IoT Device Identification from Network Traffic with K - Nearest Neighbors Machine Learning Algorithm

Machine Learning Mid-Term Exam Project
By:
GORASH PLATON
Student no. 22605107

Supervised by:
Hamid Tahaei, Aminreza Karamoozian

Abstract

This study explores the use of the K-Nearest Neighbors (KNN) algorithm for Internet of Things (IoT) device identification based on network traffic patterns. Traditional identification methods, such as signature-based detection, are resource-intensive and not adaptable to the growing diversity of IoT devices. In this work, KNN, a simple yet effective machine learning algorithm, is used to classify IoT devices using features extracted from network traffic data, including protocol details, packet size, and transmission rates.

The work involves preprocessing the dataset, which includes 137 attributes and captures network traffic traces of 18 IoT devices. Data cleaning, feature selection through Random Forest, and hyperparameter tuning of the KNN model are applied to optimize performance. The best model achieved an accuracy of 99.984%, demonstrating the feasibility of using KNN for IoT device identification. The results show that KNN can offer a computationally efficient and accurate alternative for real-time IoT traffic analysis. The findings provide insights into the potential for scalable, machine learning-driven solutions for network security and management in dynamic IoT environments.

Introduction

The popularity of **Internet of Things (IoT)** devices has created an exponential increase in network traffic and, in turn, the requirement for efficient ways through which traffic is monitored and managed. It is important to **identify IoT devices** in networking for network security, performance optimization, and efficient utilization of resources. One such way of identification is the classification of devices in the network into different categories through features extracted from their **network traffic**, which includes packet size, transmission rates, and communication protocol. The features, if recorded correctly, give patterns distinctive for every IoT device, providing the possibility for auto-identification. Large amount of repetitive data with discernible

patterns and the task of classifying IoT devices make this situation a perfect fit for using **machine learning** techniques, both for analysis and actual application.

This assignment is a study that utilizes the **K-Nearest Neighbors (KNN)** machine learning algorithm, a simple but efficient classifying method, and explores how effectively it can differentiate between different types of IoT devices in real network environments. KNN offers simplicity, adaptability, and interpretability (Zhang, 2016), making it an effective and scalable choice for identifying diverse IoT devices based on network traffic without requiring complex training or assumptions about data distribution (Sun, Du, Shi, 2018). The goal is to develop a machine learning model using KNN that can accurately identify IoT devices based on their network traffic patterns, achieving high accuracy by applying appropriate machine learning algorithms, feature selection techniques, and parameter tuning.

The motivation for such a study comes from the increasing demand for efficient and scalable IoT device identification in today's network infrastructures. Signature-based detection and deep packet inspection, among other traditional methods, prove costly in terms of resources and require frequent updates to account for new devices and shifting traffic patterns. Following the exponential growth in the number of IoT devices, the need for a more adaptive approach toward identifying devices is imminent. Fast but robust machine learning methods, such as the KNN algorithm, can identify devices with the help of pattern learning in the absence of any prior knowledge about the devices.

While much research has focused on applying machine learning to IoT traffic analysis, it often centers on complex, resource-demanding algorithms like deep learning and ensemble methods (Kotak, Elovici, 2021). These techniques, while accurate, require substantial computational resources and large labeled datasets, which may not be feasible in constrained environments. Other studies focus on a small subset of the information available from packet inspection, or only some characteristics of it (Pinheiro et al., 2021). Moreover, many papers focus on binary classification or assume ideal, homogeneous IoT environments, overlooking the need for models capable of handling diverse devices and real-world cases.

Data

We will be working with a dataset capturing network traffic traces from 18 distinct IoT devices, collected over a day.

(The dataset *("[RAW] Mid-TermProject_16-09-27_Data_5percent.csv")* is available at https://github.com/gorash-bolatu/ml_midterm/blob/main/%5BRAW%5D%20Mid-TermProject_16-09-27_Data_5percent.csv. A cropped subset version with the first 200 rows is available at https://github.com/gorash-bolatu/ml_midterm/main/subset.csv).

The full dataset comprises 258000 records stored in a CSV format, where each record corresponds to a network frame and includes 137 attributes (including the label) that encapsulate multi-layered protocol details, ranging from Ethernet and ARP headers to transport-layer (TCP/UDP) and application-layer (TLS, HTTP, DNS) metadata. Key attributes include *Frame_length*, *Frame_protocols_in*, source/destination IP and MAC addresses, port numbers,

protocol-specific flags (e.g., TCP flags, ICMP codes), and expert analysis annotations (e.g., retransmission warnings, checksum status). The target variable, *DeviceName*, labels each frame with the originating IoT device (e.g. "Amazon Echo", "Samsung SmartCam", "TPLink Router Bridge LAN", etc.)

<table>
<tr><td colspan="2">Number of instances</td><td colspan="2">258000</td></tr>
<tr><td colspan="2">Number of attributes</td><td colspan="2">137</td></tr>
<tr><td>Attribute information</td><td>Label: DeviceName
Features:
• Frame_length
• Frame_protocols_in
• arp.dst_hw_mac
• arp.dst_proto_ipv4
• arp.hw_size
• arp.hw_type
• arp.opcode
• arp.proto_size
• arp.proto_type
• arp.src_hw_mac
• arp.src_proto_ipv4
• eth.addr
• eth.addr_oui
• eth.addr_oui_resolved
• eth.addr_resolved
• eth.dst
• eth.dst_ig
• eth.dst_lg
• eth.dst_oui
• eth.dst_oui_resolved
• eth.dst_resolved
• eth.ig
• eth.lg
• eth.src
• eth.src_ig
• eth.src_lg
• eth.src_oui
• eth.src_oui_resolved
• eth.src_resolved
• eth.stream
• eth.type
• icmp.checksum
• icmp.checksum_status
• icmp.code
• icmp.data
• icmp.data_len
• icmp.ident
• icmp.ident_le
• icmp.seq
• icmp.seq_le
• icmp.type
• ip.addr
• ip.checksum
• ip.checksum_status</td><td>• ip.dsfield
• ip.dsfield_dscp
• ip.dsfield_ecn
• ip.dst
• ip.dst_host
• ip.flags
• ip.flags_df
• ip.flags_mf
• ip.flags_rb
• ip.frag_offset
• ip.hdr_len
• ip.host
• ip.id
• ip.len
• ip.proto
• ip.src
• ip.src_host
• ip.stream
• ip.ttl
• ip.version
• tcp.
• tcp._ws_expert
• tcp._ws_expert_group
• tcp._ws_expert_message
• tcp._ws_expert_severity
• tcp.ack
• tcp.ack_raw
• tcp.analysis
• tcp.analysis_ack_rtt
• tcp.analysis_acks_frame
• tcp.analysis_bytes_in_flight
• tcp.analysis_flags
• tcp.analysis_initial_rtt
• tcp.analysis_push_bytes_sent
• tcp.checksum
• tcp.checksum_status
• tcp.completeness
• tcp.completeness_ack
• tcp.completeness_fin
• tcp.completeness_rst
• tcp.completeness_str
• tcp.completeness_syn
• tcp.completeness_syn_ack
• tcp.dstport
• tcp.flags
• tcp.flags_ack
• tcp.flags_ae</td><td>• tcp.flags_cwr
• tcp.flags_ece
• tcp.flags_fin
• tcp.flags_push
• tcp.flags_res
• tcp.flags_reset
• tcp.flags_str
• tcp.flags_syn
• tcp.flags_urg
• tcp.hdr_len
• tcp.len
• tcp.nxtseq
• tcp.option_kind
• tcp.option_len
• tcp.options
• tcp.options_nop
• tcp.options_timestamp
• tcp.options_timestamp_tsecr
• tcp.options_timestamp_tsval
• tcp.port
• tcp.seq
• tcp.seq_raw
• tcp.srcport
• tcp.stream
• tcp.time_delta
• tcp.time_relative
• tcp.urgent_pointer
• tcp.window_size
• tcp.window_size_scalefactor
• tcp.window_size_value
• tls.record
• tls.record_content_type
• tls.record_length
• tls.record_version
• udp.
• udp.checksum
• udp.checksum_status
• udp.dstport
• udp.length
• udp.port
• udp.srcport
• udp.stream
• udp.stream_pnum
• udp.time_delta
• udp.time_relative</td></tr>
</table>

Classes	• Class 1: Dropcam • Class 2: TPLink Router Bridge LAN (Gateway) • Class 3: Netatmo Welcome • Class 4: Amazon Echo • Class 5: Smart Things • Class 6: Withings Smart Baby Monitor • Class 7: Samsung SmartCam • Class 8: TP-Link Day Night Cloud camera • Class 9: Belkin Wemo switch	• Class 10: Triby Speaker • Class 11: HP Printer • Class 12: Belkin wemo motion sensor • Class 13: PIX-STAR Photo-frame • Class 14: TP-Link Smart plug • Class 15: Netatmo weather station • Class 16: Samsung Galaxy Tab • Class 17: NEST Protect smoke alarm • Class 18: Withings Smart scale

With 137 features, the dataset presents challenges in dimensionality reduction, noise filtering, and computational efficiency. *Frame_length*, *udp.dstport*, and *udp.length* and other columns are numerical and contain continuous values. Columns like *Frame_protocols_in* contain text-based data representing the protocols involved in network communication (e.g., "eth:ethertype:ip:tcp:tls"). Some features, such as *eth.dst_ig*, are represented as boolean values (True/False or 1/0). Many of these columns may also have missing or null values, which should be taken into consideration during data handling.

Attributes such as *eth.addr_oui_resolved* (resolved manufacturer names) and *ip.dsfield_dscp* (QoS markers) may offer discriminative power but require careful preprocessing. Sequential attributes like *tcp.time_relative* and *tcp.stream* provide temporal context, enabling analysis of session persistence and inter-packet timing.

Overall, the dataset serves as a good base for ML-driven IoT traffic analysis, with implications for network forensics, policy compliance, and anomaly detection systems.

Methodology

Our data preprocessing and machine learning pipeline will be implemented using the **Python** programming language. We will use several libraries, including:

- **pandas** for data manipulation;
- **scikit-learn** for imputation, encoding, model training, and evaluation;
- **matplotlib** and **seaborn** for visualizations;
- **numpy** for additional visualization help.

Inspecting the dataset shape, removing unreliable columns based on domain knowledge, duplicate dropping is done with **pandas**. Missing values are imputed with a custom technique and then **SimpleImputer** from **scikit-learn** (where numerical features are imputed with the mean, and categorical features are imputed with the most frequent value). **One-hot encoding** is applied to categorical columns (but only those with fewer than 50 unique values, as too many different values leads to unnecessary computations). Next, the dataset is split into features (**X**) and labels (**y**), with a **random forest classifier** used to identify the most important features. Finally, these top features are selected for the final model, and the numerical columns in them are scaled using **StandardScaler**.

We then train a **K-Nearest Neighbors (KNN) classifier** with **hyperparameter tuning** for three main parameters: **k**, **weights** (uniform/distance), and **p** (power parameter that influences calculation distance metrics). A **3D scatter** plot visualizes the relationship between hyperparameters and accuracy. The best model is then selected and tested, and its performance is reported, including execution times for each phase. The code also visualizes a **confusion matrix** with heatmaps through **seaborn**, indicating the classifier's accuracy across various device classes.

This handles preprocessing, feature selection, model tuning, and performance evaluation to create an optimized KNN classification model for identifying IoT devices based on network traffic patterns.

Implementation: Data preprocessing

Loading and exploring the dataset with pandas:

```
import pandas as pd

df = pd.read_csv('[RAW] Mid-TermProject_16-09-27_Data_5percent.csv')
print(df.info())
print(df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 258000 entries, 0 to 257999
Columns: 137 entries, DeviceName to udp.time_relative
dtypes: bool(6), float64(64), int64(5), object(62)
memory usage: 259.3+ MB
None
```

	DeviceName	Frame_length	Frame_protocols_in	arp.dst_hw_mac	arp.dst_proto_ipv4	arp.hw_size	arp.hw_type	arp.opcode	arp.proto_size	arp.proto_type	...
0	Dropcam	156	eth:ethertype:ip:tcp:tls	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
1	TPLink Router Bridge LAN (Gateway)	66	eth:ethertype:ip:tcp	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
2	Dropcam	284	eth:ethertype:ip:tcp:tls	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
3	Dropcam	156	eth:ethertype:ip:tcp:tls	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
4	TPLink Router Bridge LAN (Gateway)	90	eth:ethertype:ip:udp:ntp	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...

5 rows × 137 columns

The following features:

- eth.addr, eth.addr_oui, eth.addr_oui_resolved, eth.addr_resolved, eth.dst, eth.dst_oui_resolved, eth.dst_resolved, eth.src, eth.src_ig, eth.src_lg, eth.src_oui, eth.src_oui_resolved, eth.src_resolved;
- ip.addr, ip.dst, ip.dst_host, ip.host, ip.src, ip.src_host;
- arp.dst_hw_mac, arp.dst_proto_ipv4, arp.src_hw_mac, arp.src_proto_ipv4;

must be dropped from the dataset because they are changeable and can be spoofed. Attackers can easily fake MAC/IP addresses or manipulate ARP entries to pretend to be legitimate devices, making these features unreliable for accurately identifying IoT devices. Keeping these in would introduce vulnerabilities, as models trained on such data could mistakenly classify malicious traffic that’s pretending to be normal devices. To improve generalization and protect against spoofing attacks, device classification should focus on

behavioral patterns (e.g., protocol usage, packet timing/size) instead of easily falsifiable network-layer identifiers.

```
print("Original shape:", df.shape)
df = df.drop(['eth.addr', 'eth.addr_oui', 'eth.addr_oui_resolved',
'eth.addr_resolved',
              'eth.dst', 'eth.dst_oui_resolved', 'eth.dst_resolved', 'eth.src',
'eth.src_ig',
              'eth.src_lg', 'eth.src_oui', 'eth.src_oui_resolved',
'eth.src_resolved', 'ip.addr',
              'ip.dst', 'ip.dst_host', 'ip.host', 'ip.src', 'ip.src_host',
'arp.dst_hw_mac',
              'arp.dst_proto_ipv4', 'arp.src_hw_mac', 'arp.src_proto_ipv4'],
axis=1)
print("Shape after removing unreliable data:", df.shape)
```

```
Original shape: (258000, 137)
Shape after removing unreliable data: (258000, 114)
```

The dataset contains many redundant duplicate rows that may negatively influence classification algorithms, so we drop duplicates:

```
df = df.drop_duplicates(keep="first")
print("Shape after removing duplicate rows:", df.shape)
```

```
Shape after removing duplicate rows: (217509, 114)
```

The data has missing or null values. However, due to their large amount, simply dropping columns and rows containing them would remove crucial information for device classification. As an example, columns with the *arp* prefix have no values unless the ARP packet protocol is used in the transmission (evident from the *Frame_protocols_in*); since majority of the traffic in the dataset uses the IP protocol, removing such records with empty values is unfeasible. Instead, we may impute missing numerical values with the column's mean, and categorical values with the mode (most common value). However, the dataset also contains many columns with only one non-null value; the given method would simply duplicate this value across the entire column. To solve this problem, we use the following method:

```
from sklearn.impute import SimpleImputer

# For numerical cols with only 1 type of value (and N/A), replace N/A with zeroes
numerical_cols = df.select_dtypes(include=['float64', 'int64']).columns
for i in df[numerical_cols]:
    if (len(df[i].unique()) < 2):
        df[numerical_cols] = df[numerical_cols].fillna(0)

# For other numerical cols, simply replace empty values with the mean
num_imputer = SimpleImputer(strategy='mean')
df[numerical_cols] = num_imputer.fit_transform(df[numerical_cols])

# For string columns with only 1 type of value (and N/A), replace N/A with the
string "N/A"
```

```
object_cols = df.select_dtypes(include=['object']).columns
for a in df[object_cols]:
    if (len(df[a].unique()) < 2):
        df[a] = df[a].fillna("N/A")
# For boolean columns with only 1 type of value (and N/A), replace N/A with the
inverse of that boolean
bool_cols = df.select_dtypes(include=['bool']).columns
for b in df[bool_cols]:
    if (len(df[b].unique()) < 2):
        df[b] = df[b].fillna(not bool(df[b].mode()[0]))
categorical_cols = df.select_dtypes(include=['object', 'bool']).columns

# For other categorical columns, simply replace empty values with the mode
cat_imputer = SimpleImputer(strategy="most_frequent")
df[categorical_cols] = cat_imputer.fit_transform(df[categorical_cols])
```

Next, since the KNN algorithm that we chose is a distance-based algorithm that requires numerical input to calculate similarity between data points, categorical features need to be converted into a numerically comparable format. Encoding transforms each category into a separate binary column, allowing the KNN algorithm to treat each category independently and avoid misinterpreting their relationships. This is particularly important for nominal data, where categories have no inherent order, ensuring the algorithm doesn't mistakenly assume an ordinal relationship between them.

However, in our data, many categorical columns contain too much different values (we give 50 as an arbitrary threshold), which creates too much overhead and new columns when trying to encode. These features are mainly checksums, timestamp and IDs – not particularly useful for discerning IoT devices, so may be assumed to be safe to remove.

```
categorical_features = categorical_cols.drop(['DeviceName'])

print("Dropping categorical features with too many different values:")
for i in df[categorical_features]:
    if (len(df[i].unique()) > 50):
        print("\t", i, ": " , len(df[i].unique()), sep="")
        categorical_features = categorical_features.drop([str(i)])
df[categorical_features].head()
```

```
Dropping categorical features with too many different values:
        Frame_protocols_in: 64
        icmp.checksum: 9384
        ip.checksum: 63301
        ip.id: 62293
        tcp._ws_expert: 144
        tcp._ws_expert_message: 144
        tcp.checksum: 60297
        tcp.options: 141096
        tcp.options_timestamp: 141018
        udp.checksum: 5645
```

If we were to use labels directly as integers (e.g., mapping "TCP" = 1, "UDP" = 2, "ICMP" = 3…), KNN might mistakenly interpret the categories as having an ordinal relationship (i.e., UDP being "closer" to TCP than to ICMP), which isn't meaningful for many categorical variables. The One-Hot Encoding technique eliminates this problem by treating each category as a separate, independent entity.

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False)
# Fit and transform the categorical features
encoder.fit(df[categorical_features])
encoded = encoder.transform(df[categorical_features])
# Convert back into pandas' DataFrame
df_encoded = pd.DataFrame(encoded,
columns=encoder.get_feature_names_out(categorical_features))

print(df_encoded.head())
```

	arp.proto_type_0x0800	eth.dst_ig_False	eth.dst_ig_True	eth.dst_lg_False	eth.dst_lg_True	eth.ig_False	eth.ig_True	eth.lg_False
0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0
1	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0
2	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0
3	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0
4	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0

5 rows × 118 columns

```
print("Concatenating encoded columns with numerical...")
X = pd.concat([df.drop(columns=categorical_cols), df_encoded], axis=1)
print("Features (x):")
print(X.head())
```

```
Concatenating encoded columns with numerical...
Features (x):
```

	Frame_length	arp.hw_size	arp.hw_type	arp.opcode	arp.proto_size	eth.dst_oui	eth.stream	icmp.checksum_status	icmp.code
0	156.0	6.0	1.0	1.462687	4.0	1362976.0	0.0	1.0	0.040768
1	66.0	6.0	1.0	1.462687	4.0	3181819.0	0.0	1.0	0.040768
2	284.0	6.0	1.0	1.462687	4.0	1362976.0	0.0	1.0	0.040768
3	156.0	6.0	1.0	1.462687	4.0	1362976.0	0.0	1.0	0.040768
4	90.0	6.0	1.0	1.462687	4.0	13652648.0	1.0	1.0	0.040768

5 rows × 185 columns

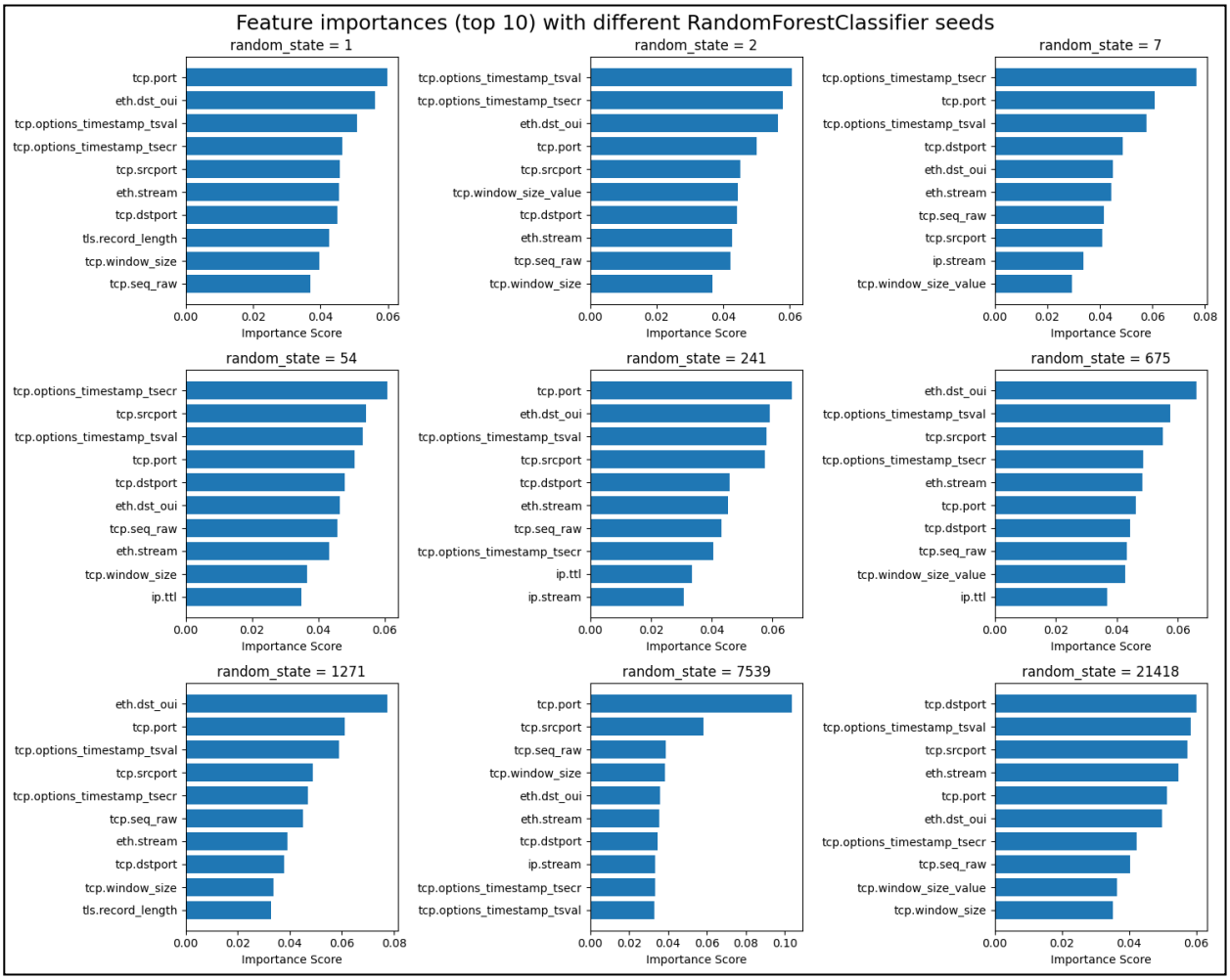
Implementation: Feature extraction and scaling

In our processed dataset, we have 185 columns (1 label columns and 184 features), which is a considerable number to train a model on. To reduce this number and keep only the necessary

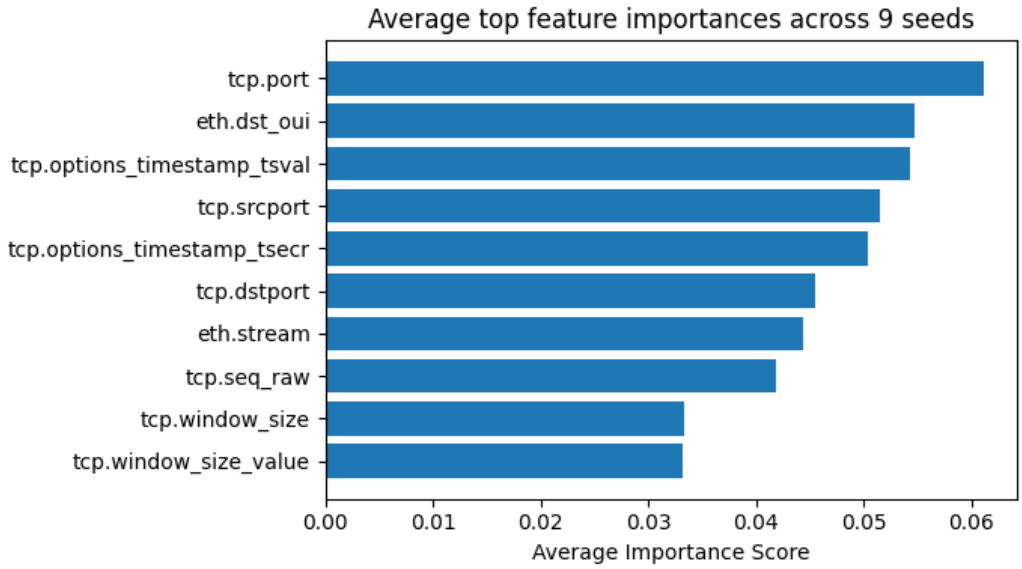
features, we perform feature extraction by using a Random Forest classifier. It identifies the most important features for device identification based on their ability to differentiate between classes, as Random Forests inherently account for interactions between features during the model's training process. This is critical for IoT traffic patterns, where device behavior may depend on combinations of features (e.g., packet size + protocol).

The Random Forest model is trained multiple times with different preset random seeds, and the feature importances are averaged across these runs to ensure stability and reliability. The top 10 most relevant features are selected based on their mean importance scores, which are then used for the final classification model. This method helps eliminate less relevant or redundant features, improving the model's efficiency and performance.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)

seeds = [1, 2, 7, 54, 241, 675, 1271, 7539, 21418] # Seeds for reproducibility
top_n = 10 # Use 10 most important features
all_importances = np.zeros((len(seeds), len(X_train.columns)))
# Store importances with every seed to compute mean later

# Create a 3x3 grid of subplots
fig, axes = plt.subplots(3, 3, figsize=(15, 12))
fig.suptitle('Feature importances (top 10) with different RandomForestClassifier
seeds', fontsize=18)

for i, seed in enumerate(seeds):
    # Train RandomForest with current seed
    rf = RandomForestClassifier(
        bootstrap=True, max_samples=0.5,
        n_estimators=100, max_depth=10,
        n_jobs=-1, random_state=seed)
    rf.fit(X_train, y_train)
    # Get and sort feature importances
    importances = rf.feature_importances_
    all_importances[i] = importances
    sorted_idx = np.argsort(importances)[::-1][:top_n]
    sorted_importances = importances[sorted_idx]
    sorted_feature_names = [X_train.columns[j] for j in sorted_idx]
    # Select subplot (3x3 grid)
    ax = axes[i//3, i%3]
    # Plot horizontal bar chart
    ax.barh(range(len(sorted_feature_names)), sorted_importances)
    ax.set_yticks(range(len(sorted_feature_names)))
    ax.set_yticklabels(sorted_feature_names)
    ax.invert_yaxis() # Most important at top
    ax.set_xlabel('Importance Score')
    ax.set_title(f'random_state = {seed}')

plt.tight_layout()
plt.show()
```




```python
# Compute mean importances across seeds
mean_importances = np.mean(all_importances, axis=0)
# Sort features by mean importance (descending)
sorted_idx = np.argsort(mean_importances)[::-1][:top_n]
sorted_importances = mean_importances[sorted_idx]
sorted_feature_names = [X.columns[j] for j in sorted_idx]
# Plot the averaged importances
plt.figure(figsize=(7, 4))
plt.barh(range(len(sorted_feature_names)), sorted_importances)
plt.yticks(range(len(sorted_feature_names)), sorted_feature_names)
plt.gca().invert_yaxis()
plt.xlabel('Average Importance Score')
plt.title('Average top feature importances across 9 seeds')
plt.tight_layout()
plt.show()
```



After performing feature extraction (cut down to 10 most important features):

```
X = X[sorted_feature_names]
print(X.shape)
print(X.head())
```

Python

```
(217509, 10)
```

	tcp.port	eth.dst_oui	tcp.options_timestamp_tsval	tcp.srcport	tcp.options_timestamp_tsecr	tcp.dstport	eth.stream	t
0	40767.000000	1362976.0	2.318916e+07	40767.000000	4.049397e+09	443.000000	0.0	9.1
1	443.000000	3181819.0	4.049398e+09	443.000000	2.318916e+07	40767.000000	0.0	6.7
2	40767.000000	1362976.0	2.318921e+07	40767.000000	4.049398e+09	443.000000	0.0	9.1
3	40767.000000	1362976.0	2.318937e+07	40767.000000	4.049398e+09	443.000000	0.0	9.1
4	22005.384225	13652648.0	1.550935e+09	22005.384225	1.568876e+09	20861.871223	1.0	1.3

Finally, we do numerical feature scaling. Distance-based algorithms like KNN are sensitive to the scale of features because they compute distances between data points. Features with larger ranges can dominate the distance calculation, leading to biased models. (We perform scaling after feature extraction with Random Forest since Tree-Based Algorithms, including Random Forests, are generally insensitive to feature scaling because they split nodes based on feature thresholds).

In this case, feature scaling is applied to the numerical features using StandardScaler from scikit-learn. This step standardizes the data by transforming the features to have a mean of 0 and a standard deviation of 1, ensuring that all numerical variables are on a similar scale. Scaling is crucial for distance-based algorithms like KNN, as it prevents features with larger ranges from dominating the distance calculations and ensures that each feature contributes equally to the model's performance.

```
# Feature scaling for numerical columns
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
numerical_cols = X.select_dtypes(include=['float64', 'int64']).columns
X[numerical_cols] = scaler.fit_transform(X[numerical_cols])
print(X[numerical_cols].head())
```

	tcp.port	eth.dst_oui	tcp.options_timestamp_tsval	tcp.srcport	tcp.options_timestamp_tsecr	tcp.dstport	eth.stream
0	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240	-0.478796
1	-1.098122e+00	0.023567	1.645411	-1.098122e+00	-1.019906e+00	1.022841	-0.478796
2	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240	-0.478796
3	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240	-0.478796
4	1.852737e-16	2.730754	0.000000	1.852737e-16	1.573181e-16	0.000000	-0.388224

And we combine it into the final processed dataset:

	DeviceName	tcp.port	eth.dst_oui	tcp.options_timestamp_tsval	tcp.srcport	tcp.options_timestamp_tsecr	tcp.dstport
0	Dropcam	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240
1	TPLink Router Bridge LAN (Gateway)	-1.098122e+00	0.023567	1.645411	-1.098122e+00	-1.019906e+00	1.022841
2	Dropcam	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240
3	Dropcam	9.554852e-01	-0.446687	-1.006126	9.554852e-01	1.636748e+00	-1.049240
4	TPLink Router Bridge LAN (Gateway)	1.852737e-16	2.730754	0.000000	1.852737e-16	1.573181e-16	0.000000

Implementation: Model training and evaluation

Model training and evaluation begin with hyperparameter tuning of the K-Nearest Neighbors classifier. Different combinations of hyperparameters, such as the number of neighbors *(k)*, the weighting scheme (*uniform* or *distance*), and the *p* value that defines the exponents in distance calculation (p=1 – Manhattan distance, p=2 – Minkowski distance), are tested. For each combination, the model is trained on the training data, and its performance is evaluated using accuracy on the test set. This process is repeated for multiple iterations using 3 different random *k* values to ensure robustness and to avoid overfitting to any particular set of random splits.

```
import time
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

print("Execution start...")
start_time = time.time()

# Values to save best hyperparameters
best_accuracy = 0
best_params = None
# Saving values to later visualize against accuracy
data = list()
```

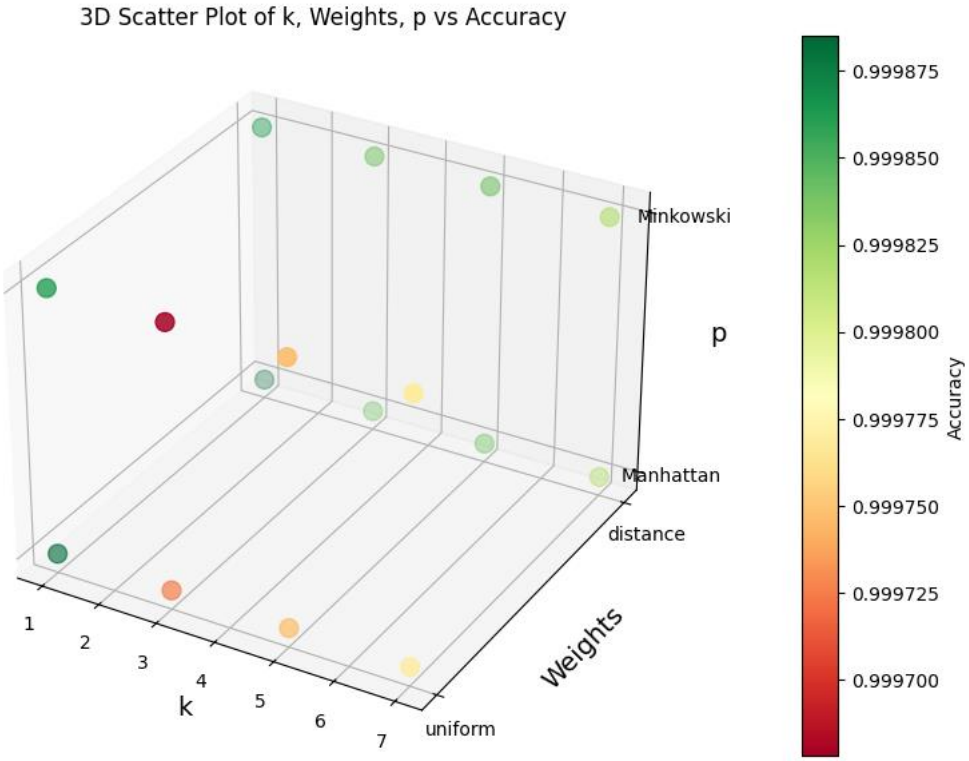
```
print("Data processing start...")
data_processing_start = time.time()

print("Test/train split...")

X = df.drop(['DeviceName'], axis=1)
y = df['DeviceName']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=80)

print("Hyperparameter tuning...")
i = 0
for k in [1, 3, 7]: # Arbitrary k values
    for weights in ['uniform', 'distance']:
        for p_val in [1, 2]: # Manhattan/Minkowski distance
            i += 1
            print(f"\tIteration {i}/12: k={k}, weights={weights}, p={p_val}")

            # Initialize the KNN classifier with current hyperparameters
            knn = KNeighborsClassifier(n_neighbors=k, weights=weights, p=p_val)

            # Train the model
            print("\t\tModel training...", end=" ")
            training_start = time.time()
            knn.fit(X_train, y_train)
            training_time = time.time() - training_start
            print(f"{training_time:.4f}sec")

            # Evaluate the model
            print("\t\tModel evaluation...", end=" ")
            evaluation_start = time.time()
            y_pred = knn.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)

            # Check if the current model is the best
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_params = (k, weights, p_val)
            evaluation_time = time.time() - evaluation_start
            print(f"{evaluation_time:.4f}sec")
            print(f"\t\tAccuracy: {accuracy:.8f}")

            # Save results
            if p_val == 1:
                p_str = "Manhattan"
            else:
                p_str = "Minkowski"
            data.append({'k': k, 'weights': weights, 'p': p_str,
                         'train_time': training_time,
                         'eval_time': evaluation_time,
                         'accuracy': accuracy})
```

```
data_processing_time = time.time() - data_processing_start
print("Data processing finished")
```

```
Execution start...
Data processing start...
Test/train split...
Hyperparameter tuning...
        Iteration 1/12: k=1, weights=uniform, p=1
                Model training... 2.8384sec
                Model evaluation... 7.5448sec
                Accuracy: 0.99983909
        Iteration 2/12: k=1, weights=uniform, p=2
                Model training... 2.1666sec
                Model evaluation... 13.4897sec
                Accuracy: 0.99981610
        Iteration 3/12: k=1, weights=distance, p=1
                Model training... 1.8600sec
                Model evaluation... 3.1648sec
                Accuracy: 0.99983909
        Iteration 4/12: k=1, weights=distance, p=2
                Model training... 2.6285sec
                Model evaluation... 3.3475sec
                Accuracy: 0.99981610
        Iteration 5/12: k=3, weights=uniform, p=1
                Model training... 1.9897sec
                Model evaluation... 7.8686sec
                Accuracy: 0.99965519
        Iteration 6/12: k=3, weights=uniform, p=2
...
```

	k	weights	p	train_time	eval_time	accuracy
0	1	uniform	Manhattan	2.838424	7.544785	0.999839
1	1	uniform	Minkowski	2.166570	13.489694	0.999816
2	1	distance	Manhattan	1.859959	3.164820	0.999839
3	1	distance	Minkowski	2.628503	3.347493	0.999816
4	3	uniform	Manhattan	1.989735	7.868621	0.999655

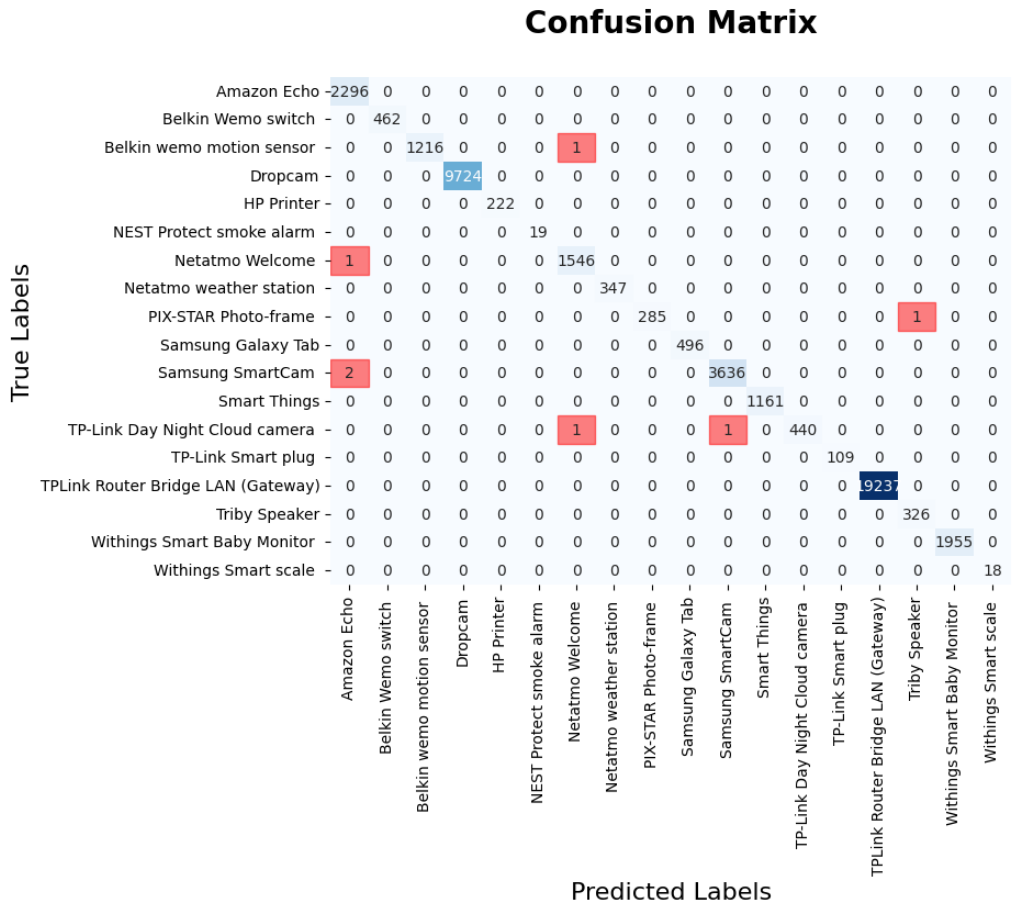
After training, the best performing combination of hyperparameters is selected based on the highest accuracy achieved during testing. This process ensures that the model generalizes well to unseen data. In our case, the parameters that give the best accuracy are k=1, *weights*='uniform' and p=1 (Manhattan distance).



The final model is then retrained using the best hyperparameters on the full training data, and its performance is evaluated once again on the test set. The accuracy score, along with other

performance metrics like confusion matrices, is used to assess the model's effectiveness in correctly classifying IoT devices.

```
# Training phase
print("Final model training...", end=" ")
training_start = time.time()
best_knn = KNeighborsClassifier(n_neighbors=best_params[0],
weights=best_params[1], p=best_params[2])
best_knn.fit(X_train, y_train)
training_time = time.time() - training_start
print(f"{training_time:.4f}sec")

# Test phase
print("Final model testing & evaluation...", end=" ")
evaluation_start = time.time()
y_pred_final = best_knn.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred_final)
evaluation_time = time.time() - evaluation_start
print(f"{evaluation_time:.4f}sec")

print("Accuracy:", accuracy)

# Report results
print(f"Final model training time: {training_time:.4f}sec")
print(f"Final model evaluation time: {evaluation_time:.4f}sec")
print("Model parameters:", best_params)
print("Accuracy score:", test_accuracy)
```



```
Final model training time: 2.6175sec
Final model evaluation time: 11.1326sec
Model parameters: (1, 'uniform', 1)
Accuracy score: 0.9998390878580296
```

The model's great performance (more than 99.9% using scikit-learn s *accuracy_score* metric) showed that even a simple KNN classifier is able to correctly classify IoT devices using correctly selected and preprocessed features from this structured dataset. Visualizing the confusion matrix, we see that in our train/test split case, only 7 devices were misclassified.

Results

The final KNN model achieved an impressive accuracy of 99.984% after extensive hyperparameter tuning, which involved testing different combinations of *k*, weighting schemes, and distance metrics. The best-performing model used *k*=1, uniform weights, and the Manhattan distance metric.

Each KNN model classifier took about 2 seconds to train and 8 seconds to train on average. The overall data handling, training and evaluation process took approximately 116 seconds, with most of the time spent on data processing (102 seconds). Time analysis reveals that the training and evaluation times are influenced by the hyperparameters. Training time remains relatively stable, with a slight increase as *k* grows, and the effect of *weights* and *p* on training time is minimal. Evaluation time, however, varies significantly: models with *weights*=uniform tend to have longer evaluation times, and the Minkowski distance metric (*p*=2) results in slower evaluations compared to the Manhattan distance (*p*=1). The most noticeable bottleneck is during the evaluation phase, particularly with the combination of larger *k*, Minkowski distance, and 'uniform' *weights*.

While the model performed exceptionally well, achieving near-perfect accuracy, there is room for improvement. Most importantly, the value of *k*=1 is associated with a high bias and the risk of overfitting; testing other hyperparameter combinations and data preprocessing techniques may help mitigate this. Further optimization could be done by experimenting with other machine learning algorithms like Random Forest or SVM to compare performance. Additionally, the model could benefit from handling more diverse real-world data to ensure its robustness in less controlled environments.

Contribution

Understanding the behavioral patterns of IoT devices through network traffic is critical for intrusion detection, anomaly identification, and policy enforcement. Using KNN, an inherently adaptive and explainable algorithm, this work proves that basic algorithms can be used proficiently for the classification of advanced types of devices based on network traffic features. Lastly, the work emphasizes the value in the use of feature selection and data preprocessing in order for the model to perform well in the IoT environments.

The study lays the groundwork for further research looking into the use of machine learning for IoT network management. It gives insight into integrating machine learning models into real-time IoT monitoring systems for the purposes of enhanced device detection and security. The insights of the research could be used to educate best practices among IoT network administrators, cybersecurity professionals, and IoT equipment manufacturers looking to enhance network security and operational effectiveness in diverse and dynamic IoT ecosystems.

Conclusion

This project aimed to develop a machine learning model using the K-Nearest Neighbors (KNN) algorithm for identifying IoT devices based on their network traffic patterns. The dataset, containing 137 features extracted from network traffic traces of 18 different IoT devices, was preprocessed by handling missing data, dropping unreliable features (like IP and MAC addresses), and performing one-hot encoding for categorical variables. Feature selection was applied using Random Forest to identify the top 10 most important features, followed by scaling the numerical features using StandardScaler to improve KNN performance.

After extensive hyperparameter tuning, the final model achieved an impressive accuracy of 99.984%. The model's performance was analyzed in terms of training and evaluation times, with a noticeable bottleneck during the evaluation phase, particularly when using larger *k* values, Minkowski distance calculation, and uniform weights. Despite the performance, the model could benefit from testing with more diverse real-world data, experimenting with other algorithms like Random Forest or SVM, and fine-tuning other hyperparameters (e.g., to help avoid the risk of overfitting when *k*'s value is as small as 1).

This project demonstrates the viability of using simpler, more efficient algorithms like KNN for IoT device identification and emphasizes the importance of feature selection and data preprocessing in achieving high performance. The work provides a solid foundation for further research into machine learning applications in IoT network security, management, and monitoring.

References

1. Zhang, Z. (2016). Introduction to machine learning: K-nearest neighbors. Annals of Translational Medicine, 4, 218.
2. Sun, J., Du, W., & Shi, N. (2018). A survey of kNN algorithm. Information Engineering and Applied Computing, 1.
3. Cheng, D., Zhang, S., Deng, Z., Zhu, Y., & Zong, M. (2014). kNN algorithm with data-driven k value. 499-512.
4. Jiang, Y. (2024) IoT Device Identification Based on IP Traffic. Journal of Computing and Electronic Information Management. 15, 138-142.
5. Kotak, J., Elovici, Y. (2021) IoT device identification using deep learning. 13th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2020), Springer International Publishing. 12. 76-86.

6. Pinheiro, A.J. et.al (2019) Identifying IoT devices and events based on packet length from encrypted traffic. Computer Communications, 144, 8-17.

7. Tahaei, H., Liu, A., Forooghikian, H., Gheisari, M., Zaki, F., Anuar, N., Fang, Z., & Huang, L. (2025). Machine learning for Internet of Things (IoT) device identification: A comparative study. PeerJ Computer Science, 11.