Reproducing the nPrint OS Detection Task Benchmark Score

Contributions

First Name	Last Name	cnet ID	Project Role
Daniel	Fields	dfields	Sole Participant

Motivation

The task of identifying the operating system of the source of a packet stream is highly relevant to computer security. Large industries that receive vast amounts of traffic might need to analyze the operating system type of incoming communications which they suspect as being malicious, in order to best plan a response and ensure the safety of their data. At the same time, groups of hackers may employ OS detection during network scans in order to determine potential targets for exploits they are planning to carry out. For these reasons, it is important to understand the current capabilities of machine learning models in detecting operating system type, and the nPrint OS detection benchmark, along with my current work, both further this goal.

Project description

In this project, I attempt to re-produce the reported leaderboard score for the nPrint OS detection task using a variety of classification models. The OS detection task consists of predicting the operating system labels of samples, each containing 100 packets from a source IP address, from the CICIDS 2017 dataset. The goal of the task is to optimize the balanced accuracy metric, and the current leaderboard score is 77.1. The researchers who obtained the current leaderboard score used the nPrintML library to complete the task, which involves converting the data to nPrint representations before using AutoML to automatically select the best models and hyperparameters. Rather than use AutoML, I aim to identify which standard classification models are most effective in this task, and use the results to better understand the structure of the dataset and which features of packet traces are most useful in predicting the source operating system type. I am also interested in understanding whether or not linear classifiers can be useful in this task, or if nonlinear decision models are necessary to achieve optimal results.

Data Processing

I followed the procedures outlined in the nPrint OS detection task website (https://nprint.github.io/benchmarks/os_detection/nprint_os_detection.html) and the paper “New Directions in Automated Traffic Analysis” (https://arxiv.org/pdf/2008.02695) to inform my data processing steps. The .pcapng file provided by the benchmark website already formats the dataset into samples of 100 packets per source IP address, and I started by downloading this .pcapng file and using the following command from the pcapML library to parse its contents.

Shell

```
  pcapml M 100-packet-traffic-pcapng -0 parsed_pcapng_data/
```

The .pcapng file includes metadata labels for each of its samples which contain a sampleID and operating system labels, and this pcapML command, when run in the Terminal, creates a new output directory containing an individual pcap file for each sample along with a single metadata CSV file. Next, I used the following shell script to convert each pcap file in the parsed_pcapng_data output directory into an nPrint representation.

Shell

```
mkdir -p nprint_samples
for f in parsed_pcapng_data/*-pcap; do
     filename=$(basename "$f" -pcap)
     nprint -4 -t --pcap_file "$f" --write_file
"nprint_samples/${filename}-npt"
done
```

This script stores the nPrint representations of each sample in the nprint_samples directory, and I uploaded both the parsed_pcapng_data and nprint_samples directories to my project Github. My notebook begins by further processing the contents of these directories in order to obtain a data matrix suitable for machine learning models. Specifically, the nPrintML source code used by the researchers creates a single vector for each sample of nPrint data by flattening the nPrint data matrix of 100 samples into one vector. To abide by these principles, I used the following code to iterate through each sample and flatten its contents, while associating it with its corresponding operating system type label and sampleID field.

Python

```
for path in nprint_files:
    base = os.path.basename(path)
    pcap_name = base[:-4] + ".pcap"
    label = pcap_to_os.get(pcap_name)
    include_columns = [index_col] + allowed_feature_columns
    nprint_df = pd.read_csv(path, usecols=include_columns)
    nprint_arr =
nprint_df[allowed_feature_columns].to_numpy(dtype=np.int16,
copy=False)
    flattened = nprint_arr.ravel()
    if flattened.shape[0] > max_length:
        max_length = flattened.shape[0]
    sample_id = pcap_name[:-5]
    data_rows.append((sample_id, label, flattened))
```

The key line in this code uses the .ravel() function to create a vector containing the nPrint representations of all 100 packets in the sample concatenated together. The max_length variable stores the maximum length of a flattened vector, in order to ensure that the overall data matrix can accommodate the longest vector (empty values for shorter vectors will be padded with -1 values). Moreover, this code also removes the disallowed features specified in the benchmark website, those which would allow the model to simply memorize which examples corresponded to which operating system. Those labels were IPv4 Source IP, IPv4 Destination IP, TCP Source Port, TCP Destination Port, TCP SEQ and TCP ACK Numbers, and I used the following code to specify these in my program.

Python

```
disallowed_features = ["ipv4_src", "ipv4_dst", "tcp_sport",
"tcp_dport", "tcp_seq", "tcp_ack", "tcp_sprt", "tcp_dprt",
"udp_sport", "udp_dport", "udp_sprt", "udp_dprt"]
def check_disallowed_feature(column_name):
    name = column_name.lower()
    return any(feature in name for feature in
disallowed_features)
```

I then created two numpy matrices X and y, which contain the data for each sample along with the corresponding operating system type label. Inspecting the shape of the outputs, I noted that the shape of X was 12439 by 79900, and the shape of y was 12439 by 1, with 13 unique labels, matching the descriptions of the data on the benchmark website. In my following classification code, I enumerated the operating system labels from 0 to 12 rather than using their string names.

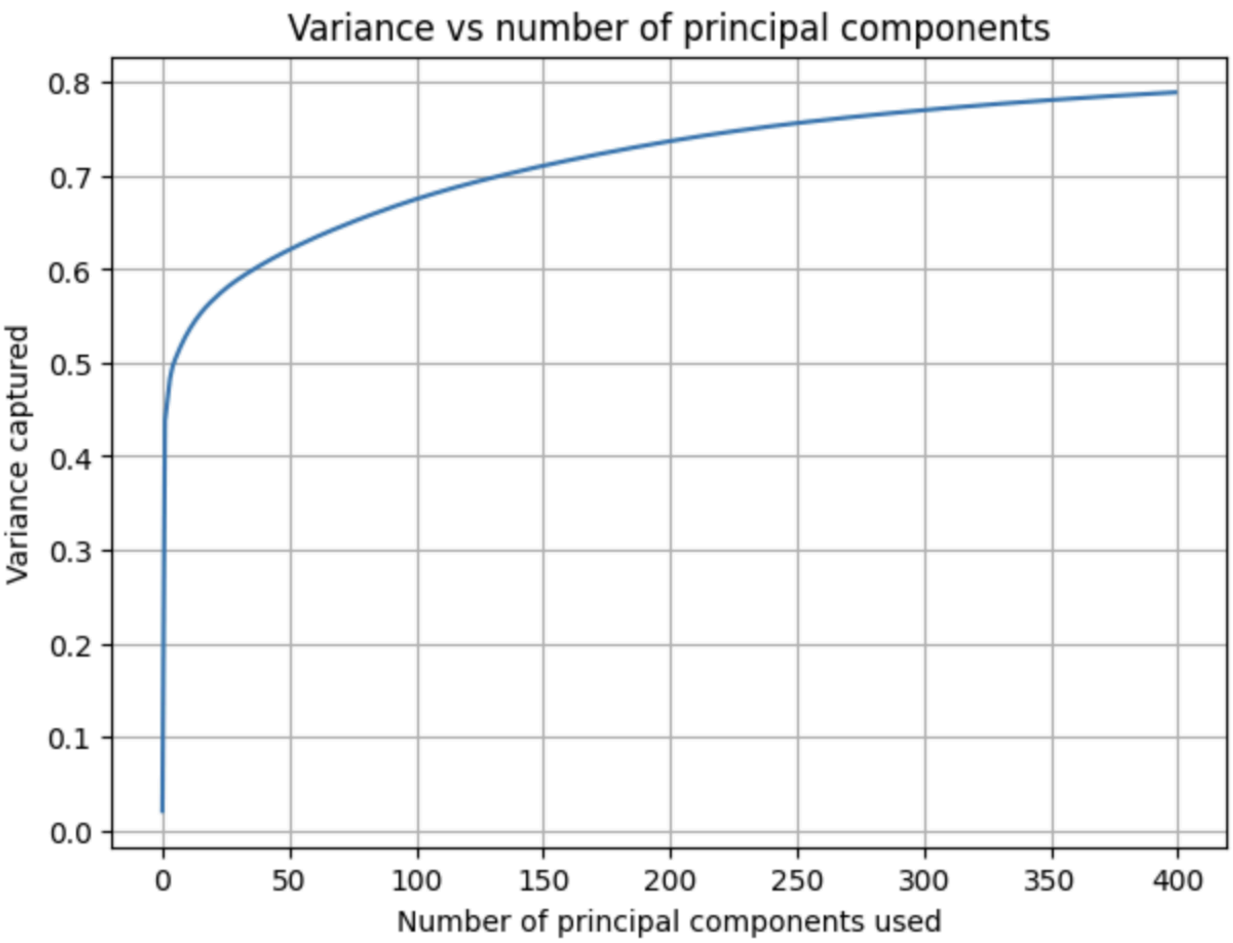
Data Analysis

My next step was to use Principal Components Analysis to investigate the number of principal directions needed to explain the variance in the data. The following code block shows my use of SciKitLearn's TruncatedSVD method to project the data onto its principal components.

Python

```
truncated_svd =
TruncatedSVD(n_components=num_principal_components,
random_state=0)
X_train_reduced = truncated_svd.fit_transform(X_train)
X_test_reduced = truncated_svd.transform(X_test)
variance_cdf = np.cumsum(truncated_svd.explained_variance_ratio_)
```

With this code, I obtained the following graph demonstrating the number of principal components needed to explain the variance in the data. I observed that there are a small number of principal components which explain around 50% of the variance, but in order to fully represent the relationships between examples, a larger number of principal directions are required. For these reasons, I decided not to use dimensionality reduction in my data processing steps, so that I could preserve the full range of information contained in the dataset.



Training Classification Models and Observing their Results

My next step was to train standard classification models on the dataset. Specifically, I aimed to analyze the results of the following classifiers: Perceptron, Logistic Regression, Support Vector Machine, RandomForest. First, I used a 75%/25% train-test-split, following the procedures of the research paper.

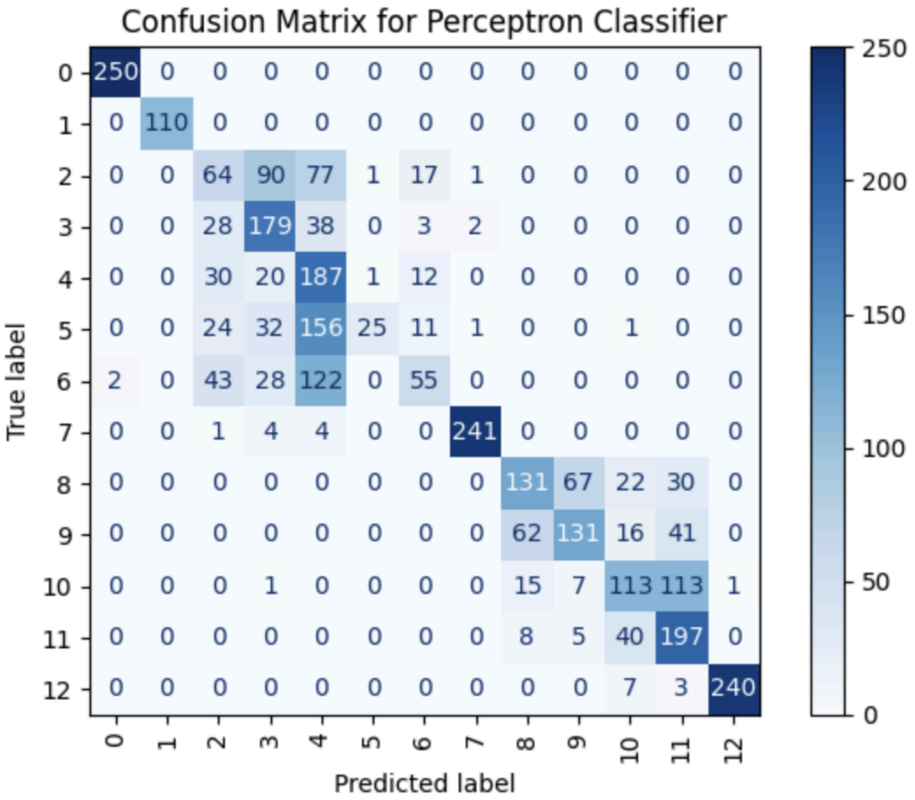
```Python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0, stratify=y)
```

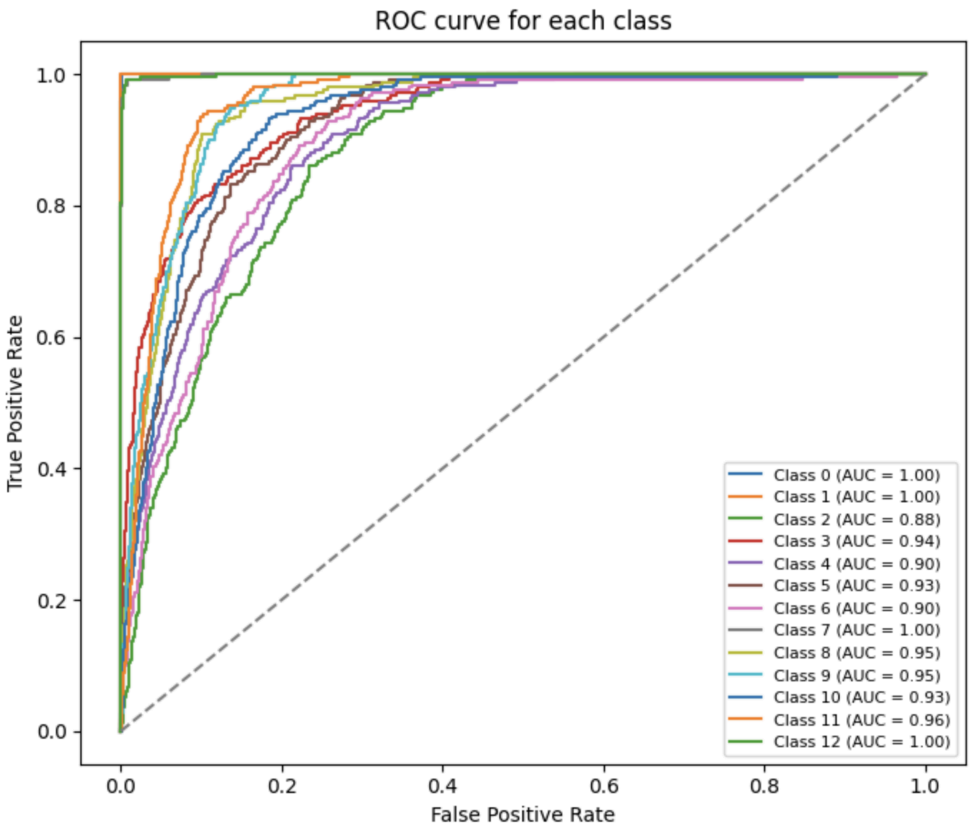
Next, I trained a Perceptron classifier on the data using the SGDClassifier class from SciKitLearn, which allows for the creation of different types of linear classifiers through specifying the appropriate loss function. Each model is learned through stochastic gradient descent on the training data, and I used 100 iterations for each of my classifiers. In this case, I made sure to use the “perceptron” loss function.

Python

```
perceptron_model = SGDClassifier(loss="perceptron", max_iter=100,
tol=1e-3, n_jobs=-1, random_state=0)
perceptron_model.fit(X_train, y_train_encoded)
y_predicted = perceptron_model.predict(X_test)
```

This classifier resulted in a balanced accuracy score of 0.6348, and I used the following confusion matrix and multi-class ROC curve plots to visualize its results.





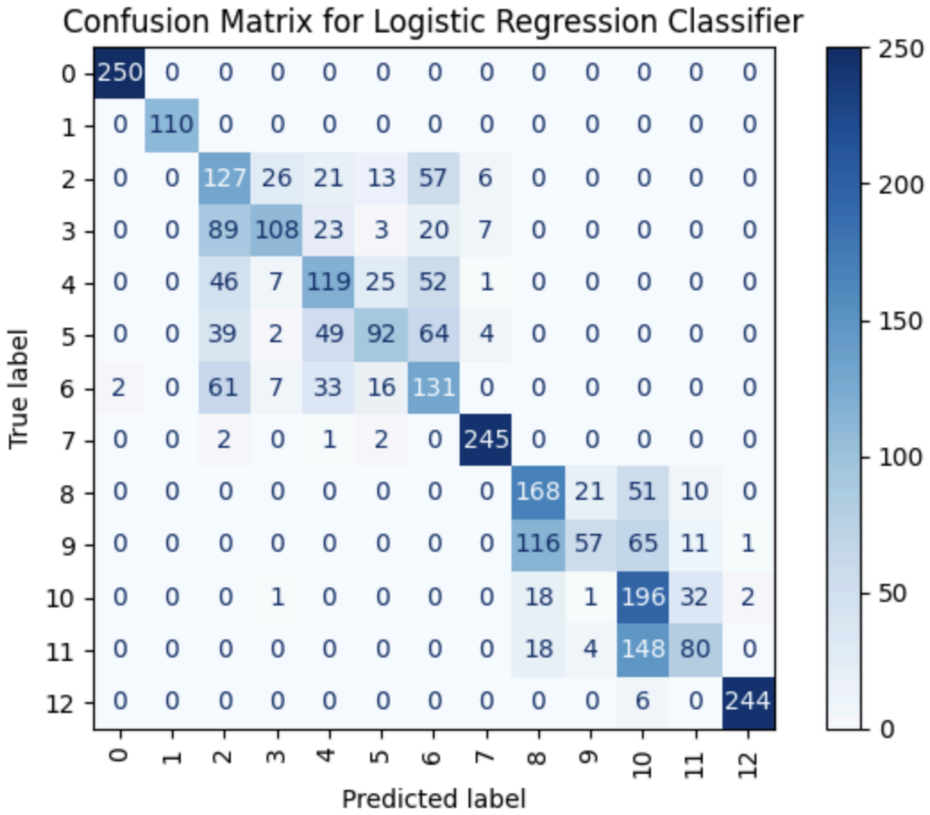
The results showed that the linear Perceptron model was able to predict the operating type of the data points with reasonable accuracy, suggesting that the data does contain features which correspond linearly with the operating system type.

Next, I trained a Logistic Regression classifier on the data by passing the logistic loss parameter into the SGDClassifier class, which results in a logistic regression model, using 100 iterations for the stochastic gradient descent training process.

Python

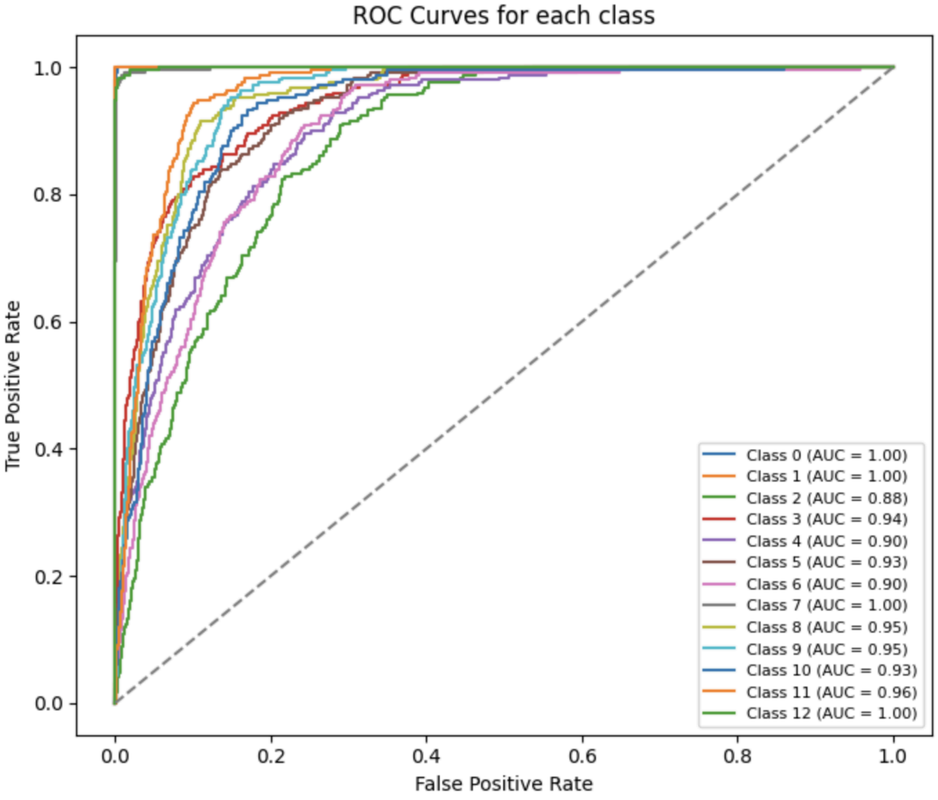
```python
logistic_regression = SGDClassifier(loss="log_loss", max_iter=100,
tol=1e-3, n_jobs=-1, random_state=0)
logistic_regression.fit(X_train, y_train_encoded)
y_predicted = logistic_regression.predict(X_test)
```

This classifier resulted in a balanced accuracy score of 0.6360, and I used the following confusion matrix and multi-class ROC curve plots to visualize its results.



Confusion Matrix for Logistic Regression Classifier

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10	11	12
0	250	0	0	0	0	0	0	0	0	0	0	0	0
1	0	110	0	0	0	0	0	0	0	0	0	0	0
2	0	0	127	26	21	13	57	6	0	0	0	0	0
3	0	0	89	108	23	3	20	7	0	0	0	0	0
4	0	0	46	7	119	25	52	1	0	0	0	0	0
5	0	0	39	2	49	92	64	4	0	0	0	0	0
6	2	0	61	7	33	16	131	0	0	0	0	0	0
7	0	0	2	0	1	2	0	245	0	0	0	0	0
8	0	0	0	0	0	0	0	0	168	21	51	10	0
9	0	0	0	0	0	0	0	0	116	57	65	11	1
10	0	0	0	1	0	0	0	0	18	1	196	32	2
11	0	0	0	0	0	0	0	0	18	4	148	80	0
12	0	0	0	0	0	0	0	0	0	0	6	0	244



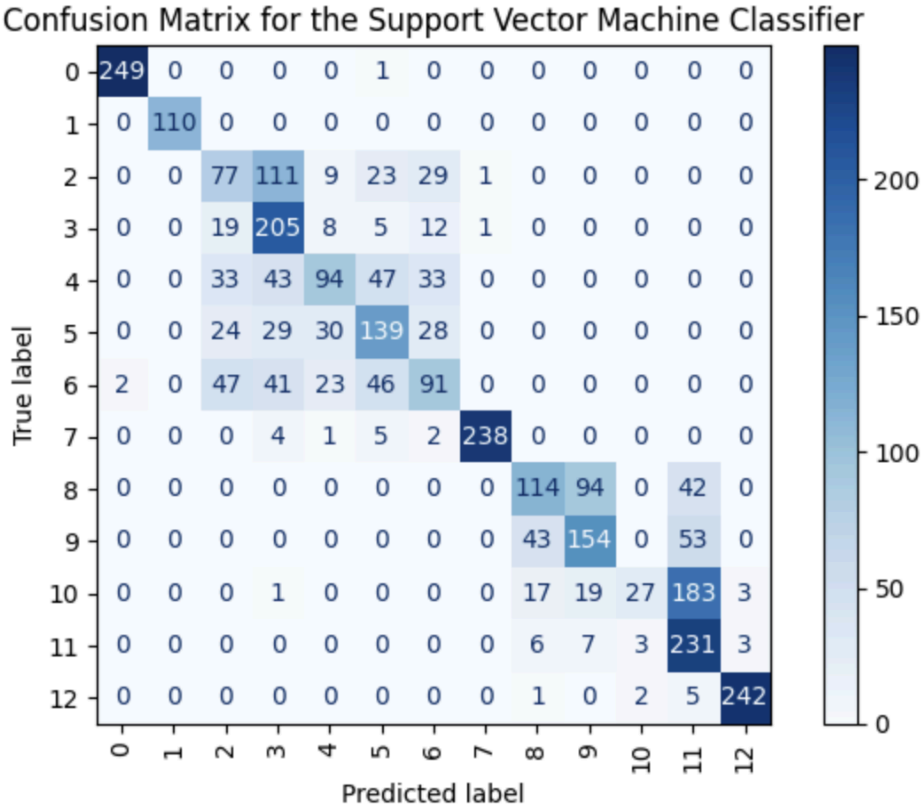
ROC Curves for each class

The results showed that the logistic regression model also performed well in the classification task, and achieved slightly better accuracy than the linear perceptron model, further verifying the linear relationships between certain features and the operating system labels.

Next, I trained a Support Vector Machine classifier on the data by passing the hinge loss parameter into the SGDClassifier class, which results in a linear model which attempts to maximize the margin of error in its predictions, using 100 iterations for the stochastic gradient descent training process.

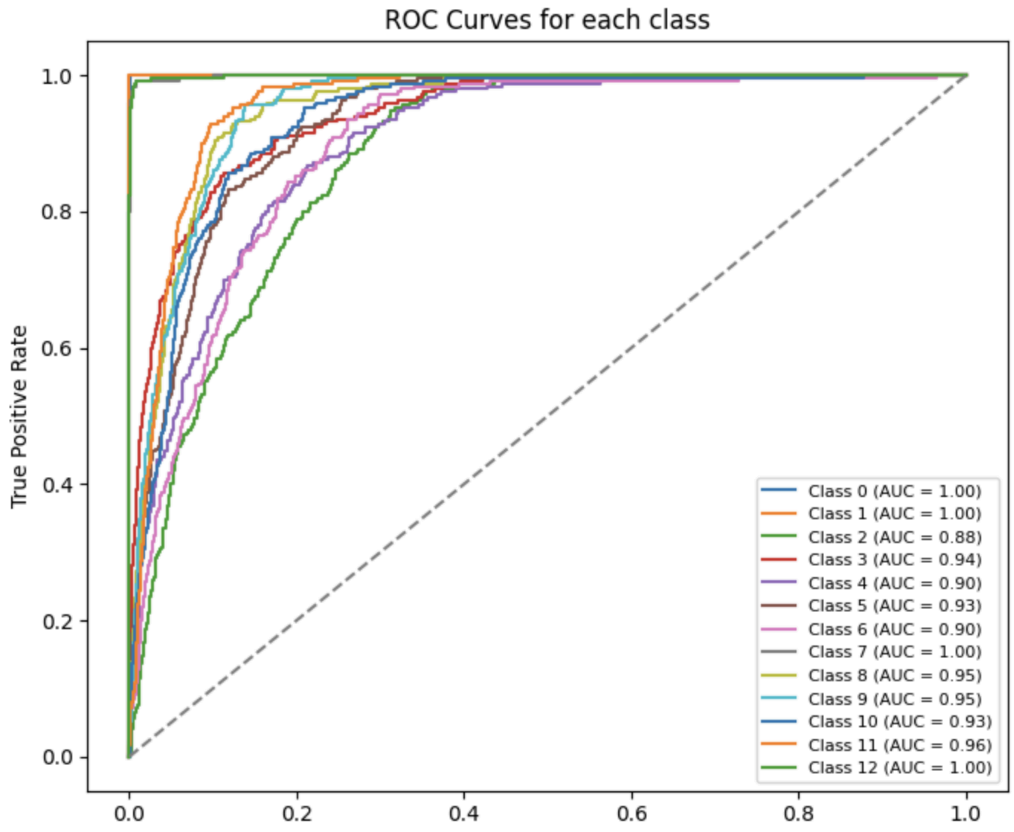
```python
support_vector_machine = SGDClassifier(loss="hinge", max_iter=100,
tol=1e-3, n_jobs=-1, random_state=0)
support_vector_machine.fit(X_train, y_train_encoded)
y_predicted = support_vector_machine.predict(X_test)
```

This classifier resulted in a balanced accuracy score of 0.6495, and I used the following confusion matrix and multi-class ROC curve plots to visualize its results.



Confusion Matrix for the Support Vector Machine Classifier

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10	11	12
0	249	0	0	0	0	1	0	0	0	0	0	0	0
1	0	110	0	0	0	0	0	0	0	0	0	0	0
2	0	0	77	111	9	23	29	1	0	0	0	0	0
3	0	0	19	205	8	5	12	1	0	0	0	0	0
4	0	0	33	43	94	47	33	0	0	0	0	0	0
5	0	0	24	29	30	139	28	0	0	0	0	0	0
6	2	0	47	41	23	46	91	0	0	0	0	0	0
7	0	0	0	4	1	5	2	238	0	0	0	0	0
8	0	0	0	0	0	0	0	0	114	94	0	42	0
9	0	0	0	0	0	0	0	0	43	154	0	53	0
10	0	0	0	1	0	0	0	0	17	19	27	183	3
11	0	0	0	0	0	0	0	0	6	7	3	231	3
12	0	0	0	0	0	0	0	0	1	0	2	5	242

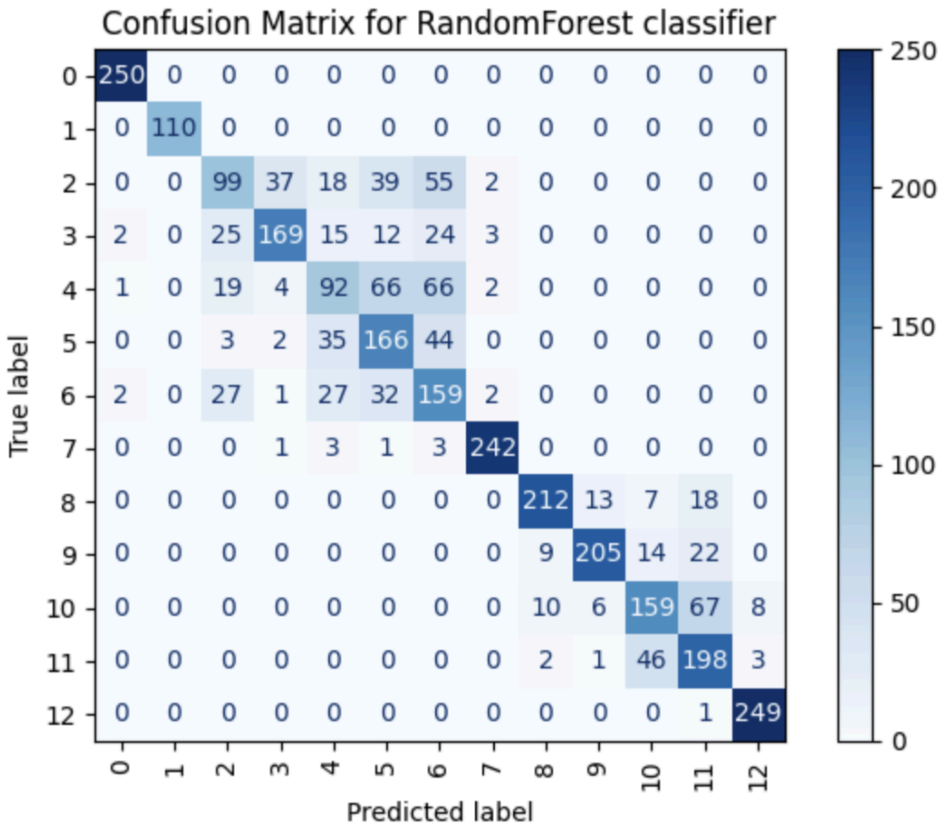


The results showed that the support vector machine also performs well in the classification task, and does better than the perceptron and logistic regression models, which makes sense given that the purpose of the support vector machine is to maximize the margin of error in its classifications, which should help in identifying the optimal boundaries between classes.

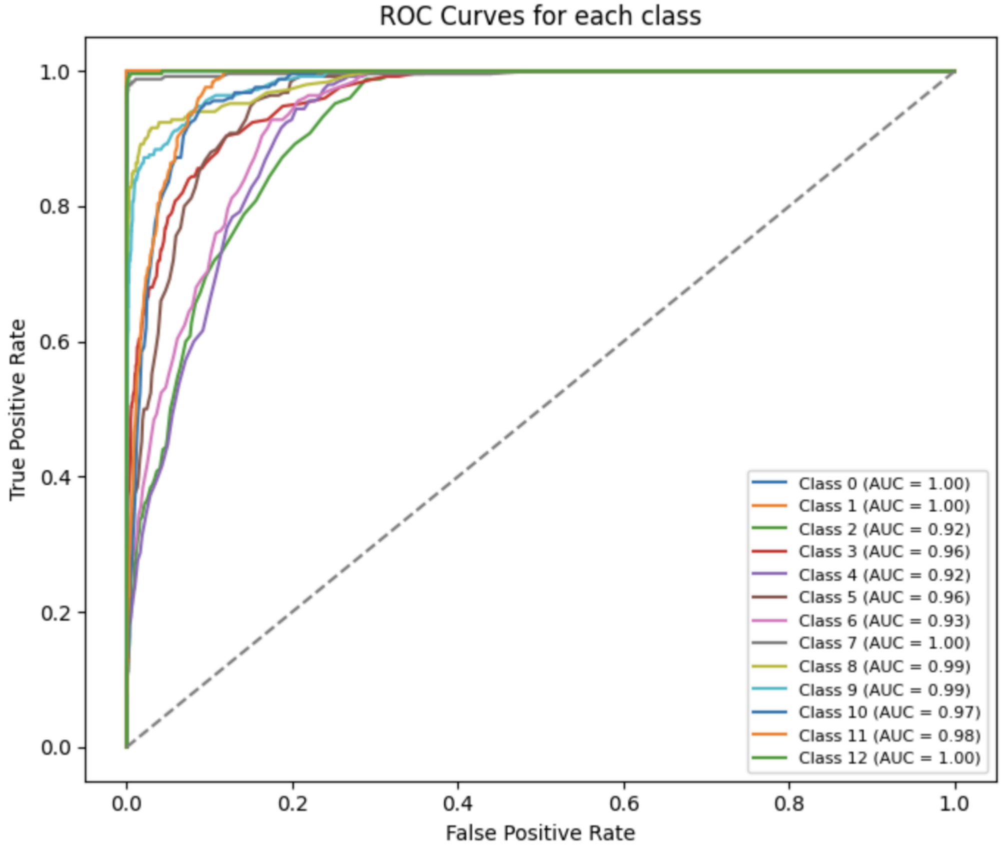
Next, I trained a RandomForest classifier on the data in order to verify whether or not using nonlinear decision models would result in a noticeable improvement from standard linear classifiers.

```python
random_forest_model = RandomForestClassifier(n_estimators=200,
max_depth=None, n_jobs=-1, random_state=0)
random_forest_model.fit(X_train, y_train_encoded)
y_predicted = random_forest_model.predict(X_test)
```

This classifier resulted in a balanced accuracy score of 0.7538, and I used the following confusion matrix and multi-class ROC curve plots to visualize its results.


Confusion Matrix for RandomForest classifier

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10	11	12
0	250	0	0	0	0	0	0	0	0	0	0	0	0
1	0	110	0	0	0	0	0	0	0	0	0	0	0
2	0	0	99	37	18	39	55	2	0	0	0	0	0
3	2	0	25	169	15	12	24	3	0	0	0	0	0
4	1	0	19	4	92	66	66	2	0	0	0	0	0
5	0	0	3	2	35	166	44	0	0	0	0	0	0
6	2	0	27	1	27	32	159	2	0	0	0	0	0
7	0	0	0	1	3	1	3	242	0	0	0	0	0
8	0	0	0	0	0	0	0	0	212	13	7	18	0
9	0	0	0	0	0	0	0	0	9	205	14	22	0
10	0	0	0	0	0	0	0	0	10	6	159	67	8
11	0	0	0	0	0	0	0	0	2	1	46	198	3
12	0	0	0	0	0	0	0	0	0	0	0	1	249


ROC Curves for each class

The results showed that the RandomForest model was able to outperform all three previous linear classifiers in terms of balanced accuracy, which reveals that even though some of the features may lend themselves well to linear classification, nonlinear classifiers are powerful in the sense that they can learn more complex relationships that relate to the true operating system labels.

Identifying the Importance of Individual Features

I then aimed to answer the question as to which features of the data are most important in predicting operating system type. I used the following code to analyze which features resulted in the greatest reduction in impurity in the RandomForest model, outputting a list of the 25 most important features.

Python

```
feature_count = len(allowed_feature_columns)
random_forest_importance = random_forest_model.feature_importances_
feature_importances = np.zeros(feature_count)
for i, importance_score in enumerate(random_forest_importance):
    feature_importances[i % feature_count] += importance_score

idx_importance = list(enumerate(feature_importances))
descending_importance = sorted(idx_importance, key=lambda x: x[1],
reverse=True)
important_features = descending_importance[:25]

print(f"The 25 most important data features in relation to operating
system label:")
for i, importance_score in important_features:
    print(f"{allowed_feature_columns[i]}: {importance_score}")
```

I obtained the following output, which shows that the most important features relate to packets’ time-to-live attribute, TCP window sizes, TCP options, and IP address checksums. This suggests that these features are especially relevant in identifying operating system type, which makes sense, because each operating system has different protocols with regards to the parameters used when sending packets. The most important attribute, time-to-live, represents the “number of hops” a packet can take from its source to its destination, and this value is predetermined by the source operating system.

None

```
The 25 most important data features in relation to operating
system label:
ipv4_ttl_0: 0.04362525575543767
tcp_wsize_1: 0.040161115194404
tcp_wsize_5: 0.03722557930422077
ipv4_ttl_1: 0.033057674362625 7
tcp_wsize_0: 0.025806975574175946
tcp_opt_42: 0.018350167342298135
tcp_opt_40: 0.018242783458630757
tcp_opt_41: 0.017995247931869324
tcp_opt_36: 0.01562198043241877
tcp_opt_34: 0.015242432564923353
ipv4_id_15: 0.013722452929194127
tcp_opt_67: 0.012071715837647124
ipv4_cksum_15: 0.012018217185875O5
ipv4_tos_4: 0.011862407308706064
ipv4_cksum_0: 0.011428291470529602
tcp_wsize_3: 0.011136352971437896
tcp_wsize_6: 0.010846328239851703
tcp_wsize_2: 0.010749857378714936
tcp_wsize_15: 0.00997642892264809
tcp_opt_20: 0.009616115766152386
tcp_opt_68: 0.00943904259724781
ipv4_id_14: 0.009421783457776181
tcp_opt_39: 0.008934015465546079
tcp_opt_44: 0.008825548939955189
ipv4_ttl_5: 0.008462549154142502
```

Analysis of Results and Comparison to Benchmark Performance

Overall, the results of my experiments helped to answer some of the motivating questions I had for this project. Specifically, I observed that the balanced accuracy scores of the perceptron, logistic regression model, support vector machine, and RandomForest were 0.6348, 0.6360, 0.6495, and 0.7538 respectively. The relatively good accuracy of the linear classifiers demonstrates that the data contains features which relate linearly to the operating system labels, but the RandomForest model's ability to learn complex, nonlinear relationships still offers a notable increase in performance. Moreover, the time it took to train the RandomForest model (~10 seconds), was much faster than that of the linear classifiers trained using stochastic gradient descent (~1 minute on average), demonstrating its efficiency. The benchmark score of 0.771 is slightly higher than that of the RandomForest model I tested, and this makes sense due to the fact that the researchers who produced that score made use of the AutoML library, which runs as part of nPrintML, to automatically determine the optimal model and hyperparameters for a given task. Given the results I have observed, I suspect that the AutoML library selects a nonlinear classifier for this task such as RandomForest or the MLP, which is likely necessary to achieve the best performance on the operating system prediction task. In general, these experiments also taught me that packet data contain unique qualities that are useful in identifying attributes of their sender, since I learned that the time-to-live attribute is useful in revealing the operating system of the packet’s source machine.