Planter: Seeding Trees Within Switches

Changgang Zheng
University of Oxford
changgang.zheng@eng.ox.ac.uk

Noa Zilberman
University of Oxford
noa.zilberman@eng.ox.ac.uk

ABSTRACT

Data classification within the network brings significant benefits in reaction time, servers offload and power efficiency. Still, only very simple models were mapped to the network. In-network classification will be useful only if we manage to map complex machine learning models to network devices. We present Planter, an algorithm for an efficient mapping of ensemble models, such as XGBoost and Random Forest, to programmable switches. By overlapping trees within match-action tables, Planter maps ensemble models to programmable switches with high accuracy and low resource overhead.

KEYWORDS

In-network computing; Machine learning; Classification; Programmable switches; P4

1 INTRODUCTION

The Internet is playing an increasingly dominant role in our lives. With more and more connected IoT devices, the amount of data flowing within the Internet is rapidly growing [8]. The traditional model of using back-end servers to process the traffic imposes a further burden on the network.

In-network computing provides the means to mitigate the problem, processing the data within network devices, as it moves through the network. In-network computing offers reduced processing delay, improved processing efficiency, and reduced network load [15].

Machine learning algorithms have long been applied to network data, e.g., for traffic classification [13] and anomaly detection [7]. However, doing so within the network is challenging. Not only network devices are resource constrained, but their architecture doesn't lend itself easily to machine learning scale and complexity.

Ensemble models, such as Random Forest [2] and XGBoost [4], are widely used in machine learning, as the combination of multiple learning models allows to improve prediction results [5, 17]. As decision trees can fit within network devices [16], the challenge becomes fitting tree-based ensemble models within the resource constrains of a switch.

This work presents Planter, an algorithm that efficiently maps tree-based ensemble models to commercially available programmable switches. It supports a variety of ensemble models such as Random Forest, XGBoost, and Isolation Forest. Planter has a low resource overhead, and minimal loss of accuracy (less than 2%) compared with fully-grown ensemble models running on a server.

2 MAPPING DECISION TREES

Running classification based on a decision tree or an ensemble model within a PISA-style switch [1] means not only mapping the model to a match-action architecture, but also attending to three constraints: 1) limited amount of memory [10] 2) limited number of stages and 3) limited mathematical operations.

One approach to mapping tree-based models to switches, used by SwitchTree [11] and pForest [3], uses a match-action stage for each level in the tree. As shown in Figure 1(a), The number of stages matches the depth of a tree. At each stage, the previous decision and branch ID decide the next branch ID or the classification result. However, these approaches consume a large number of logic operations and stages. Furthermore, each tree in an ensemble model is implemented independently, limiting the number of trees per pipeline, as a pipeline has only 12-20 stages [9].

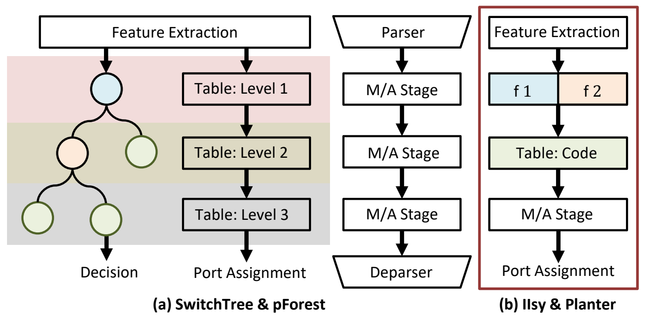

Figure 1: The difference in mapping a decision tree to a match-action pipeline between (a) SwitchTree & pForest and (b) IIsy & Planter.

In IIsy [16], we proposed a different approach to mapping decision trees, which encodes the model using a table per feature, coding the decision into the tree level, as shown in Figure 1(b). Planter extends this idea, offering an algorithm that can both map ensemble models to switches, and do so efficiently. In Planter, the number of stages is independent of a tree's depth.

3 ENSEMBLE MODELS IN PLANTER

Planter maps ensemble models to programmable switches by overlapping trees encoding within feature tables, and using a single table per tree to decode a tree's decision. In this section we explain the algorithm.

In Planter, a forest model consisting of N trees with M features requires $M+N$ match-action (M/A) tables. The first M tables, named feature tables, use the value of a feature as the key. The action of the feature table is divided into multiple fields, each with a code indicating the respective branches taken in each of the tree models. The following N tables, named code tables, use as the key to the table the concatenated code (per tree) of all features. The resulting action is the leaf node (decision) of the tree. There is an alternative form of the Planter algorithm which requires $M + 1$ tables, with a single code table for all trees. This alternative is feasible only when the number of entries in the code table is small enough.

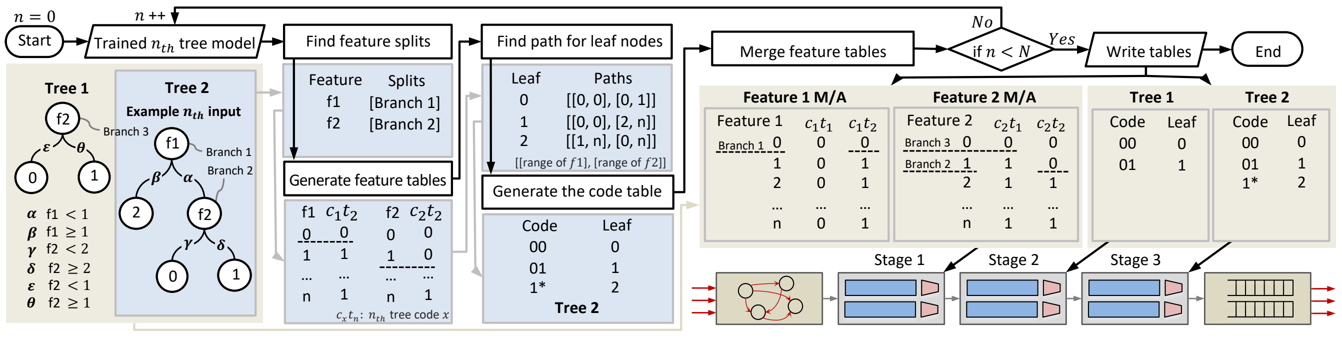

Figure 2: Workflow of the Planter algorithm.

Figure 2 shows a simple example of mapping an ensemble model using two trees based on the $M + N$ format of the algorithm, with arrows indicating Planter's workflow. Grey boxes illustrate M/A tables contents. Planter starts by iterating on all trees. In the example, $n = 1$, meaning that the table entries are indicated for the second tree. The algorithm goes through all the branches of the n^{th} tree and records the condition values of each split in the tree. These splits are then used to generate feature tables. Once all feature tables have been generated, the algorithm goes through all the leaf nodes, and records all the branches taken along the path. This is used to generate a code table for the n^{th} tree, encoding the path to the leaf node by combining the codes from the feature tables and finding their respective leaf in the code table. When all n trees have been processed, Planter collects all code tables and merges the feature tables of each tree model. These tables are then loaded to the network device.

Different types of ensemble models are differentiated in Planter by the action of the code table and final decision method. Random forest uses the resulting label from each code table as a vote, and counts the votes from all the trees to obtain the final label. In XGBoost, the code table stores the (normalized) probability of a selected leaf, sums per class the probabilities from all trees, and sets the label by comparing classes' summed probabilities. For both models, decision confidence can be included in the code table. Isolation Forest stores the depth of the leaf node in the code table, adds the depth from all tree models, and decides on a class based on the total depth.

4 PRELIMINARY EVALUATION

The Planter algorithm is implemented in Python. The machine learning training is done using Scikit-learn [14] and P4 is used for the switch data plane. The generated program runs on $64 \times 100GE$ Intel Tofino platform. Planter is evaluated using two datasets: a traditional machine learning dataset, Iris [6], and a network intrusion detection dataset [12].

Our preliminary evaluation uses 4 features and 3 trees for the IRIS dataset (due to the small size of the dataset), and 5 features and 6 trees for anomaly detection, with a depth of 4, implementing Random Forest and XGBoost. Note that these results use the number of trees sufficient for classification, and not the maximum number possible. For example, in the anomaly detection use case, we can fit 7 trees with (up to) 1000 leaves. A baseline model running on a host is fully grown (i.e. order of 100 trees and 10,000 leaves).

All the generated models consume less than 7% of the memory compared with Intel's reference design, switch.p4, and have a negligible effect on the latency of a switch pipeline — significantly lower than the reference design latency. The programs run at line rate, evaluated using DPDK's pktgen and both synthetic and anomaly detection traces [12]. As Planter tables can share stages with standard network functionality tables, this may be transparent to the user. The accuracy loss compared with models running on a server is less than 2%, and similarly for metrics such as F1.

5 CONCLUSION AND FUTURE WORK

We present Planter, an algorithm for efficient mapping of ensemble models to programmable data planes. Planter increases the number of trees that can be mapped to a network device by overlapping trees within match-action tables.

Planter's development is currently focused on the optimization of the algorithm: minimizing the number of entries in code tables and maximizing the number of trees that can fit a pipeline. Future work will explore more use cases, both network-focused and traditional machine learning use cases. It will further explore if ensemble models can or should be split between multiple switches.

Acknowledgements We acknowledge support from VMware.

REFERENCES

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[2] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[3] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680* (2019).
[4] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 785–794.
[5] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.
[6] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[7] Sohaila Eltanbouly, May Bashendy, Noora AlNaimi, Zina Chkirbene, and Aiman Erbad. 2020. Machine Learning Techniques for Network Anomaly Detection: A Survey. In *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. IEEE, 156–162.
[8] Peter Ffoulkes. 2017. The Intelligent Use of Big Data on an Industrial Scale. *insideBIGDATA* (2017), 1–10.
[9] Vladimir Gurevich and Andy Fingerhut. 2021. P4-16 Programming for Intel Tofino Using Intel P4 Studio. In *2021 P4 Workshop*.
[10] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores

with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[11] Jong-Hyouk Lee and Kamal Singh. 2020. SwitchTree: in-network computing and traffic analyses with Random Forests. *Neural Computing and Applications* (2020), 1–12.

[12] Nour Moustafa and Jill Slay. 2015. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 military communications and information systems conference (MilCIS)*. IEEE, 1–6.

[13] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cedric Baudoin, and Jose Aguilar. 2018. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1988–2014.

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

[15] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[16] Zhaoqi Xiong and Noa Zilberman. 2019. Do switches dream of machine learning? Toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33.

[17] Zhi-Hua Zhou. 2012. *Ensemble methods: foundations and algorithms*. CRC press.