

```
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 class BasicNN(nn.Module):
382     def __init__(self, input_size, output_size):
383         super(BasicNN, self).__init__()

```

```

384     self.layer1 = nn.Linear(input_size, 64)
385     self.layer2 = nn.Linear(64, 32)
386     self.layer3 = nn.Linear(32, output_size)
387
388     def forward(self, x):
389         x = F.relu(self.layer1(x))
390         x = F.relu(self.layer2(x))
391         x = self.layer3(x)
392         return x
393
394     # Specify the input and output dimensions based on your dataset
395     input_size = X_train.shape[1] # Number of features
396     output_size = len(np.unique(y_train)) # Number of unique classes
397
398     # Instantiate the model
399     model = BasicNN(input_size, output_size).to(device)
400     print("Model defined.")
401
402     %%
403     # Cell 19: Train the Model on df3
404
405     # Convert the training dataset to PyTorch tensors and move to the device
406     X_train_tensor = torch.tensor(X_train.values, dtype=torch.float).to(device)
407     y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)
408
409     # Create a TensorDataset and DataLoader for batching
410     train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
411     train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
412     # Adjust batch_size as needed
413
414     # Define the loss function and optimizer
415     criterion = nn.CrossEntropyLoss()
416     optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # Adjust learning
417     rate as needed
418
419     # Train the model
420     def train_model(model, criterion, optimizer, train_loader, epochs=10):
421         model.train()
422         for epoch in range(epochs):
423             for inputs, labels in train_loader:
424                 inputs, labels = inputs.to(device), labels.to(device)
425                 optimizer.zero_grad()
426                 outputs = model(inputs)
427                 loss = criterion(outputs, labels)
428                 loss.backward()

```

```

427         optimizer.step()
428
429         # Logging
430         if (epoch+1) % 1 == 0:
431             print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
432
433         # Run the training function
434         train_model(model, criterion, optimizer, train_loader, epochs=10)
435
436         ###
437         # Cell 20: Evaluate the Model on the Test Set of df3
438
439         # Move the model to CPU for evaluation (if it's on GPU)
440         model = model.to('cpu')
441
442         # Ensure data is in CPU for evaluation
443         X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')
444         y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
445
446         # Evaluation mode
447         model.eval()
448
449         # Disable gradient calculation
450         with torch.no_grad():
451             # Forward pass
452             outputs = model(X_test_tensor)
453             _, predictions = torch.max(outputs, 1)
454
455             # Calculate accuracy
456             accuracy = (predictions == y_test_tensor).sum().item() / y_test_tensor.size(0)
457             print(f'Accuracy on test set: {accuracy:.2f}')
458
459         ###
460         # Cell 21: Enhanced Evaluation with Additional Performance Measures
461
462         # Ensure data is in CPU for evaluation
463         X_test_tensor = torch.tensor(X_test.values, dtype=torch.float).to('cpu')
464         y_test_tensor = torch.tensor(y_test, dtype=torch.long).to('cpu')
465
466         # Evaluation mode
467         model.eval()
468
469         # Disable gradient calculation
470         with torch.no_grad():
471             # Forward pass

```

```

472 outputs = model(X_test_tensor)
473 _, predictions = torch.max(outputs, 1)
474
475 # Convert predictions and y_test to NumPy arrays for sklearn metrics
476 predictions_np = predictions.numpy()
477 y_test_np = y_test_tensor.numpy()
478
479 # Calculate precision, recall, and F1-score
480 precision = precision_score(y_test_np, predictions_np, average='weighted')
481 recall = recall_score(y_test_np, predictions_np, average='weighted')
482 f1 = f1_score(y_test_np, predictions_np, average='weighted')
483
484 # Display metrics
485 print(f'Accuracy: {accuracy:.2f}')
486 print(f'Precision: {precision:.4f}')
487 print(f'Recall: {recall:.4f}')
488 print(f'F1 Score: {f1:.4f}')
489
490
491 ###
492 # Cell 22: Further Analysis and Visualization
493
494 import matplotlib.pyplot as plt
495 import seaborn as sns
496
497 # Assuming 'model' is your trained neural network model
498 # and 'X_train' is your training dataset.
499
500 # Make sure the model is in evaluation mode and on the right device
501 model.eval()
502 model.to(device) # Ensure the model is on the correct device
503
504 # Generate predictions
505 with torch.no_grad():
506     # Ensure the data tensor is on the same device as the model
507     outputs = model(X_train_tensor.to(device))
508     _, predictions = torch.max(outputs, 1)
509
510 # Convert predictions to numpy array for use with matplotlib
511 predictions_np = predictions.cpu().numpy()
512
513 # Plotting the histogram of predictions
514 plt.figure(figsize=(10, 6))
515 plt.hist(predictions_np, bins=len(np.unique(predictions_np)), color='skyblue')
516 plt.title('Histogram of Predicted Classes')

```

```
517 plt.xlabel('Classes')
518 plt.ylabel('Frequency')
519 plt.grid(True)
520 plt.show()
521
522 # Additional analysis can be added here depending on the specific requirements
or objectives of your project.
523
524 ###
525 # Cell 23: Create DataFrame df4 with Network Anomalies Only
526
527 # Exclude 'BENIGN' traffic to focus only on network anomalies
528 df4 = df2[df2[' Label'] != 'BENIGN']
529
530 # Check the new shape and distribution of the labels
531 print("Shape of df4:", df4.shape)
532 print("\nCounts of each label in df4:")
533 print(df4[' Label'].value_counts())
534
535 # Plotting the distribution of network anomalies
536 plt.figure(figsize=(12, 8))
537 sns.countplot(y=df4[' Label'], order = df4[' Label'].value_counts().index)
538 plt.title('Distribution of Network Anomalies')
539 plt.xlabel('Frequency')
540 plt.ylabel('Anomaly Type')
541 plt.show()
542
543 ###
544 # Cell 24: Clearing DataFrame 'df2' and 'df3' from memory
545
546 # Delete the DataFrames
547 del df2, df3
548
549 # Import the garbage collector module
550 import gc
551
552 # Manually trigger garbage collection to free up memory
553 gc.collect()
554
555 print("DataFrames 'df2' and 'df3' have been deleted and memory cleared.")
556
557 ###
558 # Cell 25: Inspect DataFrame and Check Environment
559
560 import pandas as pd
```

```

561 import seaborn as sns
562 import matplotlib.pyplot as plt
563 import os
564 import psutil
565
566 # Check if df4 is defined and display its first few rows, shape, and label
distribution
567 if 'df4' in locals():
568     print("First few rows of df4:")
569     print(df4.head())
570     print("\nShape of df4:", df4.shape)
571
572     # Display label distribution
573     print("\nLabel distribution in df4:")
574     label_counts = df4['Label'].value_counts() # Adjust the column name if
necessary
575     print(label_counts)
576
577     # Plotting the label distribution
578     plt.figure(figsize=(10, 6))
579     sns.barplot(x=label_counts.index, y=label_counts.values, palette='viridis')
580     plt.title('Distribution of Labels in df4')
581     plt.xlabel('Labels')
582     plt.ylabel('Frequency')
583     plt.xticks(rotation=45)
584     plt.show()
585
586     # Display descriptive statistics for numerical features
587     print("\nDescriptive Statistics:")
588     print(df4.describe())
589 else:
590     print("df4 is not defined.")
591
592 # Display current memory usage
593 process = psutil.Process(os.getpid())
594 print(f"Current memory usage: {process.memory_info().rss / 1024 ** 2:.2f}
    MB")
595
596 ###
597 # Cell 26: Splitting Data into Training and Testing Sets
598
599 from sklearn.model_selection import train_test_split
600 from sklearn.preprocessing import LabelEncoder
601
602 # Features and Labels

```

```

603 X = df4.drop(' Label', axis=1) # Drop the label column to isolate features
604 y = df4[' Label'] # Isolate the label column
605
606 # Encoding the Labels
607 encoder = LabelEncoder()
608 y_encoded = encoder.fit_transform(y)
609
610 # Splitting the data
611 # 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.
612 X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2,
random_state=42, stratify=y_encoded)
613
614 # Printing shapes of the splits to verify
615 print(f'Training set shape: X_train: {X_train.shape}, y_train: {y_train.shape}')
616 print(f'Test set shape: X_test: {X_test.shape}, y_test: {y_test.shape}')
617
618 ###
619 # Cell 27: Define the Basic Neural Network Model
620
621 class BasicNN(nn.Module):
622     def __init__(self, input_size, output_size):
623         super(BasicNN, self).__init__()
624         self.layer1 = nn.Linear(input_size, 128) # First hidden layer
625         self.layer2 = nn.Linear(128, 64) # Second hidden layer
626         self.output_layer = nn.Linear(64, output_size) # Output layer
627         self.relu = nn.ReLU() # ReLU activation function
628
629     def forward(self, x):
630         x = self.relu(self.layer1(x))
631         x = self.relu(self.layer2(x))
632         x = self.output_layer(x)
633         return x
634
635 # Initialize the model
636 input_size = X_train.shape[1] # Number of features
637 output_size = len(np.unique(y_train)) # Number of unique classes
638 model = BasicNN(input_size, output_size).to(device)
639
640 print("Model defined and moved to device:", device)
641
642 ###
643 # Cell 28: Train the Model
644
645 # Convert training data to tensors and move them to the appropriate device

```

```

646 X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32).to(device)
647 y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)
648
649 # DataLoader for batching
650 train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
651 train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
652
653 # Loss function and optimizer
654 criterion = nn.CrossEntropyLoss()
655 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
656
657 # Training loop
658 def train_model(model, criterion, optimizer, train_loader, epochs=10):
659     model.train()
660     for epoch in range(epochs):
661         for data, target in train_loader:
662             data, target = data.to(device), target.to(device)
663             optimizer.zero_grad()
664             output = model(data)
665             loss = criterion(output, target)
666             loss.backward()
667             optimizer.step()
668         if epoch % 1 == 0:
669             print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}')
670
671 # Start training
672 train_model(model, criterion, optimizer, train_loader, epochs=3)
673
674 # %%
675 # Cell 29: Enhanced Model Evaluation
676
677 # Ensure the model is in evaluation mode
678 model.eval()
679
680 # Move the model to CPU for evaluation
681 model.to('cpu')
682
683 # Convert test data to tensors and ensure they are on the CPU
684 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
685 y_test_tensor = torch.tensor(y_test, dtype=torch.long)
686
687 # Disable gradient calculation for evaluation
688 with torch.no_grad():
689     outputs = model(X_test_tensor)
690     _, predictions = torch.max(outputs, 1)

```



```

691
692 # Calculating performance metrics
693 from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
694 import seaborn as sns
695
696 # Converting tensors to numpy arrays for use with sklearn functions
697 predictions_np = predictions.numpy()
698 y_test_np = y_test_tensor.numpy()
699
700 # Calculating accuracy, precision, recall, and F1-score
701 accuracy = accuracy_score(y_test_np, predictions_np)
702 class_report = classification_report(y_test_np, predictions_np, target_names=np.
    unique(y_test).astype(str))
703
704 # Display metrics
705 print(f'Accuracy: {accuracy:.4f}')
706 print('Classification Report:\n', class_report)
707
708 # Confusion Matrix
709 conf_matrix = confusion_matrix(y_test_np, predictions_np)
710 plt.figure(figsize=(10, 8))
711 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.
    unique(y_test), yticklabels=np.unique(y_test))
712 plt.title('Confusion Matrix')
713 plt.ylabel('Actual Labels')
714 plt.xlabel('Predicted Labels')
715 plt.show()
716
717 ###
718 # Cell 29: Enhanced Model Evaluation
719
720 # Ensure the model is in evaluation mode
721 model.eval()
722
723 # Convert test data to tensors and ensure they are on the CPU
724 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
725 y_test_tensor = torch.tensor(y_test, dtype=torch.long)
726
727 # Disable gradient calculation for evaluation
728 with torch.no_grad():
729     outputs = model(X_test_tensor)
730     _, predictions = torch.max(outputs, 1)
731
732 # Calculating performance metrics

```

```

733 from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
734 import seaborn as sns
735
736 # Converting tensors to numpy arrays for use with sklearn functions
737 predictions_np = predictions.numpy()
738 y_test_np = y_test_tensor.numpy()
739
740 # Calculating accuracy, precision, recall, and F1-score
741 accuracy = accuracy_score(y_test_np, predictions_np)
742 class_report = classification_report(y_test_np, predictions_np, target_names=np.
    unique(y_test).astype(str), zero_division=0)
743
744 # Display metrics
745 print(f'Accuracy: {accuracy:.4f}')
746 print('Classification Report:\n', class_report)
747
748 # Confusion Matrix
749 conf_matrix = confusion_matrix(y_test_np, predictions_np)
750 plt.figure(figsize=(10, 8))
751 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.
    unique(y_test), yticklabels=np.unique(y_test))
752 plt.title('Confusion Matrix')
753 plt.ylabel('Actual Labels')
754 plt.xlabel('Predicted Labels')
755 plt.show()
756
757 #%%
758 # Cell 30: Define a More Complex BNN
759 class ComplexNN(nn.Module):
760     def __init__(self, input_size, output_size):
761         super(ComplexNN, self).__init__()
762         self.layer1 = nn.Linear(input_size, 128)
763         self.relu1 = nn.ReLU()
764         self.dropout1 = nn.Dropout(0.5)
765         self.layer2 = nn.Linear(128, 64)
766         self.relu2 = nn.ReLU()
767         self.dropout2 = nn.Dropout(0.3)
768         self.layer3 = nn.Linear(64, output_size)
769
770     def forward(self, x):
771         x = self.dropout1(self.relu1(self.layer1(x)))
772         x = self.dropout2(self.relu2(self.layer2(x)))
773         x = self.layer3(x)
774         return x

```

```

775
776 # Instantiate the model
777 input_size = 44 # Adjust this based on the number of features
778 output_size = len(np.unique(y_train)) # Adjust this based on the number of
       unique labels/classes
779
780 model = ComplexNN(input_size, output_size).to(device)
781 print("Complex model defined and moved to:", device)
782
783 ###
784 # Cell 31: Train the More Complex BNN
785 def train_complex_model(model, criterion, optimizer, train_loader, epochs=10):
786     model.train()
787     for epoch in range(epochs):
788         for inputs, labels in train_loader:
789             inputs, labels = inputs.to(device), labels.to(device)
790             optimizer.zero_grad()
791             outputs = model(inputs)
792             loss = criterion(outputs, labels)
793             loss.backward()
794             optimizer.step()
795             if (epoch+1) % 1 == 0:
796                 print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
797
798 # Assuming train_loader is already defined
799 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
800 criterion = nn.CrossEntropyLoss()
801
802 train_complex_model(model, criterion, optimizer, train_loader, epochs=10)
803
804 ###
805 # Cell 32: Enhanced Model Evaluation
806
807 # Ensure the model is in evaluation mode
808 model.eval()
809
810 # Move the model to CPU for evaluation
811 model.to('cpu')
812
813 # Convert test data to tensors and ensure they are on the CPU
814 X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
815 y_test_tensor = torch.tensor(y_test, dtype=torch.long)
816
817 # Disable gradient calculation for evaluation
818 with torch.no_grad():

```

```
819 outputs = model(X_test_tensor)
820 _, predictions = torch.max(outputs, 1)
821
822 # Calculating performance metrics
823 from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
824 import seaborn as sns
825
826 # Converting tensors to numpy arrays for use with sklearn functions
827 predictions_np = predictions.numpy()
828 y_test_np = y_test_tensor.numpy()
829
830 # Calculating accuracy, precision, recall, and F1-score
831 accuracy = accuracy_score(y_test_np, predictions_np)
832 class_report = classification_report(y_test_np, predictions_np, target_names=np.
    unique(y_test).astype(str), zero_division=0)
833
834 # Display metrics
835 print(f'Accuracy: {accuracy:.4f}')
836 print('Classification Report:\n', class_report)
837
838 # Confusion Matrix
839 conf_matrix = confusion_matrix(y_test_np, predictions_np)
840 plt.figure(figsize=(10, 8))
841 sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=np.
    unique(y_test), yticklabels=np.unique(y_test))
842 plt.title('Confusion Matrix')
843 plt.ylabel('Actual Labels')
844 plt.xlabel('Predicted Labels')
845 plt.show()
846
```