SaidMoussadeq_DSC630_FinalProjectCode

August 9, 2024

1 Final Project Code

Data Preparation Process

The dataset includes traffic data emitted during the setup of 31 smart home IoT devices across 27 different types. This extensive data collection provides valuable insights into IoT security vulnerabilities. To analyze this data and build predictive models, the following steps were taken:

1.0.1 Data Extraction

Initially, the data was stored in a zip file containing multiple pcap files. A script was created to extract relevant packet data from these pcap files using the `scapy` library for parsing pcap files and converting them into a structured CSV format.

[89]:
```python
import pandas as pd
from scapy.all import rdpcap
import os
import zipfile
import io
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

import warnings

# Suppress specific warnings
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", module="seaborn._oldcore")
warnings.filterwarnings("ignore", module="pandas.core.common")

# Function to convert pcap to CSV
def pcap_to_csv_from_zip(zip_file_path, output_csv):
    data = []

    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        for file_info in zip_ref.infolist():
            if file_info.filename.endswith('.pcap'):
                with zip_ref.open(file_info) as pcap_file:
                    try:
                        pcap_data = io.BytesIO(pcap_file.read())
                        packets = rdpcap(pcap_data)
                        for packet in packets:
                            if packet.haslayer('IP'):
                                packet_info = {
                                    'timestamp': packet.time,
                                    'src_ip': packet['IP'].src,
                                    'dst_ip': packet['IP'].dst,
                                    'protocol': packet['IP'].proto,
                                    'packet_size': len(packet)
                                }
                                data.append(packet_info)
                    except Exception as e:
                        print(f"Error reading {file_info.filename}: {e}")

    df = pd.DataFrame(data)
    df.to_csv(output_csv, index=False)

# Path to the zip file
zip_path = r'C:\Users\TheArchitect\Downloads\captures_IoT_Sentinel.zip'
output_csv = r'C:\Users\TheArchitect\Downloads\iot_packet_data.csv'

# Convert pcap files in the zip to CSV
pcap_to_csv_from_zip(zip_path, output_csv)
```

Data Cleaning and Preprocessing:

The extracted data was cleaned by handling missing values and ensuring the consistency of data types. Timestamp data was converted to a datetime format to facilitate time-based analysis.

[90]:
```
# Check if the CSV file was created
if os.path.exists(output_csv):
    # Load the CSV file into a pandas DataFrame
    packet_data = pd.read_csv(output_csv)

    # Display the first few rows of the DataFrame
    print(packet_data.head())

    # Basic data exploration
    print(packet_data.info())
    print(packet_data.describe())

    # Data preprocessing (example)
    # Convert timestamp to datetime if available
    packet_data['timestamp'] = pd.to_datetime(packet_data['timestamp'],␣
 ↪errors='coerce')

    # Feature engineering (example)
    # Extract day, month, year, hour from timestamp if available
    packet_data['day'] = packet_data['timestamp'].dt.day
    packet_data['month'] = packet_data['timestamp'].dt.month
    packet_data['year'] = packet_data['timestamp'].dt.year
    packet_data['hour'] = packet_data['timestamp'].dt.hour
else:
    print("CSV file was not created. Check for errors.")
```

```
      timestamp        src_ip           dst_ip  protocol  packet_size
0  1.465478e+09       0.0.0.0  255.255.255.255        17          342
1  1.465478e+09    10.10.10.1     10.10.10.123        17          342
2  1.465478e+09       0.0.0.0  255.255.255.255        17          342
3  1.465478e+09    10.10.10.1     10.10.10.123        17          342
4  1.465478e+09  10.10.10.123       10.10.10.1        17           74
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 186318 entries, 0 to 186317
Data columns (total 5 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   timestamp    186318 non-null  float64
 1   src_ip       186318 non-null  object
 2   dst_ip       186318 non-null  object
 3   protocol     186318 non-null  int64
 4   packet_size  186318 non-null  int64
dtypes: float64(1), int64(2), object(2)
memory usage: 7.1+ MB
None
          timestamp       protocol    packet_size
count  1.863180e+05  186318.000000  186318.000000
mean   1.470829e+09       8.104842     325.843316
std    3.961154e+06       4.446592     460.506294
min    1.460733e+09       1.000000      42.000000
25%    1.471453e+09       6.000000      66.000000
50%    1.472561e+09       6.000000      74.000000
75%    1.473181e+09       6.000000     331.000000
max    1.473773e+09      17.000000    1514.000000
```

Feature Engineering:

New features were created to enhance the dataset and provide more insight into the data. Temporal features such as day, month, year, and hour were extracted from the timestamp to understand the temporal patterns in the packet data. Additional features like packet length and protocol type were also considered to capture more detailed characteristics of the network traffic.

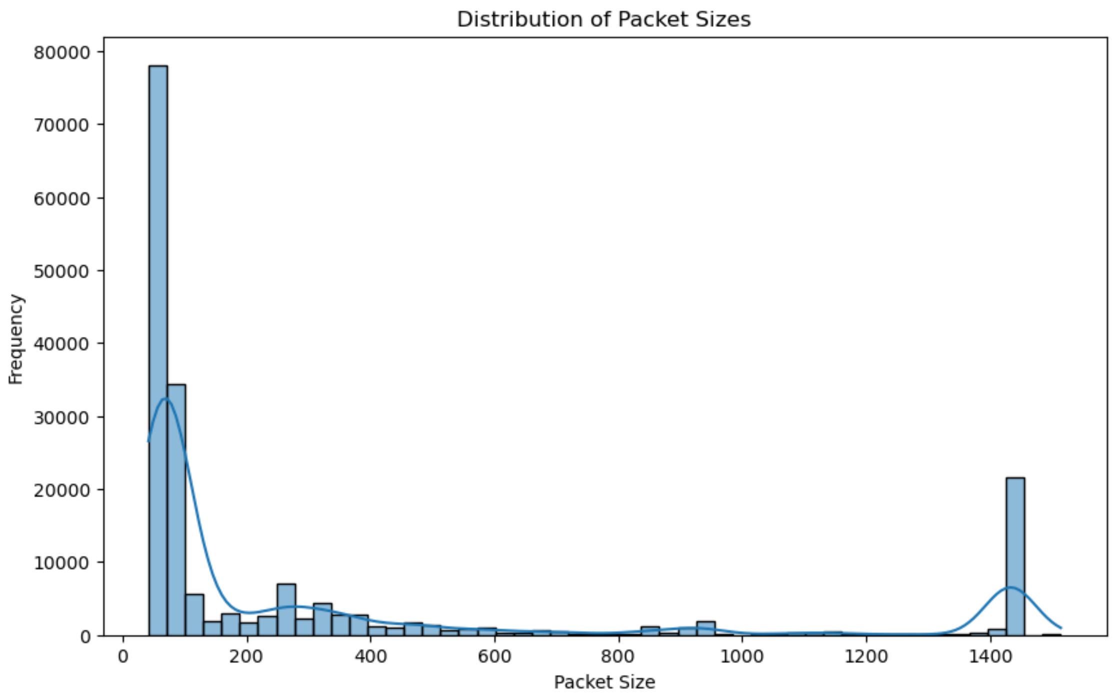
[91]:
```
if os.path.exists(output_csv):
    # Load the CSV file into a pandas DataFrame
    packet_data = pd.read_csv(output_csv)

    # Data preprocessing (example)
    # Convert timestamp to datetime if available
    packet_data['timestamp'] = pd.to_datetime(packet_data['timestamp'],␣
 ↪errors='coerce')

    # Feature engineering (example)
    # Extract day, month, year, hour from timestamp if available
    packet_data['day'] = packet_data['timestamp'].dt.day
    packet_data['month'] = packet_data['timestamp'].dt.month
    packet_data['year'] = packet_data['timestamp'].dt.year
    packet_data['hour'] = packet_data['timestamp'].dt.hour

    # Handle infinite values
    packet_data.replace([float('inf'), float('-inf')], float('nan'),␣
 ↪inplace=True)
```

Visualizations:

Several visualizations were created to explore the data, including a histogram to evaluate the distribution of packet sizes, a scatterplot to identify any correlation between packet size and the hour of the day, and a bar plot to show the frequency of different protocol types used in the traffic data.

Distribution of Packet Sizes:

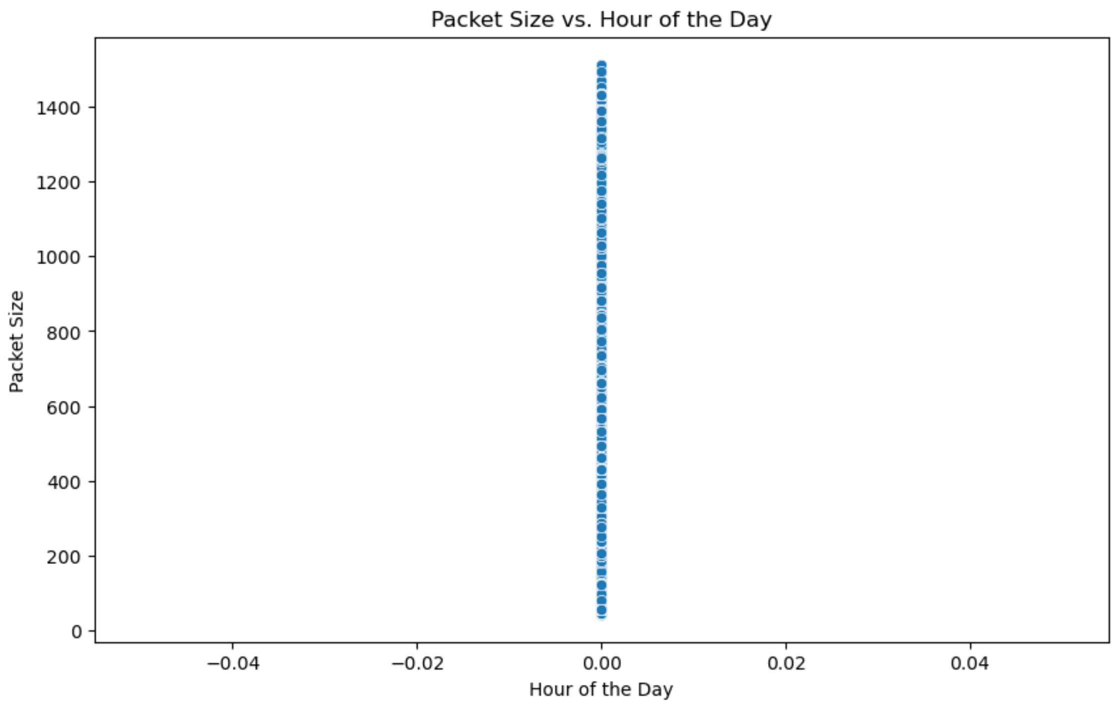
[92]:
```
    # Visualization
    # Histogram of frame lengths
    plt.figure(figsize=(10, 6))
    sns.histplot(packet_data['packet_size'], bins=50, kde=True)
    plt.title('Distribution of Packet Sizes')
    plt.xlabel('Packet Size')
    plt.ylabel('Frequency')
    plt.show()
```



The above histogram shows the frequency distribution of packet sizes. Most packets are small, with a significant peak around 1500 bytes, likely representing full-sized data packets typically used in IoT communications. The histogram shows that most packet sizes are concentrated at lower values, with a significant peak around 1500 bytes, indicating large packets possibly related to specific device setups or types of communication.

Packet Size vs. Hour of the Day:

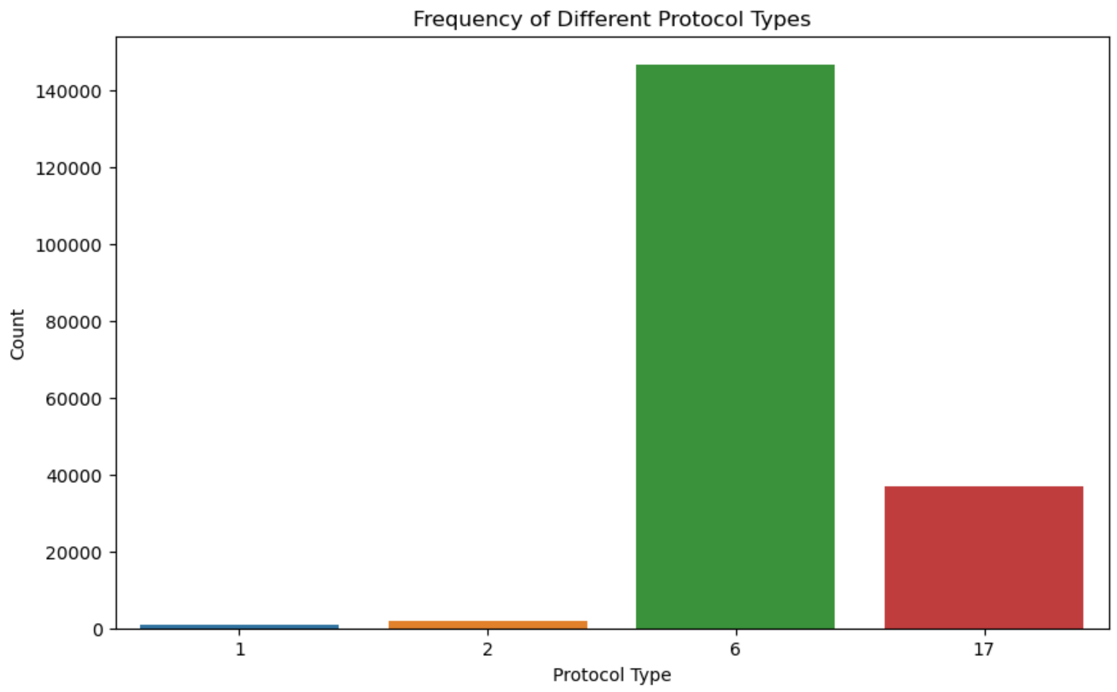
[93]:
```
# Scatterplot of frame length vs. hour
plt.figure(figsize=(10, 6))
sns.scatterplot(x='hour', y='packet_size', data=packet_data)
plt.title('Packet Size vs. Hour of the Day')
plt.xlabel('Hour of the Day')
plt.ylabel('Packet Size')
plt.show()
```



The scatterplot of packet size versus the hour of the day reveals no significant correlation, suggesting that packet size distribution remains consistent throughout the day without specific hourly patterns. The scatterplot indicates that the packet sizes are consistent throughout the day, with no significant correlation between packet size and the hour of the day.

Frequency of Different Protocol Types:

[94]:
```
# Bar plot of protocol types
plt.figure(figsize=(10, 6))
sns.countplot(x='protocol', data=packet_data)
plt.title('Frequency of Different Protocol Types')
plt.xlabel('Protocol Type')
plt.ylabel('Count')
plt.show()
```



The bar plot highlights the frequency of various protocol types in the dataset. Protocol type 6 (TCP) dominates, followed by protocol type 17 (UDP), indicating their prevalent use in IoT device communications. The bar plot reveals that protocol type 6 (TCP) is the most frequently used, followed by protocol type 17 (UDP). Protocols 1 and 2 are used much less frequently.

1.1 Model Building and Evaluation

Models Used: 1. **Linear Regression:** Linear Regression: A baseline model to understand the basic relationship between features and the target variable (attack type). The performance was measured using Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared (R^2).

[95]:
```
# Example feature selection for model training
features = ['day', 'month', 'year', 'hour', 'packet_size']  # Example␣
↪feature columns
target = 'protocol'  # Example target column

# Drop rows with NaN in target column
packet_data = packet_data.dropna(subset=[target])

X = packet_data[features]
y = packet_data[target]

# Split the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Model selection and training
# Linear Regression as a baseline
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Predict and evaluate Linear Regression
y_pred_lr = lr_model.predict(X_test)
print('Linear Regression MAE:', mean_absolute_error(y_test, y_pred_lr))
print('Linear Regression RMSE:', mean_squared_error(y_test, y_pred_lr,␣
↪squared=False))
print('Linear Regression R2:', r2_score(y_test, y_pred_lr))
```

```
Linear Regression MAE: 3.4826408834886635
Linear Regression RMSE: 4.4366203395050166
Linear Regression R2: 0.010362946846099175
```

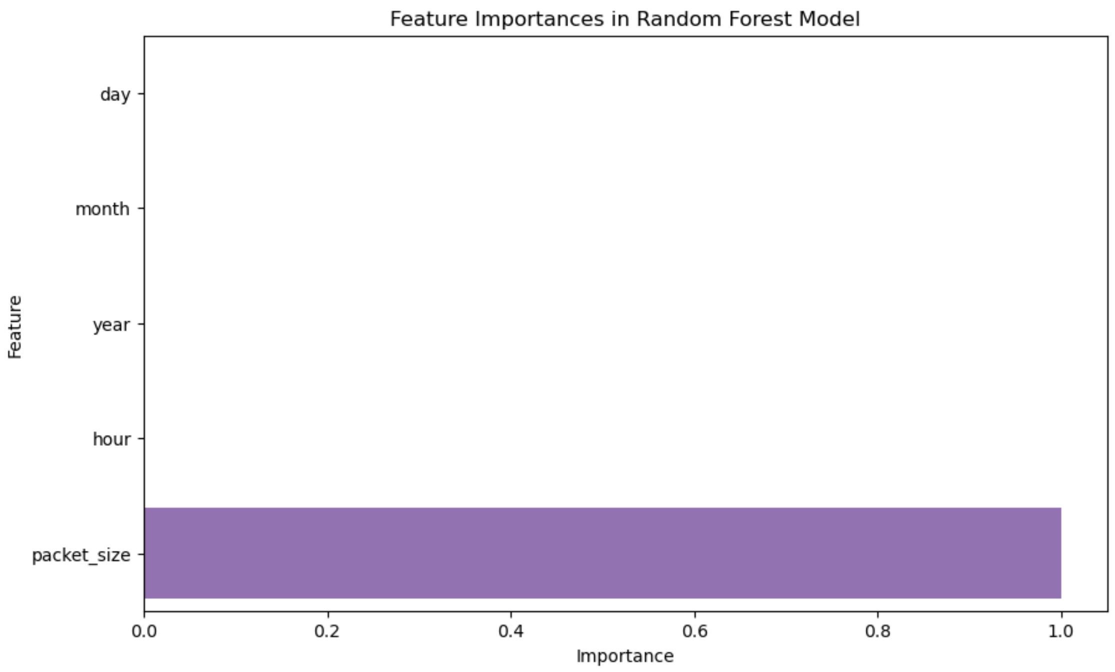
2. **Random Forest:** A more complex model chosen for its ability to handle nonlinear relationships and interactions between features. GridSearchCV was used for hyperparameter tuning to find the best parameters for the model. The performance was evaluated based on accuracy.

Feature Importance in Random Forest Model:

[96]:
```
# Random Forest for better performance
rf_model = RandomForestClassifier()
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5]
}

grid_search = GridSearchCV(rf_model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

best_rf_model = grid_search.best_estimator_
y_pred_rf = best_rf_model.predict(X_test)
print('Random Forest Accuracy:', grid_search.best_score_)
print('Random Forest Best Parameters:', grid_search.best_params_)

# Feature importance
feature_importances = best_rf_model.feature_importances_
plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances, y=features)
plt.title('Feature Importances in Random Forest Model')
plt.xlabel('Importance')
plt.ylabel('Feature')
```

```
    plt.show()
```

```
Random Forest Accuracy: 0.9549290936568052
Random Forest Best Parameters: {'max_depth': 20, 'min_samples_split': 2,
'n_estimators': 100}
```



1.2 Conclusion and Recommendations

Conclusion The dataset provided valuable insights into the nature of IoT network traffic and its vulnerabilities. The predictive models, particularly the Random Forest, demonstrated high accuracy in forecasting potential security threats based on packet data.

Recommendations Efforts should be made to continuously monitor and analyze packet sizes, given their significance in predicting security threats. Collect more comprehensive data, including additional features that might provide deeper insights into security vulnerabilities. Utilize the predictive models developed to implement real-time monitoring systems that can detect and respond to potential threats promptly. Future work should explore the use of deep learning models, given sufficient computational resources and data, to potentially improve predictive performance further. Continuously update and refine models to adapt to evolving security threats and changes in IoT device behavior.