'Training set shape: X_train: {X_train.shape}, y_train: {y_train.shape}')
376 print(f'Test set shape: X_test: {X_test.shape}, y_test: {y_test.shape}')
377
378 ###
379 # Cell 18: Define the Neural Network Model for df3
380
381 def __init__(self, input_size, output_size):
382     super(BasicNN, self).__init__()
383     self.layer1 = nn.Linear(input_size, 64)

```

```

384     self.relu = nn.ReLU()
385     self.layer2 = nn.Linear(64, output_size)
386     self.sigmoid = nn.Sigmoid() # Only use if output_size == 1 for binary
classification
387     self.initialize_weights()
388
389     def forward(self, x):
390         x = self.relu(self.layer1(x))
391         x = self.layer2(x)
392         if self.layer2.out_features == 1: # Assuming binary classification
393             x = self.sigmoid(x)
394         return x
395
396     def initialize_weights(self):
397         for m in self.modules():
398             if isinstance(m, nn.Linear):
399                 nn.init.kaiming_normal_(m.weight, mode='fan_out')
400                 if m.bias is not None:
401                     nn.init.constant_(m.bias, 0)
402
403     # Define device
404     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
405
406     # Initialize model and move to the correct device
407     model = BasicNN(input_size=X_train.shape[1], output_size=len(np.unique(
408         y_train)))
409     model.to(device)
410     print("Model defined.")
411
412     ###
413     # Cell 19: Train the Model on df3
414
415     # Convert the training dataset to PyTorch tensors and move to the device
416     X_train_tensor = torch.tensor(X_train.values, dtype=torch.float).to(device)
417     y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)
418
419     # Create a TensorDataset and DataLoader for batching
420     train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
421     train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
# Adjust batch_size as needed
422
423     # Define the loss function and optimizer
424     criterion = nn.CrossEntropyLoss()
425     optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # Adjust learning

```

```

425 rate as needed
426
427 # Train the model
428 def train_model(model, criterion, optimizer, train_loader, epochs=10):
429     model.train()
430     for epoch in range(epochs):
431         for inputs, labels in train_loader:
432             inputs, labels = inputs.to(device), labels.to(device)
433             optimizer.zero_grad()
434             outputs = model(inputs)
435             loss = criterion(outputs, labels)
436             loss.backward()
437             optimizer.step()
438
439         # Logging
440         if (epoch+1) % 1 == 0:
441             print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
442
443 # Run the training function
444 train_model(model, criterion, optimizer, train_loader, epochs=10)
445
446 ###
447 # Cell 20: Evaluate the Model on the Test Set of df3
448
449 # Move the model to CPU for evaluation (if it's on GPU)
450 model = model.to('cpu')
451
452 # Ensure data is in CPU for evaluation
453 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')
454 y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
455
456 # Evaluation mode
457 model.eval()
458
459 # Disable gradient calculation
460 with torch.no_grad():
461     # Forward pass
462     outputs = model(X_test_tensor)
463     _, predictions = torch.max(outputs, 1)
464
465     # Calculate accuracy
466     accuracy = (predictions == y_test_tensor).sum().item() / y_test_tensor.size(0)
467     print(f'Accuracy on test set: {accuracy:.2f}')
468
469 ###

```

```
470 # Cell 21: Enhanced Evaluation with Additional Performance Measures
471
472 # Ensure data is in CPU for evaluation
473 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')
474 y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
475
476 # Evaluation mode
477 model.eval()
478
479 # Disable gradient calculation
480 with torch.no_grad():
481     # Forward pass
482     outputs = model(X_test_tensor)
483     _, predictions = torch.max(outputs, 1)
484
485     # Convert predictions and y_test to NumPy arrays for sklearn metrics
486     predictions_np = predictions.numpy()
487     y_test_np = y_test_tensor.numpy()
488
489     # Calculate precision, recall, and F1-score
490     precision = precision_score(y_test_np, predictions_np, average='weighted')
491     recall = recall_score(y_test_np, predictions_np, average='weighted')
492     f1 = f1_score(y_test_np, predictions_np, average='weighted')
493
494     # Display metrics
495     print(f'Accuracy: {accuracy:.2f}')
496     print(f'Precision: {precision:.4f}')
497     print(f'Recall: {recall:.4f}')
498     print(f'F1 Score: {f1:.4f}')
499
500
501 #%%
502 # Cell 22: Further Analysis and Visualization
503
504 import matplotlib.pyplot as plt
505 import seaborn as sns
506
507 # Assuming 'model' is your trained neural network model
508 # and 'X_train' is your training dataset.
509
510 # Make sure the model is in evaluation mode and on the right device
511 model.eval()
512 model.to(device) # Ensure the model is on the correct device
513
514 # Generate predictions
```

```
515 with torch.no_grad():
516     # Ensure the data tensor is on the same device as the model
517     outputs = model(X_train_tensor.to(device))
518     _, predictions = torch.max(outputs, 1)
519
520 # Convert predictions to numpy array for use with matplotlib
521 predictions_np = predictions.cpu().numpy()
522
523 # Plotting the histogram of predictions
524 plt.figure(figsize=(10, 6))
525 plt.hist(predictions_np, bins=len(np.unique(predictions_np)), color='skyblue')
526 plt.title('Histogram of Predicted Classes')
527 plt.xlabel('Classes')
528 plt.ylabel('Frequency')
529 plt.grid(True)
530 plt.show()
531
532 # Additional analysis can be added here depending on the specific requirements
    or objectives of your project.
533
534 """
535 # Cell 23: Create DataFrame df4 with Network Anomalies Only
536
537 # Exclude 'BENIGN' traffic to focus only on network anomalies
538 df4 = df2[df2['Label'] != 'BENIGN']
539
540 # Check the new shape and distribution of the labels
541 print("Shape of df4:", df4.shape)
542 print("\nCounts of each label in df4:")
543 print(df4['Label'].value_counts())
544
545 # Plotting the distribution of network anomalies
546 plt.figure(figsize=(12, 8))
547 sns.countplot(y=df4['Label'], order = df4['Label'].value_counts().index)
548 plt.title('Distribution of Network Anomalies')
549 plt.xlabel('Frequency')
550 plt.ylabel('Anomaly Type')
551 plt.show()
552
553 """
554 # Cell 24: Clearing DataFrame 'df2' and 'df3' from memory
555
556 # Delete the DataFrames
557 del df2, df3
558
```

```

559 # Import the garbage collector module
560 import gc
561
562 # Manually trigger garbage collection to free up memory
563 gc.collect()
564
565 print("DataFrames 'df2' and 'df3' have been deleted and memory cleared.")
566
567 ###
568 # Cell 25: Inspect DataFrame and Check Environment
569
570 import pandas as pd
571 import seaborn as sns
572 import matplotlib.pyplot as plt
573 import os
574 import psutil
575
576 # Check if df4 is defined and display its first few rows, shape, and label
distribution
577 if 'df4' in locals():
578     print("First few rows of df4:")
579     print(df4.head())
580     print("Shape of df4:", df4.shape)
581
582     # Display label distribution
583     print("Label distribution in df4:")
584     label_counts = df4['Label'].value_counts() # Adjust the column name if
necessary
585     print(label_counts)
586
587     # Plotting the label distribution
588     plt.figure(figsize=(10, 6))
589     sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')
590     plt.title("Distribution of Labels in df4")
591     plt.xlabel('Labels')
592     plt.ylabel('Frequency')
593     plt.xticks(rotation=45)
594     plt.show()
595
596     # Display descriptive statistics for numerical features
597     print("Descriptive Statistics:")
598     print(df4.describe())
599 else:
600     print("df4 is not defined.")
601

```

```

602 # Display current memory usage
603 process = psutil.Process(os.getpid())
604 print(f'Current memory usage: {process.memory_info().rss / 1024 ** 2:.2f}
MB')
605
606 ###
607 # Cell 26: Splitting Data into Training and Testing Sets
608
609 from sklearn.model_selection import train_test_split
610 from sklearn.preprocessing import LabelEncoder
611
612 # Features and Labels
613 X = df4.drop(' Label', axis=1) # Drop the label column to isolate features
614 y = df4[' Label'] # Isolate the label column
615
616 # Encoding the Labels
617 encoder = LabelEncoder()
618 y_encoded = encoder.fit_transform(y)
619
620 # Splitting the data
621 # We use stratify to ensure our training and test sets have approximately the
same percentage of samples of each target class as the complete set.
622 X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2,
random_state=42, stratify=y_encoded)
623
624 # Printing shapes of the splits to verify
625 print(f'Training set shape: X_train: {X_train.shape}, y_train: {y_train.shape}')
626 print(f'Test set shape: X_test: {X_test.shape}, y_test: {y_test.shape}')
627
628 ###
629 # Cell 27: Define the Basic Neural Network Model
630
631 def __init__(self, input_size, output_size):
632     super(BasicNN, self).__init__()
633     self.layer1 = nn.Linear(input_size, 64)
634     self.relu = nn.ReLU()
635     self.layer2 = nn.Linear(64, output_size)
636     self.sigmoid = nn.Sigmoid() # Only use if output_size == 1 for binary
classification
637     self.initialize_weights()
638
639 def forward(self, x):
640     x = self.relu(self.layer1(x))
641     x = self.layer2(x)
642     if self.layer2.out_features == 1: # Assuming binary classification

```

```

643     x = self.sigmoid(x)
644     return x
645
646 def initialize_weights(self):
647     for m in self.modules():
648         if isinstance(m, nn.Linear):
649             nn.init.kaiming_normal_(m.weight, mode='fan_out')
650             if m.bias is not None:
651                 nn.init.constant_(m.bias, 0)
652
653 # Define device
654 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
655
656 # Initialize model and move to the correct device
657 model = BasicNN(input_size=X_train.shape[1], output_size=len(np.unique(
    y_train)))
658 model.to(device)
659
660 print("Model defined.")
661
662 ###
663 # Cell 28: Train the Model
664
665 # Convert training data to tensors and move them to the appropriate device
666 X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32).to(device)
667 y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)
668
669 # DataLoader for batching
670 train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
671 train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
672
673 # Loss function and optimizer
674 criterion = nn.CrossEntropyLoss()
675 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
676
677 # Training loop
678 def train_model(model, criterion, optimizer, train_loader, epochs=10):
679     model.train()
680     for epoch in range(epochs):
681         for data, target in train_loader:
682             data, target = data.to(device), target.to(device)
683             optimizer.zero_grad()
684             output = model(data)
685             loss = criterion(output, target)
686             loss.backward()

```



```

687     optimizer.step()
688     if epoch % 1 == 0:
689         print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}')
690
691     # Start training
692     train_model(model, criterion, optimizer, train_loader, epochs=3)
693
694     ###
695     # Cell 29: Enhanced Model Evaluation
696
697     # Ensure the model is in evaluation mode
698     model.eval()
699
700     # Move the model to CPU for evaluation
701     model.to('cpu')
702
703     # Convert test data to tensors and ensure they are on the CPU
704     X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
705     y_test_tensor = torch.tensor(y_test, dtype=torch.long)
706
707     # Disable gradient calculation for evaluation
708     with torch.no_grad():
709         outputs = model(X_test_tensor)
710         _, predictions = torch.max(outputs, 1)
711
712     # Calculating performance metrics
713     from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
714     import seaborn as sns
715
716     # Converting tensors to numpy arrays for use with sklearn functions
717     predictions_np = predictions.numpy()
718     y_test_np = y_test_tensor.numpy()
719
720     # Calculating accuracy, precision, recall, and F1-score
721     accuracy = accuracy_score(y_test_np, predictions_np)
722     class_report = classification_report(y_test_np, predictions_np, target_names=np.
unique(y_test).astype(str))
723
724     # Display metrics
725     print(f'Accuracy: {accuracy:.4f}')
726     print('Classification Report:\n', class_report)
727
728     # Confusion Matrix
729     conf_matrix = confusion_matrix(y_test_np, predictions_np)

```

```

730 plt.figure(figsize=(10, 8))
731 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.
    unique(y_test), yticklabels=np.unique(y_test))
732 plt.title('Confusion Matrix')
733 plt.ylabel('Actual Labels')
734 plt.xlabel('Predicted Labels')
735 plt.show()
736
737 %%
738 # Cell 29: Enhanced Model Evaluation
739
740 # Ensure the model is in evaluation mode
741 model.eval()
742
743 # Convert test data to tensors and ensure they are on the CPU
744 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
745 y_test_tensor = torch.tensor(y_test, dtype=torch.long)
746
747 # Disable gradient calculation for evaluation
748 with torch.no_grad():
749     outputs = model(X_test_tensor)
750     _, predictions = torch.max(outputs, 1)
751
752 # Calculating performance metrics
753 from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
754 import seaborn as sns
755
756 # Converting tensors to numpy arrays for use with sklearn functions
757 predictions_np = predictions.numpy()
758 y_test_np = y_test_tensor.numpy()
759
760 # Calculating accuracy, precision, recall, and F1-score
761 accuracy = accuracy_score(y_test_np, predictions_np)
762 class_report = classification_report(y_test_np, predictions_np, target_names=np.
    unique(y_test).astype(str), zero_division=0)
763
764 # Display metrics
765 print(f'Accuracy: {accuracy:.4f}')
766 print('Classification Report:\n', class_report)
767
768 # Confusion Matrix
769 conf_matrix = confusion_matrix(y_test_np, predictions_np)
770 plt.figure(figsize=(10, 8))
771 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.

```

```

771 unique(y_test), yticklabels=np.unique(y_test))
772 plt.title('Confusion Matrix')
773 plt.ylabel('Actual Labels')
774 plt.xlabel('Predicted Labels')
775 plt.show()
776
777 #%%
778 # Cell 30: Define a More Complex BNN
779 class ComplexNN(nn.Module):
780     def __init__(self, input_size, output_size):
781         super(ComplexNN, self).__init__()
782         self.layer1 = nn.Linear(input_size, 128)
783         self.relu1 = nn.ReLU()
784         self.dropout1 = nn.Dropout(0.5)
785         self.layer2 = nn.Linear(128, 64)
786         self.relu2 = nn.ReLU()
787         self.dropout2 = nn.Dropout(0.3)
788         self.layer3 = nn.Linear(64, output_size)
789
790     def forward(self, x):
791         x = self.dropout1(self.relu1(self.layer1(x)))
792         x = self.dropout2(self.relu2(self.layer2(x)))
793         x = self.layer3(x)
794     return x
795
796 # Instantiate the model
797 input_size = 44 # Adjust this based on the number of features
798 output_size = len(np.unique(y_train)) # Adjust this based on the number of
unique labels/classes
799
800 model = ComplexNN(input_size, output_size).to(device)
801 print("Complex model defined and moved to:", device)
802
803 #%%
804 # Cell 31: Train the More Complex BNN
805 def train_complex_model(model, criterion, optimizer, train_loader, epochs=10):
806     model.train()
807     for epoch in range(epochs):
808         for inputs, labels in train_loader:
809             inputs, labels = inputs.to(device), labels.to(device)
810             optimizer.zero_grad()
811             outputs = model(inputs)
812             loss = criterion(outputs, labels)
813             loss.backward()
814             optimizer.step()

```

```

815     if (epoch+1) % 1 == 0:
816         print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
817
818     # Assuming train_loader is already defined
819     optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
820     criterion = nn.CrossEntropyLoss()
821
822     train_complex_model(model, criterion, optimizer, train_loader, epochs=10)
823
824     ###
825     # Cell 32: Enhanced Model Evaluation
826
827     # Ensure the model is in evaluation mode
828     model.eval()
829
830     # Move the model to CPU for evaluation
831     model.to('cpu')
832
833     # Convert test data to tensors and ensure they are on the CPU
834     X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
835     y_test_tensor = torch.tensor(y_test, dtype=torch.long)
836
837     # Disable gradient calculation for evaluation
838     with torch.no_grad():
839         outputs = model(X_test_tensor)
840         _, predictions = torch.max(outputs, 1)
841
842     # Calculating performance metrics
843     from sklearn.metrics import classification_report, confusion_matrix,
        accuracy_score
844     import seaborn as sns
845
846     # Converting tensors to numpy arrays for use with sklearn functions
847     predictions_np = predictions.numpy()
848     y_test_np = y_test_tensor.numpy()
849
850     # Calculating accuracy, precision, recall, and F1-score
851     accuracy = accuracy_score(y_test_np, predictions_np)
852     class_report = classification_report(y_test_np, predictions_np, target_names=np.
        unique(y_test).astype(str), zero_division=0)
853
854     # Display metrics
855     print(f'Accuracy: {accuracy:.4f}')
856     print('Classification Report:\n', class_report)
857

```

```
858 # Confusion Matrix
859 conf_matrix = confusion_matrix(y_test_np, predictions_np)
860 plt.figure(figsize=(10, 8))
861 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.
    unique(y_test), yticklabels=np.unique(y_test))
862 plt.title('Confusion Matrix')
863 plt.ylabel('Actual Labels')
864 plt.xlabel('Predicted Labels')
865 plt.show()
866
```