Connor McGraw, Hewitt Watkins, Faiz Hilaly
Machine Learning for Computer Systems
12 December 2025

Recreating "*NetML: A Challenge for Network Traffic Analytics*"

Introduction

This project aims to reproduce and extend the baseline classification results from NetML: A Challenge for Network Traffic Analytics (Barut et al., 2020) for the application identification task. The original paper introduced three network traffic datasets and established baseline performance using Random Forest, SVM, and Multilayer Perceptron models. As in many machine learning papers, however, the reproducibility of these results depends on implementation details that are not fully specified, such as feature choices, data splits, and training procedures.

Reproducibility is a cornerstone of scientific research, yet machine learning research often faces reproducibility challenges because results can hinge on specific tools, hyperparameters, random seeds, and preprocessing steps. By attempting to reproduce the NetML style baseline results on the corresponding pcapML Application Identification benchmark, we aim to (1) validate the published findings at least qualitatively, (2) identify what information is sufficient or missing for faithful reproduction, (3) surface practical challenges in network traffic analysis experiments, and (4) provide a clearer baseline that can be reused or extended in future work.

The underlying problem is application identification in network traffic flows, a critical task for network management, security monitoring, and quality of service. Traditional port based methods perform poorly for modern applications that use dynamic ports and encryption, so operators increasingly rely on machine learning models that infer application classes from flow level features. In this project we work with the pcapML Application Identification dataset that is derived from the NetML VPN nonVPN traffic captures and focuses on the non vpn 2016 subset used in Section 3.3 of the original paper. Rather than studying VPN detection directly, we follow the NetML setup for application identification and concentrate on how well standard models can separate application classes at three label granularities (easy, medium, hard). The results we we seek to reproduce are as follows:

Task	Metric	RF (Target)	SVM (Target)	MLP (Target)
non-vpn-t (7 classes)	F1	0.6273	0.5868	0.6066
	mAP	0.3257	0.1934	0.2304
non-vpn-m (18 classes)	F1	0.3693	0.3441	0.3609
	mAP	0.3223	0.1398	0.2041
non-vpn-f (31 classes)	F1	0.2486	0.2036	0.2359

	mAP	0.2127	0.0768	0.1404

Beyond simple reproduction, we also extend the baseline in a way that connects directly to the nPrint project. In addition to the metadata feature set used in the NetML paper, we generate nPrint bitmap representations for each flow and train comparable models on both feature spaces. Comparing Random Forest and SVM performance on metadata features versus nPrint bitmaps allows us to evaluate how much benefit these richer, packet level representations provide over simpler flow statistics, and to reflect on the tradeoffs between accuracy and feature complexity in practical network analytics systems.

Data Processing

Our analysis begins with a pcapML dataset, provided as a single compressed packet capture (PCAP) file. pcapML is a system designed to improve reproducibility in traffic analysis by leveraging the pcapng file format to encode metadata directly into raw traffic captures; this removes ambiguity about which packets belong to which traffic sample while remaining compatible with standard tools like tshark and tcpdump. The dataset organizes traffic into pre-segmented 'traffic samples,' where each sample corresponds to one application-level communication instance and comes with a hierarchical ground-truth label (easy / medium / hard) that identifies the application at different levels of specificity. We extract features exclusively from packet sizes and timestamps, avoiding any use of payload content (as we have done in this class, in case some packets are encrypted). This design keeps the feature set lightweight, broadly applicable, and usable in settings where inspecting packet contents is unavailable or undesirable. This format is powerful because it gives us supervised labels, but it is also restrictive; unlike the Barut paper, which relies on an Intel proprietary flow feature extraction tool to compute rich, engineered flow descriptors, pcapML does not hand us a ready-made feature table. Because we do not have access to Intel's extractor (and because the raw PCAP does not contain the proprietary flow-level features used in the paper), we must construct our own feature set directly from the observables available to us: packet size and timing behavior within each sample.

The sections that follow describe the major conceptual ‘families’ of features we engineered from these packet sequences; we group them at a high level because enumerating every individual feature would be overly detailed (and far too long).

From Packets to Flow-Level Signals

Once pcapML gives us a traffic sample, what we actually have in our hands is a packet sequence: a list of packet timestamps and packet sizes. This is the entire observable “behavior” of the application in this dataset. In turn, the conceptual move we make is to treat each sample like a time series generated by an unknown process (the application). Feature extraction is the step where we compress that time series into a fixed set of numbers that preserve the most application-identifying structure. Concretely, for every sample we iterate through its packets, record (1) size_i = len(pkt.raw_bytes) and (2) t_i = pkt.ts. Every feature we compute is a function

of one or both of those arrays. The categories below reflect different “views” of the same underlying behavior: volume, size shape, time shape, rate, and burstiness. Each view captures a distinct axis along which applications differ, even when payloads and protocol semantics are hidden.

Basic Volume and Duration Features

We start with three backbone descriptors: num_pkts (packet count), total_bytes (sum of packet sizes), and duration (time between first and last packet). These features capture the scale and timespan of a flow, which is often the first separation between application regimes. For example, streaming tends to produce long, sustained activity, while interactive apps can be shorter and more sporadic. Even when they can’t uniquely identify an application, volume-and-duration features anchor the model so “shape-only” similarities don’t collapse distinct behaviors.

Packet Size Distribution Features

Next we treat packet sizes as a distribution and summarize its center, spread, and shape (mean/std/median/quantiles, plus skew/kurtosis and ratios like CV and IQR). The point is that applications don’t just differ in who much they send; they differ in how they packetize info, which remains visible under encryption. Quintiles and medians keep these summaries robust when sizes are heavy-tailed or multimodal, and ratio features capture variability relative to scale, which often separates steady streaming from bursty message traffic.

Inter-Arrival time Features

Packet sizes miss timing, so we compute inter-arrival times (IATs) between consecutive packets and summarize that distribution with the same toolkit (mean/std/quantiles/skew/kurtosis/CV). These features capture cadence: whether packets arrive steadily or in stop-and-go bursts. A low iat_cv suggests regular pacing (common in streaming), while high dispersion suggests bursty interaction or scheduling effects. In practice, timing features often distinguish flows that look similar in size statistics.

Throughput-Style Rate Features

Rate features normalize volume by time to approximate intensity (packets or bytes per second). They separate flows that transmit similar total data but differ in delivery speed, which strongly correlates with application intent.

Coarse Packet-Size Histogram Features

Broad packet-size bucket proportions preserve multi-modal structure that summary statistics can hide. This captures mixtures of small control packets and larger data packets in a stable, low-dimensional form.

Temporal Concentration Features

By measuring what fraction of packets and bytes occur early versus late in a normalized timeline, we capture coarse phase structure without modeling full sequences. These features distinguish front-loaded, steady, and end-heavy communication patterns common across applications.

Fano Factors

Fano factors summarize how unevenly packets or bytes are distributed across time bins, providing a global measure of burstiness. They complement local timing statistics by capturing large-scale spiking behavior across the entire flow.

Second-Pass Derived Features: Why We Add Ratios, Logs, and Flags After the First Extraction

After baseline extraction, we add transforms that make patterns easier for the three models we train on to identify potential patterns and correlations. Log transforms stabilize heavy-tailed features like bytes and duration so the model doesn't over-prioritize extreme flows. Ratios like bytes_perpkt and bytes_per_time compress multiple signals into interpretable intensities, while variability ratios like size_std_over_mean capture dispersion independent of scale. Simple flags isolate special regimes that can otherwise distort continuous summaries. This pass doesn't add new information, it re-expresses what's already there in more separable forms.

Feature Set Summary and Transition to Modeling Results

Overall, these feature families convert each packet sequence into a consistent, flow-level summary of size, timing, and burstiness. While we cannot reproduce Intel's proprietary extractor, this set is our closest behavioral proxy using only the data within our PCAP file and the skills we have learned in this class. With features finalized, we now move to results for the three models (as in the paper) across the easy, medium, and hard label levels.

Baseline models and results

Random Forest

Random Forests are ensemble models that train many decision trees on bootstrap resamples of the data and then average their predictions. Each tree considers only a random subset of features at each split, which decorrelates the trees and usually improves generalization. We implement the model using scikit learn's RandomForestClassifier. For each label granularity we create a stratified 80 percent train, 20 percent test split from the metadata DataFrame, preserving class proportions.

With the default hyperparameters (Number of estimators: 100, Maximum depth: 10) the F1 scores were far below the NetML RF baselines, so we performed a light hyperparameter search using RandomizedSearchCV. For each of the three tasks we searched over the number of trees, maximum depth, fraction of features considered at each split, and the minimum number of samples required to split or stay in a leaf. We optimized macro F1 using three fold cross validation on the training set. The search used configurations similar to the following pattern:

```python
param_dist = {
    "n_estimators":     [100, 200, 400, 600],
    "max_depth":        [10, 20, 30, None],
    "max_features":     ["sqrt", 0.3, 0.5],
    "min_samples_split": [2, 5],
    "min_samples_leaf":  [1, 2, 4],
    "bootstrap":        [True],
}

rf_base = RandomForestClassifier(
    n_jobs=-1,
    random_state=0,
)

search = RandomizedSearchCV(
    estimator=rf_base,
    param_distributions=param_dist,
    n_iter=12,
    cv=3,
    scoring="f1_macro",
    verbose=1,
    n_jobs=-1,
    random_state=0,
)

search.fit(X_train, y_train)
best_rf = search.best_estimator_
y_test_pred = best_rf.predict(X_test)
```

For the easy, 7 class task, the best parameters were 200 trees, unlimited depth, max_feaures = “sqrt”, min_samples_leaf = 2, and min_samples_split = 2. With that model we obtained macro F1 = 0.3486, weighted F1 = 0.4437, micro F1 = 0.4104, and macro mAP = 0.4561 on the test set. In comparison, the paper reports a single RF F1 of 0.6273 and mAP of 0.3257 for this task. Our observed F1 scores are still substantially lower than their F1, but our mAP is noticeably higher.

For the medium, 18 class task, the best parameters were 400 trees, max_depth = 20, max_features = 0.3, min_samples_split = 5, and min_samples_leaf = 1. With that model we obtained macro F1 = 0.3563, weighted F1 = 0.3235, micro F1 = 0.2798, and macro mAP = 0.4387. The NetML baseline paper reported F1 = 0.3693 and mAP = 0.3223. Here, our record macro F1 is very close to the paper's F1, and we again see a much higher mAP score. Since our search explicitly optimizes macro F1, it encourages configurations that treat minority classes more fairly even if that hurts overall accuracy, which likely explains the macro vs micro discrepancy.

For the hard, 31 class task, the best parameters were 200 trees, unlimited depth, max_features = 0.3, min_samples_split = 2, and min_samples_leaf = 2. With that model we obtained macro F1 = 0.2921, weighted F1 = 0.1669, micro F1 = 0.1414, and macro mAP = 0.3571. The NetML baseline paper reported F1 = 0.2486 and mAP = 0.2127. Once again, our macro F1 is slightly higher than the reported F1, while micro F1 is much lower and mAP is significantly higher. The classification report shows that the forest is doing quite well on some small, well defined classes such as netflix, torrent, or specific file transfer subclasses, but performs poorly on very frequent, overlapping traffic types like facebook-audio and hangouts-audio. Because we tuned for macro F1, the hyperparameter search favors this behavior: improving performance on rare classes at the expense of the dominant ones yields a better average across classes even if it reduces overall accuracy.

Below are the full results:

Task	Metric	RF (Original)	RF (Ours)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.6273	0.3486, 0.4437, 0.4104
	mAP	0.3257	0.4561
non-vpn-m (18 classes)	F1 (Macro, Micro, Weighted)	0.3693	0.3563, 0.3235, 0.2798
	mAP	0.3223	0.4387
non-vpn-f (31 classes)	F1 (Macro, Micro, Weighted)	0.2486	0.2921, 0.1669, 0.1414
	mAP	0.2127	0.3571

In summary, after hyperparameter tuning our Random Forest reproduces the qualitative trends of the NetML paper: performance deteriorates as the label space expands from 7 to 31 classes, and imbalanced classes are particularly challenging. Quantitatively, our macro F1 scores for the medium and hard tasks are close to or slightly better than the published F1, while micro F1 is consistently lower and mAP is consistently higher. This pattern is consistent with using a different objective (macro F1) in the hyperparameter search and with small differences in preprocessing and train test splits, and it highlights how sensitive reported "single" F1 numbers can be to averaging conventions and optimization choices.

Support Vector Machine (SVM)

A Support Vector Machine is a large margin classifier that tries to find a decision boundary that separates classes while maximizing the margin between them and the closest training points, called support vectors. In the linear case this boundary is a hyperplane, and prediction is based on the sign of w·x + b. Because application identification is not linearly separable in the raw feature space, we use a kernel SVM with a radial basis function (RBF) kernel, which lets the model learn non linear boundaries in a higher dimensional implicit feature space.

To follow the NetML setup, we implement the same 80 percent, 10 percent, 10 percent split of the data. For each class separately, 10 percent of its flows are assigned to a “test challenge” set, another 10 percent to a “test standard” set, and the remaining 80 percent are used as the training pool. The NetML paper then uses only 10 percent of that training pool to actually fit the SVM and keeps the remaining 90 percent as a validation set, because training SVMs on the full dataset is expensive and scales poorly with the number of samples. We mirror that logic in our code.

```python
def netml_split_80_10_10(y, random_state=0):
    """
    Implement the NetML style split:
      - For each class separately:
        * randomly select 10 percent of samples to test_chal
        * randomly select 10 percent to test_std
        * remaining 80 percent to train
    Returns three index arrays: idx_train, idx_test_std, idx_test_chal
    """
    rng = np.random.RandomState(random_state)
    y = np.asarray(y)
    unique_classes = np.unique(y)

    idx_train = []
    idx_test_std = []
    idx_test_chal = []

    for cls in unique_classes:
        cls_idx = np.where(y == cls)[0]
        n = len(cls_idx)
        # shuffle indices for this class
        perm = rng.permutation(cls_idx)

        # number for each split (floor so small classes can end up with zero in
a test split)
        n_chal = int(0.1 * n)
```

```
        n_std = int(0.1 * n)

        chal_idx = perm[:n_chal]
        std_idx = perm[n_chal:n_chal + n_std]
        train_idx = perm[n_chal + n_std:]

        idx_test_chal.append(chal_idx)
        idx_test_std.append(std_idx)
        idx_train.append(train_idx)

    idx_train = np.concatenate(idx_train)
    idx_test_std = np.concatenate(idx_test_std)
    idx_test_chal = np.concatenate(idx_test_chal)

    return idx_train, idx_test_std, idx_test_chal
```

Given this split, we train and evaluate an RBF SVM using a scikit learn Pipeline. We standardize the features with StandardScaler, then apply SVC with C = 1.0, gamma = "scale", and probability = True so that we can compute mean average precision. Only 10 percent of the training pool (X_fit, y_fit) is used to fit the model, and the remaining 90 percent (X_val, y_val) is used as a validation set. We then evaluate on the test standard set. Below is the code for the train_eval_svm_for_label function, which performs the above described steps for any of the 3 label granularities given as an input to the function:

Python

```
def train_eval_svm_for_label(df, feature_cols, label_col, random_state=0):
    """
    Match the paper:
      - Per class 80 percent train, 10 percent test_std, 10 percent test_chal
      - Only 10 percent of the training set is used to fit SVM, remaining 90
percent is validation
      - RBF SVM with C = 1.0, gamma = 'scale'
      - Report F1 (macro, weighted, micro) and mAP (macro) on validation and
test_std
    """

    # 1. Build X, y
    X = df[feature_cols].values
    y = df[label_col].values

    # 2. NetML style 80 / 10 / 10 split per class
```

```
    idx_train, idx_test_std, idx_test_chal = netml_split_80_10_10(y,
random_state=random_state)

    X_train_full = X[idx_train]
    y_train_full = y[idx_train]

    X_test_std = X[idx_test_std]
    y_test_std = y[idx_test_std]

    X_test_chal = X[idx_test_chal]
    y_test_chal = y[idx_test_chal]

    print(f"[{label_col}] total samples: {len(df)}")
    print(
        f"[{label_col}] train: {len(X_train_full)}, "
        f"test_std: {len(X_test_std)}, "
        f"test_chal: {len(X_test_chal)}"
    )

    # 3. Inside training set, take random 10 percent for fitting SVM, 90
percent for validation
    rng = np.random.RandomState(random_state)
    n_train = len(X_train_full)
    n_fit = max(1, int(0.1 * n_train))

    perm_train = rng.permutation(n_train)
    idx_fit_local = perm_train[:n_fit]
    idx_val_local = perm_train[n_fit:]

    X_fit = X_train_full[idx_fit_local]
    y_fit = y_train_full[idx_fit_local]

    X_val = X_train_full[idx_val_local]
    y_val = y_train_full[idx_val_local]

    print(f"[{label_col}] SVM fit subset: {len(X_fit)} (10 percent of train)")
    print(f"[{label_col}] validation subset: {len(X_val)} (90 percent of
train)")

    # 4. SVM pipeline: StandardScaler + RBF SVM
    svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svc", SVC(
            kernel="rbf",
```

```
            C=1.0,
            gamma="scale",
            probability=True,  # needed for mAP
        )),
    ])

    # 5. Train on the 10 percent fit subset
    svm_clf.fit(X_fit, y_fit)

    def compute_metrics(model, X_split, y_split, split_name):
        """
        Compute F1 macro, weighted, micro and mAP macro for a split.
        """
        y_pred = model.predict(X_split)

        f1_macro = f1_score(y_split, y_pred, average="macro")
        f1_weighted = f1_score(y_split, y_pred, average="weighted")
        f1_micro = f1_score(y_split, y_pred, average="micro")

        # mAP (macro) one versus rest
        classes = np.unique(y_fit)  # classes seen in training
        y_true_bin = label_binarize(y_split, classes=classes)
        y_scores = model.predict_proba(X_split)

        # Some classes might not appear in this split; average="macro" handles
that
        mAP_macro = average_precision_score(y_true_bin, y_scores,
average="macro")

        print(f"\n[{label_col}] {split_name} F1 (macro):     {f1_macro:.4f}")
        print(f"[{label_col}] {split_name} F1 (weighted):  {f1_weighted:.4f}")
        print(f"[{label_col}] {split_name} F1 (micro):     {f1_micro:.4f}")
        print(f"[{label_col}] {split_name} mAP (macro):    {mAP_macro:.4f}")

        return {
            "f1_macro": f1_macro,
            "f1_weighted": f1_weighted,
            "f1_micro": f1_micro,
            "mAP_macro": mAP_macro,
            "y_pred": y_pred,
        }

    # 6. Metrics on validation and test_std
    val_metrics = compute_metrics(svm_clf, X_val, y_val, "validation")
```

```
    test_std_metrics = compute_metrics(svm_clf, X_test_std, y_test_std,
"test_std")

    print("\nValidation classification report:")
    print(classification_report(y_val, val_metrics["y_pred"]))

    print("\nTest_std classification report:")
    print(classification_report(y_test_std, test_std_metrics["y_pred"]))

    cm_test_std = confusion_matrix(y_test_std, test_std_metrics["y_pred"])

    return {
        "model": svm_clf,
        "idx_train": idx_train,
        "idx_test_std": idx_test_std,
        "idx_test_chal": idx_test_chal,
        "X_fit": X_fit,
        "y_fit": y_fit,
        "X_val": X_val,
        "y_val": y_val,
        "X_test_std": X_test_std,
        "y_test_std": y_test_std,
        "X_test_chal": X_test_chal,
        "y_test_chal": y_test_chal,
        "val_metrics": val_metrics,
        "test_std_metrics": test_std_metrics,
        "cm_test_std": cm_test_std,
    }
```

Below are the results achieved by our SVM compared with the model from the paper:

Task	Metric	SVM (Original)	SVM (Ours)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.5868	0.2310, 0.6149, 0.7211
	mAP	0.1934	0.2854
non-vpn-m (18 classes)	F1 (Macro, Micro, Weighted)	0.3441	0.1875, 0.3472, 0.4274
	mAP	0.1398	0.2446
non-vpn-f (31 classes)	F1 (Macro, Micro, Weighted)	0.2036	0.1251, 0.2453, 0.3218
	mAP	0.0768	0.1556

Overall, our SVM reproduces the general pattern reported in the NetML paper, but the exact scores differ in interesting ways. As the task becomes harder (7 to 18 to 31 classes), all of our F1 variants and mAP decrease, which matches the trend in the original results. For the easiest setting, our micro F1 of 0.6149 is close to the paper's single F1 of 0.5868, and for the medium task our micro F1 of 0.3472 is very close to their 0.3441. This suggests that the F1 reported in the paper is most likely a micro averaged F1 or an accuracy style metric, since that is the variant where our reproduction lines up best.

At the same time, our macro F1 scores are much lower than the original F1 values, especially on the medium and hard label sets. Macro F1 gives each class equal weight, so it penalizes the many rare applications that only appear a handful of times. Because we only use 10 percent of the training pool to actually fit the SVM, some minority classes end up with very few support vectors, and their per class F1 is often close to zero. This drags macro F1 down even when the model performs reasonably well on the head classes, which is reflected in our much higher weighted F1 values. In other words, our SVM is capturing the dominant applications fairly well but is struggling on the tail of the label distribution.

The mAP comparison tells a different story. For all three label granularities our macro mAP scores are noticeably higher than those reported in the paper, for example 0.2854 vs 0.1934 on the 7 class task and 0.2446 vs 0.1398 on the 18 class task. Since mAP evaluates the ranking of class probabilities rather than a single decision threshold, this suggests that our SVM's probability outputs are better calibrated or more informative, even though the hard class assignments give lower macro F1. Taken together, these results indicate that we have qualitatively reproduced the behavior of the original SVM, particularly in terms of micro level performance and the way accuracy degrades with more classes, but that differences in feature engineering, hyperparameters, and the exact F1 definition make exact numerical reproduction difficult

Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is a feed-forward neural network that stacks one or more fully connected layers and learns non-linear decision boundaries by backpropagation. Compared to Random Forests and SVMs, which work with hand-crafted splits or kernel similarity, an MLP learns its own intermediate representation of the features in a hidden layer. Following the NetML paper, we use a single hidden layer with 121 units, L2 regularization, and the Adam optimizer, combined with standardized input features.

We implement the MLP with scikit-learn's MLPClassifier. For each label granularity (easy, medium, hard) we reuse the same flow-level metadata features described in the Data Processing section. Starting from the feature table df_mlp, we take X to be all non-label, non-ID columns (48 metadata features total) and set y to the chosen label column (easy_label, medium_label, or hard_label) encoded with LabelEncoder. We then perform a stratified 80/20 train–test split and standardize all features so that each column has zero mean and unit variance. The core of the training function is:

Python

```python
def train_eval_mlp_for_label(df, feature_cols, label_col,
random_state=42):
    # Build X and encoded labels
    X = df[feature_cols].values
    y_raw = df[label_col].astype(str).values
    le = LabelEncoder()
    y = le.fit_transform(y_raw)

    # 80/20 stratified split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=0.2,
        stratify=y,
        random_state=random_state,
    )

    # Standardize features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # MLP with NetML-style parameters
    mlp = MLPClassifier(
        hidden_layer_sizes=(121,),  # one hidden layer, 121 units
        alpha=0.0001,               # L2 regularization
        solver="adam",              # Adam optimizer
        max_iter=500,
        random_state=random_state,
        early_stopping=True,        # 10% of training used as
validation
        validation_fraction=0.1,
        n_iter_no_change=20,
        verbose=False,
    )

    mlp.fit(X_train_scaled, y_train)
```

```
    # Predictions and metrics (macro / micro / weighted F1, macro
mAP)
    ...
```

This setup mirrors the NetML description in terms of network size, regularization, optimizer, and preprocessing, but uses a simpler split than the SVM section: a single 80/20 stratified train–test partition, with the full 80% used for training and only the internal early-stopping split acting as validation.

On the metadata feature set, the MLP sees all 158,355 flows (48 features plus labels) and learns one classifier per label level. For the easy, top-level labels (easy_label), this corresponds to 7 classes with 126,684 training examples and 31,671 test examples. For the medium level (medium_label), there are 18 classes under the same split, and for the hard, fine-grained level (hard_label), there are 31 classes under the same split.

For the easy, 7-class task, the MLP achieves macro F1 of 0.3860, micro F1 of 0.7322, weighted F1 of 0.6369, and macro mAP of 0.4151. The NetML paper reports a single F1 of 0.6066 and mAP of 0.2304 for this task. Our macro F1 is noticeably lower than 0.6066, but our micro F1 is actually higher (0.7322), and our mAP is substantially higher (0.4151 vs 0.2304). The classification report shows why: the MLP is excellent on the dominant audio class (F1 ≈ 0.84, recall ≈ 0.99) and solid on p2p and video, but almost never correctly predicts the large file-transfer class (F1 ≈ 0.02). Because macro F1 weights each class equally, poor performance on file-transfer and the tiny tor class pulls the macro score down even when overall accuracy (≈ 0.73) and performance on head classes look good.

For the medium, 18-class task, we obtain macro F1 of 0.2948, micro F1 of 0.4382, weighted F1 of 0.3610, and macro mAP of 0.3272. Compared to the NetML baseline of F1 = 0.3609 and mAP = 0.2041, our macro F1 is somewhat lower, but once again our mAP is clearly higher. Per-class scores show that the MLP does reasonably well on larger classes like facebook, torrent, and voipbuster (torrent reaches F1 ≈ 0.57), but it struggles on small, infrequent classes such as aim and ftps, where F1 hovers near zero. As the label space grows and more rare classes are introduced, this head–tail imbalance becomes even more severe.

For the hard, 31-class task, the MLP yields macro F1 of 0.2136, micro F1 of 0.3350, weighted F1 of 0.2668, and macro mAP of 0.2793. The corresponding NetML results are F1 = 0.2359 and mAP = 0.1404. Here our macro F1 is slightly below the paper's value, but our mAP is again roughly doubled. Looking at the fine-grained report, the model retains reasonable performance on structured classes like torrent (F1 ≈ 0.66) and voipbuster, while many tiny Tor-related subclasses (for example, tor-facebook with just one sample) receive effectively zero F1. This leads to the familiar pattern: high micro F1 and accuracy on the head classes, low macro F1 once all rare labels are included, and surprisingly strong macro mAP because the probability rankings are often sensible even when the final top-1 prediction is wrong.

We summarize the MLP metadata results alongside the NetML baselines below (our F1 numbers are given as macro, micro, weighted):

Task	Metric	MLP (Original)	MLP (Ours)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.6066	0.3860, 0.7322, 0.6369
	mAP	0.2304	0.4151
non-vpn-m (18 classes)	F1 (Macro, Micro, Weighted)	0.3609	0.2948, 0.4382, 0.3610
	mAP	0.2041	0.3272
non-vpn-f (31 classes)	F1 (Macro, Micro, Weighted)	0.2359	0.2136, 0.3350, 0.2668
	mAP	0.1404	0.2793

Overall, the metadata MLP behaves similarly to the Random Forest and SVM baselines: performance degrades as we move from 7 to 18 to 31 classes, and macro F1 is consistently lower than the single F1 reported in the paper, while micro F1 and mAP are comparable or better. The gap between macro and micro F1 highlights how strongly the model favors the frequent applications. At the same time, the consistently higher macro mAP suggests that our MLP's probability outputs are informative—useful for ranking and thresholding—even when top-1 accuracy on minority classes is weak.

nPrint Extension

Our baseline experiments only used the 63 metadata features that summarize each flow with simple statistics like packet counts and durations. These are cheap to compute but throw away most of the packet level structure that might distinguish similar applications. The nPrint project was designed to fix that, by turning raw packet headers into fixed length bitmaps that preserve much richer information about how flows behave over time. To see how much this representation matters for application identification, we generated nPrint feature vectors for each flow, joined them with the original NetML labels, and then ran the same Random Forest, SVM, and MLP models on these new features.

nPrint Feature Generation: Labeled Packets vs Flows

To generate nPrint features, we first had to step outside of the pcapML/NetML 'sample' abstraction and work directly with packet-level traffic derived from the underlying PCAP.

Practically, this meant reconstructing each labeled traffic instance as an explicit packet sequence with consistent ordering and flow identity, and then materializing those instances into individual PCAP files. Doing this gave us full control over segmentation, and it allowed us to run standard tooling on each example without depending on any proprietary feature extractor.

Once each flow was represented as a standalone PCAP, we ran the nPrint pipeline to convert raw packet headers into fixed-length representations. nPrint encodes header bits across the packet sequence into a bitmap-like structure, and we enforced consistent truncation/padding so every example produced the same dimensional output. The result was a new feature table where each row corresponded to a flow's nPrint vector, preserving substantially more of the within-flow packet structure than the baseline summary statistics.

After extraction, we joined these nPrint features back to the original NetML labels to ensure supervision stayed aligned with the exact traffic used for featurization. Because nPrint produces packet-level representations, this join is not one flow → one feature. Instead, each flow produces a sequence of packet-derived nPrint rows, and we propagate the parent flow/application label down to every packet row. This yields a supervised packet dataset where each training example is a single packet's nPrint feature vector paired with the application label of the flow it came from.

Furthermore, this packet-level setup changes what the models are learning and how results should be interpreted. Because the number of training examples becomes proportional to the number of packets, we hypothesize that long or bursty flows will contribute many more labeled samples than short flows, shifting the effective class balance and potentially biasing learning toward traffic that naturally generates more packets. It also introduces strong within-flow correlation: packets from the same flow appear in both sets. At the same time, packet-level nPrint can capture early-stage or fine-grained header patterns that flow summary statistics blur out, which is exactly the representational advantage we want to test; yet, it requires careful evaluation. We want to ensure gains reflect generalization across flows, not oversampling within them.

Random Forest

Task	Metric	RF (Original)	RF (Ours)	RF (nPrint)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.6273	0.3486, 0.4437, 0.4104	0.7891 (weighted)
	mAP	0.3257	0.4561	0.7761
non-vpn-m (18 classes)	F1 (Macro, Micro, Weighted)	0.3693	0.3563, 0.3235, 0.2798	0.6764 (weighted)
	mAP	0.3223	0.4387	0.8703
non-vpn-f (31 classes)	F1 (Macro, Micro, Weighted)	0.2486	0.2921, 0.1669, 0.1414	0.6678 (weighted)

	mAP	0.2127	0.3571	0.8622

For Random Forest, nPrint features turn a relatively weak baseline into a very strong classifier. On the 7 class task, the original NetML paper reports a single RF F1 of 0.6273 and mAP of 0.3257. Our metadata RF lands below that in terms of F1, with macro, micro, and weighted F1 scores of 0.3486, 0.4104, and 0.4437, although it already improves mAP to 0.4561. When we swap in nPrint features, the weighted F1 jumps to 0.7891 and mAP to 0.7761, so both overall accuracy and ranking quality are now clearly better than the original baseline. The effect is even more pronounced as the label space grows. On the 18 class task the original RF has F1 of 0.3693 and mAP 0.3223, our metadata RF is similar with macro, micro, and weighted F1 of 0.3563, 0.2798, and 0.3235 and mAP 0.4387, but the nPrint RF reaches weighted F1 of 0.6764 and mAP 0.8703. On the 31 class task the original RF reports F1 of 0.2486 and mAP 0.2127, while our metadata RF gets macro F1 0.2921, micro 0.1414, weighted 0.1669 and mAP 0.3571. With nPrint, weighted F1 climbs to 0.6678 and mAP to 0.8622. In other words, once we give the forest packet level structure instead of just metadata, its performance roughly doubles across the board, and in some cases more than triples the original mAP, especially on the medium and hard label sets.

Support Vector Machine

Task	Metric	SVM (Original)	SVM (Ours)	SVM (nPrint)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.5868	0.2310, 0.6149, 0.7211	0.6395, 0.8010, 0.8296
	mAP	0.1934	0.2854	0.7468
non-vpn-m (18 classes)	F1 (Macro, Micro, Weighted)	0.3441	0.1875, 0.3472, 0.4274	0.5921, 0.6527, 0.6541
	mAP	0.1398	0.2446	0.8781
non-vpn-f (31 classes)	F1 (Macro, Micro, Weighted)	0.2036	0.1251, 0.2453, 0.3218	0.5370, 0.5940, 0.6324
	mAP	0.0768	0.1556	0.7867

SVM shows the same pattern, but even more sharply. On the 7 class task the original SVM baseline has F1 of 0.5868 and mAP 0.1934. Our metadata SVM has macro, micro, and weighted F1 of 0.2310, 0.6149, and 0.7211, with mAP 0.2854. That is, our micro F1 is close to or slightly better than the original F1, but macro F1 is much lower, indicating that rare classes are not handled well. With nPrint features the SVM moves into a different regime: macro, weighted, and micro F1 become 0.6395, 0.8010, and 0.8296, and mAP jumps to 0.7468. Similar trends appear on the harder tasks. For 18 classes the original SVM has F1 0.3441 and mAP 0.1398, our metadata SVM gets macro, micro, and weighted F1 of 0.1875, 0.3472, and 0.4274 with mAP 0.2446, while the nPrint SVM reaches macro, weighted, and micro F1 of 0.5921, 0.6527, and 0.6541 and mAP 0.8781. For 31 classes, the original SVM has F1 0.2036 and mAP

0.0768. Our metadata SVM manages macro, micro, and weighted F1 of 0.1251, 0.2453, and 0.3218 with mAP 0.1556, but the nPrint SVM jumps to macro, weighted, and micro F1 of 0.5370, 0.5940, and 0.6324, with mAP 0.7867. Across all three tasks, nPrint roughly triples macro F1 and mAP compared to the metadata SVM, and in the medium and hard settings the nPrint SVM not only surpasses our metadata reproduction, it also clearly outperforms the original NetML SVM baseline. Together with the Random Forest results, this strongly suggests that representation is the main bottleneck: once we expose packet level structure through nPrint, both models become competitive and robust even in the most fine grained label configuration.

Multi-Layer Perceptron

Task	Metric	MLP (Original)	MLP (Ours, Metadata)	MLP (nPrint)
non-vpn-t (7 classes)	F1 (Macro, Micro, Weighted)	0.6066	0.3860, 0.7322, 0.6369	0.7831, 0.9100, 0.9103
	mAP	0.2304	0.4151	0.8257
non-vpn-m (18/15 classes)	F1 (Macro, Micro, Weighted)	0.3609	0.2948, 0.4382, 0.3610	0.8359, 0.8705, 0.8699
	mAP	0.2041	0.3272	0.8933
non-vpn-f (31/24 classes)	F1 (Macro, Micro, Weighted)	0.2359	0.2136, 0.3350, 0.2668	0.8129, 0.8641, 0.8634
	mAP	0.1404	0.2793	0.8539

MLP follows the same pattern, but with an even bigger jump once we move to nPrint. On the 7-class task, the original NetML MLP baseline has F1 of 0.6066 and mAP of 0.2304. Our metadata MLP sits below that in terms of macro F1 but above it in micro F1, with macro, micro, and weighted F1 of 0.3860, 0.7322, and 0.6369 and mAP of 0.4151. When we swap in nPrint features, the model moves into a new regime: macro, micro, and weighted F1 become 0.7831, 0.9100, and 0.9103, and mAP rises to 0.8257, so both accuracy and ranking quality are now far beyond the original baseline. The gap widens on the harder tasks. For the medium-granularity labels, the NetML MLP reports F1 of 0.3609 and mAP of 0.2041, while our metadata MLP reaches macro, micro, and weighted F1 of 0.2948, 0.4382, and 0.3610 with mAP 0.3272; on the corresponding nPrint subset (15 classes instead of 18), the MLP jumps to macro, micro, and weighted F1 of 0.8359, 0.8705, and 0.8699 and mAP 0.8933. For the fine-grained task, the original MLP baseline has F1 of 0.2359 and mAP of 0.1404, and our metadata MLP gets macro, micro, and weighted F1 of 0.2136, 0.3350, and 0.2668 with mAP 0.2793; on the nPrint subset (24 classes instead of 31), the same architecture reaches macro, micro, and weighted F1 of 0.8129, 0.8641, and 0.8634 and mAP 0.8539. Across all three label levels, nPrint more than

doubles, and often triples, macro F1 and mAP compared to the metadata MLP, and the nPrint MLP comfortably outperforms the original NetML baseline in every setting. Taken together with the Random Forest and SVM results, this makes the point stark: once we expose packet-level structure through nPrint, the MLP becomes our strongest model, and representation, not the choice of classifier, is the real bottleneck.

Conclusion

We started with a simple goal: replay the NetML baselines on the non-vpn-2016 application identification task and see how far we could get with the tools and data we actually had. Doing that forced us to confront what “reproduce a paper” really means in practice. The original results rest on a proprietary feature extractor, unspecified train–test splits, and a single F1 number per model and task, with no clear averaging scheme. Our project sits in that gap: same dataset family, similar models, different features and experimental choices.

On the metadata features, all three models—Random Forest, SVM, and MLP—told the same story. We recovered the qualitative trends from the paper:

- performance drops as we move from 7 → 18 → 31 classes,
- minority classes are hard across the board,
- the three models are in the same rough performance band.

Quantitatively, our scores diverge in consistent ways. Our macro F1 values tend to come in below the paper’s single F1 numbers, while our micro F1 and weighted F1 are often close or higher. Our mAP scores are almost always higher. This pattern shows up for RF, SVM, and MLP: when we average over all classes equally, we pay for the long tail of tiny labels; when we weight by frequency or look at ranking quality instead of hard decisions, our models look much healthier. It also strongly suggests that the F1 reported in the original NetML tables is closer to an accuracy or micro-style metric than to macro F1.

The more important result, though, is that representation beats model choice. When we swap the 60-ish flow statistics for nPrint packet-level bitmaps, every classifier we tried jumps to a different regime. Random Forest goes from a middling baseline on metadata to a very strong nPrint model with much higher F1 and mAP at all label granularities. The SVM, which had struggled on rare classes, becomes competitive across the label space and posts large gains in macro F1 and mAP. The MLP benefits the most: on nPrint features it reaches macro F1 and mAP well above 0.8 on the medium and fine-grained tasks, far beyond what we could do with flow summaries alone and well beyond the original MLP baselines.

That contrast cuts to the heart of the experiment. With metadata features, we could tune hyperparameters, tweak splits, and debate macro vs micro averaging and still live in roughly the same accuracy band as the paper. With nPrint, we did not change the model families at all; we only changed what they see. The jump in performance—especially in macro F1 and mAP—makes it clear that the bottleneck was not the classifier, but the information we were

feeding it. Once we exposed packet-level structure, all three models had enough signal to separate even fine-grained classes much more cleanly.

Along the way, we also got a concrete feel for the fragility of reproducibility in ML:

- Small changes in the split (stratified 80/20 vs the NetML 80/10/10 protocol) move metrics by non-trivial amounts.
- Choosing macro vs micro F1 changes the story you tell about “how good” a model is.
- Relying on a proprietary feature extractor makes exact reproduction impossible for anyone outside the original environment.

None of these choices are unusual in a research paper, but together they make exact numerical reproduction a moving target. Our results show that you can match the shape of a result—relative ordering of models, degradation with task difficulty—without matching the number to the third decimal. For applied work, that shape may matter more.

There are clear limitations to what we did. We only worked on the non-vpn-2016 subset, did not attempt heavy hyperparameter searches for SVM and MLP, and treated nPrint features at the packet level rather than enforcing a strict flow-level train/test separation. That design makes it easier to train and shows the representational gains clearly, but it also means our packet-level experiments may overestimate how well models generalize across flows from the same application. A more stringent setup would keep flows, not packets, as the unit of evaluation and might close some of the gap we see.

If we had more time, the next steps are straightforward. Combine metadata and nPrint features to see whether flow-level statistics still add value on top of rich packet-level views. Tighten the evaluation so that flows, not packets, define the split. Explore sequence models that treat the packet stream directly instead of flattening it into a bitmap. Most importantly, write down every experimental detail—splits, averaging schemes, and feature definitions—so that the next group does not have to reverse-engineer our setup from a few tables.

In the end, our reproduction effort did two things. It confirmed that the NetML baselines are directionally sound on this dataset, and it showed that better features, not fancier models, are the fastest way to push performance forward. For network traffic analytics, that is a useful lesson: if you want a stronger baseline, start by changing what the model sees, not how you optimize it.

Works Cited

Barut, O., Luo, Y., Zhang, T., Li, W., & Li, P. (2020). *NetML: A challenge for network traffic analytics*. arXiv preprint arXiv:2004.13006. https://doi.org/10.48550/arXiv.2004.13006

Acknowledgements

We wanted to acknowledge that we completed this assignment with the aid of ChatGPT and Claude.