# IIsy: Practical In-Network Classification

Changgang Zheng[†], Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane[†], Antoine Bernabeu[§], Shay Vargaftik[◇], Yaniv Ben-Itzhak[◇], and Noa Zilberman[†]

[†]University of Oxford, [§]École Centrale de Nantes, [◇]VMware Research
{changgang.zheng, noa.zilberman}@eng.ox.ac.uk, antoine.bernabeu@eleves.ec-nantes.fr,
riyad.bensoussane@worc.ox.ac.uk, {shayv, ybenitzhak}@vmware.com

## ABSTRACT

The rat race between user-generated data and data-processing systems is currently won by data. The increased use of machine learning leads to further increase in processing requirements, while data volume keeps growing. To win the race, machine learning needs to be applied to the data as it goes through the network. In-network classification of data can reduce the load on servers, reduce response time and increase scalability.

In this paper, we introduce IIsy, implementing machine learning classification models in a hybrid fashion using off-the-shelf network devices. IIsy targets three main challenges of in-network classification: (i) mapping classification models to network devices (ii) extracting the required features and (iii) addressing resource and functionality constraints. IIsy supports a range of traditional and ensemble machine learning models, scaling independently of the number of stages in a switch pipeline. Moreover, we demonstrate the use of IIsy for hybrid classification, where a small model is implemented on a switch and a large model at the backend, achieving near optimal classification results , while significantly reducing latency and load on the servers.

## 1 INTRODUCTION

Machine learning (ML) is increasingly applied to every aspect of our lives, leading to huge processing requirements. In data centers, ML has become a prominent workload [21]. To alleviate compute requirements and improve latency-sensitive applications' performance, ML is pushed to the edge [13] and to end-user devices [7]. The performance requirements of ML have driven the development of a range of ML accelerators, including GPUs [2], FPGA [6] and custom ASICs [26]. While state-of-the-art accelerators can run trillions of operations per second, their throughput is still limited by their network interface. Network devices offer an untapped resource for scaling ML, and in particular – classification. The use of programmable network devices for in-network computing applications, such as caching [25], consensus [15] and network services [27], provides orders of magnitude throughput
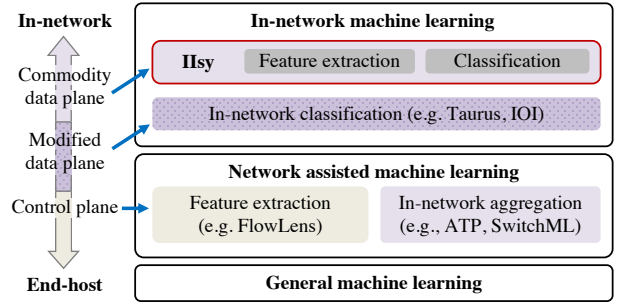


**Figure 1: The positioning of IIsy**

increase and latency reduction, combined with significant power savings [50].

Combining ML and networking is not a new trend (e.g. [33]), with most of the work focusing on ML on the end host. Newer works had succeeded in creating network-assisted machine learning, as shown in Figure 1, using network devices either for aggregation [28, 41], or for feature extraction [5]. Despite these previous successes, running ML within network devices has proven hard to tackle. While ML accelerators rely on multiplication and matrix multiplication [26], network devices do not support such operations. Several works have tried to attend to this limitation by modifying the data plane or designing new hardware modules [48, 61].

A few attempts have been made to run ML models within the network (top of Figure 1), as detailed in Table 4 and §9. The first class of works [40, 44–46], implemented binary neural networks on network interface cards (NICs), FPGA or in a software environment. Their attempts to implement on a switch-ASIC have failed both in scale and performance, as it is significantly more constrained in resources and functionality. The second class of works [30, 48, 61] enabled ML by modifying the hardware or using FPGA. These are experimental, not off-the-shelf solutions, and can not be easily and cheaply adopted. The last class of solutions has focused on implementing Random Forests on switches. These solutions had failed to scale on a switch-ASIC [12] or were independent of most constraints by running in a software environment or on a NIC [29, 55]. Harnessing the power

of network processing for ML within commodity switches remains a challenge.

To that end, we present IIsy, supporting *off-the-shelf programmable switches* (e.g., Intel Tofino) to employ a range of ML classification methods. IIsy supports decision-tree, Random-Forest, Isolation-Forest, XGBoost, Support Vector Machine (SVM), Naïve Bayes and K-Means. IIsy is generalizable to other classification methods, but does not support neural network models, to avoid performance compromises.

The design of IIsy follows the following guidelines:

**Low-resource ML models** ML models vary in types and nature, requiring complex mathematical operations, unsupported by switch-ASIC, or consume significant resources (e.g., tree-based models). The scarcity of switch resources means that not every model mapping will be feasible. Beyond the need to reserve resources for networking functionality, some mappings are impractical, such as using features of hundreds of bytes as lookup keys. Therefore, any model or class of models requires a tailored mapping to a switch architecture. We address these challenges in §4.

**Easy ML model updates** As over time data changes and ML models need to be re-trained, a quick and easy deployment of ML models is sought. While switches are programmable, using them in a production environment means that ideally the switch's (P4) program should not be changed, and that only common operations, such as table updates, should be allowed. Moreover, the deployment of an updated classification model should be quick, and minimize traffic disruption. These challenges are addressed in §4.

**Machine learning performance** For classification purposes, the same level of ML performance, e.g., accuracy and precision, as running on a CPU or a GPU is targeted. While this is highly desirable, it is sometimes practical to trade some accuracy for resources, e.g., saving half the memory resources while giving up 1% of accuracy. To address this challenge, several possible in-network ML deployments are listed in §2.2, and §7 focuses on the hybrid deployment option, offering competitive ML performance with low-resource in-network classification consumption.

**Feature extraction** While packet-header features can be easily extracted, more complex features are needed to support many ML models. In PISA-style devices [8] this means using the parser to extract specific data from the packet, and the match-action pipeline to turn this extracted data into a feature and store the information. Additionally, it is required to process data stored deep within the payload. These challenges are addressed in §5.

In summary, this paper presents IIsy – a framework for automated mapping of trained classification models to commodity, off-the-shelf, switch ASIC. IIsy generates both data plane and control plane programs from the output of a common ML training framework, and does not require any modifications to tools, network devices, or protocols.

In particular, our main contributions in this paper are:

- Introducing a mapping to programmable network devices of a range of classification methods, including decision-tree, Random-Forest, Isolation-Forest, XGBoost, Support Vector Machine (SVM), Naïve Bayes and K-Means.
- Presenting a mapping algorithm that is independent of the number of stages in the switch pipeline, which is critical for scaling ensemble models.
- Demonstrating feature extraction on packet, flow, aggregate and file granularity.
- Demonstrating IIsy's usability for in-network classification using a hybrid model, consisting of a small model on a network devices, and large model over the hosts. We demonstrate it can reduce backend's load, and reduce the classification latency for time-sensitive applications.

## 2 IN-NETWORK CLASSIFICATION OVERVIEW

### 2.1 Potential Benefits

Network devices have two major advantages over any other type of a computing device: location and data-processing speed. Any cloud-processed user-generated data goes through the network first. This means that i) the rate of data that can be processed is capped by the network, ii) the latency from the user to the processing node will always be higher than the latency to any network device along its path and iii) network devices are already part of the infrastructure carrying user data, and do not need to be newly added. This leads to the observation that *the rate of classification decisions is bounded by network devices' data rate.*

From a system perspective, in-network computing was already shown to be beneficial, freeing cycles on the CPU and providing high power efficiency per operation [50]. Improved throughput and latency are also known advantages [15, 25], but need to be considered per use-case. As we show later, in-network classification can significantly reduce the amount of traffic that gets to servers, and that requires further processing.

Using network devices for in-network ML suggests a few more benefits. Firstly, being able to classify data before it reaches the host can be essential for some use-cases. For example, distributed denial of service (DDoS) mitigation, where malicious traffic has to be dropped as close as possible to the source. Secondly, automatically converting and loading ML training results to (local and remote) network devices [56],

can speed up the reaction to events in the network, and shorten the time for detection and mitigation.

## 2.2 Deployments Scenarios

In-network classification is possible in different deployment scenarios, including: (1) a native switch operation, (2) a switch acting as an endpoint accelerator, (3) smart NICs, and (4) a hybrid ML model, combining the in-network classification model with a traditional ML model deployed at the end-point.

**Native switch.** A switch in its native usage model has some, or most, of its resources dedicated to networking operations, meaning that in-network classification needs to be resource efficient. Using in-network classification within the switch does not require extra cost or space, and the power overheads are small [50]. The location of a switch has affects its benefits; A switch very close to the user is most useful for data reduction, ultra low latency applications, and to mitigate the effects of distributed events (e.g., DDoS attacks). On the other hand, a switch within a data center can support more complex applications. For example, assuming that the switch is located after a load balancer, decrypted traffic can sometimes be assumed [3, 36], allowing to apply in-network classification to use-cases otherwise prohibited by traffic encryption.

**Endpoint accelerator.** The switch as an endpoint accelerator refers to using a switch purely for ML purposes. This model is already in use for some applications, such as load balancing [11]. Unlike other deployment scenarios, here the switch adds space, power, and cost overheads.

**Smart NIC.** Smart NICs support lower bandwidth than switches but benefit from better resources availability, such as on-board memory and encryption modules. The host's proximity means that the devices see only a subset of the overall network traffic. Host offloading and power savings are reduced relative to a switch [50]. Our solution is applicable to architectures like the Portable NIC Architecture (PNA) [9].

**Hybrid.** The resource-constrained nature of network devices means that in-network ensemble models are smaller than full-grown ensemble models, hence their ML performance may be sub-par. In such cases, a *hybrid ML model* can be used to achieve close to optimum ML performance, while still benefiting from the performance of in-network classification. The hybrid deployment scenario employs a small in-network ML model and a large ML model over the end-point.

*2.2.1 Hybrid Deployments.* In many ML ensemble models, such as Random-Forest and XGBoost, the ML model can provide a classification with a corresponding confidence level – the probability that the classification is correct.

In this paper, we adopt the hybrid ML model concept [43, 52], by implementing a small in-network ensemble model (e.g., by limiting the number of trees in an ensemble, or using a subset of features [49]), and running a large ML model at the end-point. To cope with the lower ML performance of the small in-network ML model, classifications by the small model are considered valid only if their corresponding confidence is above a given (high) threshold. Invalid classifications by the small in-network ML model (i.e., confidence below the threshold) are forwarded for re-classification by the large ML model deployed at the back-end.

Previous ML works [43, 52] have shown that most of the queries in a given data-set can be classified by a small ML model with high confidence level. Hence, a hybrid ML deployment reduces both the classification latency and the load over the back-end servers (by forwarding only "hard" queries for re-classifications), as compared to a monolithic ML model deployment at the back-end. In §7, we demonstrate these benefits using two use-cases from different domains, cyber-security and finance.

## 3 IISY ARCHITECTURE

IIsy is a framework that automatically maps trained classification models to programmable network devices, and in particular to off-the-shelf switch ASIC. IIsy takes the output of a common ML training framework, and converts it to data plane code and control plane code.
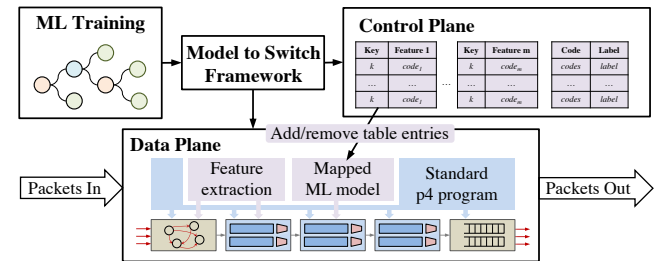


**Figure 2: The architecture of IIsy**

The architecture of IIsy, shown in Figure 2, is composed of four components: machine learning training, mapping tool to map the trained models to a target network device (the core of IIsy's architecture), a data plane implementation on a hardware target, and a control plane component for populating table entries.

IIsy's mapping tool takes a standard ML training output (e.g., pickle file) and generates from it two components: an implementation of the data plane, and the entries for the lookup tables used by the data plane which are loaded by the control plane. The hardware target and the control plane component contain standard elements that are target specific, such as the architecture of the network device (e.g.,

Tofino Native Architecture, TNA). In the following section, we present IIsy's mapping tool in details.

## 4 MAPPING MODELS TO SWITCHES

IIsy's mapping of ML classification model to network devices is guided by several insights [56]:

- Use lookup tables to implement mathematical operations, for example multiplication and exponents.
- Optimize the use of on-chip resources, and reduce lookup tables size by using codes or indicators instead of explicit calculation results.
- Use a lookup tables to decide a classification result at the end of the pipeline. The key to this table is composed of a set of indicators collected from the previous stages.
- Be willing to lose some accuracy to save resources. Decide on the (accuracy) price to pay to fit a model into your network device, or use a hybrid deployment.
- Break the dependency between tree-depth and pipeline-stages by looking up values instead of conditions.
- Optimize the use of on-chip resources by sharing features lookup tables between multiple trees or models.

In the following, we present our mapping method for different ML models to a network device. We discuss the mapping challenges involved with each ML model and the corresponding mapping solution.

### 4.1 Decision Tree

Decision-trees naturally fit into network devices, since their most basic functionality is packet classification – every ingress packet is assigned to an output port (i.e., class in terms of ML). To that end, lookup tables are used for classifying packets to their corresponding output port. For instance, in a layer-2 Ethernet switch, the feature used for classification is the destination MAC address, and the MAC table is used to decide the output port – the classification's result.

IIsy uses a more efficient mapping of a decision tree to a network device by using a single table per feature[1], and a single classification table.

IIsy's mapping is demonstrated using a simple example in Figure 3. The example shows a decision tree with a depth of three, with six leaf nodes, and using two features. The color of each branch in the tree indicates the feature used, with three branches using feature 1 ($f1$), and two branches using feature 2 ($f2$). The branches using feature 1 are mapped to Table 1, with feature 1 used as the key to the table. The table has 4 ranges, covering the potential outcomes of branches 1,2 and 4.Each range is associated with a code as the resulting action. Similarly, Table 2 uses feature 2 as the key, with 3 ranges,corresponding to branches 3 and 5. The action of

---

[1]Features are extracted as in §5

this table is a second 2-bit code. While the tables are shown as exact match, a ternary implementation is possible. The third table, the classification table, uses as the key the codes (actions) of Table 1 and Table 2. A match on this key results in a leaf node, the result of the classification.

Each leaf node is denoted by a code word, and the code word indicates the branches taken in the tree. This code word is the result of the feature tables lookup. Therefore the number of pipeline stages consumed by a decision tree is independent of the depth of the tree, overcoming previous limitations [12, 29].

**Table size analysis.** Assume a tree with $B$ branches using $F$ features, where each feature $f_i$ is $w_i$ bits wide and used in $b_i$ branches. The number of entries in a ternary feature table for $f_i$ will be $O(b_i \times w_i)$, while an exact match table will contain all the feature's values that are possible for the use case. This number can end up small, as shown in §7.3, for example if most values are mapped to a default entry.

The number of entries in the classification table depends on the depth and shape of the tree. The worst case is when the branches are evenly divided between all features, so features require the same number of (multiple) bits for the code. The best case is where $F - 1$ features are used only once in branches, requiring a single bit code, and the last feature is used $B - (F - 1)$ times. This can be written as:

$$2^{\lceil \log_2 \frac{B}{F} \rceil^F} \geq Entries \geq 2^{B-1+\lceil \log_2(B-(F-1)) \rceil} \qquad (1)$$

### 4.2 Ensemble Tree-Based Methods

Ensemble methods improve ML prediction results by combining multiple learning models [62]. We consider two types of ensemble methods: Bagging and Boosting.

The size of an ensemble is known to improve a prediction's accuracy, which stands in contradiction to the scarcity of resources on network devices. To attend to this challenge, we maintain the guidelines presented in §4, such as using a single table per feature and combining multiple models within a single table. In this manner, we increase tables' depth and the amount of metadata used but decouple the number of trees from the number of stages required in the device.

**Bagging.** In bagging, multiple learners are used, and each learner has an equal weight in the final decision. Each learner is trained using a different sample with replacement of the training data. We use Random Forest [10], which is built from multiple decision trees, as an example of mapping a bagging model to a network device.

The mapping of a model to a switch is oblivious to differences between bagging models in terms of training related parameters such as sample selection and training method. The mapping is influenced only by the constraints of the

**Decision Tree**

Branch 1
- $f1 < 5$ → Branch 2
- $f1 \geq 5$ → Branch 3

Branch 2
- $f1 < 3$ → 0
- $f1 \geq 3$ → 1

Branch 3
- $f2 < 2$ → Branch 4
- $f2 \geq 2$ → Branch 5

Branch 4
- $f1 < 7$ → 2
- $f1 \geq 7$ → 3

Branch 5
- $f2 < 6$ → 4
- $f2 \geq 6$ → 5

Input          : [Feature 1: $f1$, Feature 2: $f2$]
Example Input: [Feature 1: 4,  Feature 2: 7  ]

**Table 1: Feature 1**

|          | Feature 1 | Code1 |
|----------|-----------|-------|
|          | 0 | 00 |
|          | 1 | 00 |
| Branch 2 | 2 | 00 |
|          | 3 | 01 |
| Branch 1 | 4 | 01 |
|          | 5 | 10 |
| Branch 4 | 6 | 10 |
|          | 7 | 11 |
|          | … | … |
|          | n | 11 |

**Table 2: Feature 2**

|          | Feature 2 | Code2 |
|----------|-----------|-------|
|          | 0 | 00 |
| Brach 3  | 1 | 00 |
|          | 2 | 01 |
|          | 3 | 01 |
|          | 4 | 01 |
| Branch 5 | 5 | 01 |
|          | 6 | 10 |
|          | 7 | 10 |
|          | … | … |
|          | n | 10 |

**Table 3: Classification**

| Code1, Code2 | Leaf |
|--------------|------|
| 00, ** | 0 |
| 01, ** | 1 |
| 1*, 00 | 2 |
| 1*, 01 | 4 |
| 1*, 10 | 5 |

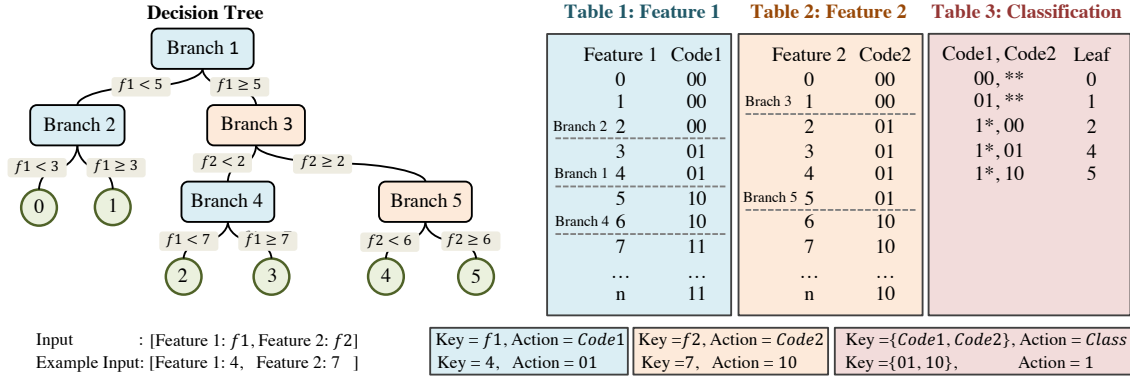| Key = $f1$, Action = $Code1$ | Key = $f2$, Action = $Code2$ | Key = {$Code1, Code2$}, Action = $Class$ |
|---|---|---|
| Key = 4,   Action = 01 | Key = 7,   Action = 10 | Key = {01, 10},          Action = 1 |

**Figure 3: An Example of a Decision Tree Mapping.**

training outcome, e.g., selected features, number and depth of trees.

While each decision tree in a Random Forest can be independently mapped to a device, as in §4.1, this is inefficient. For example, a Random Forest of ten trees, each using five features, will require fifty feature tables and eleven decision actions (one per tree, plus one for the entire forest).

IIsy significantly reduces resources requirements by sharing feature tables between trees. This means that for the previous example, IIsy will require just five feature tables instead of fifty. This mapping is not free; the number of entries in each table will increase, as well as the action's width. The result of each lookup in a feature's table is the series of action codes (as defined in §4.1), one per tree. Trees can be pruned to create action codes of feasible length. As demonstrated for a single tree, IIsy requires a table per tree to turn the action code into the decision of a tree. The classification result of the entire ensemble is based on the collective classification results of all trees in the ensemble, and can be implemented as a table or using a sum and conditions.

**Boosting.** Boosting methods are different from bagging [42], as the ensemble is built by training new learners to focus on misclassifications by previous learners. Gradient boosting is often built from an ensemble of decision trees, where a small decision tree (e.g., with 8-32 terminal nodes) is added at each iteration and scaled by a constant factor. Then, a new tree is grown to reduce the loss function of the previous trees. In boosting, new trees are trained with a focus on previous misclassifications. The decision is based on the weighted outcome of each tree.

Despite the differences in training, the mapping of a generated XGBoost model is mostly identical to a Random Forest (§4.2), using a table per feature, and a table per tree. The difference is that the leaf nodes are weighted. The weighting can be applied either when constructing the tree table, which is typically more resource-efficient, or at the decision stage. When the number of classes is small, it is effective (§7) to add the weighted results across trees, either directly or using a lookup tables, and setting the class at the final stage. Isolation Forests are implemented similarly, but summing the depth of terminal nodes, rather than their weights.

## 4.3 Classical models: SVM, Naïve Bayes and K-Means

Classical classification algorithms can all be mapped using similar methodologies. These are applied to SVM, Naïve Bayes and K-Means. A mapping takes one of two forms. The first holds one table per feature. The result of looking up the feature in the table is a code, or a value that is normalized. If the result of a lookup is a code, the last stage in the pipeline will use a lookup tables with a key of the codes of all features. If the result of a lookup is a value, then the last stage in the pipeline will operate on all values, typically adding them up and comparing the results across classes.

For example, in SVM, the key to a feature's table is the feature's value, and lookup's result is a vector of calculated values $a_i \times x_i$, where $x_i$ is the value of the feature (potentially normalized or binned). The value of an SVM hyperplane, separating two classes, is calculated as the sum of vectors from all feature tables. This can be optimized by summing the features in each pipeline stage.

A second approach, which is not always feasible, holds a table per class or class indicator. For example, in SVM there will be a table per hyperplane. The lookup key is the value all the features. The result of the lookup will be an indicator, such as if the entry belongs within or outside the hyperplane (for SVM) or the distance from a center of a cluster (for K-Means).

Our experience shows that the first approach provides (relatively) shallow tables, proportional to the number of classes. However, this approach may experience some loss of accuracy, explored in §7.7. The second approach is feasible only when the use of multiple features leads to a feasible key size and table depth. It provides higher accuracy results and requires fewer operations at the last stage.

Clearly, no single solution fits all use-cases. The approach fits some classification, regression, and clustering models. Iterative models are less suitable, though they may be feasible, e.g., using recirculation. Importantly, the framework takes care that the type of feature or its range will not affect the accuracy of the classification (e.g., through normalization).

## 4.4 Retraining and Updates

ML models often need to be retrained and the resulting classification model needs to be updated. IIsy enables updating a model deployed on a switch using only table updates, without changes to the deployed program.

For a given use case, a user defines the features that need to be extracted, the type of the model, and, in the case of an ensemble model, the constraints on the model (e.g., number of trees). This leads to a generated P4 program for the network device. As long as the user maintains the constraints, retraining the model will not change the P4 program. Still, retraining will result in a different ML model, mapping to different actions in the features table, and in different code-to-classification entries in the tree and decision tables (as in Figure 3). These can be updated by table updates, a common management operation.

In a hybrid deployment, traffic can be directed to the backend during updates, to avoid misclassification.

## 5 FEATURE EXTRACTION

Network devices are designed to extract headers from packets. However, the research community has already gone beyond packet headers for applications ranging from telemetry [27] to in-network computing [50]. In this section, we discuss how features can be extracted from data on different levels of granularity. While we use PISA and P4-nomenclature[8], similar concepts apply to other targets.

## 5.1 Packet & flow level features

Extracting packet-level features is native to network devices, with packet header extraction done in the parser, and features are stateless. Such features include, for example, protocol type or source and destination port number. Packet level features also refer to features that describe the packet, such as packet size, switch source port, or timestamp.

Flow level features are stateful, and information is collected and stored across multiple packets. Examples of flow-level features include flow size, flow duration, and flow data rate. Heavy hitters detection is one line of research where flow-level features are already extracted and used [47].

We distinguish between two types of flow-level features: counted features (e.g., flow size, packets count), and time-related features (e.g., flow's start time, inter-packet gap).

## 5.2 Aggregate level features

Aggregate level features consider a group of flows,an aggregation of traffic (e.g., traffic from/to port $X$) or the network as a whole. Examples of features useful for ML purposes include traffic volume from a group of subnets, inter-arrival time toward a specific application or a histogram of source and destination ports. Aggregate level features are mostly similar in implementation to flow level features, however they may require additional operations, such as mapping flow-identifiers to an aggregated-feature identifier.

## 5.3 File level features

Extracting features from a file is more complex than any previous case, yet building upon flow-level feature extraction makes the challenge simpler. We distinguish between four stages of file processing:
• Start of a file, where file header needs to be processed, and initial resources need to be assigned. This is similar to a start of a flow but with a more complex parsing of the header.
• Looking into the file's payload. If a packet exceeds the size of the programmable data plane bus, then it may need to be recirculated (target dependent).
• Examining payload across packets. As a file is likely to be broken across many packets, extracting features from a file means that contents at the end of a previous packet need to be stitched with the contents at the head of the next packet.
• End of file. This is similar to the end of a flow and may allow to free up some resources.

To be clear, it is feasible to extract data from a subset of filetypes, not from all file types. Text-based files, such as txt, xml, html and csv, are straightforward to process. File types that use many objects, such as docx, pdf and xlsx, are very hard to process due to the complex structure and required resources. Certain image file types, e.g., png and tif, or audio files such as mp3, have a complex file structure and require significant switch resources, making them impractical. Video files, composed of frames of images, will be even harder to process. One exception to image file types is JPEG, which is relatively easier to process. Extracting a feature from the JPEG file, such as the average value or the value of a certain pixel, is possible. However, extracting more complex features will likely be beyond the resource budget [18].

Handling files raises other concerns. First, we assume that files are not encrypted (§2.2). Second, privacy and legal rights to process the data need to be addressed by the operator. Third, we assume no packet reordering, e.g., direct-attached SmartNIC or endpoint accelerator. Last, we assume file-type specific feature extraction.

We focus on two of the challenges raised by file processing: looking within the packet's payload and examining payload across packets. To look deep into the packet, it needs to be

recirculated, with bytes that were already processed stripped from the packet. This process is destructive, as the removed data can not be returned when the processing is done. Furthermore, recirculation can lead to significant bandwidth loss.

Examining payload across packets[2] creates resource challenges. Assume that you want to access a feature partly stored in the last two bytes of a packet and partly in the first 2 payload bytes of the next packet. You need to save the last 2 bytes of the packet in the memory (e.g., register) and the amount of data that was saved (i.e. two bytes). However, this information can be accessed only in a single place in the pipeline. If the last two bytes were saved at a stage toward the end of the pipeline, it would not be possible to extract this data until the new packet reaches this stage. One solution is recirculation, where the first pass through the pipeline extracts the missing information, potentially adding it as metadata or a header. Another solution is to extract this information in the Ingress pipeline and process it in the Egress pipeline, but this limits the program's functionality.

There is a trade-off in functionality, performance, and resource efficiency when applied to specific file-level use cases. As previous works have suggested, an easy get away is to manipulate the file sent at the host's side before entering the network so that some challenges can be avoided.

## 6  IMPLEMENTATION

IIsy's framework uses four components, as described in §3. In this section, we describe the implementation of our prototype. Further generalization of IIsy's framework is described in [59, 60].

The prototype's machine learning training framework is based on *scikit-learn* [37]. Our implementation enables fast development and prototyping of different models and, in particular, the hybrid approach. The training of the hybrid models used scikit-learn 0.24.1 and XGBoost 1.3.3, running over a c4.8xlarge AWS EC2 instance with 36 vCPUs and 60 GB RAM running Ubuntu 16.04 LTS.

The switch implementation run on two platforms: Intel's Barefoot Tofino (ASIC), and NetFPGA-SUME [63](FPGA). All the models are mapped to both targets, except for boosting, which targets only Tofino. The NetFPGA implementation enables exploring the limits of feature extraction. This includes also complex stateful features (§5), such as jitter, inter-arrival time, and data rate. On Tofino, packet-level, flow and aggregate features are supported, with further focus on files. Data is extracted from text files, both where the size of a feature is known (constant) and for unknown feature length (e.g., words separated by delimiters). Our implementation currently supports features of up to 15 ASCII characters

per feature. In addition, it supports features split between packets and features implemented deep within the packet (§7.1.2).

The simplicity of the mapping enables to auto-generate the data-plane and the control-plane, using a python script and a configuration file. A user defines in a configuration file design constraints, such as maximum number of trees, and the tool takes the output of the training stage (pickle file) and uses it to generate both the data plane (P4 files) and the control plane (table entries, in json format).

The system test environment uses $64\times100G$ ports Barefoot Tofino. P4-NetFPGA [22] is used for FPGA development. Four servers with 100G NICs are used to send and receive traffic from the switch. To test full throughput, we use a snake configuration, where traffic is looped from each port to the following one, enabling traffic across all 64 ports, which is a common practice [15]. As a baseline, we measure 6.2Tbps on the switch when running simple forwarding.

## 7  EVALUATION

In this section, we evaluate in-network classification for feasibility, performance, resource consumption and ML performance. For brevity, this section focuses on Intel Tofino, and details of the NetFPGA evaluation are provided in [57].

### 7.1  Use cases

Our evaluation is driven by two use cases: network anomaly detection using the UNSW-NB15 dataset [35], and time sensitive financial transactions using the Jane Street Market Prediction [19]. For each of these use cases, described below, we explore the classification performance of the switch alone, as well as part of a hybrid model4.2.

*7.1.1  Anomaly detection - Reducing back-end resource consumption.* Anomaly detection, such as intrusion detection and prevention, is typically done at the back-end and can consume significant compute or acceleration resources [58]. All network traffic toward certain application servers needs to be examined, and non-human or malicious traffic needs to be filtered. Our goal is to provide a scalable solution, whereby normal traffic is allowed in by the switch, and anomaly traffic is either dropped (where all decisions are taken by the switch) or sent to the back-end (in a hybrid mode). The aim of the system is to allow in all normal traffic, which is the majority of traffic volume. In the hybrid mode, traffic that is classified as anomalous or with low confidence is sent to the backend for deeper inspection. In this manner, the switch does not block (drop) legitimate traffic and offloads significant processing from the backend, as most traffic is normal. This use-case is an example where in-network classification saves resources compared with host-based solutions while also scaling with the network's bandwidth.

---

[2]For recirculated packets one can strip data on feature boundaries.

To demonstrate this use-case, we use the UNSW-NB15 dataset [35] that contains a mix of normal traffic and different types of attacks. The goal of the prediction is to detect attack traffic, which we label "anomaly" in our evaluation.

The use case is explored for the various ML models (§4), for feasibility study purposes. From ML perspective, Random Forest is the most suitable for this use-case, as it offers low variance in its classifications. This leads to a more predictable fraction of the traffic that is correctly classified as normal (unless the traffic distribution changes dramatically – which requires retraining the model).

Our learning uses 80% of the data for training and 20% for testing. The model running on the back-end is using a Random Forest of 200 trees (estimators) and 10,000 leaf nodes, and all the features in the dataset.

*7.1.2 Financial transactions - Reducing latency.* Low latency financial transactions, such as algorithmic trading, are very sensitive to latency. The lower the latency of a transaction, the higher the potential gain, even with slight reduction in latency. For top 10% financial traders, the a decision latency is less than 42 microseconds [4] from a passive order to an active transaction.

Typically, a large backend is used to provide real time classification for all transactions. In this case, the switch can be used to identify and tag high priority transactions, while other transactions are sent to the backend for fine-grain classification. The tagged high priority transactions can be forwarded to a different for immediate execution. Moreover, tagged queries can be prioritized over a dedicated link(s), avoiding congestion. While the switch may miss some high-priority transactions, those, in turn, will undergo the regular classification path. The gain in assigning many of the time-sensitive requests to a special fast processing path may introduce significant financial benefits with low resource consumption. This is an example where the advantage is the latency of classifying high-priority events, while the change in the backend's load is small.

To demonstrate this use-case, we use the Jane Street Market Prediction dataset [19], a recent trading finance dataset. Each trade in this dataset contains 130 anonymized features representing real stock market data and two output values ('weight' and 'resp') representing the trade's return. According to these two output values, we label the transactions in the dataset by recommended actions: 'Strong sell or buy', and 'Sell/Hold/Buy'. Financial transactions are typically a feed of individual trade instructions, arriving from the stock exchange. Such a feed is not openly available. The Jane Street dataset is the most recent and open information available from a trading company, presenting pre-processed transactions.

Our goal is to minimize the latency experienced by transactions marked as "strong or sell" (accounts for ≈13.1% of the total transactions). We assume the switch is located in such point in the network that any incoming transaction must go through it, so any classification by the switch has an additive latency of close to zero[3].

In terms of ML performance, while we evaluate with different models, the target for this use case is XGBoost, commonly used in financial applications as boosting offers a controlled bias which is more suitable for identifying minority.

Our learning uses 80% of the dataset for training and 20% for testing. The model running on the back-end is using all 130 features, with XGBoost of 100 trees (estimators) and a maximum depth of 8 (XGBoost trees tend to be shallow).

## 7.2 Feature Extraction

In the anomaly detection use-case, we implement on Tofino support for packet level features (e.g., source and destination port, protocol, service, and ports equivalence) and flow level features (e.g., duration, flow size in bytes and packets in each direction). While flow level features can improve the quality of the prediction, they cost two stages within the switch: to hash the flow ID, and to update a register holding the features value (e.g. flow size). Choosing between the two options requires weighting also other considerations, such as if flow ID is needed for "standard" processing purposes. Our resource consumption evaluation uses the features 'sport', 'dsport', 'proto', 'service', and 'is_sm _ips_ports' (Table 1), and the study of ensemble models (Table 3) uses the features 'sport', 'dsport', 'proto', 'service', 'sbytes'. This is as the 'sbytes' feature only improves the performance of the ensemble models.

The Jane Street dataset contains 130 numerical features, which we process two ways: either as a packet containing the features as numerical values, or in its original csv format, demonstrating the feasibility of file processing. For ease of exploration we reformat the file as columns of eight characters, but note that other implementations under this work are not of fixed size or known delimiter location. Both numerical and csv formats allow to explore feature extraction from deep within the packet. We succeed in extracting features from any of the 130 columns, without recirculation. The limitation to the extraction is not the location of the extracted feature, but rather the maximum number of features extracted and their size, which are limited by parser's resources. As financial transactions are typically a feed of individual trade instructions (§7.1.2), and the size of an entry in the Jane street dataset, with 130 columns, barely fits within an MTU packet (1522B), we send each transaction as

---

[3]Some settings use L1 switches, such as Cisco Nexus 3550. This is a different use case.

| Model | SVM | Bayes | KMeans | DT | RF | XGB |
|-------|-----|-------|--------|-----|-----|-----|
| Tables | 5 | 7 | 5 | 6 | 10 | 10 |
| Memory | 5.7% | 15.7% | 6.0% | 2.5% | 9.4% | 3.6% |
| Stages | 5 | 6 | 6 | 3 | 5 | 4 |
| Latency | 34.2% | 40.8% | 34.6% | 33.6% | 45.9% | 39.4% |
| F1 | 0.815 | 0.846 | 0.815 | 0.848 | 0.848 | 0.849 |

**Table 1: Anomaly Detection - Resource consumption of the models on Intel Tofino, using 5 features. 2 Stages are required for feature extraction and sending the packet. Resource Consumption is relative to switch.p4 reference program.**

| Model | SVM | Bayes | KMeans | DT | RF | XGB |
|-------|-----|-------|--------|-----|-----|-----|
| Tables | 5 | 7 | 5 | 6 | 15 | 9 |
| Memory | 1.8% | 14.7% | 1.7% | 12.1% | 5.5 % | 3.9% |
| Stages | 5 | 6 | 6 | 3 | 3 | 3 |
| Latency | 27.4% | 33.9% | 27.7% | 32.9% | 32.2% | 32.2% |
| F1 | 0.639 | 0.696 | 0.705 | 0.678 | 0.884 | 0.677 |

**Table 2: Financial Transactions - Resource consumption of the models on Intel Tofino, using 5 features. 1 stage is required for sending the packet. Resource Consumption is relative to switch.p4 reference program.**
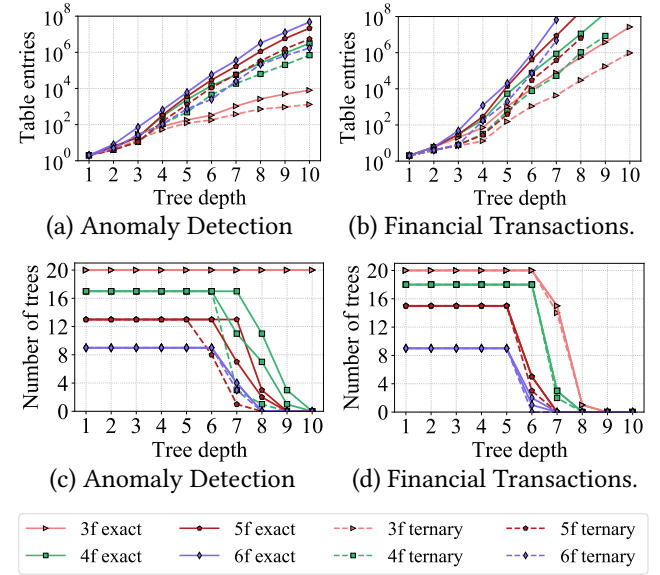
a separate packet. The features that we use are features 42, 43, 45, 124 and 126, demonstrating our ability to look deep into the data.

## 7.3 Resource Consumption

We run an exploratory experiment where the goal is to maximize the performance of the ML prediction, while still fitting the design within the ingress pipeline. SVM, Naïve Bayes, K-Means and Decision Tree (DT) fit within the ingress pipeline. While the Egress pipeline can be used, it is not recommended, as discussed in §8.

Table 1 and Table 2 summarize the resource consumption of the anomaly detection and financial transactions implementations, respectively. The tables show, for each model, the maximum size of the model that fits within an ingress pipeline switch using 5 features. The memory and latency are measured in comparison with Tofino's switch.p4 reference design. The number of tables does not directly map to the number of stages, and can be significantly higher. Our p4 programs see multiple feature tables mapped to the same stage.

As the results show, the memory requirements (all SRAM) are quite low in comparison with switch.p4, despite their potential to scale. This demonstrates the efficiency of our mapping algorithm. For Bayes, the results are relatively high as we maximize the accuracy in our features tables, and as we use two table to calculate the final probability (multiplication). In contrast, SVM and K-Means can have their results added up, without a decision, which saves significant resources.

## 7.4 Scalability

The size of a model that can be fit within a switch depends not only on the type of the model, but also on the dataset and its features. This is demonstrated in Figure 4 (a) & (b), which shows how memory requirements of a decision tree scale with the number of features for each of the use cases. These requirements depend on the depth of the feature tables, as



(a) Anomaly Detection    (b) Financial Transactions.

(c) Anomaly Detection    (d) Financial Transactions.

**Figure 4: Ensemble scaling with the number of features.**
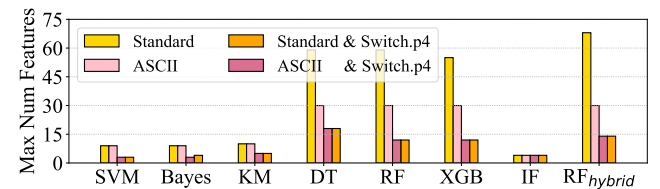


**Figure 5: The maximum number of features available in all IIsy's supported ML models under the financial transaction use case.**

well as on the depth of the decision table. In the finance use case, all the features are similar, and adding another feature increases the memory requirements roughly in a consistent manner. In the anomaly detection use case, on the other hand, features vary significantly in their memory requirement. For example, protocol type requires significantly less entries than source or destination port. Consequently, the anomaly detection use case requires less memory than financial transactions.

Fitting an ensemble model within a pipeline requires attending to several constraints: available memory (i.e., table size), number of stages in a pipeline, size constraints on metadata and lookup keys, and logic resources. The overall number of tables used is a soft constraint; as parallel lookups may happen within the same stage, the overall number of stages is a stronger constraint.

The difference in the model size that can be fit is demonstrated in Figure 4. The figure shows the number of trees that can be fit for each of the use cases, as well as their depth, depending on the number of features and the type of memory used (exact match, TCAM or a mix of both). As the figure shows, using up to 6 features, one can fit up to 20 trees. Increasing tree depth means that fewer trees can fit within the switch, due to the size of the decision table. Figure 5 shows the maximum number of features allowed in the Tofino pipeline under four implementation variations. Tree models are usually able to utilise more features compare to classical models due to stage sharing. Among these, the standard DT can fit up to 60 features which is due to the number of tables constrained per stage. This number of DT under ASCII implementation is reduced to 30 features, which is because of the depth limitation the parser can parse from the payload. As the results in Table 3 show, scaling the number of trees and features has a limited effect on performance, and therefore a smaller switch model may be more efficient when the hybrid model is possible. While these results are specific to the switch used, similar constraints exist on different targets.
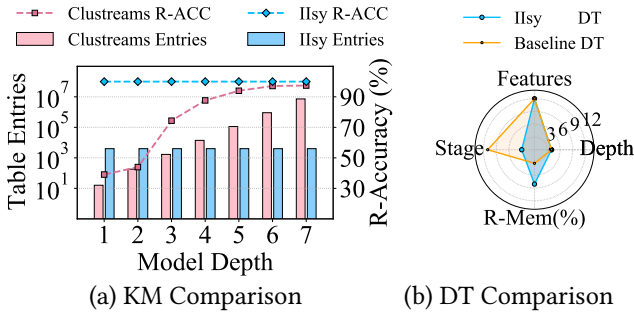


(a) KM Comparison      (b) DT Comparison

**Figure 6: Compare to Baseline - (a) IIsy KM compare to Clustreams KM in terms of Table Entries and (R)elative Accuracy (%). (b) The IIsy DT compare to SwitchTree DT in terms of Stage and (R)elative (Mem)ory (%)**

## 7.5 Baseline Comparison

We compare IIsy with two typical in-network Ml research, Clustreams and SwitchTree, in terms of resource consumption under financial transaction use case. As shown in Figure 6, to achieve the same accuracy, the IIsy's K-means algorithm

Anomaly Detection, Random Forest, 0.7 confidence

|  | Small | Medium | Large | Baseline |
|---|---|---|---|---|
| Features | 4 | 5 | 6 | 25 |
| Trees | 6 | 10 | 14 | 200 |
| Max Depth | 4 | 5 | 6 | — |
| Accuracy | 97.05 | 97.17 | 97.78 | 99.51 |
| Precision | 98.06 | 98.12 | 98.60 | 99.67 |
| Recall | 88.55 | 89.04 | 91.36 | 99.75 |
| F1 score | 92.60 | 92.94 | 94.58 | 98.88 |
| Hybrid Accuracy | 98.58 | 98.94 | 99.31 | — |
| Hybrid F1 | 96.64 | 97.53 | 98.41 | — |

Financial Transactions, XGBoost, 0.7 confidence

|  | Small | Medium | Large | Baseline |
|---|---|---|---|---|
| Features | 4 | 5 | 6 | 130 |
| Trees | 6 | 10 | 14 | 200 |
| Max Depth | 4 | 5 | 6 | – |
| Accuracy | 72.48 | 72.65 | 73.73 | 77.34 |
| Precision | 68.48 | 68.76 | 70.05 | 74.43 |
| Recall | 66.51 | 65.69 | 68.09 | 72.76 |
| F1 score | 67.16 | 65.51 | 68.78 | 73.43 |
| Hybrid Accuracy | 77.31 | 77.30 | 77.26 | — |
| Hybrid F1 | 73.41 | 73.43 | 73.40 | — |

**Table 3: Scalability of ensemble models and resulting ML performance.**



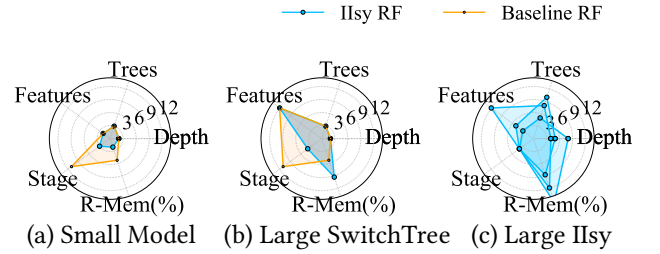(a) Small Model    (b) Large SwitchTree    (c) Large IIsy

**Figure 7: Compare to Baseline - The (R)elative (Mem)ory and stage consumption of IIsy RF and SwitchTree RF (Baseline) under three sets of hyperparameters.**

requires significantly less memory consumption compared to Clustreams. In Figure 6 (b), compared to the SwitchTree, IIsy's DT implementation requires significantly fewer stages (save 8) with only 5% more relative memory. When it comes to the ensemble model (i.e. RF), as shown in Figure 7 (a), with the small model size, IIsy shows merit in controlling both memory and stage consumption. For the maximum size available for SwitchTree (Figure 7 (b)), in comparison, with only 4% more memory requirement, IIsy has a significant benefit in stage consumption. In the larger model size (Figure 7 (c)), SwitchTree is unable to map to commodity switches,

however, IIsy is able to map and has excellent control over resource consumption.

## 7.6 Throughput and Latency

For both use cases, and for all models, the programs are designed to meet to line rate, with no recirculations or packet drop. The programs meet Tofino's timing for a minimum packet size of 100Gbps per port. For the anomaly detection scenario, we use UNSW's pcap traces [35], and for the financial transactions, we used the test dataset and send it over UDP. In both cases, we measure identical throughput to running the same traces through a simple layer 2 forwarding program, with no packet drop on any of the switch's 64 ports. Observe the throughput of the financial transaction use case, Figure 8 (a), which shows that the switch implementation achieves a 25 to 80000-fold throughput improvement over the CPU implementation.
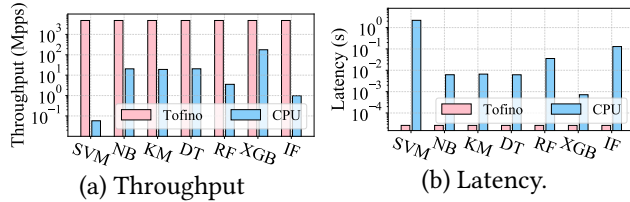


(a) Throughput

(b) Latency.

**Figure 8: Throughput and latency of ML algorithms on Tofino and CPU under financial transaction use case.**

The latency of a design is the number of clock cycles in the pipeline reported by the switch compiler, compared those to the switch.p4 reference design. For anomaly detection, the latency of the mapped models in the ingress pipeline is between 27.4% and 45.9% of the reference design. This means that mapping a model to a switch will have a negligible effect on latency if a switch is deployed in its native usage model (§2.2)[4]. A switch acting as an accelerator will add latency to the end-to-end traversal time at the scale of a microsecond [1]. Compared with the latency of current financial trading systems [4], as shown in Figure 8 (b), this will save an order of magnitude in latency. Operating on a stock exchange feed, the action on a classified packet can be the actual buy/sell order packet.

## 7.7 ML performance

We explore the performance of in-network classification for two scenarios: the classification is done solely in the switch, and the hybrid model. In the anomaly detection use case, the dataset is biased, meaning that most of the traffic is normal traffic. Although SVM, Naïve Bayes, and K-Means achieve a precision of 0.76−0.91 and F1 score of 0.81−0.85, they classify most of the anomaly traffic as normal. To correctly identify

---

[4]We do not claim to reduce the overall latency of the switch

anomalies, we focus on the ensemble models. In the finance use case, SVM, Naïve Bayes, and K-Means and achieve a precision of 0.70−0.73 and F1 score of 0.64−0.71.
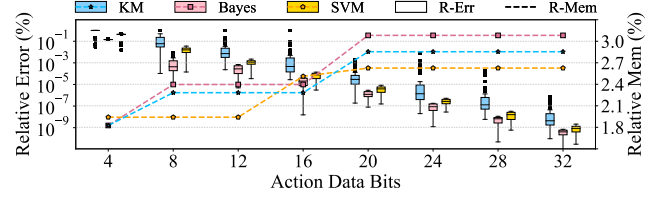


**Figure 9: Calculation error in SVM, Bayes and K-Means.**

Our implementations of non-tree based models may introduce an error. We study this error on two front: calculation error and classification error. The calculation error, shown in Figure 9 for the anomaly detection use case, is the relative error of a result calculated on a switch (e.g., hyperplane equation in SVM), compared with the same equation calculated on a server. While this error is small (less than 0.001%), the more important result is the misclassification due to calculation error: zero for SVM and K-Means, and 0.00003% for Naïve Bayes when action data bits is 16. This error is due to extremely low probabilities, and is further eliminated by coding the results of Naïve Bayes calculations, rather than normalizing values. Moreover, as shown in Figure 9, the increase in action data bits has a minor effect in terms of memory consumption but significantly reduces the calculation error.

Next, we consider the use of a hybrid deployment. The baseline is the full ensemble model running on back-end servers. We implement on the switch a smaller model that classifies a subset of the traffic, where all traffic not classified or classified with low confidence goes to the back-end as before. A confidence level is set in the switch to determine the threshold for classification on the switch. The performance of the small model, running on a switch alone, compared with the full model, is shown in Table 3.

Figure 10 (a) shows for the anomaly detection use case using Random Forest, the fraction of traffic offloaded by the switch and the corresponding misclassification rate, as a function of the switch classification confidence threshold. The baseline results in a misclassification rate of 0.49%, and F1 score of 0.9888. In comparison, with a confidence threshold of 0.7, 84.5% of the traffic is handled by the switch, achieving a misclassification rate of 1.03% and F1 score of 0.976. These improve as the confidence threshold increases, but the fraction of traffic handled by the switch decreases. Where Figure 10 (b) shows a similar result in Financial transaction use case.

Figure 11 presents the effect of confidence threshold on the performance of financial transactions use case. Figure 11)(a) shows that the XGBoost model running in the back-end, using 100 trees with a maximum depth of 8, achieves an error
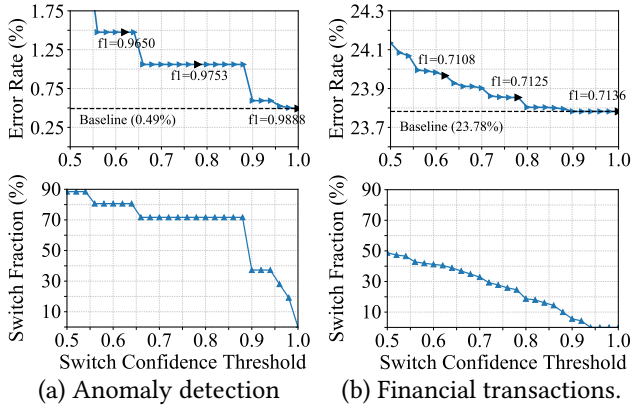
(a) Anomaly detection     (b) Financial transactions.

**Figure 10: Fraction of traffic handled by the switch and misclassification rate.**

rate of 0.244. In comparison, the hybrid model achieves an error rate 0.255 with a confidence threshold of 0.5. Increasing the confidence level to 0.65 reduces the error rate to 0.23. However, there is a trade-off here, shown in Figure 11)(b): with a threshold of 0.5, 23.9% of strong buy/sell transactions are classified by the switch, whereas at 0.66 confidence, only 3.25% of the transactions are classified by the switch. To put these results in context, consider Figure 11)(c), which shows the error rate for classifications done by the switch compared with the error rate for the same transactions, if done by the host. As the graph shows, transactions that achieve low confidence (below 0.55) on the switch, are also more likely (over 30%) to be misclassified by the full-grown model running on the server. In fact, starting 0.6 confidence threshold (where 10.8% of strong sell/ buy are being served by the switch model) the difference in error rate between the server and the switch is very small, and some traders may even find that the error difference at 0.54 is still small enough to provide higher transactions rate for 18.4% of the transactions.

## 7.8 Optimizations

The results presented in this section are the output of IIsy. These results can be further optimized by the user according to needs, either by changing IIsy's configurations (in most cases) or manually (rarely).

The easiest resource optimization for ensemble models is reducing the number of trees or their depth in the training stage, thereby reducing ML performance. As demonstrated in Table 3, in a hybrid deployment, there is a minor added benefit to using *Medium* or *Large* switch models over a *Small* model. This can further be tweaked by changing the confidence threshold, and is configurable.

Memory resources can be saved by binning, i.e. mapping more entries in the features tables to the same code. This in

turn reduces the size of classification tables and tree tables. This may lead to loss of ML performance and is controlled by a configuration.

Overall resources, including both memory and the number of tables, can also be reduced by combining feature tables, meaning using as a key to a table the concatenation of multiple features. This currently requires a manual change of one line in the p4 code and a small change in the mapping tool.

## 7.9 ML Model Updates

We measure the update time of our ML ensemble models on a switch. Our assumption is that due to changes in data over time, the result of the training has changed, but not the type of the model (e.g., XGBoost) or the constraints used for the training (e.g., maximum tree depth). The update time varies based on the size of the model: from $50ms$ for a small model, for several seconds for a large one (Table 3).

## 8 DISCUSSION

**Generalization** The focus of this paper is on the methodology of mapping ML models to network devices. IIsy uses a simple data plane, with complexity mainly in the algorithms mapping from the trained models to table entries. Porting between targets is straight forward, as was the case in porting between NetFPGA and Tofino. It requires syntax changes in the P4 code generator and a script generating control-plane commands, but not to the mapping tool. Planter [59, 60] builds upon IIsy to support more models and targets.

**Benefits** A lesson of this work is that despite resource constraints, network switches can serve as important classification components in hybrid deployments. Saving microseconds (or more) of latency in time-sensitive applications or reducing back-end servers load by tens of percent, without adding new hardware to the infrastructure is a key element. While classification can not be implemented within a fully utilized switch, our results show that the resource overheads of adding classification functionality to a switch are practical.

**Scope** This paper focuses on mapping trained ML models to network devices. The work does not seek to improve the quality of training ML models, nor to contribute to a specific use case. Applying the methodology to certain applications, such as congestion control, is beyond the scope of the paper. While the methodology offered in this work is can not be directly applied to neural networks models, our choice of ensemble models is because they provide the best results for the example use cases.

**Egress Pipeline** Using the egress pipeline is unlikely in most use cases, as typically the output port is determined in the ingress pipeline. While we explored using an egress pipeline, it did not improve the scalability of IIsy.
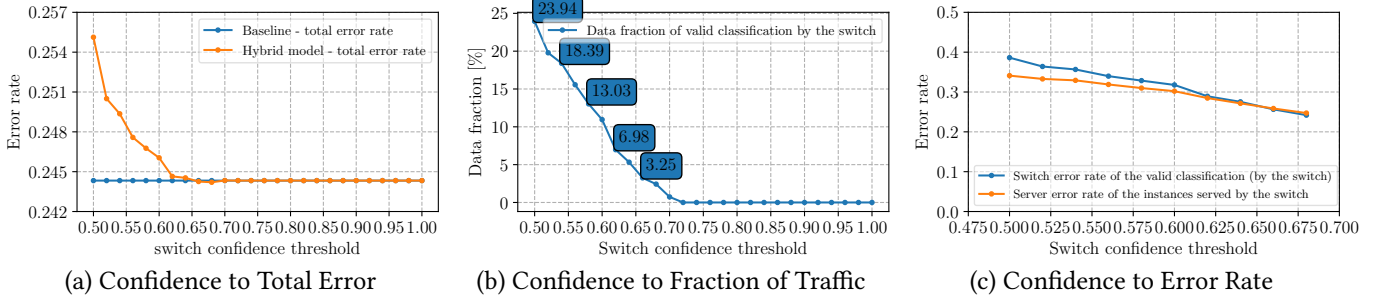
(a) Confidence to Total Error

(b) Confidence to Fraction of Traffic

(c) Confidence to Error Rate

**Figure 11: Latency sensitive financial transactions: The effect of confidence threshold**

| Project | Target | Models | OTS | Const. | OA |
|---|---|---|---|---|---|
| BaNaNa [40, 44] | RMT, NIC | BNN | ✓ | P | ✗ |
| N3IC [45, 46] | NIC, FPGA | BNN | ✓ | P | ✗ |
| Qin [38] | bmv2, NIC | BNN | ✓ | ✗ | ✓ |
| IOI[61] | ASIC | NN | ✗ | — | ✗ |
| iSwitch [30] | FPGA | RL | — | P | ✗ |
| Taurus [48] | ASIC | DNN, SVM, KM, LTSM | ✗ | ✓ | ✗ |
| pForest [12] | bmv2, ASIC | RF | ✓ | ✓ | ✗ |
| SwitchTree [29] | bmv2 | RF | — | ✗ | ✓ |
| NERDS[55] | bmv2, NIC | RF | ✓ | P | ?[5] |
| Clustreams [17] | ASIC | KM | ✓ | ✗ | ✗ |
| IIsy | ASIC, FPGA | SVM, KM, NB RF, XGB, IF | ✓ | ✓ | ✓ |

**Table 4: A comparison of in-network classification solutions. Legend: OTS - Off the shelf. Const. - Resource constrained. OA - Open Access. NN - Neural Network. BNN/DNN - Binary/Deep NN. RF - Random Forest. NB - Naïve Bayes. KM- K-Means. XGB - XGBoost. IF - Isolation Forest. P - Partial.**

**Limitations** Some of the limitations discussed in this work, e.g., the number of tables or features, are property of the target platform and will change on a different platform. For example, NetFPGA is mostly limited by memory and logic resources, while on Tofino, memory and logic resources are rarely limiting us, and we are limited by different constraints, such as the number of stages.

## 9  RELATED WORK

The application of ML to network traffic, and in particular the use of ML for traffic classification, has been of an interest for a long time (e.g., [14, 33]). Using ML for scheduling and congestion control (e.g., [16]) was also studied. The focus of most of these works has been on using ML over traditional computing platforms.

The challenges of ML have led researchers to explore new approaches to resource constrained ML, using devices of limited resources [54]. Such approaches are popular with IoT

---

devices (e.g., [39]). This work focuses on network switches, which have more resources than some of these devices, and also much higher processing rate and a different architecture.

A related thread of research is using programmable switches to accelerate ML frameworks. These works focus on parameters servers and in-network aggregation [28, 41] in the training stage, rather than the classification.

A few works tried to implement ML models within network devices. Implementing binary neural networks was explored in N2Net [44] and BaNaNa Split [40], with limited performance benefits. pForest [12], SwitchTree [29] and NERDS [55] explored mapping random forests, with only pForest attempting ASIC implementation. Their methodology is different, encoding each decision tree in separate tables, with a table for every tree level, achieving lower scalability (e.g., depth of 4 in pForest). Li [30] added an acceleration module within a switch for reinforced learning, and Taurus[48] suggested adding a map-reduce module. Taurus did not report ML performance results for the models supported by IIsy. IIsy's use of unmodified network devices is complementary to these works.

Using programmable network devices for anomaly detection was explored both at the host side [58] and within switch-ASIC (e.g., [31]). Our work is orthogonal to these non-ML based efforts, as it enables ML based anomaly detection solutions (e.g., [23]) to be migrated to the network.

ML for financial transactions has been widely researched, with XGBoost and SVM often used [51]. Acceleration of financial transactions has mostly focused on the back-end, e.g. using FPGA [53]. The closest programmable switches project is the publish-subscribe system [24], for the NASDAQ Market data feed filter and router.

## 10  CONCLUSION

The toll of running ML workloads is high. In this paper, we have made the case for in-network classification. By mapping multiple ML models, including ensemble models, to

---

[5]The repository linked by the authors is not available at this time

off-the-shelf network switch, we have demonstrated the feasibility and benefits of running classification within network devices.

This paper complies with all applicable ethical standards of the authors' home institution.

## REFERENCES

[1] Arista. 2018. *Arista 7170 Multi-function Programmable Networking*. White Paper, https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf.

[2] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. 2017. An In-depth Performance Characterization of CPU- and GPU-based DNN Training on Modern Architectures. In *MLHPC*. ACM.

[3] AWS. 2019. New – TLS Termination for Network Load Balancers. Retrieved January 2021 from https://aws.amazon.com/blogs/aws/new-tls-termination-for-network-load-balancers/

[4] Matthew Baron, Jonathan Brogaard, Björn Hagströmer, and Andrei Kirilenko. 2019. Risk and Return in High-Frequency Trading. *JFQA* 54, 3 (2019), 993–1024.

[5] Diogo Barradas, Nuno Santos, Luıs Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. 2021. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *NDSS*.

[6] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, et al. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *TRETS* (2018).

[7] Amirali Boroumand, Saugata Ghose, Youngsok Kim, et al. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*. 316–331.

[8] 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. *SIGCOMM CCR* 44, 3 (2014), 87–95.

[9] Gordon Brebner. 2018. Extending the Range of P4 Programmability. In *Keynote in the EuroP4*.

[10] Leo Breiman. 2001. Random Forests. *Machine learning* 45, 1 (2001), 5–32.

[11] Businesswire. 2020. *InsidePacket extends SONiC use cases, enabling new edge-cloud network services.*

[12] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pForest: In-Network Inference with Random Forests. *arXiv:1909.05680* (2019).

[13] Stephen Cass. 2019. Taking AI to the edge: Google's TPU now comes in a maker-friendly package. *IEEE Spectrum* 56, 5 (2019), 16–17.

[14] Alberto Dainotti, Antonio Pescape, and Kimberly C Claffy. 2012. Issues and Future Directions in Traffic Classification. *IEEE network* 26, 1 (2012), 35–40.

[15] 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 TON* 28, 4 (2020), 1726–1738.

[16] Vojislav Dukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *NSDI*. 565–580.

[17] Roy Friedman, Or Goaz, and Ori Rottenstreich. 2021. Clustreams: Data Plane Clustering. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 101–107.

[18] René Glebke, Johannes Krude, Ike Kunze, Jan Rüth, Felix Senger, and Klaus Wehrle. 2019. Towards Executing Computer Vision Functionality on Programmable Network Devices. In *ENCP*.

[19] Jane Street Group. 2020. *Jane Street Market Prediction*. [Online; accessed January 2021].

[20] David J Hand and Keming Yu. 2001. Idiot's Bayes-Not So Stupid After All? *International statistical review* 69, 3 (2001), 385–398.

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*. IEEE, 620–629.

[22] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilbermann. 2019. The P4→NetFPGA Workflow for Line-Rate Packet Processing. In *FPGA*. ACM, 1–9.

[23] MohammadNoor Injadat, Fadi Salo, Ali Bou Nassif, Aleksander Essex, and Abdallah Shami. 2018. Bayesian Optimization with Machine Learning Algorithms Towards Anomaly Detection. In *GLOBECOM*. IEEE.

[24] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and Routing with Packet Subscriptions. In *CoNEXT*. 282–294.

[25] 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 *SOSP*. ACM, 121–136.

[26] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. IEEE, 1–12.

[27] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band Network Telemetry via Programmable Dataplanes. In *SIGCOMM*.

[28] C. Lao, Y Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*.

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

[30] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-switch Computing. In *ISCA*. 279–291.

[31] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security*.

[32] Melvin Earl Maron. 1961. Automatic Indexing: An Experimental Inquiry. *J. ACM* 8, 3 (1961), 404–417.

[33] Andrew W Moore and Denis Zuev. 2005. Internet Traffic Classification Using Bayesian Analysis Techniques. In *SIGMETRICS PER*, Vol. 33. ACM, 50–60.

[34] Nour Moustafa. 2015. *The UNSW-NB15 Dataset Description*. [Online; accessed January 2021].

[35] Nour Moustafa and Jill Slay. 2015. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *MilCIS*. IEEE.

[36] NGINX. [n.d.]. SSL Termination for TCP Upstream Servers. Retrieved May 2021 from https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-tcp/

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay.

2011. Scikit-learn: Machine Learning in Python . *JMLR* 12 (2011), 2825–2830.

[38] 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 *IFIP Networking*. IEEE.

[39] Ramon Sanchez-Iborra and Antonio F Skarmeta. 2020. TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities. *IEEE Circuits and Systems Magazine* (2020).

[40] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network be the AI Accelerator?. In *NetCompute*.

[41] 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 *NSDI*.

[42] Robert E Schapire and Yoav Freund. 2013. Boosting: Foundations and Algorithms. *Kybernetes* (2013).

[43] Carlos N Silla and Alex A Freitas. 2011. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery* 22, 1 (2011).

[44] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network Neural Networks. *arXiv:1801.05731* (2018).

[45] 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.

[46] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. 2020. Running Neural Networks on the NIC. *arXiv:2009.02353* (2020).

[47] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *SOSR*. ACM, 164–176.

[48] 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.

[49] Jiliang Tang, Salem Alelyani, and Huan Liu. 2014. Feature Selection for Classification: A Review. *Data classification: Algorithms and applications* (2014), 37.

[50] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *EuroSys*.

[51] Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. 2017. Using Deep Learning to Detect Price Change Indications in Financial Markets. In *EUSIPCO*. IEEE, 2511–2515.

[52] Shay Vargaftik, Isaac Keslassy, Ariel Orda, and Yaniv Ben-Itzhak. 2021. RADE: Resource-efficient supervised anomaly detection using decision tree-based ensemble methods. *Springer Machine Learning* (2021), 1–32.

[53] Raja Velu. 2020. *Algorithmic Trading and Quantitative Strategies*. CRC Press.

[54] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. 2019. Adaptive Federated Learning in Resource Constrained Edge Computing Systems. *J-SAC* 37, 6 (2019), 1205–1221.

[55] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. 2021. Programmable Switches for in-Networking Classification. In *INFOCOM WKSHPS CCC*. IEEE, 1–10.

[56] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *HotNets*. 25–33.

[57] Zhaoqi Xiong and Noa Zilberman. 2019. IIsy's GitHub Repository. https://github.com/cucl-srg/IIsy

[58] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *OSDI*.

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

[60] Changgang Zheng and Noa Zilberman. 2021. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 12–14.

[61] Zhizhen Zhong, Weiyang Wang, Manya Ghobadi, Alexander Sludds, Ryan Hamerly, Liane Bernstein, and Dirk Englund. 2021. IOI: In-network Optical Inference. In *SIGCOMM OptSys*.

[62] Zhi-Hua Zhou. 2012. *Ensemble Methods: Foundations and Algorithms*. CRC press.

[63] Noa Zilberman, Yury Audzevich, G.Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (2014), 32–41.

# A MAPPING MODELS - ADDITIONAL INFORMATION

## A.1 SVM

Support vector machines (SVM) use hyperplanes to separate between classes, where the output of the training stage is the equations of the hyperplanes , such as:

$$\begin{cases} a_1 x_1 + b_1 x_2 + \dots z_1 x_n + d_1 = 0 \\ a_2 x_1 + b_2 x_2 + \dots z_2 x_n + d_2 = 0 \\ \quad \dots \\ a_m x_1 + b_k x_2 + \dots z_m x_n + d_m = 0 \end{cases}$$

where $x_i$ is the value of feature i, $n$ is the number of features, $k$ is the number of classes and $m = k * (k - 1)/2$.

There are two ways to map SVM to a network device. First, to hold a table per feature, and second, to hold a table per hyperplane. A table per feature means that the key to a table is the feature's value, and the output of the lookup is a vector of calculated values $a_i \times x_i$. The value of each hyperplane is calculated as the sum of vectors from all feature tables, and a decision is taken. This can be optimized by adding up the features in each pipeline stage. A table per hyperplane
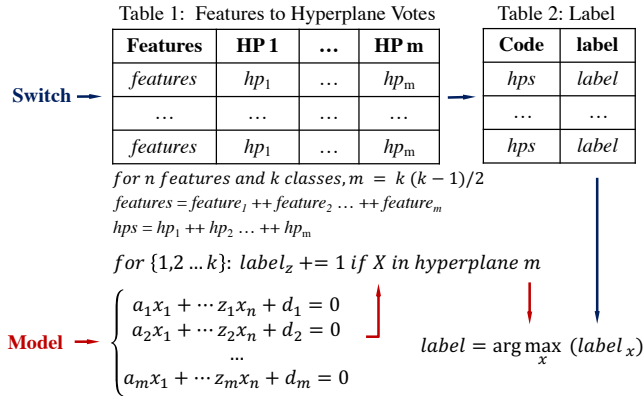
the concatenation of all features can yield the classification within a single table. However, this table is likely to be very large and less resource-efficient than distributing across a few smaller tables.

The main advantage of the table per hyperplane approach is that there is no unintentional loss of accuracy; the output of the table is a vote, not a value.It is possible to purposely lose some accuracy, e.g., if one wants to reduce the number of table entries by merging multiple ranges of different "votes" into a single entry (e.g., if keys 0-1111 and 1113-32767 are mapped to class 1, and key 1112 to class 2). In contrast, a solution using a table per feature may lose some accuracy, as the result of a lookup is a calculated value (and not a code), which has an accuracy limited by its number of bits. The final classification decision may not be affected by the loss of accuracy in calculations along the pipe, but this is not guaranteed.

Using a table per feature will still be favored in some cases, e.g., if working with eight features, each of eight-bit, so each table is only (and at most) 256 entries deep, and features can be looked up in parallel. The table per hyperplane equivalent will be multiple ($m$) tables of a 64-bit key.



| Features | HP 1 | ... | HP m |
|---|---|---|---|
| *features* | $hp_1$ | ... | $hp_m$ |
| ... | ... | ... | ... |
| *features* | $hp_1$ | ... | $hp_m$ |

Table 1: Features to Hyperplane Votes

| Code | label |
|---|---|
| *hps* | *label* |
| ... | ... |
| *hps* | *label* |

Table 2: Label

*for n features and k classes, $m = k(k-1)/2$*
*features = feature₁ ++ feature₂ ... ++ featureₘ*
*hps = hp₁ ++ hp₂ ... ++ hpₘ*

*for {1,2 ... k}: labelz += 1 if X in hyperplane m*

$$\begin{cases} a_1 x_1 + \cdots z_1 x_n + d_1 = 0 \\ a_2 x_1 + \cdots z_2 x_n + d_2 = 0 \\ \quad \dots \\ a_m x_1 + \cdots z_m x_n + d_m = 0 \end{cases}$$

$$label = \arg\max_x (label_x)$$

**Figure 12: Mapping an SVM Model.**

means that $m$ lookup tables are used, one per hyperplane and the outcome of the lookup indicates on which side of a hyperplane is a given input. The key to a table is a set of features, and the action is a "vote". A "vote" is a one-bit value mapped to the metadata bus that indicates if the input belongs within or outside a hyperplane. The "votes" from all $m$ tables are counted in the last stage, and the class with the highest count of "votes" is the classification result.

The table per hyperplane approach is feasible only when the concatenation of all features does not lead to a too wide key. If the features used are, for example, source and destination port, protocol, and some IP flags, the key will be relatively small, and the solution will be feasible. Theoretically,

| Features | $xa_1$ | ... | $xa_m$ |
|---|---|---|---|
| *feature₁* | $xa_1$ | ... | $xa_m$ |
| ... | ... | ... | ... |
| *feature₁* | $xa_1$ | ... | $xa_m$ |

| Features | $xz_1$ | ... | $xz_m$ |
|---|---|---|---|
| *featureₙ* | $xz_1$ | ... | $xz_m$ |
| ... | ... | ... | ... |
| *featureₙ* | $xz_1$ | ... | $xz_m$ |

Table 1 to Table n: Features 1 to Feature n to polynomial components
*for n features and k classes, $m = k(k-1)/2$*

*for {1,2 ... k}: labelz += 1 if X in hyperplane m*

$$\begin{cases} a_1 x_1 + \cdots z_1 x_n + d_1 = 0 \\ a_2 x_1 + \cdots z_2 x_n + d_2 = 0 \\ \quad \dots \\ a_m x_1 + \cdots z_m x_n + d_m = 0 \end{cases}$$

$$label = \arg\max_x (label_x)$$

**Figure 13: Mapping an SVM Model (Second Option)**

## A.2 Naïve Bayes

For a Naïve Bayes classifier [32], we assume a Gaussian distribution of independent features [20]. Similar concepts apply to related methods, such as kernel estimation [33]. Under this assumption, the likelihood of feature $x_i$ is expressed as:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

And the classification rule is:

$$\hat{y} = arg\,max_y P(y) \prod_{i=1}^{n} P(x_i|y)$$

If there are $n$ features and $k$ classes, there are $k \times n$ pairs of $(\mu_y, \sigma_y)$.



Table 1: Class 1 Probability

| Features | Class 1 |
|----------|---------|
| *features* | *prob₁* |
| ... | ... |
| *features* | *prob₁* |

Table k: Class k Probability

| Features | Class K |
|----------|---------|
| *features* | *probₖ* |
| ... | ... |
| *features* | *probₖ* |

$features = feature_1 \,{+}{+}\, feature_2 \ldots {+}{+}\, feature_m$

$$label = \arg\max_y P(y) \prod_{i=1}^{n} p(x_i|y) \quad \longrightarrow \quad label = \arg\max_x P_x$$

$$p(x_i \,|\, y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2})$$
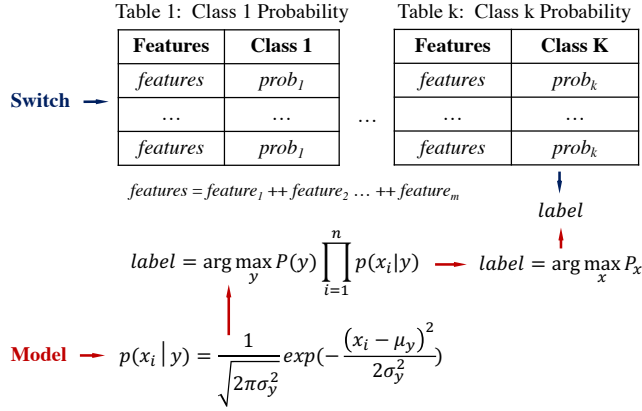
**Figure 14: Mapping a Bayes Model**

A mapping based on a table per feature is possible but can be both inefficient and inaccurate. Here, the result of each feature-value lookup will be a vector of probabilities. As the number of bits per vector is limited, there will be some accuracy loss. Even if the target allows for any vector length and a fixed point notation is used, the amount of metadata that needs to be carried between stages will be higher than other solutions, and depending on the number of features and classes, exceeding allowed resources. In this approach, each class will require a table at the end of the pipe to calculate its overall probability, bringing the overall number of tables required to $O(n + k)$. Unless there is a compromise on accuracy, the number of entries in each such table will be large, as the key is the concatenation of all probabilities per class. Finally, at the end of the pipeline, a comparison is required to find $max_y P(y)$.
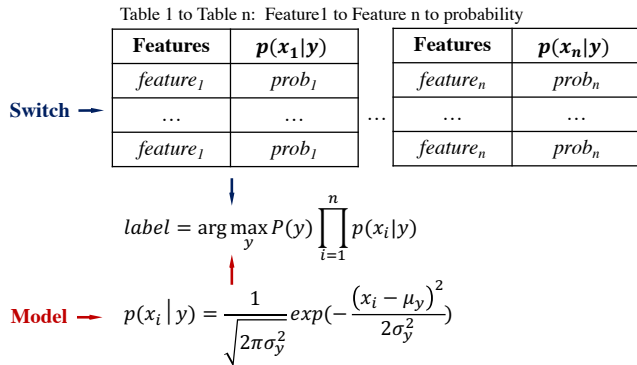


Table 1 to Table n: Feature1 to Feature n to probability

| Features | $p(x_1|y)$ |
|----------|-----------|
| *feature₁* | *prob₁* |
| ... | ... |
| *feature₁* | *prob₁* |

| Features | $p(x_n|y)$ |
|----------|-----------|
| *featureₙ* | *probₙ* |
| ... | ... |
| *featureₙ* | *probₙ* |

$$label = \arg\max_y P(y) \prod_{i=1}^{n} p(x_i|y)$$

$$p(x_i \,|\, y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2})$$

**Figure 15: Mapping a Bayes Model (Second Option)**

## B MAPPING MODELS - ADDITIONAL INFORMATION

A better approach is to use one table per class, with all the features as the key, and with the result being the probability of that class. The disadvantage is the size of the required table: it uses a very wide key (a form of a concatenation of all input features values), and its depth is proportional to this width unless a compromise is made for accuracy. The resulting probability does not need to be presented as a fraction, and an integer value can be used that symbolizes the probability. As long as the same notation is used across all tables, the final comparison of $P(y)$ and the classification result will be correct. Figure 14 in Appendix A illustrates this implementation.

### B.1 K-Means

An example of unsupervised learning mapped to a network device uses K-means clustering. In K-means, $k$ classes are represented by $k$ centers of clusters, with each center defined by $n$ coordinate values, one per feature. A data point will be mapped to a class based on its nearest center of a cluster. The distance from cluster $i$ is denoted by:

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

where $x_1$ to $x_n$ are the values of the data point's features. Obviously, to find the nearest cluster, it is sufficient to consider the square distances.

As in previous examples, there are two ways to map the model to a network device. One option is using a table per feature, with the lookup's result of table $i$ being a vector of $\{(x_i - c_i^1)^2, (x_i - c_i^2)^2, ..., (x_i - c_i^k)^2\}$. Here, the last stage will need to sum up all $D_i$ and find the smallest one[6]. As before, the challenges here are the accuracy of the calculation and the width of the required metadata bus.

The second approach uses a table per class, with the key being the concatenation of all features (presenting a challenge of key width). The result of each such table lookup is the distance of the data point from the center of the cluster. As proposed above, this distance can be consistently represented by an integer value across all tables, allowing for easy comparison and selection at the last stage. The approaches are illustrated in Appendix A.

## C FEATURE EXTRACTION ON NETFPGA

Using our implementation on NetFPGA, we explore the feasibility of implementing the full range of features included in the UNSW dataset [34], regardless of their contribution to the Random Forest model's performance.

---

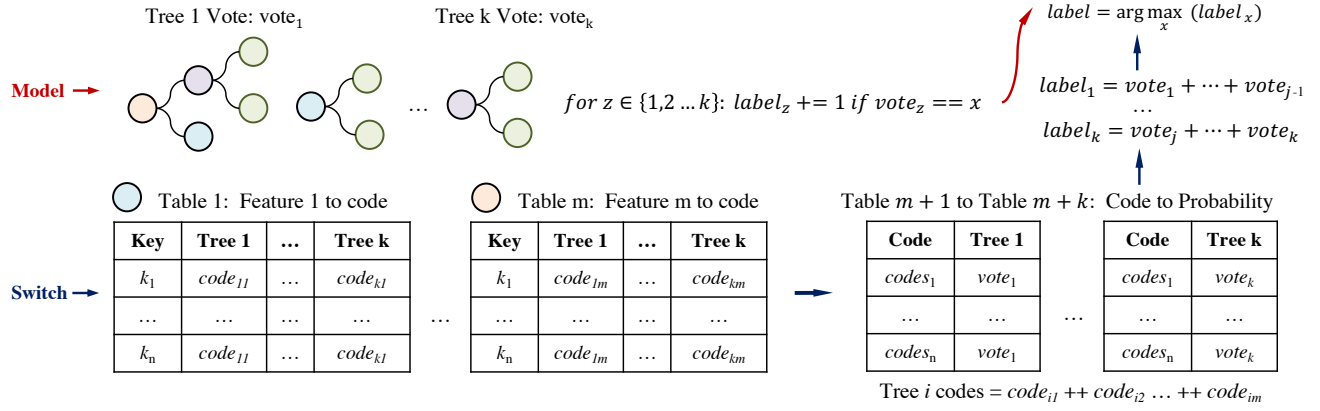[6]Values can also be summed up in each stage.
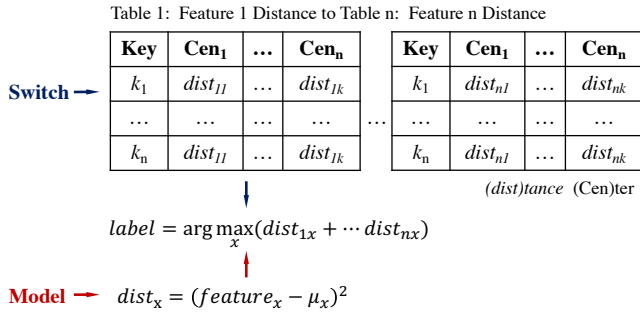
Figure 16: Mapping a Random Forest Model



Figure 17: Mapping a K-means Model

The dataset includes 47 features, which we categorize into three groups according to their implementation complexity, with a fourth group including with unclear description. We classify 11 features as easy, 15 features as medium complexity and 18 features as hard or impossible to implement on a switch. Three more features are unclear. Out of these, we implement 15 features on NetFPGA.

In general, easy features are stateless, though some require operating on the data, e.g., comparison. All the features implemented under this class are packet-level features.

The medium level features are stateful, and include both counted and time related flow level features. Among the features that we implement under this category we include flow duration, flow record start time and last time, packet inter-arrival time, packets loss per flow, and data rate (bits per second). For the data rate, we use both flow duration and a counter of bytes per flow, and estimate data rate by the ratio of byte per flow to duration. The most complex feature implemented is jitter, where we use an multiple bins to store packets inter arrival time per flow, and consider the number of packet in each bin according to the classifier.

The hard to impossible features require tracking state machines, and information that is not available on the switch.

The implementation that supports all features uses 13 externs in total, where some of the extrens are used by more than one feature (with a single access in the pipeline). The types of externs use are hash, read/write memory (equivalent to registers) and read-modify-write memory (atomic operation). The types of externs available on NetFPGA are described in [22].

Extracting a jitter feature illustrates the complexity of some flow-level features. The jitter feature, in the context of this example , considers the difference in inter-arrival time between packets of the same flow. As a new packet arrives, its flow-identifier fields are extracted in the Parser, and an extern (e.g., timestamp mechanism) is used to get the arrival time. A hashing function turns the flow identifier fields into a flow-id. These steps are common in non-ML use cases. Next, the previous arrival time in the flow is read from a memory, and replaced with the new timestamp (e.g., using read-modify-write). The difference between the current and the previous arrival time is the inter-arrival gap. To account for the jitter, bins can be used, e.g., number of packets with less than 1ms, 1ms to 10ms, or more than 10ms inter-arrival time. Each bin requires a per-flow entry in the memory. Given $K$ flows and $N$ bins means that the jitter feature will required $K \times (N + 1)$ memory entries.

## D RELATED WORK - EXTENSION

Figure 19 illustrates the positioning of IIsy's contribution relative to the works listed in Table 4. As the figure shows, IIsy is the only work to present a generic solution for a range of ML methods.

While multiple works have considered Random Forest, they focused on bmv2 and smart NICs. pForest has also focused on bmv2, and presented a non-optimized implementation on Tofino with a depth of 4, significantly less than IIsy. Solutions such as [29, 55] did not attend to resource
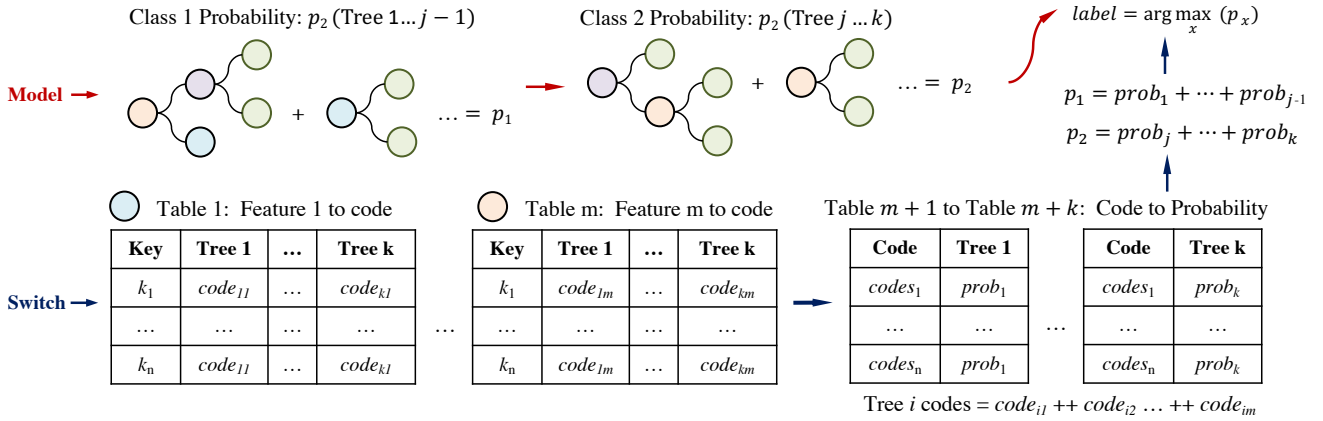
Class 1 Probability: $p_2$ (Tree $1...j-1$)          Class 2 Probability: $p_2$ (Tree $j...k$)          $label = \arg\max_{x} (p_x)$

$... = p_1$          $... = p_2$

$p_1 = prob_1 + \cdots + prob_{j-1}$

$p_2 = prob_j + \cdots + prob_k$

Table 1: Feature 1 to code          Table $m$: Feature m to code          Table $m+1$ to Table $m+k$: Code to Probability

| Key | Tree 1 | ... | Tree k |
|-----|--------|-----|--------|
| $k_1$ | $code_{11}$ | ... | $code_{k1}$ |
| ... | ... | ... | ... |
| $k_n$ | $code_{11}$ | ... | $code_{k1}$ |

| Key | Tree 1 | ... | Tree k |
|-----|--------|-----|--------|
| $k_1$ | $code_{1m}$ | ... | $code_{km}$ |
| ... | ... | ... | ... |
| $k_n$ | $code_{1m}$ | ... | $code_{km}$ |

| Code | Tree 1 |
|------|--------|
| $codes_1$ | $prob_1$ |
| ... | ... |
| $codes_n$ | $prob_1$ |

| Code | Tree k |
|------|--------|
| $codes_1$ | $prob_k$ |
| ... | ... |
| $codes_n$ | $prob_k$ |

Tree $i$ codes $= code_{i1} ++ code_{i2} \ldots ++ code_{im}$

**Figure 18: Mapping an XGBoost Model.**

constraints, and don't scale as well as IIsy. Even in the bmv2 implementation, SwitchTree [29] was studied with only 5 trees and depth of 10, where each tree is coded independently.

While Taurus supports SVM and KMeans, it is not possible to compare to it, as it did not report ML performance results for these models. Moreover, it relies on a modification to the silicon design.



**Figure 19: The positioning of IIsy**

## E   TEST SETUP

Our system test environment uses APS-Networks BF6064X, an Intel Barefoot Tofino platform with $64 \times 100G$ ports. Barefoot's SDE 9.2.0 is used on the switch, and we further experiment with SDE 9.6.0 in the software development environment. P4-NetFPGA [22] with SDNet 2018.2 compiler is used for the FPGA development.

ESC4000A-E10 servers using AMD EPYC 7302P CPUs with 256GB RAM, Ubuntu 20.04LTS, and equipped with Mellanox ConnectX-5 100G NICs are used to send traffic to the switch using DPDK 20.11.1 and PktGen 21.03.0. Four CPU cores are dedicated per port.

To test full throughput, we use a snake configuration, where traffic is looped from each port to the following one, enabling traffic across all 64 ports, which is a common practice [15]. A set of python scripts is used to generate, capture and check traffic. As a baseline, we measure 6.2Tbps on the switch when running simple forwarding.

In this section we include additional illustrations of mapping different machine learning models to network devices. The methodolog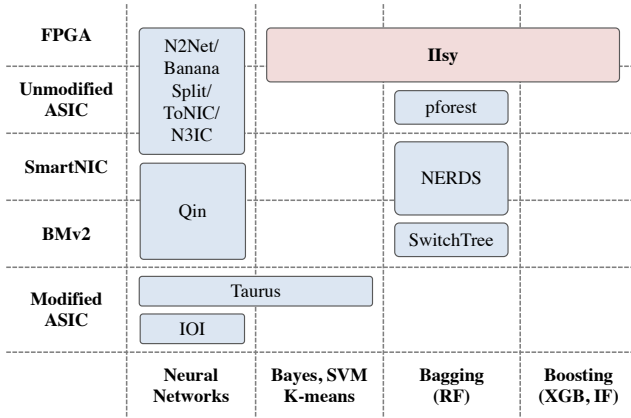y of mapping these models is described in 4.