

Simulating and Capturing Attack Traffic

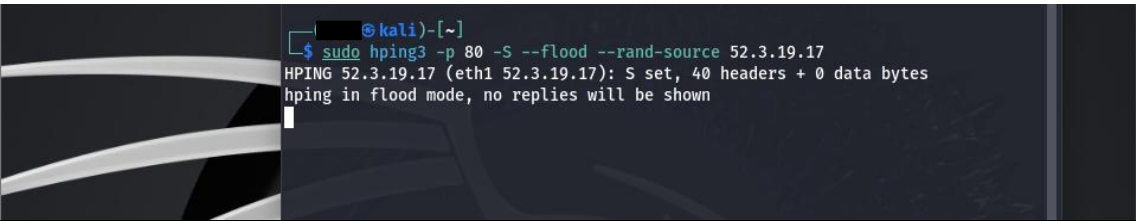
This section outlines the process of generating and collecting network traffic for analysis. DDoS attacks were simulated using hping3, executing TCP, UDP, and ICMP flood commands to replicate real-world attack patterns. Network packets were captured using tcpdump, providing raw traffic data for further examination. The captured packets were then saved in pcap format and stored in an AWS S3 bucket, ensuring accessibility for subsequent processing and analysis.

1. To simulate DDoS attacks, hping3 was used to generate high volumes of malicious traffic. The attacks included TCP SYN flood, UDP flood, and ICMP flood, each executed with randomized source IP addresses. Figure 17 shows the process of simulating DDoS attacks using hping3.

Commands:

- TCP Flood Attack: `sudo hping3 -p 80 -S --flood --rand-source <IP_ADDRESS>`
- UDP Flood Attack: `sudo hping3 -p 53 --udp --flood --rand-source <IP_ADDRESS>`
- ICMP Flood Attack: `sudo hping3 --icmp --flood --rand-source <IP_ADDRESS>`

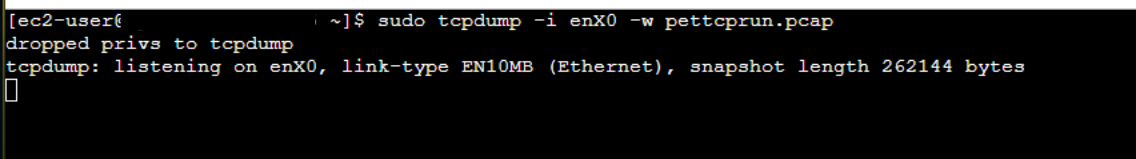
Figure 1
Simulating DDoS Attacks using hping3



- 2. Network traffic was captured using tcpdump. The command monitored incoming packets on interface enX0 and saved them in a .pcap file for further analysis.

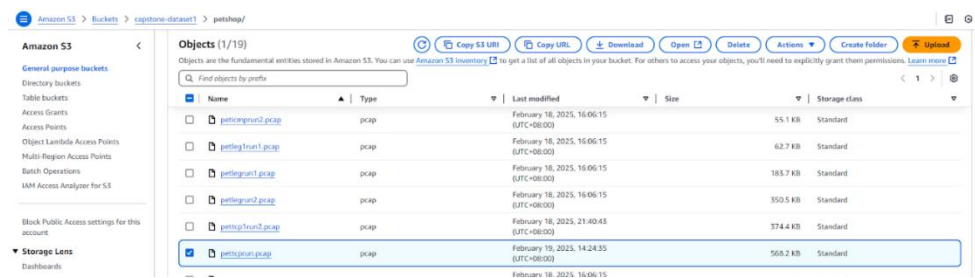
Figure 18 illustrates the process of capturing network packets using tcpdump.

Figure 2
Capturing Network Packets using tcpdump



- 3. The captured .pcap files were uploaded to an AWS S3 bucket for secure storage and further processing. Figure 19 shows the process of saving the captured packets as .pcap files in the S3 bucket.

Figure 3
Saving Captured Packets as pcap

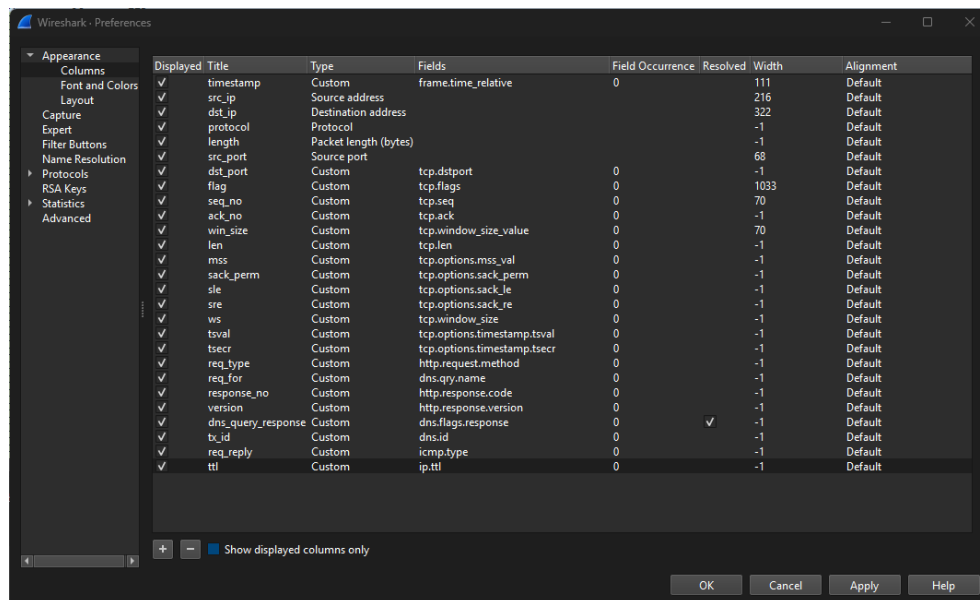


Filtering Packets Using Wireshark

This section outlines the refinement of captured network traffic using Wireshark. Columns were adjusted for consistency, filters were applied to extract TCP, UDP, and ICMP packets, and the processed data was exported as a csv file. The dataset was further refined by retaining relevant packets and converting 6-bit TCP flags into readable text for analysis.

1. Wireshark column preferences were customized to align with the training dataset format. This step ensured that the captured data could be processed and analyzed consistently. Figure 20 shows the process of adjusting Wireshark column preferences.

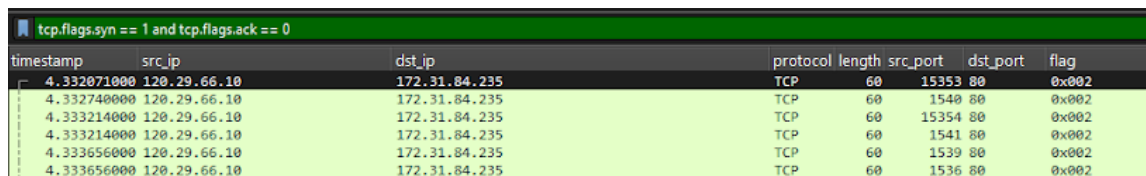
Figure 4
Adjusting Wireshark Column Preferences



2. Wireshark filters were applied to extract specific network traffic. Figure 21 shows the application of these filters in Wireshark. Filtering consists of:

- TCP: tcp.flags.syn == 1 and tcp.flags.ack == 0
- UDP: udp && !icmp
- ICMP: icmp

Figure 5
Applying Filters

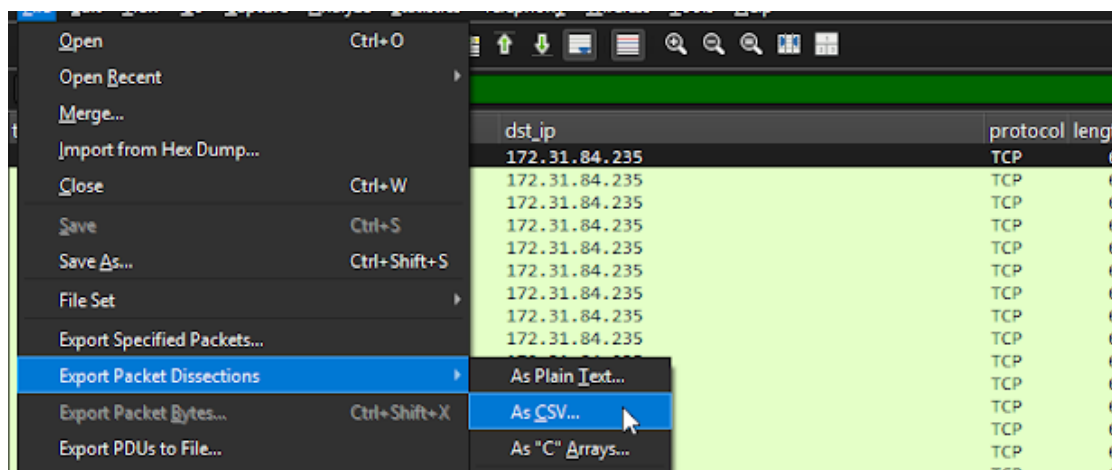


The image shows the Wireshark interface with a filter bar at the top containing the filter `tcp.flags.syn == 1 and tcp.flags.ack == 0`. Below the filter bar, a list of filtered packets is displayed. The table has columns for timestamp, src_ip, dst_ip, protocol, length, src_port, dst_port, and flag.

timestamp	src_ip	dst_ip	protocol	length	src_port	dst_port	flag
4.332071000	120.29.66.10	172.31.84.235	TCP	60	15353	80	0x002
4.332740000	120.29.66.10	172.31.84.235	TCP	60	1540	80	0x002
4.333214000	120.29.66.10	172.31.84.235	TCP	60	15354	80	0x002
4.333214000	120.29.66.10	172.31.84.235	TCP	60	1541	80	0x002
4.333656000	120.29.66.10	172.31.84.235	TCP	60	1539	80	0x002
4.333656000	120.29.66.10	172.31.84.235	TCP	60	1536	80	0x002

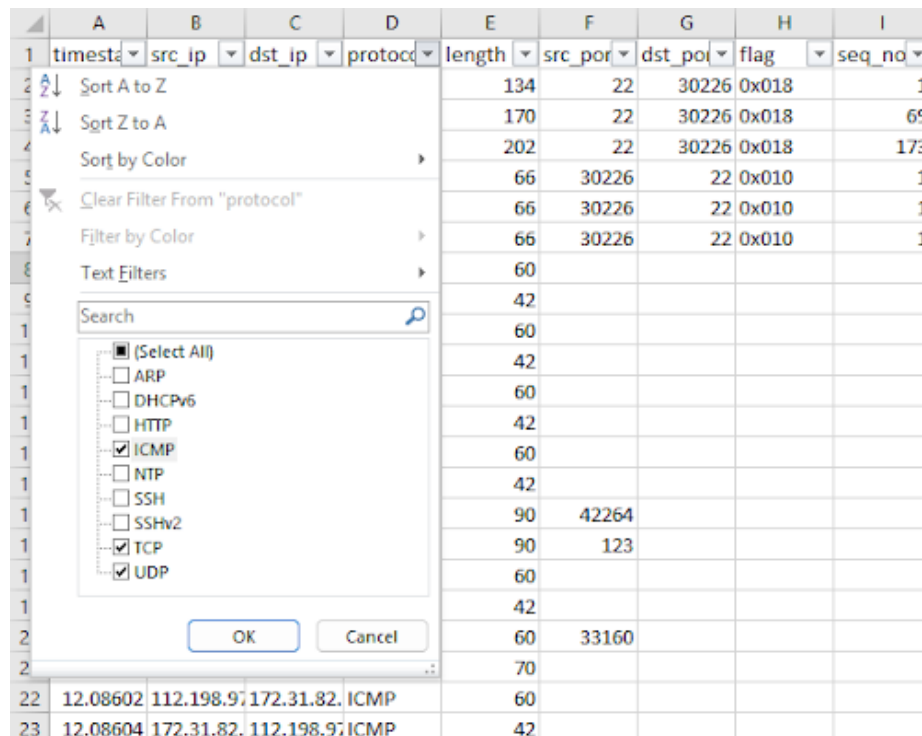
3. Filtered packets were exported from Wireshark as a csv file, allowing the processed data to be integrated into the dataset for further analysis. Figure 22 illustrates the process of exporting the filtered packets.

Figure 6
Exporting Filtered Packets as .csv



4. The filtered dataset retained only ICMP, TCP, and UDP packets, ensuring that only relevant traffic was included in the analysis. Figure 23 shows the process of filtering the dataset to retain these specific packets.

Figure 7
Filtering Dataset



	A	B	C	D	E	F	G	H	I
	timestamp	src_ip	dst_ip	protocol	length	src_port	dst_port	flag	seq_no
1					134	22	30226	0x018	1
2					170	22	30226	0x018	69
3					202	22	30226	0x018	173
4					66	30226	22	0x010	1
5					66	30226	22	0x010	1
6					66	30226	22	0x010	1
7					60				
8					42				
9					60				
10					42				
11					60				
12					42				
13					60				
14					42				
15					60				
16					42				
17					90	42264			
18					90	123			
19					60				
20					42				
21					60	33160			
22					70				
23	12.08602	112.198.97	172.31.82	ICMP	60				
24	12.08604	172.31.82	112.198.97	ICMP	42				

5. Figure 24 shows the mapping of these 6-bit TCP flags into readable text. Converted 6-bit TCP flags into readable text for analysis, with the following conversions:

- 0x002 = [SYN]
- 0x004 = [RST]
- 0x010 = [ACK]
- 0x011 = [FIN_ACK]

- 0x012 = [SYN_ACK]
- 0x014 = [RST_ACK]
- 0x018 = [PSH_ACK]

Figure 8
Mapping 6-bit TCP Flags

C	D	E	F	G	H	I
ip	protocol	length	src_port	dst_port	flag	seq_no
1.31.82	TCP	Sort A to Z				1
1.31.82	TCP	Sort Z to A				1
1.31.82	TCP	Sort by Color				1
1.31.82	ICMP	Clear Filter From "flag"				
1.198.97	ICMP	Filter by Color				
1.198.97	ICMP	Text Filters				
1.31.82	ICMP	Search				
1.198.97	ICMP	<input checked="" type="checkbox"/> (Select All)				
1.31.82	ICMP	<input checked="" type="checkbox"/> 0x002				
1.198.97	ICMP	<input checked="" type="checkbox"/> 0x004				
1.31.82	ICMP	<input checked="" type="checkbox"/> 0x010				
1.198.97	ICMP	<input checked="" type="checkbox"/> 0x011				
1.31.82	ICMP	<input checked="" type="checkbox"/> 0x012				
1.198.97	ICMP	<input checked="" type="checkbox"/> (Blanks)				
1.31.82	UDP	OK				
1.198.97	ICMP	Cancel				
1.31.82	ICMP					
1.198.97	ICMP	42				
1.31.82	UDP	60	63374			
1.198.97	ICMP	70				

Training of the model

This section outlines the DDoS attack detection model training process. The dataset is loaded and preprocessed, retaining only TCP traffic and encoding categorical data. Features and labels are separated, and a Random Forest model is either trained or loaded if pre-existing. The trained model is tested on new data, with performance evaluated using accuracy, precision, recall, and a confusion matrix to assess its effectiveness.

1. The training and testing datasets are read from CSV files. Any missing values in the dataset are replaced with the most common value in their respective columns. Since we are focusing on TCP traffic, only rows where the protocol is 'TCP' are kept. This step is illustrated in Figure 25, which shows the dataset after preprocessing, where only TCP traffic is retained, and missing values are filled with the most common values.

Figure 25
Cleaned BUET-DDoS2020 Train Dataset

#	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC
1	timestamp	src_ip	dst_ip	protocol	length	src_port	dst_port	flag	seq_no	ack_no	win_size	len	ms	sack_perm	sre	total	tsort	req_type	req_for	response_version	dns_querytx_id	req_replyttl	req_resp_label						
2	86.27917	24.34.143.172	172.16.0.2	TCP	174	2651	80	[SYN]	0		64	120	0																1
3	86.27913	172.16.0.2	24.34.143.172	TCP	58	80	2651	[SYN_ACK]	0	29200	0	1460																	1
4	86.27913	216.161.11	172.16.0.2	TCP	174	2652	80	[SYN]	0		64	120	0																1
5	86.27913	123.177.41	172.16.0.2	TCP	174	2653	80	[SYN]	0		64	120	0																1
6	86.27856	172.16.0.2	123.177.41	TCP	58	80	2653	[SYN_ACK]	0	29200	0	1460																	1
7	86.27879	252.97.241	172.16.0.2	TCP	174	2654	80	[SYN]	0		64	120	0																1
8	86.27879	172.16.0.2	252.97.241	TCP	58	80	2654	[SYN_ACK]	0	29200	0	1460																	1
9	86.27892	0.180.73.2	172.16.0.2	TCP	174	2655	80	[SYN]	0		64	120	0																1
10	86.27894	254.31.161	172.16.0.2	TCP	174	2656	80	[SYN]	0		64	120	0																1
11	86.27896	172.16.0.2	254.31.161	TCP	58	80	2656	[SYN_ACK]	0	29200	0	1460																	1
12	86.27913	73.73.26.1	172.16.0.2	TCP	174	2657	80	[SYN]	0		64	120	0																1
13	86.27916	172.16.0.2	73.73.26.1	TCP	58	80	2657	[SYN_ACK]	0	29200	0	1460																	1
14	86.27933	211.116.61	172.16.0.2	TCP	174	2658	80	[SYN]	0		64	120	0																1
15	86.27936	172.16.0.2	211.116.61	TCP	58	80	2658	[SYN_ACK]	0	29200	0	1460																	1
16	86.27952	354.118.11	172.16.0.2	TCP	174	2659	80	[SYN]	0		64	120	0																1
17	86.27956	172.16.0.2	354.118.11	TCP	58	80	2659	[SYN_ACK]	0	29200	0	1460																	1
18	86.27981	40.123.131	172.16.0.2	TCP	174	2660	80	[SYN]	0		64	120	0																1
19	86.27983	172.16.0.2	40.123.131	TCP	58	80	2660	[SYN_ACK]	0	29200	0	1460																	1
20	86.27996	149.103.11	172.16.0.2	TCP	174	2661	80	[SYN]	0		64	120	0																1
21	86.28	172.16.0.2	149.103.11	TCP	58	80	2661	[SYN_ACK]	0	29200	0	1460																	1

2. Some columns may contain text instead of numbers. These text values are converted into numbers using a technique called label encoding. This ensures that the machine learning model can understand and process them properly. Figure 26 illustrates the Python code used to transform categorical variables into numerical representations using label encoding.

Figure 10

Python code: Encoding Categorical Data

```
1 import pandas as pd
2 from sklearn.ensemble import RandomForestClassifier
3 from sklearn.metrics import (
4     confusion_matrix, accuracy_score, precision_score, recall_score,
5     f1_score, classification_report
6 )
7 from sklearn.preprocessing import LabelEncoder
8 import time
9 import joblib # Import joblib for saving and loading the model
10
11 # Load the datasets
12 train_data = pd.read_csv('train.csv', low_memory=False)
13 test_data = pd.read_csv('gocomixRerun2.csv', low_memory=False)
14
15 # Handle missing values
16 train_data = train_data.fillna(train_data.mode().iloc[0])
17 test_data = test_data.fillna(test_data.mode().iloc[0])
18
19 # Filter TCP traffic
20 train_data = train_data[train_data['protocol'] == 'TCP']
21 test_data = test_data[test_data['protocol'] == 'TCP']
22
23 # Combine data for encoding
24 combined_data = pd.concat([train_data, test_data], axis=0)
25
26 # Identify categorical columns
27 categorical_columns = combined_data.select_dtypes(include=['object']).columns.tolist()
28
29 # Apply Label encoding
30 label_encoders = {}
31 for col in categorical_columns:
32     le = LabelEncoder()
33     combined_data[col] = le.fit_transform(combined_data[col].astype(str))
34     label_encoders[col] = le
```

3. The dataset consists of multiple columns, but only one of them (called 'label') is what we want to predict. We separate this column as our target variable, while the rest of the columns serve as input features. The process of separating features and labels is shown in Figure 27.

Figure 11

Python code: Separating Features and Labels

```
29 # Apply label encoding
30 label_encoders = {}
31 for col in categorical_columns:
32     le = LabelEncoder()
33     combined_data[col] = le.fit_transform(combined_data[col].astype(str))
34     label_encoders[col] = le
35
36 # Split data back
37 train_data = combined_data.iloc[:len(train_data)]
38 test_data = combined_data.iloc[len(train_data):]
39
40 # Separate features and target variable
41 X_train = train_data.drop('label', axis=1)
42 y_train = train_data['label']
43 X_test = test_data.drop('label', axis=1)
44 y_test = test_data['label']
```

4. If a previously trained model already exists, it is loaded to save time. Otherwise, a new Random Forest model is created and trained using the training data. Once trained, the model is saved so it can be reused later. Figure 28 demonstrates the implementation of the Random Forest model training process, including parameter tuning and model saving. Figure 29 shows the saved Random Forest model file, which can be reused for predictions without retraining.

Figure 12

Python code: Training the Model

```
46 # Check if model exists, load or train
47 try:
48     rf_classifier = joblib.load('rf_model.joblib')
49     print("Model loaded from rf_model.joblib")
50 except FileNotFoundError:
51     rf_classifier = RandomForestClassifier(n_estimators=10, random_state=42)
52     rf_classifier.fit(X_train, y_train)
53     joblib.dump(rf_classifier, 'rf_model.joblib')
54     print("Model trained and saved as rf_model.joblib")
```

Figure 13*Produced RF Model: rf_model.joblib*

<input type="checkbox"/> petshopRerun2.csv	yesterday	678.7 KB
<input type="checkbox"/> rf_model.joblib	9 seconds ago	133 KB
<input type="checkbox"/> test working.csv	7 minutes ago	6.3 MB
<input type="checkbox"/> train.csv	3 days ago	49.9 MB
<input type="checkbox"/> xmodel.py	49 seconds ago	2.9 KB

5. The trained model is tested on new data to see how well it performs. The time it takes to make predictions is recorded. Various evaluation metrics like accuracy, precision, recall, and the confusion matrix are calculated to understand how effectively the model can detect attacks. Figure 30 contains the Python script used to test the trained model and compute evaluation metrics. Figure 31 presents the performance results of the trained model, including accuracy, precision, recall, and confusion matrix visualization.

Figure 14*Python code: Making Predictions and Measuring Performance*

```

56 # Measure prediction time
57 start_time = time.time()
58 y_pred = rf_classifier.predict(X_test)
59 end_time = time.time()
60 print(f"\nTime taken to predict: {end_time - start_time:.4f} seconds")
61
62 # Evaluate model
63 conf_matrix = confusion_matrix(y_test, y_pred)
64 TN, FP, FN, TP = conf_matrix.ravel()
65 accuracy = accuracy_score(y_test, y_pred)
66 precision = precision_score(y_test, y_pred)
67 recall = recall_score(y_test, y_pred)
68 specificity = TN / (TN + FP)
69 f1 = f1_score(y_test, y_pred)
70 fpr = FP / (FP + TN)
71 fnr = FN / (FN + TP)
72 class_report = classification_report(y_test, y_pred)

```

Figure 15

Test Result of the Model rf_model.joblib

```
(venv) sh-4.2$ python3 xmodel.py
Model trained and saved as rf_model.joblib

Time taken to predict: 0.1953 seconds

Confusion Matrix:
[[13406    0]
 [    7 55110]]

True Positives (TP): 55110
False Positives (FP): 0
False Negatives (FN): 7
True Negatives (TN): 13406

Accuracy: 0.9999
Precision: 1.0000
Recall: 0.9999
Specificity: 1.0000
F1 Score: 0.9999
False Positive Rate (FPR): 0.0000
False Negative Rate (FNR): 0.0001

Classification Report:

```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	13406
1	1.00	1.00	1.00	55117
accuracy			1.00	68523
macro avg	1.00	1.00	1.00	68523
weighted avg	1.00	1.00	1.00	68523

Testing of the model

This section outlines the model testing process using a separate dataset. The data is preprocessed, retaining TCP traffic, encoding categorical values, and separating features and labels. The trained Random Forest model is loaded and applied to the test set, with prediction time recorded. Performance is evaluated using accuracy, precision, recall, and other metrics across multiple simulation tests for Gocomix, Petshop, and Dental datasets.

1. The dataset is read from test.csv using `pandas.read_csv()`. Missing values are handled by replacing them with the mode of each column. Data is filtered to retain only TCP traffic by selecting rows where the 'protocol' column is 'TCP'. Figure 32 shows the loading and preprocessing of the test dataset.

Figure 16

Load Test Dataset

```
1 import pandas as pd
2 from sklearn.ensemble import RandomForestClassifier
3 from sklearn.metrics import (
4     confusion_matrix, accuracy_score, precision_score, recall_score,
5     f1_score, classification_report
6 )
7 from sklearn.preprocessing import LabelEncoder
8 import time
9 import joblib # Import joblib for saving and loading the model
10
11 # Load the datasets
12 train_data = pd.read_csv('train.csv', low_memory=False)
13 test_data = pd.read_csv('petshop.csv', low_memory=False)
```

2. Categorical columns are identified using `select_dtypes(include=['object'])`. Label encoding is applied to transform categorical variables into numerical values. The dataset is processed to ensure consistency with the training data.

Figure 33 illustrates the label encoding process for categorical values in the test dataset.

Figure 17

Label Encoding Test Dataset

```
29 # Apply label encoding
30 label_encoders = {}
31 for col in categorical_columns:
32     le = LabelEncoder()
33     combined_data[col] = le.fit_transform(combined_data[col].astype(str))
34     label_encoders[col] = le
```

3. The 'label' column is separated as the target variable (y_{test}), while the remaining columns serve as input features (X_{test}). The process of preparing features and labels is shown in Figure 34.

Figure 18

Features and Labels Test Dataset

```
36 # Split data back
37 train_data = combined_data.iloc[:len(train_data)]
38 test_data = combined_data.iloc[len(train_data):]
39
40 # Separate features and target variable
41 X_train = train_data.drop('label', axis=1)
42 y_train = train_data['label']
43 X_test = test_data.drop('label', axis=1)
44 y_test = test_data['label']
```

4. The trained model (`rf_model.joblib`) is loaded using `joblib.load()`. This pre-trained model is used to make predictions on the test dataset without the need for retraining. Figure 35 demonstrates the loading of the trained Random Forest model.

Figure 19

Load rf_model.joblib

```
46 # Check if model exists, load or train
47 try:
48     rf_classifier = joblib.load('rf_model.joblib')
49     print("Model loaded from rf_model.joblib")
50 except FileNotFoundError:
51     rf_classifier = RandomForestClassifier(n_estimators=10, random_state=42)
52     rf_classifier.fit(X_train, y_train)
53     joblib.dump(rf_classifier, 'rf_model.joblib')
54     print("Model trained and saved as rf_model.joblib")
```

5. The model is applied to X_test to generate predictions (y_pred). The prediction time is measured using the time module to evaluate efficiency. The predicted labels are then compared against y_test to assess model performance.
6. The model is evaluated using several metrics. Figure 36 presents the prediction and evaluation metrics, including accuracy, precision, recall, and the confusion matrix.

Figure 20

Prediction and Evaluation Metrics

```
56 # Measure prediction time
57 start_time = time.time()
58 y_pred = rf_classifier.predict(X_test)
59 end_time = time.time()
60 print(f"\nTime taken to predict: {end_time - start_time:.4f} seconds")
61
62 # Evaluate model
63 conf_matrix = confusion_matrix(y_test, y_pred)
64 TN, FP, FN, TP = conf_matrix.ravel()
65 accuracy = accuracy_score(y_test, y_pred)
66 precision = precision_score(y_test, y_pred)
67 recall = recall_score(y_test, y_pred)
68 specificity = TN / (TN + FP)
69 f1 = f1_score(y_test, y_pred)
70 fpr = FP / (FP + TN)
71 fnr = FN / (FN + TP)
72 class_report = classification_report(y_test, y_pred)
73
74 # Print evaluation metrics
75 print("\nConfusion Matrix:")
76 print(conf_matrix)
77 print(f"\nTrue Positives (TP): {TP}")
78 print(f"False Positives (FP): {FP}")
79 print(f"False Negatives (FN): {FN}")
80 print(f"True Negatives (TN): {TN}")
81 print(f"\nAccuracy: {accuracy:.4f}")
82 print(f"Precision: {precision:.4f}")
83 print(f"Recall: {recall:.4f}")
84 print(f"Specificity: {specificity:.4f}")
85 print(f"F1 Score: {f1:.4f}")
86 print(f"False Positive Rate (FPR): {fpr:.4f}")
87 print(f"False Negative Rate (FNR): {fnr:.4f}")
88 print("\nClassification Report:\n", class_report)
```

7. Multiple simulation tests were conducted on the Gocomix, Petshop, and Dental datasets. The following figures present the results for each test scenario.

Figure 37 displays the results of the first simulation test for the Gocomix dataset.

Figure 21

Gocomix Simulation 1 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0054 seconds

Confusion Matrix:
[[ 715  272]
 [   0 2000]]

True Positives (TP): 2000
False Positives (FP): 272
False Negatives (FN): 0
True Negatives (TN): 715

Accuracy: 0.9089
Precision: 0.8803
Recall: 1.0000
Specificity: 0.7244
F1 Score: 0.9363
False Positive Rate (FPR): 0.2756
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00      0.72      0.84         987
     1           0.88      1.00      0.94        2000

   accuracy              0.91         2987
  macro avg              0.94      0.86      0.89         2987
weighted avg              0.92      0.91      0.90         2987
```


The second simulation results for the Gocomix dataset are shown in Figure 38.

Figure 22

Gocomix Simulation 2 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0039 seconds

Confusion Matrix:
[[ 685  309]
 [   5 2495]]

True Positives (TP): 2495
False Positives (FP): 309
False Negatives (FN): 5
True Negatives (TN): 685

Accuracy: 0.9101
Precision: 0.8898
Recall: 0.9980
Specificity: 0.6891
F1 Score: 0.9408
False Positive Rate (FPR): 0.3109
False Negative Rate (FNR): 0.0020

Classification Report:
              precision    recall  f1-score   support

     0           0.99       0.69       0.81         994
     1           0.89       1.00       0.94        2500

   accuracy          0.91         0.91         0.91        3494
  macro avg           0.94         0.84         0.88        3494
 weighted avg           0.92         0.91         0.90        3494
```

Figure 39 presents the third simulation results for the Gocomix dataset.

Figure 23

Gocomix Simulation 3 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0043 seconds

Confusion Matrix:
[[1558  201]
 [   0 2500]]

True Positives (TP): 2500
False Positives (FP): 201
False Negatives (FN): 0
True Negatives (TN): 1558

Accuracy: 0.9528
Precision: 0.9256
Recall: 1.0000
Specificity: 0.8857
F1 Score: 0.9614
False Positive Rate (FPR): 0.1143
False Negative Rate (FNR): 0.0000

Classification Report:

```

	precision	recall	f1-score	support
0	1.00	0.89	0.94	1759
1	0.93	1.00	0.96	2500
accuracy			0.95	4259
macro avg	0.96	0.94	0.95	4259
weighted avg	0.96	0.95	0.95	4259

The first simulation results for the Petshop dataset are shown in Figure 40.

Figure 24

Petshop Simulation 1 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0041 seconds

Confusion Matrix:
[[1545  133]
 [   10 1990]]

True Positives (TP): 1990
False Positives (FP): 133
False Negatives (FN): 10
True Negatives (TN): 1545

Accuracy: 0.9611
Precision: 0.9374
Recall: 0.9950
Specificity: 0.9207
F1 Score: 0.9653
False Positive Rate (FPR): 0.0793
False Negative Rate (FNR): 0.0050

Classification Report:

```

	precision	recall	f1-score	support
0	0.99	0.92	0.96	1678
1	0.94	0.99	0.97	2000
accuracy			0.96	3678
macro avg	0.97	0.96	0.96	3678
weighted avg	0.96	0.96	0.96	3678

Figure 41 displays the second simulation test results for the Petshop dataset.

Figure 25

Petshop Simulation 2 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0048 seconds

Confusion Matrix:
[[ 675  320]
 [   0 2500]]

True Positives (TP): 2500
False Positives (FP): 320
False Negatives (FN): 0
True Negatives (TN): 675

Accuracy: 0.9084
Precision: 0.8865
Recall: 1.0000
Specificity: 0.6784
F1 Score: 0.9398
False Positive Rate (FPR): 0.3216
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00       0.68      0.81         995
     1           0.89       1.00      0.94        2500

   accuracy          0.91         3495
  macro avg          0.94         0.84      0.87         3495
 weighted avg          0.92         0.91      0.90         3495
```

The third simulation results for the Petshop dataset are presented in Figure 42.

Figure 26

Petshop Simulation 3 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0062 seconds

Confusion Matrix:
[[ 729  309]
 [   0 2500]]

True Positives (TP): 2500
False Positives (FP): 309
False Negatives (FN): 0
True Negatives (TN): 729

Accuracy: 0.9127
Precision: 0.8900
Recall: 1.0000
Specificity: 0.7023
F1 Score: 0.9418
False Positive Rate (FPR): 0.2977
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00       0.70       0.83       1038
     1           0.89       1.00       0.94       2500

   accuracy          0.91
  macro avg           0.94       0.85       0.88
weighted avg           0.92       0.91       0.91
```

Figure 43 presents the first simulation results for the Dental dataset.

Figure 27

Dental Simulation 1 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0055 seconds

Confusion Matrix:
[[ 488   30]
 [    0 2000]]

True Positives (TP): 2000
False Positives (FP): 30
False Negatives (FN): 0
True Negatives (TN): 488

Accuracy: 0.9881
Precision: 0.9852
Recall: 1.0000
Specificity: 0.9421
F1 Score: 0.9926
False Positive Rate (FPR): 0.0579
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00       0.94       0.97         518
     1           0.99       1.00       0.99        2000

 accuracy                   0.99         2518
  macro avg              0.99       0.97       0.98         2518
  weighted avg           0.99       0.99       0.99         2518
```

The second simulation results for the Dental dataset are shown in Figure 44.

Figure 28

Dental Simulation 2 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0057 seconds

Confusion Matrix:
[[ 808  294]
 [   0 2500]]

True Positives (TP): 2500
False Positives (FP): 294
False Negatives (FN): 0
True Negatives (TN): 808

Accuracy: 0.9184
Precision: 0.8948
Recall: 1.0000
Specificity: 0.7332
F1 Score: 0.9445
False Positive Rate (FPR): 0.2668
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00       0.73       0.85        1102
     1           0.89       1.00       0.94        2500

 accuracy                   0.92         3602
 macro avg              0.95         0.87         0.90         3602
 weighted avg           0.93         0.92         0.91         3602
```

Figure 45 displays the third simulation test results for the Dental dataset.

Figure 29

Dental Simulation 3 Results

```
(venv) sh-4.2$ python3 xmodel.py
Model loaded from rf_model.joblib

Time taken to predict: 0.0050 seconds

Confusion Matrix:
[[ 488   62]
 [    0 2500]]

True Positives (TP): 2500
False Positives (FP): 62
False Negatives (FN): 0
True Negatives (TN): 488

Accuracy: 0.9797
Precision: 0.9758
Recall: 1.0000
Specificity: 0.8873
F1 Score: 0.9878
False Positive Rate (FPR): 0.1127
False Negative Rate (FNR): 0.0000

Classification Report:
              precision    recall  f1-score   support

     0           1.00       0.89       0.94         550
     1           0.98       1.00       0.99        2500

   accuracy                   0.98        3050
  macro avg           0.99       0.94       0.96        3050
 weighted avg           0.98       0.98       0.98        3050
```