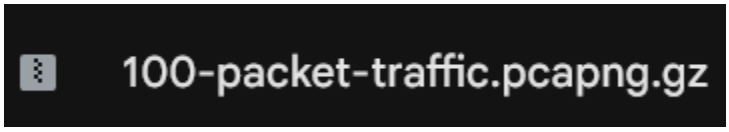
Using pcapML and nprint for OS detection

For this term's project I chose to try and reproduce the best-known results of Machine Learning in Computer Networking, specifically in detecting operating systems. There were two main tools to help complete this project, pcapML and nPrint .



We were given data on many different traffic traces, already encoded by pcapML. What this allowed was for metadata to be stored alongside the packets, which then created the modified .pcapng file. In this specific scenario, the information stored allowed for proper labeling, dictating the sample_id and the operating system.

To create the labels(assuming pcapMLis installed of course), all one has to do is run

./pcapml -M (dataset encoded with pcapML) -O (output directory)

This command splits the large pcapML dataset into many individual PCAP files, each placed in the output directory, with filenames or a metadata csv reflecting their associated labels. However, although this is nice information, it still isn't quite what we are looking for.

nPrint is a feature extraction tool that transforms each raw PCAPinto a fixed-size, numerical representation suitable for machine learning. It hashes packet header into a vector, allowing for packet traces at different lengths to run and train off the same data. Thus, in order to turn all these packet traces into usable data, we have to use nPrint to convert.

Something that can also be done at the same time is filtering out. There are some constraints with the headers we are allowed to use. In this situation, these features trivialize

finding the OS, and thus would embody the spirit of the project to leave them out.

```
nprint -P "$f" \
       -4 -t -R -F 0 \
       -x "src_ip|dst_ip|tcp_src_port|tcp_dst_port|tcp_seq|tcp_ack" \
       -W "$OUTPUT_DIR/${fname}.nprint" \
       -V
```

Finally we get to the actual development of the model. Each nPrint file is already standardized to a fixed size, meaning every trace produces a matrix with the same number of rows and columns. This makes the dataset uniform and comparable across samples. However, the Autogluon model which is highlighted in the original experiment takes single dimensional vectors, thus requiring a flattening.

```
# Keep only numeric columns and flatten
flat = nprint_data.select_dtypes(include=[np.number]).values.flatten()
X.append(flat)
```

With the traces being converted into vectors, the next challenge was the high dimensionality of the feature space. Even with the standardized nPrints, each vector can contain tens of thousands of numeric features(in this case over 90k), making direct training computationally expensive and potentially prone to overfitting. To address this, dimensionality reduction using Truncated SVD was applied.

```
svd = TruncatedSVD(n_components=2000, random_state=42)
X_reduced = svd.fit_transform(X_features_filled)
```

This crunched the feature space while retaining the important patterns, reducing the effect from noise.

The final step was Autogluon, which is an automated machine-learning framework, specifically made for tabular data. This is why I had to convert from nPrints into those flat Dataframes, otherwise it wouldn't be compatible with a multitude of tests that this framework runs. An important feature of Autogluon are the varied presets that can reduce training time, along with a changeable training time limit. These are so that you can optimize between training time and balanced accuracy, meaning that a normal computer can actually run these tests without dying(had my computer run a test for multiple hours before giving up)

```
presets='good_quality',
# increase training time if you want higher accuracy
# unfortunately my computer is garbage so this took me really long
time_limit=600
```

Although overall I got a pretty terrible accuracy score, considering the tradeoffs of having your computer crunch for hours, I'd say I'm happy with this result.

```
TabularPredictor saved. To load, use: predictor = TabularPredictor.load("/home/doawgh/nprint/AutogluonM
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
{'balanced_accuracy': np.float64(0.43589743589743607), 'accuracy': 0.45, 'mcc': 0.402978130659247}
```