Planter: Rapid Prototyping of In-Network Machine Learning Inference

Changgang Zheng[†], Mingyuan Zang[§], Xinpeng Hong[†], Liam Perreault[†], Riyad Bensoussane[†],
Shay Vargaftik[◇], Yaniv Ben-Itzhak[◇], and Noa Zilberman[†]

[†]University of Oxford, [§]Technical University of Denmark, [◇]VMware Research (by Broadcom)
{changgang.zheng, xinpeng.hong, noa.zilberman}@eng.ox.ac.uk, minza@dtu.dk, {liam.perreault,
riyad.bensoussane}@worc.ox.ac.uk, {shay.vargaftik, yaniv.ben-itzhak}@broadcom.com

ABSTRACT

In-network machine learning inference provides high throughput and low latency. It is ideally located within the network, power efficient, and improves applications' performance. Despite its advantages, the bar to in-network machine learning research is high, requiring significant expertise in programmable data planes, in addition to knowledge of machine learning and the application area. Existing solutions are mostly one-time efforts, hard to reproduce, change, or port across platforms. In this paper, we present Planter: a modular and efficient open-source framework for rapid prototyping of in-network machine learning models across a range of platforms and pipeline architectures. By identifying general mapping methodologies for machine learning algorithms, Planter introduces new machine learning mappings and improves existing ones. It provides users with several example use cases and supports different datasets, and was already extended by users to new fields and applications. Our evaluation shows that Planter improves machine learning performance compared with previous model-tailored works, while significantly reducing resource consumption and co-existing with network functionality. Planter-supported algorithms run at line rate on unmodified commodity hardware, providing billions of inference decisions per second.

KEYWORDS

In-Network Computing; Machine Learning; Dimension Reduction; Modular Framework; Machine Learning Compilers; Programmable Switches; P4.

1 INTRODUCTION

The rapid growth of data volume and the increasing demands for data exploitation are creating an ever-increasing processing burden on computing systems [96]. The need to scale computing resources and the emergence of programmable network devices drove researchers to leverage the underused processing resources within the network [18, 41]. Processing within the network, also known as in-network computing, was shown to boost performance, while at the same time increasing power efficiency [84]. It was demonstrated to improve a range of applications, from network services and monitoring [3, 42, 45, 55] to caching and consensus [18, 41].

In-network computing was suggested as a means to improve machine learning (ML) performance, both in the acceleration of host-based ML training using in-network aggregation [29, 48, 73], and ML inference by using in-network classification [72, 91, 98]. In-network ML inference benefits from its deployment location, illustrated in Figure 1, as data can travel directly from source to destination, without additional hops through inference servers. This provides latency benefits and high throughput, without introducing additional traffic into the network, as in server or GPU-based inference.

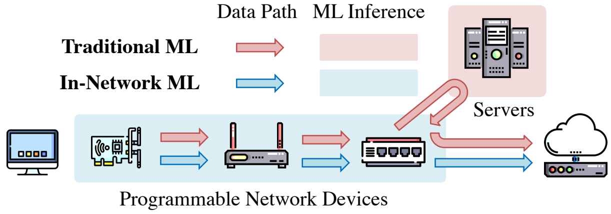


Figure 1: Difference in traffic flow between traditional ML in the network domain and in-network ML.

While in-network ML is appealing, its practical adoption within the research community was limited for two reasons. First, from the *design and realization* perspective, there was no general solution or a standard methodology for mapping different ML models to programmable data planes. Furthermore, many mapping solutions suffered from stage and memory explosion [7, 26], thereby limiting model size and consequently compromising ML performance. Second, from the *prototyping and applications' development* perspective, developing in-network ML solutions requires a breadth of expertise, in programmable data planes, ML and the application field. Consequently, in-network ML was limited to a small number of research groups. Moreover, the one-time tailored-design nature of in-network ML works made reproducibility hard, let alone comparing and contrasting solutions on different targets.

There is a significant demand for rapid prototyping, agile deployment, fair comparison, and seamless portability of in-network ML solutions, which remained unfulfilled by previous research efforts.

In this paper, we present Planter, a modular framework for rapid in-network ML inference prototyping. Planter supports the development, testing and deployment process end-to-end, in an automated manner. By defining three general mapping methodologies, Planter enables a large variety of ML models. It supports a range of tree-based models, statistical, neural network, clustering, dimensionality reduction and other models, listed in Table 1. The modular design of the Planter framework enables support of multiple architectures and target devices, as shown in the table. Furthermore, it supports several use cases (e.g., anomaly detection, financial market prediction) and was already extended by early adopters. Planter is modular and scalable, extendable to future in-network ML models

Machine Learning Models	
Decision Tree (DT), Random Forest (RF), XGBoost (XGB), Isolation Forest (IF), Support Vector Machine (SVM), Neural Networks (NN), k-Nearest Neighbors (KNN), k-means (KM), Naïve Bayes (NB), Autoencoder (AE), Principal Component Analysis (PCA)	
Architectures	Targets
PSA, v1model, Intel TNA, AMD XSA, NVIDIA Spectrum	P4Pi (CPU), BMv2 (CPU), T4P4S (CPU), Intel Tofino (Switch), Tofino2 (Switch), AMD Alveo U280 (FPGA), NVIDIA BlueField2 (DPU)

Table 1: Planter supported ML models, data plane architectures, and programmable network targets to data, which can and will be further extended.

and use cases. It allows reproducibility and comparison of different in-network ML mapping methods.

The main contributions of this paper are:

- Presenting Planter, a framework facilitating the deployment of in-network ML inference across architectures and targets. Planter is modular, allowing easy integration of new models, targets, architectures, and use-cases.
- Defining three general model mapping methodologies, enabling the mapping of a wide range of ML inference algorithms onto programmable data planes.
- Introducing four newly mapped inference algorithms, enhancing the efficiency of six previously proposed mappings, and supporting four more state-of-the-art mapping solutions.
- Comparing the performance of multiple previously proposed solutions on a single platform, for several use cases. Our evaluation highlights the need to support different models, and shows Planter high resource efficiency compared with previous proposals while achieving the same or better ML inference performance.
- Making Planter openly available to the research and user community, and enabling in-network ML development and deployment without expert knowledge. Our open framework enables transparent verification, collaborative refinement, and broader adoption of novel in-network ML.

2 BACKGROUND AND MOTIVATION

2.1 Background

Programmable Network Devices: The programmable network devices have enabled users to create customized data planes, driven further by the introduction of the P4 programming language [5]. Today, programmable data planes are supported on a range of hardware and software targets, using different pipeline architectures (PSA [62], TNA [39], v1model [67], and others). Despite the use of a common language, compiling programs to different targets turned out to be complex, requiring specialized solutions (e.g., Lyra [28], μP4 [79]).

In-Network ML Inference: In-network ML inference is the partial or full offloading of ML inference algorithms to run within network devices [7, 50, 76, 81, 91, 92, 98, 102]. In this work, we limit the scope to the forward classification process of ML algorithms being offloaded to the data plane, while the training part remains on the host (including accelerators) or in the control plane. In-network ML solutions follow an *offline training, online (in-band) inference* pattern. Feature extraction can be done either by parsing within the data plane or using customized headers [7, 26]. A mapped ML model is typically implemented within the Match-Action (M/A) pipeline, and the decision can be stored in a header or turned into an action within the network device [102]. A related research area, in-network aggregation [48, 73], leverages programmable devices to tackle the communication bottleneck in training aggregation. However, in-network aggregation is outside the scope of this work. In-network aggregation is used for ML training, and is based on using registers within the data plane. In contrast, this work focuses on inference and mainly uses match-action tables (§3), an inherently different research domain.

2.2 In-Network ML Motivation

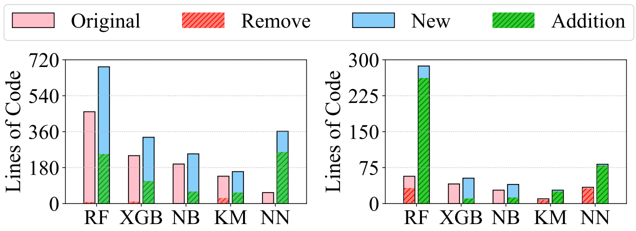
Programmable network devices primarily provide switching and routing-support functions. These functions require a significant portion of the programmable devices' resources, but often don't exhaust them, as demonstrated by Intel Tofino's *switch.p4* reference design (L2/L3 switch). In-network ML tasks can utilize remaining resources, co-existing with mandatory and traditional switch functions (see §5.2.1).

Motivating Use Cases. Several use cases are commonly tied with ML for networking, and are applicable to in-network ML. *Traffic Engineering*: The use of ML to improve traffic engineering can reduce communication overheads between cloud and edge [63, 64], improve heavy hitter detection [95] and quality of experience (QoE) prediction [86], and support IoT classification at line-rate [91]. *Anomaly Detection*: ML for networking is often used for network security [2, 16, 50, 88], allowing early detection and fast mitigation, potentially preventing distributed attacks. *Resource Management*: ML has been used to optimize scheduling algorithms in network infrastructure [38, 56] and to drive automatic congestion control [21, 46, 105]. *Financial Prediction*: ML models are widely used to perform financial tasks, importantly stock market forecasting [14, 33, 51, 75]. In parallel, hardware and software solutions are used to accelerate financial applications [49, 54]. In a field where every nanosecond counts, the two-fold goal is increasing prediction accuracy, while minimizing latency.

2.3 The Gap in In-Network ML

After five years of research on in-network ML, we find it challenging to develop and prototype in-network ML, especially along with its application across use cases. A gap exists in efficiently producing high-performance in-network ML prototypes on commodity platforms, which can be classified into two folds, ML performance and ML deployment.

In-Network ML Performance. Although in-network ML inference mapping (e.g. SVM [91], KM [26], DT [50], RF [7], and NN [76]) have been proposed and show promise, commodity hardware resources restrict the size and complexity of the models, resulting in compromised accuracy [76]. To obtain high performance: *1.* Individual model mapping should be inherently efficient. *2.* Options for diverse models and mapping approaches are essential for mitigating



(a) Data Plane. (b) Control Plane.

Figure 2: LOC changes as a result of single setup change (UNSW dataset [57]). RF - depth of 2 to 5, XGB - 2 to 6 trees, NB - 2 to 5 features, KM - *v1model* to *TNA*, NN - 2 to 5 layers.

the limitations of a single model or mapping. *3.* Fair comparisons are essential to ensure the selection of optimal solutions across scenarios. Such demands are not yet fulfilled by existing work [26, 50, 91]: *1.* Many in-network ML works suffer from stage or entry explosion [7, 26, 91]. *2.* Most existing works support only one type of ML model [50, 76]. The absence of a discussion on mapping methodologies further constrains the extension of existing research to additional ML models. *3.* Existing endeavors lack comprehensive comparisons of different models and solutions [7, 77].

In-Network ML Deployment. When it comes to practical use case deployment, diverse use cases and time-varying data inputs require ML models to be adjusted accordingly. Model deployment, comparison, tuning, or retraining is inevitable, but it is cumbersome for current solutions to realize such procedures on the data plane. Figure 2 demonstrates the effect of ML model changes in terms of Lines of Code (LOC). Common changes like increasing model size, adding more features, or moving between targets require changing hundreds of LOC in the data plane, the same scale as the entire original code. Changes also affect M/A table rules, added or removed through the control plane. A framework for rapid prototyping is imperative. Otherwise, intricate modifications can result in endless debugging and limit the efficiency of model deployment. The absence of a framework further hinders efficient model comparison, selection, and replacement, as well as swift migration among use cases.

2.4 Planter Design Goals

Planter aims to narrow the gap to production of in-network ML inference, and sets the following design goals:

Efficient In-Network ML Mappings (§3). *1. General mapping methodology.* Fundamental characteristics of ML inference algorithms should be identified and their data plane mapping should be simplified. This can be used as a guidance for in-network ML realization and drive the implementation of new models and their variations (§3.1). *2. Optimized models.* ML inference models mapped to programmable data planes should provide high ML and system performance, with minimum resource overheads. As programmable network devices are primarily designed for packet processing, they have limited resources, and support a constrained set of mathematical operations. Mapped ML algorithms need to trade off model size and ML performance to fit on a network device, and should not degrade network performance. Thus, Planter should support a wide range of predefined mapped ML models, with optimized resource efficiency, that can co-exist with mandatory network functionality (§3.2 - 3.4).

Ease of Use (§4). *1. One-click framework.* Mapping ML models to programmable network devices using P4 should be easy. However, the deployment of in-network ML models itself is complex. New or changed targets, architectures, and models may result in significant changes in (i) trained models, (ii) P4 programs, (iii) table entries & register values, and (iv) tables' update process. An easy-to-operate solution is needed to handle the deployment process and flexibly adapt to changes (§4.1). *2. Extensibility and Portability.* Extending and porting different inference solutions should require minimum effort. ML algorithms are emerging rapidly and new programmable network targets are reaching the market. The framework should be able to support new ML algorithms, architectures, and targets. It should be easy to add new application scenarios and to port designs between different devices. This calls for a modular design with elements easily added, updated, or replaced, independently of other components (§4.2 - §4.3). *3. Automated Framework.* Not all in-network ML users have ML expertise. It is critical to have a tool to handle compilation failures and automatically select the best hyperparameters under constraints. An automated front-end is needed to drive the in-network ML generation framework in realizing models (§4.4).

3 PLANTER'S ML MODEL MAPPING

3.1 Mapping Methodologies

Mapping ML inference models to the data plane can be challenging, therefore many previous works [7, 26, 50, 69, 76, 77] have focused on a single model. While a single-model approach has benefits, it also limits the agility and adaptability to use cases, creating a barrier to adoption of new algorithmic solutions. Planter tackles this challenge by proposing three general ML model mapping methods, based on similarities in models' structures.

A model's mapping needs to attend to the constraints of programmable network devices, such as pipeline architectures, resources (e.g., stages and memory), and limited mathematical operations. To enable the mapping of multiple different models under these constraints, we propose three general mapping methodologies: direct-mapping (DM), encode-based (EB), and lookup-based (LB). Some models have a clear sequential inference process, where a DM (direct-mapping) solution can be used (§3.2). However, when the inference model has no assumption on data distribution and the inference is based on splitting the feature space, DM solutions may result in high pipeline-stage consumption. EB (encode-based) solutions in Planter mitigate this issue in applicable algorithms by using input feature-space partitioning to encode the feature space, breaking direct-mapping's connection between model size and number of stages (§3.3). When an inference model uses complex mathematical operations (e.g. logarithms, root) that are not supported by network devices, Planter can utilize an LB (lookup-based) solution by leveraging match-action tables to lookup intermediate results (§3.4).

This section describes the three mapping methodologies, using leading examples per method. All the mappings are available in Planter's repository [104]. Table 2 demonstrates distinct implementations of ML models in Planter, categorized according to the three methodologies. As shown in the table, in Planter, four inference models are completely new mappings, six mapping proposals are improved, and six more are supported.

	Supported			Improved				New Mapping			
Types	SVM	KM	NN	DT	RF	XGB	NB	IF	KNN	PCA	AE
EB		✧$_2$		▮$_7$	▮$_8$	▮$_6$		✓$_4$	✓$_2$		
LB	✧$_2$	✧$_2$					▮$_5$			✓$_2$	✓$_2$
DM			✧$_3$	▮$_2$	▮$_4$						

Table 2: Different ML models, and their Planter implementation using the three methods. Notation: ✓new, ▮ improved, ✧ supported. ✧$_n$, ▮$_n$ or ✓$_n$ indicates n supported variations.

3.2 Direct-Mapping

Researchers have identified early on, that some ML algorithms have a structure relatively similar to a data plane [91], simplifying their mapping. However, some adaptations are still required.

3.2.1 Ensemble Trees. The direct-mapping (DM) method of the Ensemble Trees, e.g., Direct Mapped Decision Trees (DT_{DM}) and Direct Mapped Random Forest (RF_{DM}) [7, 50], shares a similar mapping method. To map a p-depth (level) model, a single DT_{DM} uses p tables. Each table uses the result from the previous level as the key. The workflow checks the current branch ID, its threshold, and the used feature. After the lookup, a comparison is done between the matched value and a threshold. The comparison's result and the current branch ID are used as the keys in deeper levels. Random Forest (RF) is an ensemble model built from a set of DT models [34]. RF_{DM} uses a similar mapping process, with an extra decision table used to conclude the labeling. DM Ensemble Trees consume relatively low memory, but the logic operations are complex and lookups need to be executed sequentially due to the strong dependency between parent and child nodes, which is stage-consuming and latency-consuming. Planter in reaction optimizes the implementation in [50] by enabling parallel trees placement.

3.2.2 Neural Networks (NN). NN stacks activators to do feature extraction and classification of input data. Due to its layer-based structure, Planter supports DM-based Binary Neural Networks (BNN), as proposed in [70, 76], by using Popcount (Hamming weight) and XNOR operations as an alternative to complex activation functions. M/A is another alternative to realize the complex activation function with the trade-off between memory (M/A) and stages (Popcount & XNOR).

Potential Extensions Planter currently supports three DM models. By using alternative operations and approximation, DNN [61], CNN [4] and LSTM [35] become candidates for additional model mappings.

3.3 Encode-Based Mapping

From an intrinsic perspective, many classification algorithms aim to find borders in either the original feature space or the mapped feature space. The area confined by a set of borders (partitions) is labeled as a class. Algorithms use different methods to define their borders. Some use complex functions, while others use linear functions for approximation. Planter proposed encode-based (EB) mapping typically uses linear borders to slice the feature space with a code to represent a certain area within the space.

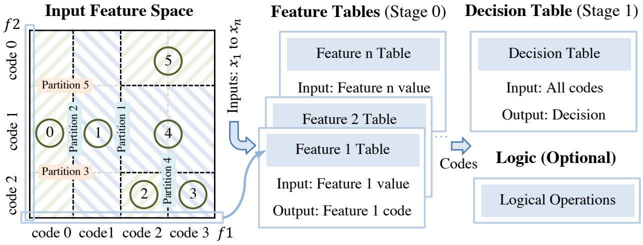


Figure 3: Mapping methodology of EB solutions.

In a general EB model, the mapping to the data plane starts with slicing input feature space into classes, using feature tables and a decision table. As shown in Figure 3, using a trained model as an example, feature space (e.g., two-dimensional space) is sliced into 6 areas (i.e., area ⓪ to area ⑤) by 5 partitions (i.e., partition Partition 1 to partition Partition 5). Mapping this ML model to a M/A pipeline requires two feature tables, recording the mapping from feature values to codes, where code pairs represent an area (e.g., area ③ coded as f1-code 3 & f2-code 2). The mapping from codes to labels is stored in a decision table. In an n dimensional feature space, the model similarly needs n feature tables and 1 decision table. EB models vary depending on how the feature space is split. The realization details of each Planter-supported model are shown as follows.

3.3.1 Decision Tree (DT). DT uses a top-down decision process, splitting the feature space at each branch (node) until reaching the leaf nodes [87]. Figure 4 shows a sample DT model and a two-dimensional input feature space split by its branches. The similarity between Figures 3 and 4 indicates that the general EB solution fits DT models. For an n features input space, an encode-based DT (DT_{EB}) requires n feature tables, encoding each feature value. The encoded feature space is mapped using a decision table to labels. All feature tables share a single pipeline stage (within target limitations) and the entire mapping requires only two logical stages.

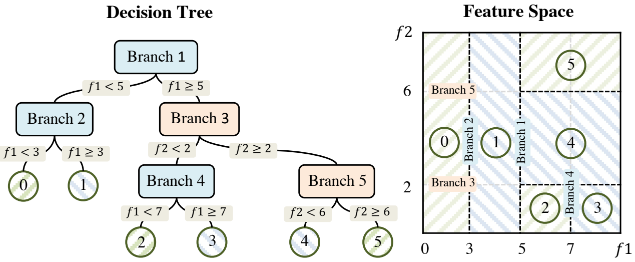
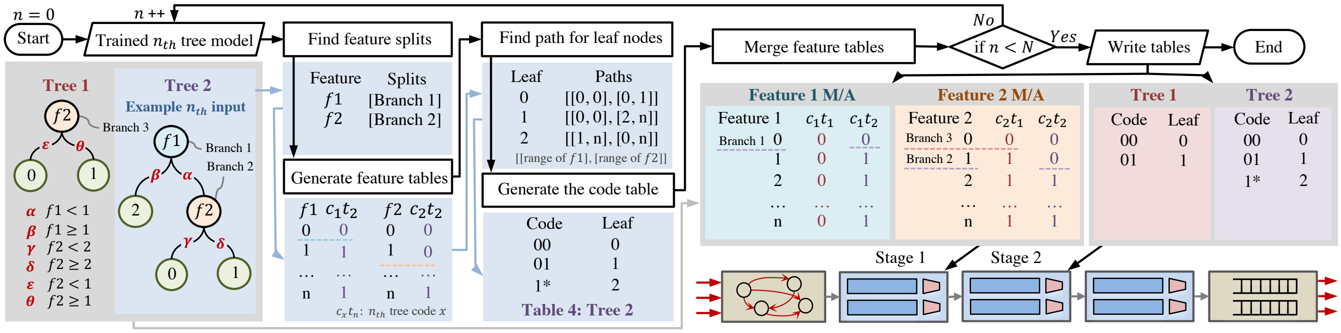


Figure 4: Feature space split in tree-based models.

To realize DT_{EB} mapping, we use four steps, as shown in Figure 5 Tree 2. The input to the process is a trained DT. In the step titled "Find feature splits", the algorithm collects all the branches related to each feature. The feature values are encoded (mapping an area to a code word) and saved as a feature table in the step "Generate feature table". The encoding is determined by the splitting conditions of the branches. The algorithm associates each area in the feature space



Feature	Splits
$f1$	[Branch 1]
$f2$	[Branch 2]

$f1$	c_1t_2	$f2$	c_2t_2
0	0	0	0
1	1	1	0
…	…	…	…
n	1	n	1

c_xt_n: n_{th} tree code x

Leaf	Paths
0	[[0, 0], [0, 1]]
1	[[0, 0], [2, n]]
2	[[1, n], [0, n]]

[[range of $f1$], [range of $f2$]]

Code	Leaf
00	0
01	1
1*	2

Table 4: Tree 2

Feature 1 M/A

	Feature 1	c_1t_1	c_1t_2
Branch 1	0	0	0
	1	0	1
	2	0	1
	…	…	…
	n	0	1

Feature 2 M/A

	Feature 2	c_2t_1	c_2t_2
Branch 3	0	0	0
Branch 2	1	1	0
	2	1	1
	…	…	…
	n	1	1

Tree 1

Code	Leaf
00	0
01	1

Tree 2

Code	Leaf
00	0
01	1
1*	2

Figure 5: Ensemble trees (Tree 1&2) and Decision Tree (Tree 2) models' mapping workflow using EB solutions.

with a leaf node, and determines the range of values it includes. Finally, "Generates the tree table" (also named code or decision table) links mapping from leaf nodes to codes pointing to their areas.

Planter's DT$_{EB}$ uses default actions in tree tables to store the most common label, along with using the ternary match, longest prefix match (LPM), or range match in all tables. These reduce the number of table entries, saving significant memory resources (§5.2.2), which are applicable for all Planter-supported models.

3.3.2 *Random Forest (RF).*

Applying EB to RF (RF$_{EB}$), we encode trees in parallel and decide the label by a voting table. In the voting table, the RF$_{EB}$ model makes the decision based on DT$_{EB}$ votes. Figure 5 Tree 1&2 shows our RF workflow and a toy example of mapping a two-tree RF model to M/A format. For an RF model with n input features and m trees, the mapped model uses n feature tables and m tree tables. In each feature table, the codes $(c_it_1, c_it_2, \ldots, c_it_m | i \in n)$ for all trees are stored as actions. As noted in Section 3.3.1, n feature tables can share a stage. Similarly, m decision tables, one per tree, can share a stage and be looked up in parallel. The voting table requires a third logical stage. Compared with if/else logic [88] and depth-based solution [50], Planter's RF$_{EB}$ is scalable and saves stages (shown in §5.2.2).

3.3.3 *XGBoost (XGB).*

XGB is a different type of DT-based ensemble model. A primary difference between XGB and RF is that XGB accumulates probabilities from each tree's leaf nodes to make the final decision [12]. However, calculating probabilities within an M/A pipeline is non-trivial or costly in resources. Instead, our EB methodology enables Planter's XGB (XGB$_{EB}$) solution to encode the probabilities in each tree (tree table in Figure 5). To create the decision (codes-to-label) table, the XGB mapping workflow calculates the cumulative probabilities and expected output label for each code combination. XGB's probabilities addition and comparison operations are therefore replaced by simple codes-to-label lookups in the final decision process. Multiple discrete probabilities are mapped to the same code if they lead to the same label mapping, thus saving resources. XGB$_{EB}$ uses the same logical stages as RF$_{EB}$.

3.3.4 *Isolation Forest (IF).*

IF is an unsupervised ensemble model based on RF [53]. To make the decision, the total number of branches used in the forest decision is compared to an anomaly threshold, as shown in Equation 1, where x is one input, $h(x)$ is the path length, t is the total number of training inputs, γ is Euler's constant, and $E(h(x))$ is the average $h(x)$ of a collection of trees.

$$E(h(x)) \leq -(2(ln(t-1)+\gamma) - 2(t-1)/t)log_2 0.5 \tag{1}$$

To map Equation 1 into M/A table, our IF (IF$_{EB}$) solution uses a method similar to XGB$_{EB}$. The number of passed branches $h(x)$ is stored in tree (code) tables, and the mapping between number of passed branches per tree and final labels is done using code-to-label look-up tables. Replacing threshold operations with look-up tables saves stages in IF$_{EB}$.

3.3.5 *K-Nearest Neighbor (KNN).*

KNN splits the feature space according to its kth closest neighbors by distance. A tree data structure provides a feature space slicing approximation and labeling [26, 71]. Planter newly maps KNN to the data plane using EB methodology and tree data structure. Specifically, in a higher dimensional feature space, at each level of the tree, the input n dimensional feature space is divided and labeled into 2^n equal parts in the same order. KNN requires a code of $d \times n$ bits to represent each area when the maximum depth is d. The feature space is split continuously until the tree reaches the maximum depth or all vertices of the current unit belong to the same class. This tree-like splitting approach enabled storing all codes in ternary tables, thereby reducing memory consumption.

3.3.6 *k-means (KM).*

Encode-based KM (KM$_{EB}$) labels the input based on the distance between the data point and each centroid [23]. By using n-dimensional tree splitting [71], Planter realizes KM$_{EB}$ based on [26], which is similar to KNN.

Potential Extensions Planter currently supports six EB models. CatBoost [68], LightGBM [43], and AdaBoost [24] are examples of additional models that can potentially be implemented in Planter using encode-based mapping.

3.4 Lookup-Based Mapping

Many ML algorithms require mathematical operations on input features to decide a label, commonly too complex to implement in a hardware data plane. Lookup-based (LB) solutions use M/A tables to store intermediate results of these operations and thus enable realizing in-network ML. As shown in Figure 6, any ML algorithms with a *Decision Process* can use LB solutions. In LB solutions, feature tables store the mapping between each input feature value and intermediate results. These intermediate values are then used for the remaining basic operations, typically addition

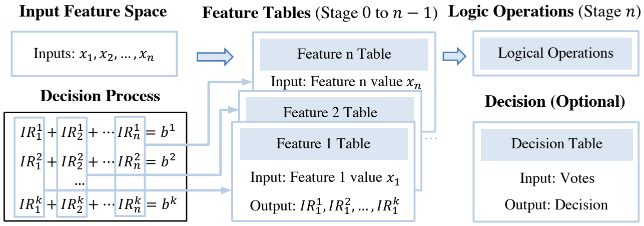

Input Feature Space
Feature Tables (Stage 0 to $n-1$)
Logic Operations (Stage n)
Inputs: $x_1, x_2, \dots, x_n$
Feature n Table
Input: Feature n value x_n
Feature 2 Table
Feature 1 Table
Input: Feature 1 value x_1
Output: $IR_1^1, IR_1^2, \dots, IR_1^k$
Logical Operations
Decision Process
$IR_1^1 + IR_2^1 + \cdots IR_n^1 = b^1$
$IR_1^2 + IR_2^2 + \cdots IR_n^2 = b^2$
...
$IR_1^k + IR_2^k + \cdots IR_n^k = b^k$
Decision (Optional)
Decision Table
Input: Votes
Output: Decision

Figure 6: Mapping methodology of LB solutions.

and comparison, as a final logic stage. Using mapped intermediate results allows for meeting high precision requirements (§5.2.1).

3.4.1 Naïve Bayes (NB). For every set of inputs, NB calculates the posterior probability of each class [20]. As shown in Equation 2, the Bayes model chooses the label that maximizes the posterior probability.

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{2}$$

One lookup-based NB mapping solution, proposed in IIsy [91], used features as the direct input to M/A tables, outputting the respective posterior probabilities of all their classes. This method only works when the range of input values is narrow. Otherwise, the table will be excessively large, as all intermediate results $P(x_i \mid y)$ are multiplied.

$$\hat{y} = \arg\max_{y} [map(log_2 P(y)) + \sum_{i=1}^{n} map(log_2 P(x_i \mid y))] \tag{3}$$

In Planter, we introduce a different lookup-based NB mapping, which uses logarithms to convert multiplication into addition, as shown in Equation 3. This mapping fits the standard LB solution, shown in Figure 6, using n feature tables (e.g., Input: x_i, Output $map(log_2 P(x_i \mid y_1)), map(log_2 P(x_i \mid y_2)), \dots, map(log_2 P(x_i \mid y_k)))$ for any k classes inference task. The logarithm operation both reduces operation complexity and lowers memory consumption. In the final logic, all intermediate results for each class are summed up and compared (replacing arg max) to get the output label. Based on this method, Planter supports variations (such as GaussianNB and BernoulliNB [66]), catering to diverse types of input variables.

3.4.2 Autoencoder (AE). AE's workflow is composed of an encoder and a decoder [52]. The forward path of the single-layer encoder network, interpreted as equation $X_{new} = XW + B$, has a similar format as the Equation in Figure 6's Decision Process (black box). Planter introduces a new AE mapping using LB. When X_{new} has k dimensions, W is a $n \times k$ weight matrix. The workflow uses n feature tables to store the intermediate results of all output feature dimensions $(x_i w_1^i, x_i w_2^i, \dots, x_i w_k^i)$ under the corresponding feature i. The final logic sums all intermediate results in each output dimension and the bias to get the value of new features. This mapping avoids multiplication operations in AE and can achieve a similar computational overhead as NB.

3.4.3 Principal Component Analysis (PCA). PCA, being an orthogonal linear transformation of the input features, is commonly used for dimensionality reduction [27]. The forward path of a trained PCA can be written as $X_{new} = (X - X_{means})Components$, where the input X is the array $[x_1, x_2, ..., x_n]$ with n input features, and $X_{means} = [x_{means}^1, x_{means}^2, ..., x_{means}^n]$ is the mean value of each feature. $Components$ is a transferring matrix with n rows and m columns. The output X_{new} is the array $[x_{new}^1, x_{new}^2, ..., x_{new}^m]$ with m output features. Based on this, Planter proposes LB PCA using n feature tables, and the intermediate result in feature table i is $IR_i^1 = (x_i - x_{means}^i)w_i^1$, $IR_i^2 = (x_i - x_{means}^i)w_i^2$, …, $IR_i^m = (x_i - x_{means}^i)w_i^m$. The final logic is similar to AE, which is the summation of intermediate results of each new dimension and the construct of new feature values. *PCA* can achieve similar memory and stage consumption as *NB*.

3.4.4 Support Vector Machine (SVM). SVM maps inputs to a space and uses hyperplanes to separate pairs of classes. For a k classification task, SVM requires $m = k(k-1)/2$ hyperplanes. Each hyperplane is equivalent to a vote. For the case of n input features, hyperplane m is $w_m^1 x_1 + w_m^2 x_2 + \dots w_m^n x_n + b_m = 0$, which similar to the Decision Process in Figure 6. The final vote can be determined by counting the votes from all hyperplanes and using logic or a decision table [17]. LB based SVM is supported in Planter based on the proposal in [101]. To achieve scalability, n feature tables will be used to store the intermediate results from all hyperplanes (e.g., Input: x_i, Output $w_1^i x_i$, $w_2^i x_i$, $\dots w_m^i x_i$). All feature tables belong to a single logical stage and use the addition operation for hyperplanes (initialized by bias b_i). This method saves memory and stages compared with other approaches [91].

3.4.5 k-means (KM). The KM workflow labels inputs according to their distance to the trained k centroid [23], as described in Equation 4. A simple LB KM (KM$_{LB}$) mapping in Figure 6, as in IIsy [91, 101], uses n feature tables to store intermediate results in parallel.

$$D_i = \sqrt{(x_1 - c_1^i)^2 + (x_2 - c_2^i)^2 + \dots (x_n - c_n^i)^2} \tag{4}$$

In Planter, using a square root operation in distance calculation can be avoided when the value under the square root is larger than zero. KM$_{LB}$ solution uses $map(.)$ operation and constructs feature tables with input x_i, output $map(x_i - c_i^1), map(x_i - c_i^2)$, $\dots, map(x_i - c_i^k)$. The $map(.)$ function maps all input value to a domain $\{1 : 2^{n_{bits}}/n\}$, where n_{bits} is the width of each action data. With this $map(.)$ operation, the square root operation is not required, lowering computational complexity.

Potential Extensions Planter currently supports five LB models. Lasso [25, 83], Linear Regression [60], and Polynomial regression [78] are examples of additional models that can potentially be implemented in Planter using lookup-based mapping, using a similar methodology.

4 PLANTER FRAMEWORK

Section 3 presented new and more efficient in-network ML mapping methodologies. Using the Planter framework, those can be rapidly prototyped on different targets. This section addresses four aspects in the design of Planter, as an automated and one-click framework: 1. What should be the functionality supported by the framework, and what should be the respective framework architecture? (§4.1)

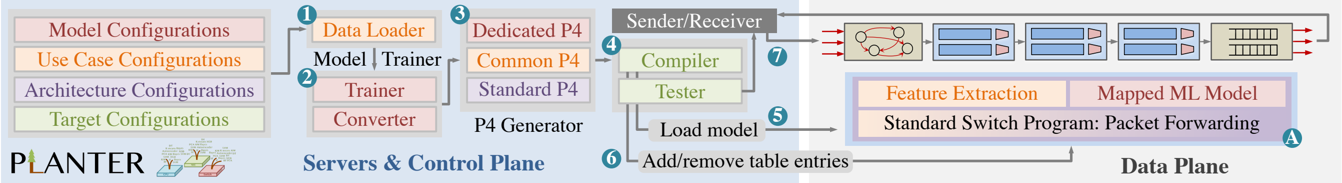


Figure 7: The Planter framework back-end components and workflow steps (❶ to ❼).

2. How to generate efficient data plane designs for different deployments? (§4.2) 3. How to create a modular framework that is extendable to different models, targets, architectures, and use cases? (§4.3) 4. How to provide ease of use, handle failures and tune hyperparameters? (§4.4). By addressing these aspects, we provide a framework that goes beyond the state-of-the-art and enables wide adoption of in-network ML.

4.1 Functionality, Workflow, and Main Components

The goal of Planter is to provide as simple as possible in-network ML inference development environment for users. For example, model developers should be able to focus on model mapping without worrying about use cases, while use case practitioners can develop just the use cases and pick from a given set of models and targets. To achieve this goal, in-Planter functions are strictly partitioned according to their type and are connected by using standard interfaces. In turn, an in-network ML application is divided into four types of elements: ML model-related, use case-related, architecture-related, and target (test)-related.

The main functions of Planter's back-end workflow are shown in Figure 7. *Use case related functions (amber)* include a Data Loader ❶ for training & testing purposes and a Common P4 ❸ to generate use case-specific P4 code (e.g., code for feature extraction from data). *Model related functions (red)* are formed by a Model Trainer & Converter ❷ for training and converting a model to M/A format and a Dedicated P4 ❸ for generating model related P4 codes. *Architecture related functions (purple)* includes a Standard P4 ❸ generator that generates architecture-related P4 code and combines the outputs from Common and Dedicated P4 to generate the complete P4 program. *Target related functions (green)* include a Compiler ❹ that compiles and runs the generated in-network ML inference program and a Tester ❹ for validating the functionality of the program.

The workflow of Planter's back-end, shown in Figure 7, has seven steps. In the first two steps, Planter loads a dataset ❶ and trains it ❷. The model is mapped to P4 ❸ using the selected architecture and target. Generated P4 code is compiled ❹ and loaded to the target's data plane ❺. In step ❻, table entries and registers are loaded through the control plane. Last, in step ❼, an auto-generated functionality test is run on the target. The generated data plane, shown in Figure 7 Ⓐ, has three components: standard switching functionality, ML feature extraction, and ML inference. The ML feature extraction and inference are in parallel to the standard functionality, while P4 parser operation is merged.

```
A  #include <core.p4>
A  #include <tna.p4>
U  const bit<16> const1 = 0x01;
M  const bit<16> const2 = 0x02;
A  struct header_t{ // Header
U      header_U_h    header_U;
M      header_M_h    header_M;
A      }
A  struct metadata_t{ // Metadata
U      bit<10> meta_data_U;
M      bit<10> meta_data_M;
A      }
A  parser SwitchParser(...){ // Parser
U      state parse_header_U{...}
M      state parse_header_M{...}
A      }
A  control SwitchIngress(...){ // Ingress
U      ... // Use case related Tables, Actions,
↪   Registers ...
M      ... // ML model related ...
A      apply{
U          ... // ML model related logic
M          ... // Use case related logic
A         }}
A  control SwitchEgress(...){ // Egress
A  U M...}
A  Switch(...) main; // Main
```

Listing 1: Sample in-network ML P4 code. A : architecture-related code, U : use case-related code, M : ML model code.

4.2 Code Generator

The data plane P4 code is determined by the ML model, P4 architecture, and selected use case (three out of four function groups in § 4.1). As shown in Listing 1, architecture-related code (A) forms the skeleton of the program, and use case (U) and model-related (M) code snippets are added into the skeleton. To simplify the coding experience, Planter provides commonly-used architecture templates as skeletons in the Standard P4 file ❸. P4's sequential execution can be a problem when assembling the code snippets. Planter ensures the correct assembly order by utilizing the Standard P4 to coordinate the Common & Dedicated P4 parts ❸. This is achieved by alternately calling specific functions that generate model and use case related code from Common & Dedicated P4 at each designated place (such as *ingress apply* block) to accurately merge the model and use case-related codes. This approach enables Planter's extensibility, fitting different models and use cases into the Standard P4 generator.

4.3 Modular Framework Design

Planter proposes a modular framework to enhance extensibility. Modules are independent and can be flexibly and easily replaced through configuration. The framework supports stand-alone modules for ML models, architectures, targets, and use cases. Additionally, Planter provides a set of common functions for all modules, such as exact-to-LPM table conversion. Planter supports generating a dependency graph, showing all the modules used, and their dependencies. This simplifies debugging and informs modules' swapping.

Python	Architecture	Target	Model	Use Case	Data	Function
Min LOC	123	277	174	65	16	6
Avg LOC	137	931	555	211	93	151
Max LOC	145	2115	835	860	420	1877
Modules	6	9	64	14	30	20

Table 3: The number of LOC and modules (including module variations) in Planter modular design.

To illustrate the modularity of Planter, Table 3 presents the average and maximum lines of code (LOC) of different modules in Planter. ML models require an average of 555 LOC, and at most 835 LOC. Supporting a new P4 architecture requires less than 150 LOC, and supporting a target requires about a thousand LOC. This lightweight implementation of architectures, targets, and models is a key advantage of Planter. Shared framework functionality ("Function") requires 377 LOC in total.

4.4 Planter Front-end

Although the Planter back-end (§4.1-4.3) enables the realization of ML models within the data plane, fitting the models on resource-constrained (e.g., commodity) hardware with ML performance goals needs to be addressed. The key challenge is hyperparameters selection, combining the desired inference accuracy and minimum resource consumption. To address these challenges, Planter provides a front-end optimizer that automates hyperparameter tuning, and fixes compilation failures.

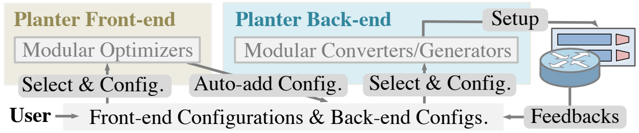


Figure 8: The combined Planter framework.

The front-end applies modular optimization models, such as Bayesian Optimization, to optimize parameters based on a predefined objective function. The optimization objective can be expressed as metrics such as Accuracy (acc), F1 Score ($F1$), or the difference between the target accuracy ($target$) and the model accuracy $n^{-|acc-target|}$ ($n > 1$). As illustrated in Figure 8, Planter requires users' input for front-end configurations, including datasets and model constraints. Based on these configurations and the selected objective function, the front-end identifies optimal hyperparameter settings. Subsequently, as shown in Figure 8, the front-end automatically generates back-end configurations, utilizes the back-end (Figure 7) to produce data plane code & setup, and performs compilation and testing to verify the settings. The front-end receives the test results and triggers a subsequent training round if the outcomes are not satisfactory. For example, in case of a compilation failure, the front-end will regenerate a more moderately-sized model. The Planter front-end is also part of the modular design in Section 4.3. While the back-end supports developers with customizing model's details, the front-end is designed to help users with limited ML experience conduct intent-based development.

4.5 Implementation

The Planter framework is implemented in 57k LOC in Python, and is available at Planter's GitHub repository [104]. The framework trains a model using Python-based learning frameworks including PyTorch and Scikit-learn (Sklearn). Training parameters can be set via the Planter front-end or through parameter tuning tools [37, 82]. A trained model is mapped into M/A format, and the framework saves table entries and generated weights (for NN) into `JSON/txt` (target dependent) files. Data plane P4 code is then automatically generated. Planter further generates `Bash` scripts to interact with the target for model deployment and verification. The provided control plane support is target-dependent, e.g., loading tables using P4Runtime. Currently supported targets include Intel Tofino and Tofino2 (switch-ASIC), AMD Alveo U280 (FPGA) over OpenNIC [89], P4Pi-BMv2 and P4Pi-T4P4S on Raspberry Pi [47], BMv2 on NVIDIA BlueField2 [6], and BMv2. Pipeline architectures include v1model [67], Intel TNA [39], PSA [62], AMD XSA [90] and NVIDIA spectrum. Supported ML modules are listed in Table 1 as well as their variations in Table 2. The Planter supported use cases can be found in Section 5.1 and reference applications are detailed in Section 6.

5 EVALUATION

The evaluation of Planter focuses on the following aspects: general ML performance (§5.2), scenario-specific ML performance (§5.3), scalability performance (§5.4), general system performance (§5.5), target-specific system performance (§5.6), and framework performance (§5.7). This section also evaluates Planter's new and improved algorithm mappings, as in Table 2.

5.1 Methodology and Testbed Setup

Workloads: Planter has already been applied in several works for various scenarios [11, 32, 36, 92, 93, 99–102]. Example use cases and datasets include attack detection (using AWID3 [10], CICIDS 2017 [74], KDD99 [80], and UNSW-NB15 [57]), finance (NASDAQ TotalView-ITCH [58], Jane Street Market Prediction [30]), QoE (Requet [31]) and flowers classification (Iris [22]). For brevity, this section focuses on the results for attack detection (throughput & latency-sensitive, using CICIDS and UNSW-NB15) and high frequency trading (latency-sensitive, using NASDAQ TotalView-ITCH).

Packet-level attack detection uses 5 features, commonly used for traffic classification [40, 69]: Source IP (first 8 bits), Destination IP (first 8 bits), Source Port, Destination Port, and protocol and classifies traffic as either normal or malicious. The first octet of the IP address is a coarse indicator of source/destination network. In the Jane Street Market Prediction dataset, five features (42, 43, 120, 124 and 126) from 130 anonymized real stock market data are

		Accuracy										Resource Performance										
		CICIDS				UNSW						UNSW										
		Switch (M)		Sklearn (M)		Switch (M)		Sklearn (M)		Server (H)		ACC (Switch)		Memory (%)			Latency (Relative)			Stages (Tofino)		
Work	Model	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	S	L	S	M	L	S	M	L	S	M	L
SwitchTree [50]	DT_{DM}	99.92	99.92	99.92	99.92	99.40	**94.53**	99.40	**94.53**	99.40	94.31	99.34	99.41	1.46	1.72	1.98	81.16	88.36	88.36	11	13†	15†
pForest [7]	RF_{DM}	99.80	99.79	99.80	99.79	99.38	94.44	99.38	94.44	99.42	94.51	99.25	99.39	7.71	14.01	NF	88.36	89.04	NF	11†	14†	NF
IIsy [91]	SVM	59.24	37.20	95.04	94.94	97.31	49.32	99.23	93.51	99.23	93.51	97.31	99.23	2.81	3.23	4.12	26.37	35.27	35.30	9	9	9
N3IC [76]	NN[‡]	92.09	92.00	99.96	99.96	98.33	85.68	99.25	93.67	99.25	93.68	98.33	97.50	NF	NF	NF	NF	NF	NF	NF	NF	NF
IIsy [91]	KM_{LB}	58.40	56.80	58.40	56.80	71.28	41.88	71.28	41.88	71.28	41.88	71.55	71.28	3.13	3.96	5.78	21.58	21.58	21.58	7	7	7
Clustreams [26]	KM_{EB}	56.92	55.75	58.40	56.80	72.69	42.37	71.28	41.88	71.28	41.88	77.21	71.30	0.40	3.16	NF	**19.52**	**19.52**	NF	2	2	NF
Planter	DT_{EB}	99.92	99.92	99.92	99.92	99.40	**94.53**	99.40	**94.53**	99.40	94.31	99.34	99.41	1.18	**1.34**	**1.34**	26.37	26.37	26.37	2	2	**2**
Planter	RF_{EB}	99.80	99.79	99.80	99.79	99.37	94.41	99.38	94.44	99.42	94.51	99.25	99.39	1.81	2.59	3.94	39.04	39.40	45.89	3	4	4
Planter	XGB	**99.98**	**99.98**	**99.98**	**99.98**	**99.42**	**94.53**	**99.42**	**94.53**	**99.43**	**94.59**	**99.40**	**99.45**	1.70	6.65	NF	33.22	45.78	NF	3	5	NF
Planter	NB	98.99	98.95	98.99	98.96	99.25	93.68	99.25	93.68	99.25	93.68	99.25	99.25	3.28	4.22	6.20	28.77	28.77	28.77	8	8	8
Planter	IF	44.89	35.35	37.90	31.08	84.86	58.90	63.83	45.07	86.33	55.05	81.74	NF	2.01	9.01	NF	36.30	43.33	NF	5	5	NF
Planter	KNN	69.33	60.63	99.38	99.36	87.51	31.55	99.30	93.17	99.30	93.17	78.24	92.73	**0.23**	1.89	21.01	20.74	20.74	22.22	**1**	**1**	5
Planter	PCA*	76.12	74.92	76.19	75.00	97.45	65.42	97.89	67.73	97.89	67.73	97.29	97.47	5.78	5.78	5.78	20.89	20.89	**20.89**	6	6	6
Planter	AE*	99.92	99.92	99.92	99.92	99.24	93.55	99.24	93.55	99.28	93.53	99.23	99.28	5.89	5.89	5.89	21.58	21.58	21.58	7	7	7

[‡] NN is trained with PyTorch instead of Sklearn. **Bold** and underline indicate the top and second top performance among all models, respectively. * Results of PCA and AE are the accuracy of (S)mall DT using new features after dimensionality reduction. KM, PCA, and AE are unsupervised learning while others are supervised learning.

Table 4: Accuracy (ACC), resources and latency relative to switch.p4. Using (S)mall, (M)edium, (L)arge and (H)uge models. Some models are not feasible (NF) on Tofino but are feasible (†) on Tofino2.

used to predict buy or sell for each trading opportunity [30]. Three packet-level fields (order side, size, and price) are used as features with the NASDAQ dataset, to predict stock price movement: up, down, or stationary. Feature selection is further explored in §5.3. Results for other use cases can be found in [102].

Testbed setup: The testbed uses two servers for traffic generation and monitoring (ESC4000A-E10, AMD EPYC 7302P CPUs, 256GB RAM, Ubuntu 20.04LTS, ConnectX-5 NICs). PTP with timestamping in the NICs is used for latency measurements. The evaluated platforms are a 1) Tofino switch (APS-Networks BF6064X, SDE 9.6.0) using a snake configuration for throughput tests. 2) AMD Alveo U280 FPGA 3) NVIDIA BlueField-2 DPU. In addition, 4) P4Pi [47] running on Raspberry Pi 4 Model B with 8GB RAM is evaluated twice: using v1model over BMv2 and using T4P4S [85]. The P4Pi testbed is connected to a server with an Intel Xeon W-2133 CPU and 64 GB RAM. Planter-generated in-network ML models were also deployed on Dell IoT Gateway [19] and have been evaluated in [92].

Parameter settings: Mapped in-network ML models are explored using four different model sizes: small (S), medium (M), large (L), and huge (H), e.g., 6/9/12/200 trees with depth of 4/5/6/30, correspondingly. Detailed setups of model parameters are provided in the repository [104]. The model size refers to the converted data plane model size, which is a function of both training and conversion parameters. Small to large in-network ML models are expected to fit on the target data plane. Huge models represent the maximum inference potential of a model (per dataset) running on a server.

Evaluation metrics: The following metrics are used in the evaluation:

1. ML performance: *Accuracy* and *F1 score* are used to evaluate ML inference performance.
2. Scalability performance: *Memory utilization, Table entries*, and *Number of stages* are used to evaluate scalability.
3. System performance: *Throughput* and *Latency* are used to evaluate the system performance of mapped models.
4. Framework performance: *Model training time* and *trained model conversion time* are used to assess Planter's run time performance.

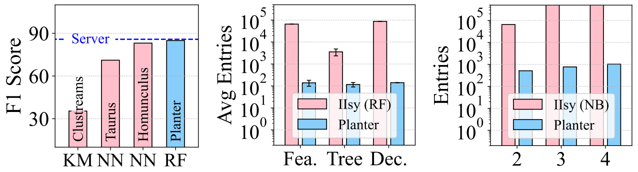
On Tofino, following NDA, we report the latency relative to Intel's *switch.p*4 reference program (L2/L3 switch).

5.2 General ML Performance

5.2.1 ML Inference Accuracy. The inference performance evaluation explores if the mapped in-network ML models have similar inference accuracy as running the same inference task on a server, and how the size of the model affects the accuracy. It is not suggested to compare *between* ML models in Table 4 as the competence preference of models varies among different use cases.

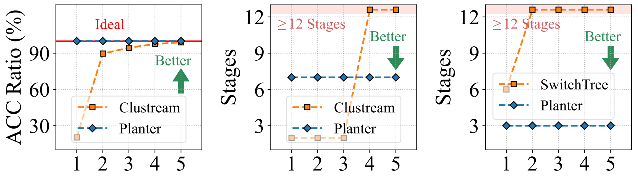
The results are presented in Table 4. The upper part shows previously proposed mappings [7, 26, 76, 91, 101, 103], and the lower part shows Planter optimized mappings. As the Accuracy column (left side of the table) shows, for the same model size, all the models have a similar accuracy performance on the programmable switch as on Sklearn or a baseline server, verifying Planter's mapping barely causes accuracy loss. In the tested use case, it has been demonstrated that smaller models (e.g. 6 trees for RF) can perform just as well as larger models (e.g. 200 trees for RF) on a server, which indicates that the size of Planter's model is applicable with satisfactory accuracy. The accuracy is checked using 10-fold cross-validation, with a standard deviation of less than 0.05% for top performing models, indicating its statistical significance.

The resource performance column (right side of the table) compares the resource performance for different *sizes* of models. As model size increases, some models achieve slightly higher accuracy with more switch resources required. All small and medium-sized models proposed/upgraded by Planter are feasible on commodity hardware with less than 8 stages, 9.1% memory, and 45.78% of relative latency. The optimized mapping methods of Planter can achieve comparable accuracy with existing mapping solutions, but with significantly lower resource consumption. For example, large DT model in Planter can reach the same accuracy as large SwitchTree,



(a) Models. (b) Table Types. (c) Num. Features.

Figure 9: Comparison of accuracy and table entries with State-of-the-Art. UNSW dataset is used.



(a) Model Depth. (b) Model Depth. (c) Model Depth.

Figure 10: Comparison of accuracy and stages used with State-of-the-Art. UNSW dataset is used.

while reducing memory by 30%, latency by 70%, and stages by 87%. Resource consumption varies between Planter targets. For example, deploying a DT model on U280 FPGA has resource usage as low as 15% LUTs and 11% registers.

Planter was integrated with *switch.p4*, an Intel reference L2/L3 switch. The *switch.p4* functionality coexists with Planter's new or upgraded models with no or minimal cost in stages and latency, but with higher resource utilization.

5.2.2 Comparison with State-of-the-Art. We compare Planter with IIsy's DT, SVM, NB and KM [91, 101], SwitchTree/pForest DT and RF [7, 50], Clustreams KM [26], N3IC's NN [76], and Homunculus/Taurus's NN [81, 82] in terms of inference performance and scalability. The Planter-based models are as in Table 4. Figures 9 and 10 demonstrate that Planter outperforms prior works in both accuracy and resource utilization. This is due to the optimized mapping methods introduced in Planter (§3).

Figure 9 (a) compares the accuracy performance of Planter on Tofino and existing solutions where Clustreams is deployed on Tofino, and Taurus and Homunculous are implemented on Taurus back-end. Planter achieves higher accuracy (84.88%) than Clustreams (35.4%), Taurus (71.1%) and Homunculous (83.1%), and similar to server accuracy. Compared with IIsy on Tofino, Planter achieves same or higher accuracy for lower memory consumption as shown in Figure 9 (b) & (c). Planter's evaluated models (RF_{EB} and NB_{LB} respectively) reduce the table entries in both cases, and prevents NB_{LB} table entry explosion by turning multiply operations to additions.

Figures 10 (a) - (c) compare Planter with Clustreams using KM (*k*-means) and SwitchTree using DT (Decision Tree). Clustreams has limited scaling capability, as table entries may explode when seeking better accuracy, compared with Planter's accurate and more resource efficient solution for larger models. Similarly, Planter maintains a constant number of stages for tree models, compared with the increasing number of stages in SwitchTree. Planter demonstrates better scalability in commodity hardware by parallelizing the inference processing within a pipeline to avoid exceeding the stage limitation as model size scales up.

		Iris				CICIDS (flow level)			
		Switch		Server		Switch		Server	
Work	Model	ACC	F1	ACC	F1	ACC	F1	ACC	F1
SwitchTree [50]	DT_{DM}	95.56	95.56	95.56	95.56	96.56	91.68	96.56	91.68
pForest [7]	RF_{DM}	95.56	95.56	95.56	95.56	96.16	87.89	96.16	87.89
IIsy [91]	SVM	97.78	97.81	97.78	97.81	96.80	91.41	96.80	91.41
N3IC [76]	NN	93.33	93.42	95.56	95.56	58.38	51.05	97.80	94.28
IIsy [91]	KM_{LB}	88.89	88.19	88.89	88.19	52.44	38.72	52.44	38.72
Clustreams [26]	KM_{EB}	77.78	75.93	88.89	88.19	53.42	39.12	52.44	38.72
Planter	DT_{EB}	95.56	95.56	95.56	95.56	96.56	91.68	96.56	91.68
Planter	RF_{EB}	95.56	95.56	95.56	95.56	96.20	87.94	96.46	89.01
Planter	XGB	97.78	97.81	97.78	97.81	97.06	91.23	97.12	92.83
Planter	NB	95.56	95.56	95.56	95.56	81.70	68.01	81.70	68.01
Planter	IF	15.56	18.18	11.11	8.48	58.42	43.63	58.42	43.63
Planter	KNN	80.0	66.43	100.0	100.0	89.48	47.41	96.86	91.21
Planter	PCA	92.86	92.67	92.86	92.67	91.52	74.85	91.67	75.41
Planter	AE	92.86	92.67	92.86	92.67	96.47	91.14	96.47	91.14

Table 5: Evaluation results of Iris and CICIDS (with flow-level features) datasets. Iris uses (M)edium size model, CICIDS uses on (L)arge size model for SVM, KM, KNN, and (S)mall for tree models, due to the model's increased complexity.

5.3 ML Performance in Different Scenarios

As different use cases may require different *types* of input features, we evaluate Planter's support for flow-level, packet-level and time-series input features.

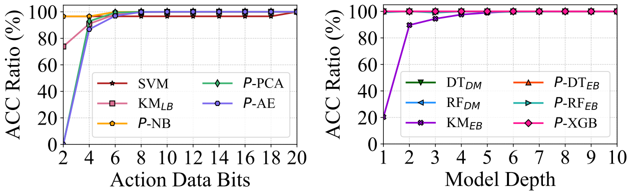
5.3.1 Flow Level. The packet-level classification of CICIDS used in Table 4 is extended to test in-network ML using flow-level classification on the same dataset. The features used include Flow Duration, mean flow Inter-Arrival Time (IAT), maximum flow IAT, minimum flow IAT, and minimum packet length (stateful information is stored in registers [9]). These five features are commonly used in anomaly detection and traffic classification tasks [40, 69], and can be extracted within the data plane. As shown in Table 5, accuracy results closely match the packet-level classification accuracy listed in Table 4. While the use of flow-level features results in a slight decrease in accuracy, it offers better generalizability. It was indicated [106] that best classification performance can be achieved when both packet and flow-level features are used.

5.3.2 Packet Level. Beyond network-centric datasets, we also evaluate Planter using the Iris dataset [22], a classic ML multiclass dataset, primarily used for analyzing four feature measurements of Iris flowers to predict their species. The Iris dataset is widely used in ML and statistics as a benchmark due to its diversity, repeatability, and broad applicability. Here it is used to evaluate Planter in multiclass tasks and to assess performance on general datasets. The experimental results are shown in Table 5, with most supported ML algorithms achieving good inference performance. A few algorithms experienced accuracy loss, primarily encode-based KM and KNN. The IF model performed poorly on this dataset because it is an outlier detection model mainly applied to imbalanced datasets.

5.3.3 Time Series. NASDAQ's Historical TotalView-ITCH sample data feeds are used for time series analysis [1]. To forecast stock future price movement using market microstructure signals, a data

Time Series Analysis - NASDAQ Stock EQIX (time series)																	
		Tofino				Tofino 2				BMv2				Server			
Work	Model	PRE	REC	F1	ACC	PRE	REC	F1	ACC	PRE	REC	F1	ACC	PRE	REC	F1	ACC
Clustreams [26]	KM	14.97	33.33	20.66	44.90	14.97	33.33	20.66	44.90	24.80	33.64	25.57	43.74	24.80	33.64	25.57	43.74
Planter	DT_{EB}	53.84	51.81	50.53	51.12	58.66	53.55	52.91	54.43	58.17	60.60	57.14	57.43	58.41	60.33	58.25	57.99
Planter	RF_{EB}	57.45	56.28	56.16	58.28	58.68	57.03	56.95	59.26	57.45	59.74	56.83	56.18	62.80	61.89	60.99	64.15
Planter	XGB	40.07	42.83	40.69	48.63	41.67	47.56	42.87	54.22	56.09	58.72	56.55	56.99	56.61	58.56	56.90	59.05
Planter	KNN	44.86	44.91	39.61	55.08	42.86	44.77	39.55	54.62	54.50	47.47	41.22	41.39	52.95	54.21	52.93	54.39

Table 6: ML prediction performance (%) with an example NASDAQ stock - EQIX. We run limited-size models on BMv2, Tofino and Tofino 2 (emulated). The benchmark runs on a server using Sklearn and unlimited-size models. PRE, REC, F1, and ACC correspond to precision, recall, f1-score, and accuracy, respectively. Note that the primary difference in results among targets mainly stems from the size of the models and the number of features utilized, which does not impact the mapping.



(a) ACC vs Action Bits.

(b) ACC vs Model Depth.

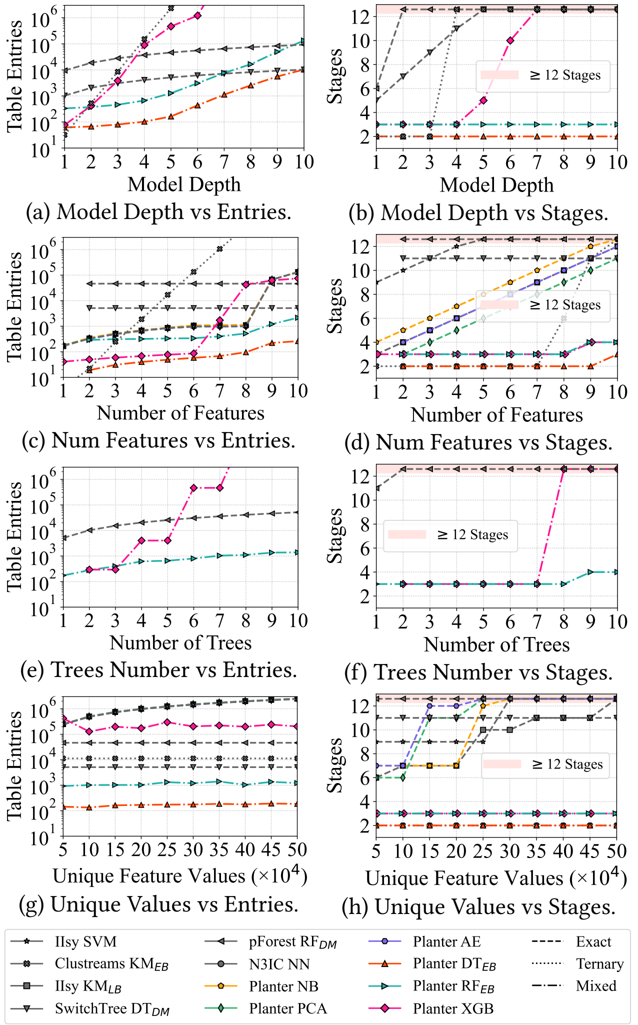
Figure 11: The ratio of accuracy (ACC), data plane model relative to the server. KM_{LB} & SVM refers to IIsy [91, 101]; DT_{DM} & RF_{DM} refers to SwitchTree [50] & pForest [7]; KM_{EB} refers to Clustreams [26]. *P*- refers to Planter proposed or upgraded model.

structure called limit order book (LOB) [8] is constructed, using the algorithm described in [36]. The LOB is updated within the data plane in real-time, from unmatched limit orders set at specific prices (stateful information is stored in registers [36]). LOBs offer a structured view of market orders, providing insights into supply and demand dynamics at different price levels over time, which is more informative for ML prediction than raw market feeds. As shown in Table 6, ML performance has a small accuracy loss across commonly used in-network models. There is a trend where more switch resources (i.e., BMv2>Tofino 2>Tofino) enable higher ML performance. The server baseline's performance is not very high, matching common results in the field [97], primarily because the financial market is influenced by numerous factors, making it challenging to predict [44]. Moreover, time-series data feeds from financial markets frequently encompass a considerable level of noise, impacting the performance of ML models.

5.4 Scalability Performance

5.4.1 Scalability and Relative Accuracy. We examine whether Planter can perform effective algorithm mapping with various hyperparameters and adapt to different resource constraints. Specifically, we study the effect of action data bits (action field's width, which can control quantization accuracy) and model depth on models' relative accuracy, comparing the switch's output with Sklearn's result on a server.

Figure 11 shows the switch accuracy relative to server accuracy. For LB (Lookup-based) models in Figure 11 (a), the relative accuracy increases as the number of action data bits increases (more accurate intermediate results are stored). Among solutions, Planter's solution can have a negligible accuracy loss with just 4 action data bits requirements, which can save 40% memory in practice (more action data bits means higher memory consumption). In Figure 11 (b),



(a) Model Depth vs Entries.

(b) Model Depth vs Stages.

(c) Num Features vs Entries.

(d) Num Features vs Stages.

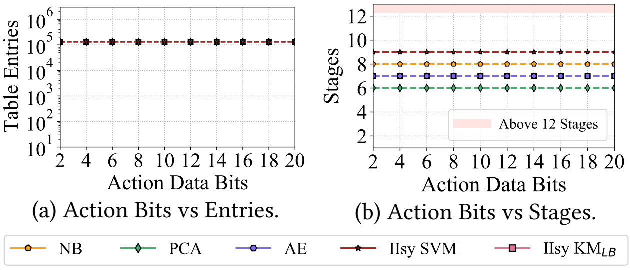
(e) Trees Number vs Entries.

(f) Trees Number vs Stages.

(g) Unique Values vs Entries.

(h) Unique Values vs Stages.

Figure 12: Memory and stage scaling with hyperparameters and feature properties (UNSW dataset). Flat or gentle curves indicate better resource scalability with model parameters. Model name references are same as Figure 11.



(a) Action Bits vs Entries.

(b) Action Bits vs Stages.

Figure 13: Action bits effect on LB solutions scalability based on the UNSW dataset.

compared to server-based models with the same model size, the accuracy of Planter's models remains stable as model depth changes. It indicates that Planter's solutions can consistently achieve high mapping accuracy across a range of model sizes, effectively adapting to different resource constraints. Compared with KM_{EB} (Clustreams [26]) and DT/RF_{DM} (SwitchTree/pForest [7, 50]) that can easily lead to table entry or stage explosion when the model depth grows, Planter's solutions improve the mapping precision when under identical memory consumption.

5.4.2 Resources Scalability. Both model/convert hyperparameters (model depth) and use case inputs (number of features) influence the applicability and scalability of Planter's in-network ML models. The resource scalability of each model is evaluated in two dimensions: the number of table entries and the number of pipeline stages. Table entries indicate the potential memory requirement from the switch, and the number of stages indicates remaining M/A stages for model growth and non-parallel functionality.

Figures 12 (a) & (b) show that as a model's depth increases, more table entries are required in all EB (encode-based) models and direct-mapping tree-based models. Among them, DM (direct mapping) solutions have a comparatively slower increment. EB tree models are more stable in terms of stage consumption. Figures 12 (c) & (d) show that except for DM tree-based models, models consume more table entries as the number of features increases. In terms of stage consumption, only LB-based models have a strong correlation to the number of features. Figures 12 (e) & (f) show that as the number of trees increases, EB tree models require 8 stages less than DM tree models, unless the number of table entries is excessive. In Figures 12 (g) & (h), the feature range, which is the number of unique feature values per feature, only influences LB models' stage and memory consumption. Figures 13 (a) & (b) show that the number of action data bits does not influence the required number of table entries and the required number of stages. Note that the evaluated models are those where action bits are a parameter.

Some insights based on this evaluation are: 1. The scalability of EB mappings is mostly affected by model's parameters (e.g., number of trees/model depth) and less by the use case (e.g., range/number of features). 2. LB resources change with use case properties (e.g., range/number of features), and typically not with model parameters. 3. DM scalability is model dependent. DM is usually bounded by stages, and less by memory. Planter's widely supported model types and characters make it highly adaptable to a wide range of use cases.

5.5 General System Performance

In terms of system performance, the throughput evaluation of different models is shown under the attack detection use case (UNSW), which is a volumetric use case. Latency is shown using price movement prediction use case (Jane Street Market Prediction), which is latency sensitive.

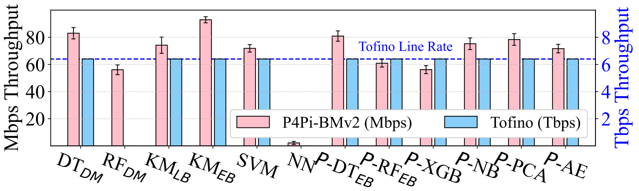


Figure 14: Throughput of ML algorithms for attack detection on Tofino (in Tbps) and P4Pi (in Mbps). DT_{DM} and RF_{DM} refers to SwitchTree [50] & pForest [7]; KM_{LB} & SVM refers to IIsy [91]; KM_{EB} refers to Clustreams [26], NN refers to N3IC [76].

5.5.1 Throughput. Throughput tests record the throughput of each in-network ML algorithm on a Tofino switch and P4Pi, as shown in Figure 14. The baseline throughput of basic forwarding is 6.4Tbps on Tofino and 94Mbps on P4Pi. On a Tofino switch, full 6.4Tbps is achieved for all feasible models (Table 4). On P4Pi, which essentially runs a software switch on a CPU, the results vary between models. Seven of the models achieve more than 80% of the baseline throughput. Ensemble models (RF_{EB}, RF_{DM}, and XGB) and NN have degraded throughput on P4Pi, due to their increased use of resources. The throughput of in-network ML on other targets can be found in Section 5.6.

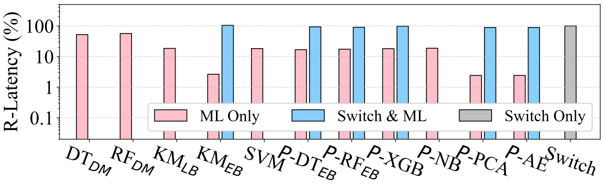
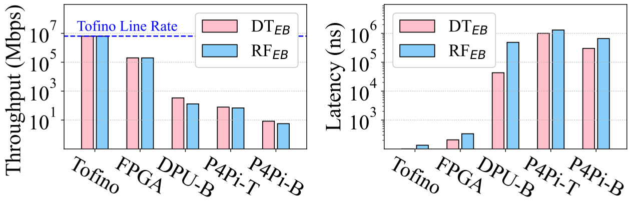


Figure 15: The relative latency (R-Latency) on Tofino in the financial prediction use case, measured for standalone ML, ML combined with *switch.p4*, and standalone *switch.p4*.

5.5.2 Latency. Latency tests are conducted with Jane Street Market Prediction dataset where Planter's models can achieve sub-microsecond latency. In compliance with the NDA, we report the relative latency in Figure 15. The baseline is the latency of *switch.p4*. When only the ML models are deployed, without additional functions, the latency in most models is less than 22% of *switch.p4*. When the ML models are combined with switch.p4, there is an overhead of less than 4.7% for all feasible algorithms. Compared with previous works, Planter's mappings require less logic for similar models and are more compatible with other switch functions in resource-constrained targets. The latency of in-network ML on other targets can be found in Section 5.6.



(a) Targets vs Throughput. (b) Targets vs Latency.

Figure 16: Throughput and latency of DT$_{EB}$ and RF$_{EB}$ on different target devices. -B refers to BMv2 and -T refers to T4P4S.

5.6 System Performance on Different Targets

The system performance of two sample in-network ML algorithms, DT$_{EB}$ and RF$_{EB}$, is evaluated on different targets. As illustrated in Figure 16, hardware targets, such as Tofino and FPGA, achieve line rate throughput (6.4Tbps and 100Gbps, correspondingly). In contrast, software switches (BMv2 or T4P4S), running on the ARM cores of P4Pi and DPU, reach a throughput in the range of tens to hundreds of Mbps. Similarly, hardware targets achieve microsecond-scale latency, whereas software targets achieve sub-millisecond latency. Furthermore, on software targets the complexity of the model impacts system performance; the more complex RF algorithm has lower throughput and higher latency on these targets. The model complexity does not have a notable impact on the performance of hardware targets. To provide more information about each of the targets:

5.6.1 FPGA. Planter's FPGA support is evaluated using AMD Alveo U280. Vitis Networking P4 is used to compile the generated P4 code to an IP block with standard AXI interfaces for the OpenNIC shell. Taking EB (encode-based) DT and RF as examples, the baseline latency through a forwarding-only program is about a microsecond. The latency added by a DT model is 170 nanoseconds, and RF latency is approximately 320 nanoseconds, aligned with compiler prediction. Both In-network ML models achieve 100Gbps, full line rate.

5.6.2 DPU. Planter currently supports in-network ML on NVIDIA BlueField-2 DPU using BMv2 running on its ARM cores. P4C is used to compile the P4 code with v1model architecture to a BMv2 software switch. Under this setup, compared to the baseline of simple forwarding, DT$_{EB}$ introduces an additional latency of approximately 43 µs, and RF$_{EB}$'s latency is approximately 487 µs. In terms of throughput, using BMv2 performance configuration [16], DT can reach about 360 Mbps and RF has approximately 131 Mbps.[1]

5.6.3 P4Pi-T4P4S. T4P4S [85] is an open-source compiler that generates a target-agnostic software switch using Data Plane Development Kit (DPDK). In this scenario, T4P4S is running on top of P4Pi [47]. Taking the same encode-based DT and RF models, the baseline latency is around 1 ms with basic forwarding functions. When DT_{EB} and RF_{EB} are enabled, the latency increases to 2 ms and 2.3 ms, correspondingly. The baseline switch throughput is 100 Mbps. When DT_{EB} and RF_{EB} are deployed, the throughput decreases to 78.7 Mbps and 68.8 Mbps.

5.6.4 P4Pi-BMv2. P4Pi is also evaluated using v1model over BMv2 software switch, using BMv2 performance configuration [16] and the same testbed as P4Pi-T4P4s. The encode-based DT and RF models deployed on BMv2 achieve a throughput of 80 Mbps and 60 Mbps. The latency results is 2.1 ms and 2.5 ms when DT and RF model (DT_{EB}, RF_{EB}) is deployed, with a baseline latency of 1.1 ms.

5.7 Framework Performance

5.7.1 Framework Execution Time. We measure the time required to load a dataset, train a model, convert the trained model, test table entries, compile the mapped model to a target, and load the generated tables. Among these, we focus on training and conversion time, the two time-consuming components in Planter's operation. Based on the results (shown in Figure 17), for a small model using UNSW dataset under Anomaly Detection use case, most of the small models' training time (except SVM, NN, and AE) and all of the models' conversion time are less than 10*s*, which shows Planter can prototype in-network ML fast.

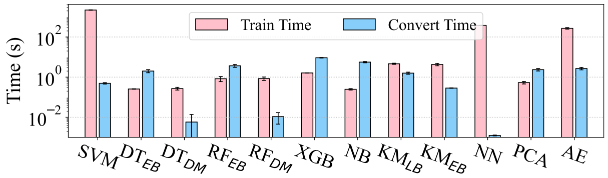


Figure 17: Algorithms' train & convert time (UNSW dataset).

5.7.2 User Experience. The efficacy of rapid prototyping was explored by checking the implementation time of an in-network ML inference prototype by two undergraduate students with no P4 knowledge and two graduates with basic P4 knowledge[2]. All test users successfully compiled and configured an in-network ML prototype on a programmable target within 10 minutes or less. The graduates estimated it would have taken them 2-3 months to study ML prerequisites and debug the code without Planter. These results demonstrate the clear advantage of rapid prototyping for users of all skill levels.

6 REFERENCE APPLICATIONS

Planter has been used in several publications, we summarize them into four use cases as reference applications of Planter: traffic classification, anomaly detection, load balancing, and financial market prediction.

Traffic Classification. P4Pir [94] and FLIP4 [93] introduce an IoT edge traffic analysis framework using ML. This framework reduces decision time for traffic analysis and classification, enabling swift traffic handling. It achieves this by concurrently applying in-network ML in the data plane alongside traffic forwarding. One challenge faced is selecting a low-overhead model. The works employ Planter for model selection and resource optimization. Leveraging Planter, the framework achieves rapid end-to-end prototyping and optimized deployment planning. As a result, the in-network deployment effectively caters to resource-constrained gateways and imposes minimal overhead on regular gateway traffic.

[1] NVIDIA's P4 compiler for DPU, currently not generally available, will enable higher performance.

[2] Approved by an institutional compliance team (IRB equivalent)

Anomaly Detection. Replacing traditional security measures or firewalls, in-network ML is able to detect the attack inside the network, realize the early traffic termination, and thus protect the network and user infrastructures. Building upon the Planter framework, IIsy tools [101] can realize in-network ML models with different network features and flexibly adapt to various attacks. To realize close to optimal inference, IIsy proposes a hybrid in-network ML solution. Specifically, the hybrid solution deploys a small model in the network and a large model at the backend. The system uses the small model's decision confidence to determine whether to directly apply the in-network decision or send it to the larger backend model for further processing. The Planter framework facilitates rapid exploration and identification of the optimal model size feasible for resource-constrained programmable network devices. Planter's mapping can keep the maximum portion of decision-making on the switch, so as to improve the system performance of hybrid deployment.

Load Balancing. To achieve low latency load balancing control, QCMP [99] employs the Planter framework, realizing an effective distributed load balancing solution through the introduction of two in-network Q-learning algorithms. Leveraging the mapping methodology of Planter, the proposed in-network solutions can be deployed on commodity switches (Tofino), consuming minimal resources while operating at maximum capacity (line rate). The QCMP load balancing solution effectively adapts to changing traffic patterns, surpassing the performance of traditional Equal-Cost Multi-Path (ECMP) solutions.

Financial Market Prediction. LOBIN [36] provides ML-based in-network market prediction using limit order books (LOBs) based on high-frequency market data feeds. The work constructs LOBs, extract ML features, and performs predictions within programmable switches, achieving low-latency forecasts for future stock price movements. The implementation of the prototype is facilitated by the Planter framework, with the process of LOB construction and updates integrated into the framework together with inference, streamlining the whole process and facilitating ML model selection. The outcomes demonstrate that in-network ML yields low-latency prediction with a minimal impact on ML performance in comparison with traditional server-based methods.

7 DISCUSSION

Rapid Prototyping. Planter abstracts three ML mapping methods and proposes a framework for one-click ML model deployment on programmable network devices. The framework can automatically select, generate, configure, compile, load, and run the mapped ML models on the target. It has four key design advantages: it 1) automates tasks by dividing functional blocks, 2) promotes code reusability by splitting P4 logics, 3) allows easy extension through modular design with interfaces, and 4) provides optional front-end for model tuning and failure handling, suitable for all-level users.

Optimized Mapping. The mapping optimization discussed in the text is aimed at the mapping techniques and is applicable to all targets. These optimizations primarily focus on reducing resource consumption, which can enhance scalability and improve ML inference performance. In terms of optimization on deployment, there are specific techniques tailored to each type of target. For example, on Tofino, stage assignment is enforced to ensure table allocation meets expectations in this research. However, target-specific deployment optimizations are beyond the scope of this paper.

ML Performance. ML models mapped by Planter provide in-network ML accuracy similar to running the same model on a host, as the evaluation shows. However, model size and inference performance present a trade-off. Sometimes, a large model can achieve higher accuracy for additional switch resources. Planter can handle this trade-off and find the best mapping configuration using Planter's front-end to fit on a switch and achieve high accuracy.

NN vs Traditional Models. In previous sections, Planter supports NN based on prior work [76]. Evaluation results show that NN is feasible for certain model sizes and hardware targets. For smaller NN sizes, there is a decrease in accuracy/F1 performance. While NN is extremely powerful, especially in training, research to date has shown that PISA-based ASIC is less suitable for NN [72], and this work does not challenge the claim. Instead, Planter shows that other inference models are feasible and powerful.

System Performance Models generated by Planter, when run on commodity hardware, exhibit minimal performance differences, such as identical line rate throughput and less than 100 nanoseconds of latency variation. On software targets, however, the system performance of the models varies significantly with complexity. For example, the throughput of NN on BMv2 is only one-tenth that of DT$_{EB}$. Planter's mapping facilitates more efficient algorithm mappings, meaning model implementations with lower complexity can achieve the same accuracy, which can lead to better system performance on multiple targets. Determining the smallest model required to meet specific accuracy needs can be achieved through the use of the Planter front-end.

Model Scalability. Models generated by Planter are designed for deployment on standalone target devices, while maximizing throughput and minimizing latency. There are numerous deployment techniques and strategies to deploy more complex models and further enhance ML performance, despite resource constraints. First, Planter has been used in [101] to provide a hybrid deployment, with a small model in the switch and a large model at the back-end. Second, DINC [100] uses Planter to support distributed in-network ML, allowing to deploy large models by distributing the resources across multiple devices. In addition, LOBIN [36], combined Planter and recirculation to support complex in-network ML functionality.

Pipeline Stages. The number of stages required by a model relates both to the type of mapped model and its size. For the UNSW dataset, at least 2 stages are consumed, and some models do not fit. Planter's Encode-based solutions outperform previous direct-mapping solutions. Planter shows that stages can be shared with standard switch functionality. Some designs can be hand-modified to reduce stages, e.g., where network and ML functions have similarities. Our experience is that 2-3 stages can be saved through manual optimization.

Targets. Planter is not target-specific. It currently supports a range of P4 targets, such as Intel Tofino and Tofino 2, BMv2, P4Pi [47] using either T4P4S over DPDK or BMv2, Alevo U280 FPGA over OpenNIC Shell [89], and all P4 architectures required by these targets. Planter is open to new targets and will continuously expand its support for emerging targets, keeping in-network ML vibrant. Owing to the framework's modular design, adding more targets to Planter is straightforward, primarily involving the inclusion of scripts pertaining to the target's compiler and testing environment. Detailed guidance can be found in the Planter repository [104].

Lessons Learned. Our experiences with Planter indicate that there is no single ML model that outperforms all other models for all use cases and targets. Tree-based models are easily mapped to hardware targets, and parallelism enables support of multiple trees without performance degradation. EB-based mapping is more efficient here. On the other hand, software-based switches without stage-limitations can be more resource efficient using direct-mapped models. However, increased computations degrade software switches' throughput and increase their latency. As ML performance is determined in the training stage, choosing a suitable class of models should follow ML applications' state of the art. Planter's front-end can then assist in identifying the best set of hyperparameters that will generate a feasible in-network ML solution.

Support & Use Cases. Planter provides a one-click in-network ML solution for emerging use cases. Early users of Planter have explored, for example, smart IoT gateways (e.g., P4Pir [92] and FLIP4 [93]), anomaly detection (e.g., IIsy [101]) and e-commerce bot detection (e.g., [32]), financial market prediction (e.g., LOBIN [36]), and load balancing (e.g., QCMP [99]). We believe that the wide adoption of in-network ML requires a suitable framework. Planter aims to be to in-network ML what CUDA was to GPUs [59]: the enabler for wide adoption on programmable targets, leading to a proliferation of use cases.

Benefits to Community. Planter serves as a rapid prototyping solution for in-network ML development. It empowers researchers and developers in the community to rapidly validate design ideas, conduct benchmark experiments, and evaluate the performance of the design. Apart from its primary function as a prototyping tool, Planter also serves as an educational resource, helping students understand concepts of in-network computing and gain hands-on experience. Given that in-network computing is a recent research area, Planter can play a vital role in expediting tests, validation, and standardization process. Furthermore, it encourages diverse exploration in network and interdisciplinary domains, accelerating research and development.

8 RELATED WORKS

In-network ML Models. Currently, attempts have been done to map several In-network ML models like SVM, *k*-means, NN, DT, and RF. However, most of the previous works support a single type of model (e.g. DT in SwitchTree [50], RF in pForest [7], NN in N3IC [76], KM in Clusteams [26]) and support few targets with less resource-constrained and not for commodity usage (e.g., SwitchTree [50] on BMv2, IIsy [91, 101] on NetFPGA & BMv2, and Taurus [81] & Homunculus [82] on their own customized hardware). At the same time, most of the models had limitations with scalability, especially in resource-constrained environments.

In-network ML Framework. The framework for ML on servers is flourishing, such as scikit-learn [66], Pytorch [65], and Keras [13]. However, the choice of in-network ML framework is very limited. Most of the existing tools like IIsy [91, 101], Netbeacon [106], and SwitchTree [50] are mainly use case specific, not extendable, lack automation, and are more like a demo than a tool. Homunculus [82] provides an automated parameter selection framework for in-network ML but does not provide new ML models for in-network ML mapping to accommodate various use cases.

9 CONCLUSION

This paper presented Planter, a modular framework for one-click implementation of in-network ML algorithms. Planter's modular design enables integration of new ML models, architectures, targets, and use cases. Planter implements a wide range of in-network ML algorithms, including four new algorithms, and upgrades six previously proposed mappings. The evaluation shows that Planter accurately maps trained models to a switch, can achieve high inference accuracy and line rate throughput, and can be integrated with *switch.p4* without consuming additional stages, scaling better than multiple previous works. As an open-source platform, Planter is the enabler for in-network ML research, and is available at [104].

Acknowledgments: This paper complies with all applicable ethical standards of the authors' home institution. This work was partly funded by VMware and the EU Horizon SMARTEDGE (101092908, UKRI 10056403). We acknowledge support from Intel and NVIDIA. For the purpose of Open Access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript (AAM) version arising from this submission.

REFERENCES

[1] [n. d.]. Nasdaq ITCH Data Source. ([n. d.]). https://emi.nasdaq.com/ITCH/ Accessed on 02/05/2023.

[2] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. 2022. Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 693–706. https://doi.org/10.1145/3544216.3544263

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the ACM SIGCOMM 2020 Conference (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[4] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern Recognition and Machine Learning*. Vol. 4. Springer.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[6] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–20. https://doi.org/10.1109/HCS52781.2021.9567066

[7] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pForest: In-Network Inference with Random Forests. *CoRR* abs/1909.05680 (2019). arXiv:1909.05680 http://arxiv.org/abs/1909.05680

[8] Charles Cao, Oliver Hansch, and Xiaoxin Wang. 2009. The information content of an open limit-order book. *Journal of Futures Markets: Futures, Options, and Other Derivative Products* 29, 1 (2009), 16–41.

[9] Lucas Castanheira, Ricardo Parizotto, and Alberto E Schaeffer-Filho. 2019. FlowStalker: Comprehensive Traffic Flow Monitoring on the Data Plane using P4. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[10] Efstratios Chatzoglou, Georgios Kambourakis, and Constantinos Kolias. 2021. Empirical Evaluation of Attacks Against IEEE 802.11 Enterprise Networks: The AWID3 Dataset. *IEEE Access* 9 (2021), 34188–34205.

[11] Hongyi Chen, Damu Ding, Changgang Zheng, Rana Abu Bakar, Filippo Cugini, et al. 2024. SmartEdge. (2024), 1–57. D4.1 Design of Dynamic & Secure Swarm Networking, GA 101092908.

[12] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining (KDD '16). ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[13] François Chollet et al. 2015. Keras. https://keras.io. (2015).

[14] Rohit Choudhry and Kumkum Garg. 2008. A Hybrid Machine Learning System for Stock Market Forecasting. *World Academy of Science, Engineering and Technology* 39, 3 (2008), 315–318.

[15] P4 Language Consortium. 2020. P4 Behavioral-Model. https://github.com/p4lang/behavioral-model [accessed January 26, 2022]. (2020).

[16] P4 Language Consortium. 2020. Performance of BMv2. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md [accessed January 26, 2024]. (2020).

[17] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Machine learning* 20, 3 (1995), 273–297.

[18] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as A Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1726–1738.

[19] Dell Technologies. 2022. Dell EMC Edge Gateway 5200 Software User's Guide. (2022). https://www.dell.com/support/manuals/en-ae/dell-edge-gateway-5200/egw-5200-software-users-guide

[20] Pedro Domingos and Michael Pazzani. 1997. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine learning* 29, 2 (1997), 103–130.

[21] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[22] Ronald A Fisher. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of eugenics* 7, 2 (1936), 179–188.

[23] Evelyn Fix and Joseph Lawson Hodges. 1989. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *International Statistical Review/Revue Internationale de Statistique* 57, 3 (1989), 238–247.

[24] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and An Application to Boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[25] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. 2010. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software* 33, 1 (2010), 1.

[26] Roy Friedman, Or Goaz, and Ori Rottenstreich. 2021. Clustreams: Data Plane Clustering. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 101–107. https://doi.org/10.1145/3482898.3483356

[27] Karl Pearson F.R.S. 1901. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (1901), 559–572. https://doi.org/10.1080/14786440109462720

[28] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. 435–450.

[29] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-Network Aggregation for Shared Machine Learning Clusters. *Proceedings of Machine Learning and Systems* 3 (2021), 829–844.

[30] Jane Street Group. 2020. Jane Street Market Prediction. https://www.kaggle.com/c/jane-street-market-prediction. (2020). [Online; accessed January 2021].

[31] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. 2019. Requet: Real-Time QoE Detection for Encrypted YouTube Traffic. In *Proceedings of the 10th ACM Multimedia Systems Conference (MMSys '19)*. Association for Computing Machinery, New York, NY, USA, 48–59.

[32] Masoud Hemmatpour, Changgang Zheng, and Noa Zilberman. 2024. E-Commerce Bot Traffic: In-Network Impact, Detection, and Mitigation. In *7th Conference on Innovation in Clouds, Internet and Networks (ICIN)*.

[33] Bruno Miranda Henrique, Vinicius Amorim Sobreiro, and Herbert Kimura. 2018. Stock Price Prediction Using Support Vector Regression on Daily and Up to the Minute Prices. *The Journal of finance and data science* 4, 3 (2018), 183–201.

[34] Tin Kam Ho. 1995. Random Decision Forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[35] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[36] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. 2023. LOBIN: In-Network Machine Learning for Limit Order Books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 159–166.

[37] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated Machine Learning: Methods, Systems, Challenges.* Springer Nature.

[38] Connor Imes, Steven Hofmeyr, and Henry Hoffmann. 2018. Energy-Efficient Application Resource Scheduling Using Machine Learning Classifiers. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 45, 11 pages. https://doi.org/10.1145/3225058.3225088

[39] Intel. 2021. P4_16 Intel® Tofino™ Native Architecture – Public Version. (2021). https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf

[40] Weirong Jiang and Viktor K Prasanna. 2011. Scalable Packet Classification on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 9 (2011), 1668–1680.

[41] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[42] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*. 1–12.

[43] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in neural information processing systems* 30 (2017).

[44] Wasiat Khan, Mustansar Ali Ghazanfar, Muhammad Awais Azam, Amin Karami, Khaled H Alyoubi, and Ahmed S Alfakeeh. 2020. Stock market prediction using machine learning classifiers and social media, news. *Journal of Ambient Intelligence and Humanized Computing* (2020), 1–24.

[45] Changhoon Kim, Parag Bhide, E Doe, H Holbrook, A Ghanwani, D Daly, M Hira, and B Davie. 2016. In-Band Network Telemetry (INT). *Tech. Spec* (2016).

[46] Yiming Kong, Hui Zang, and Xiaoli Ma. 2018. Improving TCP Congestion Control with Machine Intelligence. In *Proceedings of the 2018 Workshop on Network Meets AI & ML (NetAI'18)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3229543.3229550

[47] Sándor Laki, Radostin Stoyanov, Dávid Kis, Robert Soulé, Péter Vörös, and Noa Zilberman. 2021. P4Pi: P4 on Raspberry Pi for Networking Education. *ACM SIGCOMM Computer Communication Review* 51, 3 (2021), 17–21.

[48] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-Network Aggregation for Multi-Tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao

[49] Christian Leber, Benjamin Geib, and Heiner Litz. 2011. High Frequency Trading Acceleration Using FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 317–322.

[50] Jong-Hyouk Lee and Kamal Singh. 2020. SwitchTree: In-Network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications* (2020), 1–12.

[51] Carson Kai-Sang Leung, Richard Kyle MacKinnon, and Yang Wang. 2014. A Machine Learning Approach for Stock Price Prediction. In *Proceedings of the 18th International Database Engineering & Applications Symposium*. 274–277.

[52] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. 2014. Autoencoder for Words. *Neurocomputing* 139 (2014), 84–96.

[53] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[54] John W Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. 2012. A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT). In *2012 IEEE 20th annual symposium on high-performance interconnects*. IEEE, 9–16.

[55] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay Thakur, Larry Peterson, Jennifer Rexford, and Oguz Sunay. 2021. A P4-Based 5G User Plane Function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 162–168.

[56] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[57] Nour Moustafa and Jill Slay. 2015. UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection systems (UNSW-NB15 Network Data Set). In *2015 military communications and information systems conference (MilCIS)*. IEEE, 1–6.

[58] NASDAQ OMX PSX. 2014. NASDAQ OMX PSX TotalView-ITCH 5.0. http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/PSXTVITCHSpecification_5.0.pdf. (2014).

[59] Vincent Natoli. 2010. Kudos for CUDA. https://www.hpcwire.com/2010/07/06/kudos_for_cuda/. (2010).

[60] John Ashworth Nelder and Robert WM Wedderburn. 1972. Generalized Linear Models. *Journal of the Royal Statistical Society: Series A (General)* 135, 3 (1972), 370–384.

[61] Michael A Nielsen. 2015. *Neural Networks and Deep Learning*. Vol. 25. Determination press San Francisco, CA, USA.

[62] The P4.org Architecture Working Group. 2017. P4_16 PSA Specification (v1.1). (2017). https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[63] Francesco Paolucci, Federico Civerchia, Andrea Sgambelluri, Alessio Giorgetti, Filippo Cugini, and Piero Castoldi. 2019. P4 Edge Node Enabling Stateful Traffic Engineering and Cyber Security. *IEEE/OSA Journal of Optical Communications and Networking* 11 (2019), A84–A95.

[64] Francesco Paolucci, Filippo Cugini, Piero Castoldi, and Tomasz Osinski. 2021. Enhancing 5G SDN/NFV Edge with P4 Data Plane Programmability. *IEEE Network* 35, 3 (2021), 154–160. https://doi.org/10.1109/MNET.021.1900599

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in neural information processing systems* 32 (2019).

[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[67] Larry Peterson, Carmelo Cascone, Brian O'Connor, Thomas Vachuska, and Bruce Davie. 2021. *Software-Defined Networks: A Systems Approach*. Systems Approach, LLC. https://sdn.systemsapproach.org

[68] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. *Advances in neural information processing systems* 31 (2018).

[69] Qiaofeng Qin, Konstantinos Poularakis, Kin K Leung, and Leandros Tassiulas. 2020. Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning. In *2020 IFIP Networking Conference (Networking)*. IEEE, 352–360.

[70] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.

[71] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (1984), 187–260. https://doi.org/10.1145/356924.356930

[72] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network be the AI Accelerator?. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*. 20–25.

[73] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*. 785–808.

[74] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. 2018. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. *ICISSp* 1 (2018), 108–116.

[75] Shunrong Shen, Haomiao Jiang, and Tongda Zhang. 2012. Stock Market Forecasting Using Machine Learning Algorithms. *Department of Electrical Engineering, Stanford University, Stanford, CA* (2012), 1–5.

[76] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 513–533.

[77] Giuseppe Siracusano, Davide Sanvito, Salvator Galea, and Roberto Bifulco. 2018. Deep Learning Inference on Commodity Network Interface Cards. In *Proc. Workshop Syst. ML Open Source Softw. NeurIPS*.

[78] Kirstine Smith. 1918. On the Standard Deviations of Adjusted and Interpolated Values of an Observed Polynomial Function and its Constants and the Guidance they give Towards a Proper Choice of the Distribution of Observations. *Biometrika* 12, 1/2 (1918), 1–85.

[79] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing Dataplane Programs with μP4. In *ACM SIGCOMM*. 329–343.

[80] S.J. Stolfo, Wei Fan, Wenke Lee, A. Prodromidis, and P.K. Chan. 2000. Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Vol. 2. 130–144 vol.2. https://doi.org/10.1109/DISCEX.2000.821515

[81] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: A Data Plane Architecture for Per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1099–1114.

[82] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. 2023. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 329–342.

[83] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[84] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case for In-Network Computing on Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[85] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. 2018. T4p4s: A target-independent compiler for protocol-independent packet processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 1–8.

[86] Sarah Wassermann, Michael Seufert, Pedro Casas, Li Gang, and Kuang Li. 2020. ViCrypt to the Rescue: Real-Time, Machine-Learning-Driven Video-QoE Monitoring for Encrypted Streaming Traffic. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2007–2023.

[87] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. 2008. Top 10 Algorithms in Data Mining. *Knowledge and information systems* 14, 1 (2008), 1–37.

[88] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. 2021. Programmable Switches for In-Networking Classification. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[89] Xilinx. Accessed on 04/15/2023. Xilinx OpenNIC Shell. https://github.com/Xilinx/open-nic. (Accessed on 04/15/2023).

[90] Xilinx. Accessed on 04/16/2023. Vitis Networking P4 User Guide. https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide/Target-Architecture. (Accessed on 04/16/2023).

[91] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33.

[92] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. 2023. Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways. *IEEE Internet of Things Journal* (2023).

[93] Mingyuan Zang, Changgang Zheng, Tomasz Koziak, Noa Zilberman, and Lars Dittmann. 2023. Federated Learning-Based In-Network Traffic Analysis on IoT Edge. In *Security for IoT Networks and Devices in 6G (Sec4IoT), IFIP Networking*.

[94] Mingyuan Zang, Changgang Zheng, Radostin Stoyanov, Lars Dittmann, and Noa Zilberman. 2022. P4Pir: In-Network Analysis for Smart IoT Gateways. In *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*. 46–48.

[95] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. 2021. pHeavy: Predicting Heavy Flows in the Programmable Data Plane. *IEEE Transactions on Network and Service Management* (2021).

[96] Yaoxue Zhang, Ju Ren, Jiagang Liu, Chugui Xu, Hui Guo, and Yaping Liu. 2017. A Survey on Emerging Computing Paradigms for Dig Data. *Chinese Journal of Electronics* 26, 1 (2017), 1–12.

[97] Zihao Zhang, Stefan Zohren, and Stephen Roberts. 2019. Deeplob: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing* 67, 11 (2019), 3001–3012.

[98] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2023. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials* (2023), 1–1. https://doi.org/10.1109/COMST.2023.3344351

[99] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. 2023. QCMP: Load Balancing via In-Network Reinforcement Learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*.

[100] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. 2023. DINC: Toward Distributed In-Network Computing. *Proceedings of the ACM on Networking* 1, CoNEXT3 (2023), 1–25.

[101] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Transactions on Networking* (2024), 1–16. https://doi.org/10.1109/TNET.2024.3364757

[102] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating In-Network Machine Learning. (2022). arXiv:cs.NI/2205.08824

[103] Changgang Zheng and Noa Zilberman. 2021. Planter: Seeding Trees Within Switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 12–14.

[104] Changgang Zheng, et al. 2022. Planter's GitHub Repository. https://github.com/In-Network-Machine-Learning/Planter. (2022).

[105] Zhiren Zhong, Wei Wang, Yiyang Shao, Zhenyu Li, Heng Pan, Hongtao Guan, Gareth Tyson, Gaogang Xie, and Kai Zheng. 2022. Muses: Enabling Lightweight Learning-Based Congestion Control for Mobile Devices. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 2208–2217. https://doi.org/10.1109/INFOCOM48880.2022.9796880

[106] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. 2023. An Efficient Design of Intelligent Network Data Plane. In *32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association.*

A APPENDIX

Planter is an open-source framework, aiming at facilitating in-network ML development. Its code is available at GitHub [104]. Starting to use Planter requires just a configuration file and a dataset. Afterward, Planter manages the transition of ML tasks into the programmable data plane. All datasets used in our evaluation are publicly available. A comprehensive and detailed step-by-step guide on the usage of the framework is in the repository `README.md` and `./src/help`. This section provides a brief overview of the guide.

A.1 Environment Setup

Planter requires `python3` with the packages listed in the repository. To install the aforementioned packages and set up the working environment, the following command should be executed:

```
sudo pip3 install -r ./src/configs/packages.txt
```

As Planter drives network programmable targets for executing and testing use cases in the data plane, ensure the chosen target's necessary environment is installed. For example, BMv2's required environment can be set up following [15]. There are also publicly available guides for other Planter-supported targets. The introduction of all targets and their installation are presented in our repository[3]under the `src/help/Planter_Supports` folder.

A.2 Getting Started with Planter

Planter is started using the following command:

```
sudo python3 Planter.py
```

Configurations can be changed by editing the configuration file. Planter provides multiple modes, for example, manual configuration mode, where configuration is input manually with a detailed CLI. This mode can be activated by adding `-m`. One can also use `-h` to see additional command options and learn more modes.

Each run will output three ML performance reports. 1. The first matrix is the report from the `scikit-learn`. 2. The second matrix shows the simulated data plane result. 3. The third matrix reports the actual result of the model performance on the selected target. A more detailed matrix is shown in the tutorial wiki [104].

A.3 Reproduction of Evaluations

The configuration for evaluations is stored in the repository to aid the reproduction of our evaluation results. The guidance on detailed steps can be found in `evaluation.md` under the `./src/eval` folder. Run the following command to update the configurations:

```
sudo nano ./src/config/Planter_config.json
```

Then, start the framework following §A.2, and Planter will load the corresponding configurations automatically.

A.4 Adding New Modules and Designs

Planter enables the incorporation of new models, architectures, targets, use cases, and datasets. For example, to use one's own dataset, a load data file should be created under `./src/load_data` folder. It is used for loading the source data and preprocessing it based on one's needs following a standard interface. The step-by-step guides on how to add new models, architectures, targets, use cases, and datasets are all presented in our repository [104].