Predicting Website Fingerprinting with Machine Learning

Elliot Hong (kyk4ge), Ivan Kisselev (wnc8zw), Kyle Vitayanuvatti (acd6hn)
CS 4501: Privacy in the Internet Age with Dr. Yixin Sun
December 11, 2024

https://github.com/KyleVitayanuvatti/Website-Fingerprinting

Background/Purpose:

Website Fingerprinting describes a special attack technique used to deanonymize users and identify the websites these users browse. It threatens the anonymity of the users' privacy since these users often use tools like Tor or FireFox to conceal their footprint. In a previous homework, we performed traffic correlation to link source-to-entry nodes with exit-to-server nodes, but this project serves as an alternate approach through website fingerprinting: capturing packets of user traffic, extracting features from these packets, and training models on these features. In this way, the attack can be performed using traffic from a single point in the communication stream. Our goal for this project was to investigate the whole process of training a website fingerprinting ML model including packet captures, feature extraction, model creation and training, model evaluation, and overall analysis.

Methodology:

1. **Packet Captures**
    a. Code is in the Python script collect.py
    b. Collected links to 100 different websites
    c. We opted to use Firefox as our browser of choice, primarily due to its simple integration with Selenium and its similarity in some aspects to the Tor Browser's underlying structure. The success of our project would also serve as a proof of concept that a similar methodology could be applied to the Tor network.
    d. We used the Selenium and PyShark Python libraries to visit each website for 3 seconds and collect traffic info during that time, resulting in a single packet capture [1] [2]
    e. Each website was visited a total of 5 times and collected 5 .pcap files, 4 of which would be used for training data with 1 used for testing data (80:20 ratio). From our own research this seemed to be a standard distribution of training and testing data. [6]
    f. In ideal circumstances this would take around 25 minutes (5 x 100 x 3 sec = 1500 sec = 25 mins), though in reality lagging issues with Selenium and PyShark made the process take about an hour or more.

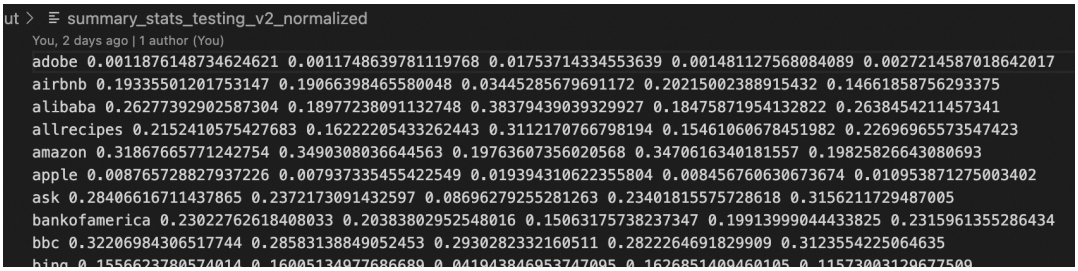
g. For this reason we implemented a “resume” feature to pick up where the captures left off to help address time constraints and interruptions
h. The packet captures took an excessive amount of space (>3GB), and therefore we did not upload them to our GitHub repository. We stored the ‘training’ and ‘testing’ packet capture folders in the root directory of our local repository. Here is a link to them on google drive: https://drive.google.com/drive/folders/1iU8otBuUv4dvWm9UXcfNNWNn-rwFe6OQ?usp=drive_link
i. Here is a brief demo of the packet capture script in action: https://drive.google.com/file/d/1r0EQh2_e56mO0RCR4qxwisjdTJzSa9fS/view?usp=sharing

2. Feature Extraction

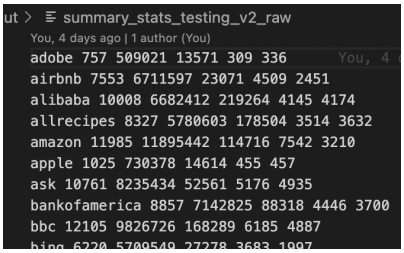
a. Code is in src/feature_extractor.py and src/feature_extractor2.py
b. For feature_extractor.py:
    i. Feature Extraction 1: Extracts and outputs summary statistics for each packet capture into four output files:
        1. summary_stats_training_raw
        2. summary_stats_training_normalized
        3. summary_stats_testing_raw
        4. summary_stats_testing_normalized
        5. Output format (per line):
        [label] [numPackets] [total_data_sent] [stdev_arrival_times] [avg_inter_arrival_time] [median_arrival_time]
    ii. Feature Extraction 2: aggregated data (data per interval) for each packet capture into four output files:
        1. aggregated_data_training_raw
        2. aggregated_data_training_normalized
        3. aggregated_data_testing_raw
        4. Aggregated_data_testing_normalized
        5. Output format (per line):
        [label] [data_sent] [data_sent_1] ... [data_sent_30]
    iii. Feature Extraction 3: Extracts and outputs summary statistics for each packet capture into four output files:

1. summary_stats_training_v2_raw
2. summary_stats_training_v2_normalized
3. summary_stats_testing_v2_raw
4. Summary_stats_testing_v2_normalized
5. Output format (per line):
   [label] [numPackets] [total_data_sent] [TLS handshake size] [num large packets (> 1000 bytes)] [num small packets (< 100 bytes)]

- All data was normalized using min/max scaling in the "normalized" files and stored without modification in the "raw" files.
- The scaler was fit and applied on training data and just applied to the testing data. Since in practice we would not have labeled testing data while making our model, it made sense not to train the scaler on it either.
- Running each of the 3 feature extraction functions took extremely long, likely due to the size of the .pcap files, which in total took up over 3 GB of data. On average, each extraction took about 3 hours to run from start to finish.
- Below is what the normalized data looked like when outputted:



- Below is what the raw data looked like when outputted:



c. For feature_extractor2.py:
   i. Extracts and outputs detailed summary statistics for each .pcap file in the specified directory (training or testing) and saves the features and labels into two output files:
      1. Features.npy: A NumPy array containing the feature vectors.

2. Labels.npy: A NumPy array containing the corresponding labels for the .pcap files.

ii. Extracted Features

1. numPackets: Total number of packets in the .pcap file.
2. avg_packet_size: Average size of packets across the entire capture.
3. max_packet_size: Maximum packet size encountered in the capture.
4. fraction_large_packets: Fraction of packets in the capture that are larger than 1000 bytes.
5. unique_ip_count: The count of unique IP addresses (both source and destination) in the capture.

iii. **Important Note**: Our team had multiple issues with the .npy based output of this feature extractor. We ran it multiple times and were unable to obtain output results which included extractions from all the data in our dataset. For this reason, and because we already had plenty of useful information from our other extractor, we didn't move forward with the data produced by this extractor. Instead, our team only used the data produced by feature_extractor.py to train our models in hopes of making the best use of our time.

3. Model Development & Training

d. Since the data was already normalized with min/max scaling in the feature extraction phase of our process, this was an unnecessary step in model development. We decided to go with a keras sequential feedforward neural network. This would support our goal of classification with a large number of classes (100). To highlight the differences in feature extractions in regard to model performance, we opted to make each model as similar as possible. Here is how we created the models trained on each feature extraction:

```
model = Sequential([
    Dense(128, activation="relu", input_shape=(values_training.shape[1],)),
    Dense(64, activation="relu"),
    Dense(32, activation="relu"),
    Dense(labels_training.shape[1], activation="softmax")
])

model.compile(optimizer=Adam(learning_rate=0.001), loss="categorical_crossentropy", metrics=["accuracy"])
```

- This model included 4 layers which were all dense (each neuron in a layer was connected to each neuron in the previous layer) [3]
    - Layer 1 (128 neurons)

        - This was the input layer and its shape was defined by the number of features in the input dataset
        - Used ReLU (Rectified Linear Unit) activation function
            - This activation function, f(x)=max(0,x), is used to introduce nonlinearity in the neural network [4]
            - We chose this primarily due to its simplicity and popularity
    - Layer 2 (64 neurons)
        - Also used ReLU (Rectified Linear Unit) activation function
    - Layer 3 (32 Neurons)
        - Also used ReLU (Rectified Linear Unit) activation function
    - Layer 4 (# neurons = # classes)
        - Output layer
        - Used softmax activation function
            - takes as input a vector of real numbers, and normalizes it into a probability distribution over our classes. This allows us to output a class prediction as the final step. [5]
- To compile the model we used the categorical cross entropy loss function which is often used for classification problems with multiple classes
- The performance of the model was judged on its prediction accuracy
- We chose the Adam (Adaptive Moment Estimation) optimizer due to its popularity and a slow learning rate of 0.001 to not accidentally overshoot the optimal solution

e. First, we loaded the feature extracted data from the corresponding feature extraction files and parsed it to store the data and labels in numpy arrays. The labels were encoded so the model could make predictions and output the labels.

f. We split up our training data into a training and validation set (3:1 ratio) to give our model some independent data to check on and reduce overfitting as much as possible. We did not use the testing data for validation in training as that could influence our models to fit to the testing data which would make it not be independent upon evaluation.

```
# Load model, training and testing datasets
# Datasets for summary stats dataset were already normalized using min-max so it doesn't need to be done again
training_file = "output/summary_stats_training_normalized"
testing_file = "output/summary_stats_testing_normalized"
model_file = "trained_model_summary_stats.keras"
columns = ["label", "numPackets", "total_data_sent", "stdev_arrival_times", "avg_inter_arrival_time", "median_arrival_time"]
data_training = pd.read_csv(training_file, sep=" ", header=None, names=columns)
data_testing = pd.read_csv(testing_file, sep=" ", header=None, names=columns)

# Separate features and labels for training and testing
values_training = data_training.iloc[:, 1:].values
labels_training = data_training.iloc[:, 0].values

values_testing = data_testing.iloc[:, 1:].values
labels_testing = data_testing.iloc[:, 0].values

# print(pd.Series(labels_train).value_counts())

# Split training dataset in 3:1 ratio to training and validation
# validation serves as an independent dataset during training
values_training, values_validation, labels_training, labels_validation = train_test_split(
    values_training, labels_training,
    test_size=0.25,
    random_state=42,
    stratify=labels_training
)

# Encode labels as integers
label_encoder = LabelEncoder()
labels_training = label_encoder.fit_transform(labels_training)
labels_validation = label_encoder.transform(labels_validation)
labels_testing = label_encoder.transform(labels_testing)
labels_training = to_categorical(labels_training)
labels_validation = to_categorical(labels_validation)
labels_testing = to_categorical(labels_testing)

# Convert to np array
values_training = np.array(values_training)
labels_training = np.array(labels_training)
values_validation = np.array(values_validation)
labels_validation = np.array(labels_validation)
values_testing = np.array(values_testing)
labels_testing = np.array(labels_testing)
```

- Above is an example of how the data was loaded, parsed, encoded, and split. We followed an identical or very similar process for each different model.

g. A total of 150 epochs were run on each model to avoid learning discrepancies and keep a baseline to make the differences in model performance limited to just the differences in training data as much as possible. Here is a brief demo of the training in action: https://drive.google.com/file/d/1faZx03WbhTbJo7vOnqc8ySoKKXEgwFi6/view?usp=sharing

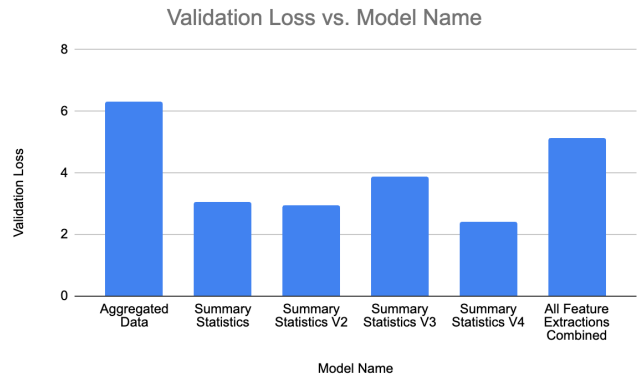
h. After each model was created and trained, we added it to our eval_acc_comparison.py file to compare their performances side-by-side.

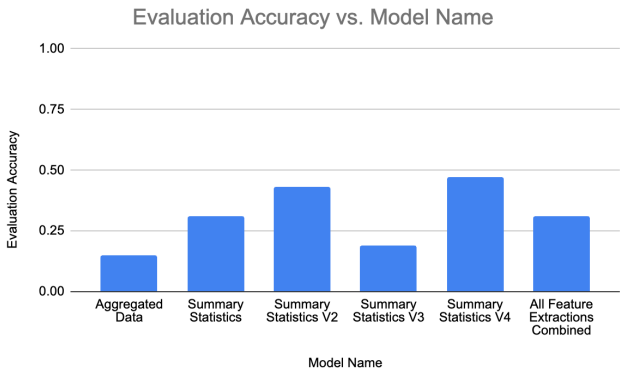
i. In addition to models based on the 3 extracted feature datasets, we created 2 extra models using combined features from multiple extractions as well as 1 model with a reduced number of features from one of the extractions. This serves to give us a better understanding of which particular features were most effective and which did not work as well for our use case.

Results:

The figure below can be replicated in your IDE's terminal by running the src/eval_acc_comparison.py file. Note that validation accuracy shows how accurate the model was on our testing data, and a higher validation loss means more prone to overfitting.

```
Each model is a keras sequential model which was ran with 150 epochs each

Aggregated Data Extraction Model Results:
(Features are data sent per 0.1 second interval within the first 3 seconds)
Validation Loss: 6.3188
Validation Accuracy: 0.1500

Summary Stats Extraction Model Results:
(Features are numPackets, total_data_sent, stdev_arrival_times, avg_inter_arrival_time, and median_arrival_time)
Validation Loss: 3.0521
Validation Accuracy: 0.3100

Summary Stats V2 Extraction Model Results:
(Features are numPackets, total_data_sent, TLS_handshake_size, num_large_packets (> 1000 bytes), and num_small_packets (< 100 bytes))
Validation Loss: 2.9432
Validation Accuracy: 0.4300

Summary Stats V3 Extraction Model Results:
(Features are just numPackets and total_data_sent)
Validation Loss: 3.8757
Validation Accuracy: 0.1900

Summary Stats V4 Extraction Model Results:
(Features are numPackets, total_data_sent, stdev_arrival_times, avg_inter_arrival_time, median_arrival_time, TLS_handshake_size, num_large_
packets (> 1000 bytes), and num_small_packets (< 100 bytes))
Validation Loss: 2.4026
Validation Accuracy: 0.4700

All Feature Extractions Combined Model Results:
Features are numPackets,total_data_sent,TLS_handshake_size,num_large_packets,num_small_packets,numPackets,total_data_sent,stdev_arrival_tim
es,avg_inter_arrival_time,median_arrival_time, data sent per 0.1 second interval within the first 3 seconds
Validation Loss: 5.1236
Validation Accuracy: 0.3100
```





The first model's (aggregated data extraction) accuracy was the lowest of the group, in addition to having the highest validation loss. This suggests that this feature extraction yielded poor generalization capabilities and was not particularly effective at distinguishing different websites using their traffic data.

The first summary statistics model was more than twice as accurate as the aggregated data model with half the validation loss. This model and the summary statistics version 2 model were trained mostly on different extracted features, yet they both proved to be effective.

Since both the summary statistics version 1 and 2 models used num_packets and total_data_sent in their training data, we trained a separate model, summary statistics version 3, on these features alone. Noticeably, this achieved lower accuracy than both the summary statistics version 1 and 2 models, suggesting that while num_packets and total_data_sent were effective features, the additional features used in versions 1 and 2 were also beneficial to the model's accuracy.

Based on this observation, we trained another model, summary statistics version 4, to use all the features used in the summary stats version 1 and 2 models. Based on the results shown above, this proved to be our most accurate model with the lowest validation loss (meaning overfitting is less of an issue and the model is better at generalizations). From this, we can conclude that this combined set of features worked best in conjunction with each other and provided the model with the most useful context of any features we tested.

Our final model was a combined set of all the features from each packet extraction we ran on the packet captures. This model showed a significant decrease in performance in both validation loss and evaluation accuracy when compared to our summary statistics version 4 model. This showcases that additional features are not always beneficial to model accuracy, likely introducing unnecessary noise into our training data. We believe that merging the summary statistics with the data-per-interval features introduced too many data points which likely made it so the model was unable to assign weights to each feature as accurately.

Based on our cumulative set of results, we concluded that performing website fingerprinting attacks are very practical and effective for identifying user website traffic. Even with our limited experience and resources throughout this project, we were able to achieve 47% accuracy with 100 different website classes (1% baseline accuracy). Based on this result we can confidently assert that professional teams with more depth of knowledge, time, and computational resources would likely be able to achieve even better results and be able to identify websites visited by different users through their traffic with a high degree of confidence.

Future Work:

- **Improving Data Collection Process**
    - We experienced many uneven packet capture lengths produced by our Selenium/PyShark script. Optimizing this process could create more useful data for the rest of our process
    - The large number of classes (100) we included meant that even with 500 packet captures, we were only able to get 5 data points per class. Increasing this, while very computationally expensive, would likely boost our results significantly
- **Optimization of Feature Extraction:**
    - Fine-tuning the feature extraction process could lead to a more meaningful representation of packet capture data. For instance, incorporating temporal trends or deep statistical measures might highlight patterns not currently utilized.
    - Include more protocol-specific attributes such as HTTP request sizes
    - Capture additional features related to packet timing, such as variance and skewness of inter-arrival times
- **Data Augmentation and Increased Collection:**
    - Expanding the dataset to include traffic from more diverse websites and under different browsing conditions could significantly improve the model's generalization capabilities. For example, collecting data from websites across varied geographic locations, using different network setups, or under various user behaviors.
    - We used min/max scaling to normalize all of our data before feeding it into our machine learning models. Using other techniques like logarithmic scaling or z-score normalization could also produce better results.
- **Experimenting with Additional Models:**
    - While the Keras sequential feedforward neural network performed adequately, other machine learning models could be explored
    - A different model architecture as well as including more/less or larger/smaller layers could also improve our accuracy. Also, manipulation of other parameters like the activation functions, loss functions, optimizers, learning rate, and more could yield better results.
- **Incorporating Dynamic Analysis:**
    - Introduce mechanisms to analyze traffic in real-time to simulate live network conditions and adapt model predictions based on dynamic feature inputs.

Sources:

[1] KimiNewt, “PyShark - Python packet parser using wireshark’s tshark,” PyShark, https://kiminewt.github.io/pyshark/ (accessed Dec. 12, 2024).

[2] jaibalaji, IslamTaha, and Louis Maddox, “How to set up a Selenium Python environment for Firefox?,” Stack Overflow, https://stackoverflow.com/questions/42204897/how-to-set-up-a-selenium-python-environment-for-firefox (accessed Dec. 1, 2024).

[3] J. Brownlee, “Multi-Class Classification Tutorial with the Keras Deep Learning Library,” MachineLearningMastery, https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/ (accessed Dec. 8, 2024).

[4] GeeksforGeeks Staff, “ReLU Activation Function in Deep Learning,” GeeksforGeeks, https://www.geeksforgeeks.org/relu-activation-function-in-deep-learning/ (accessed Dec. 11, 2024).

[5] GeeksforGeeks Staff, “Softmax Activation Function in Neural Networks,” GeeksforGeeks, https://www.geeksforgeeks.org/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/ (accessed Dec. 9, 2024).

[6] A. S. Gillis, “Data Splitting,” TechTarget, https://www.techtarget.com/searchenterpriseai/definition/data-splitting# (accessed Dec. 10, 2024).

Python dependencies:

- PyShark
- Selenium
- NumPy
- Pandas
- Scikit-learn
- Tensorflow
- Tqdm