User Manual of Planter

Version 0.1.0

Changgang Zheng

June 13, 2024

Contents

1 Introduction

This document serves as a user guide for the Planter framework, offering detailed information on implementing in-network machine learning (ML) algorithms and their related use cases using Planter.

Planter is a modular framework designed to prototype a broad range of in-network ML algorithms with an ease-of-use process. It is the outcome (artifact) of the research presented in the paper "Planter: Rapid Prototyping of In-Network Machine Learning Inference" published in SIGCOMM CCR [23] (with an earlier version available on arXiv in 2022 titled "Automating In-Network Machine Learning" [22]). This document complements the information presented in the paper and can be considered an additional resource for understanding Planter's functionality.

Planter's code is available at GitHub [25]. Starting to use Planter requires just a configuration file and a dataset. Afterward, Planter manages the transition of ML tasks into the programmable data plane. All datasets used in our evaluation are publicly available.

2 Environment Setup

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

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

As Planter drives network programmable targets for executing and testing use cases in the data plane, ensure the chosen target's necessary environment is installed. For example, BMv2's required environment can be set up following [3]. There are also publicly available guides for other Planter-supported targets. The introduction of all targets and their installation are presented in our repository[1]under the `src/help/Planter_Supports` folder.

In future releases, Planter will be packed in the virtual machine (VM), docker container, linux package, or python package.

3 Getting Started with Planter

Planter is started using the following command:

```
sudo python3 Planter.py
```

With the above command, Planter will directly load the configurations file. All the missing configurations should be input by CLI. The configuration file is under `src/config/Planter_config.json`. Planter provides multiple modes, for example, manual configuration mode, where configuration is input manually with a detailed CLI. This mode can be activated by adding `-m`. One can also use `-h` to see additional command options and learn more modes. **For the users new to the Planter framework, starting with `-m` mode is strongly recommended**.

Each run will output three ML performance reports. 1. The first matrix is the report from the `scikit-learn`. 2. The second matrix shows the simulated data plane result. 3. The third matrix reports the actual result of the model performance on the selected target. A more detailed matrix is shown in the tutorial wiki [25].

4 Running Example

This example realises an encode-based random forest on a BMv2 software switch. To run this example, all needed Python packets should be installed. Without BMv2 support can partially run this example. Now, Let's "Planting" this in-network random forest model.

Run Planter with -m (manually config) mode:

```
(base) changgang@linux:~/Planter$ python3 Planter.py -m
// Planter's logo.
```

4.1 Directory-related Part

The following will show in CLI directly after running Planter:

```
Please input the following directories:
+ Where is your data folder? (default = '/home/*/Data')
```

Press `enter` (`return` in Apple's keyboard) if the default is correct. Otherwise, input the correct directory and press `enter`:

```
+ Where is your Planter folder? (default = '/home/*/Planter')
```

Press `return` if the default is correct. Otherwise, input the correct directory and press `return`.

4.2 Model's Mapping-related Part

Continue from the above code segment:

```
+ Which model to plant? (default = DT | options = -h)
```

Input "`-h`" and press `enter` will show hits for all the available options. In this case, input "RF" and press `enter`:

```
+ Which model variation to plant? (default = 1 | options = -h)
```

Input "EB" and press `enter`. Note that for folder name `Type_<variation_name>`, the input is `<variation_name>`.

4.3 Use Case (Dataset)-related Part

Continue from the above code segment:

```
+ Which dataset to use? (default = UNSW_5_tuple | options = -h)
```

Input "`Iris`" and press `enter`:

```
+ Number of features? (default = 5)
```

Input "4" and press `enter`:

```
= src/configs/Planter_config.json is generated
+ Use the testing mode or not? (default = y)
```

Directly pressing `enter` can activate the testing mode, where the framework randomly selects 2000 samples from the dataset for fast training. To use the full dataset, input “n” and press `enter`.

4.4 Model’s Hyperparameter-related Part

Continue from the above code segment:

```
= The shape of the dataset - train x: (105, 4) ... test y: (45,)
= Add the following path: ... /Planter_v2/src/models/RF/Type_EB
+ Number of trees? (default = 5)
```

Press `enter`:

```
+ Number of depth? (default = 4)
```

Press `enter`:

```
+ Number of leaf nodes? (default = 1000)
```

Press `enter`.

4.5 Training and Mapping-related Part

Continue from the above code segment:

```
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        13
           1     0.9500    0.9500    0.9500        20
           2     0.9167    0.9167    0.9167        12

    accuracy                         0.9556        45
```

```
   macro avg     0.9556    0.9556    0.9556        45
weighted avg     0.9556    0.9556    0.9556        45
```

The matrix shows the classification result from the scikit-learn library. Continuing from the above code segment, the algorithm mapping begins:

```
Classification results are downloaded to log as
 ↪  src/logs/log.json
The table for Tree: 4 is generated
Generating vote to class table...Done
Begin transfer: Feature table 0
Input table has:  80 entries and: ... output TCAM entry has 15
80th testing sample with correct matches: 100.0 % and 0 errors.
Begin transfer: Feature table 1
Input table has:  45 entries and: ... output TCAM entry has 11
45th testing sample with correct matches: 100.0 % and 0 errors.
Begin transfer: Feature table 2
Input table has:  70 entries and: ... output TCAM entry has 16
70th testing sample with correct matches: 100.0 % and 0 errors.
Begin transfer: Feature table 3
Input table has:  26 entries and: ... output TCAM entry has 12
26th testing sample with correct matches: 100.0 % and 0 errors.

Ternary_Table is generated
Exact_Table is generated
... /Planter_v2/src/configs/Planter_config.json is generated
+ Test the table or not? (default = y)
```

Input “y” and press `enter`:

```
Test the generated tables
switch_prediction: 1, test_y: 1, with acc: 0.956 ... f1: 0.955
The accuracy of ... is 0.955 ...


               precision    recall  f1-score   support

```

```
           0     1.0000    1.0000    1.0000        13
           1     0.9500    0.9500    0.9500        20
           2     0.9167    0.9167    0.9167        12

    accuracy                         0.9556        45
   macro avg     0.9556    0.9556    0.9556        45
weighted avg     0.9556    0.9556    0.9556        45

= src/configs/Planter_config.json is generated
Exact match entries:  187
Ternary match entries:  54
```

This output matrix is the classification performance evaluation (of generated table entries). This assessment uses Python to simulate the pipeline logic.

4.6 Target & Architecture-related Part

Continue from the above code segment:

```
+ Which architecture to use? (default = tna | options = -h)
```

Input "`v1model`" and press `enter`:

```
+ Which is the use case? (default = ... | options = -h)
```

Input "`performance`" and press `enter`:

```
= Add the following path: ... /standard_classification
= Add the following path: ... /src/architectures/v1model
Generating p4 files and load data file... Done
```

At this step, both data plane and control plane codes for the in-network random forest model have been generated! Before this step, neither the specific P4 platform's environment nor the Linux operating system is required. The only requirement is a Python 3+ environment (with some frequently used packets, e.g., NumPy, scikit-learn).

4.7 Test-related Part

Continue from the above code segment.

```
+ What is the target device? (default = Tofino | options = -h)
```

Input “bmv2” and press `enter`:

```
+ Which mode to choose? (default = software | options = -h)
```

Press `enter`:

```
= Dump the targets info to src/configs/Planter_config.json
= Add the following path: ... /src/targets/bmv2/software
+ Send packets to which port? (default = 'eth0'):
```

Input the correct port (on my device is `eth0`) and press `enter`:

```
+ Please input your password for 'sudo' command:
```

Input your sudo password (`pi` in VM) and press `enter`:

```
ip link show
*** Killing stale mininet node processes
pkill -9 -f mininet:
*** Shutting down stale tunnels
pkill -9 -f Tunnel=Ethernet
pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
rm -f *.pcap
rm -rf build pcaps logs
mkdir -p build pcaps logs
p4c-bm2-ss --p4v 16 --p4runtime-files
 ↪  build/RF_performance_Iris.p4.p4info.txt -o
 ↪  build/RF_performance_Iris.json RF_performance_Iris.p4
sudo python3 ../../utils/run_exercise.py -t topology.json -j
 ↪  build/RF_performance_Iris.json -b simple_switch_grpc
```

```
Reading topology file.
Building mininet topology.
Configuring switch s1 with file s1-commands.txt
s1 -> gRPC port: 50051
**********
h1
default interface: eth0 10.0.1.1        08:00:00:00:01:01
**********
Starting mininet CLI

===============================================================
Welcome to the BMV2 Mininet CLI!
===============================================================
Your P4 program is installed into the BMV2 software switch
and your initial runtime configuration is loaded. You can
 ↪  interact
with the network using the mininet CLI below.

To inspect or change the switch configuration, connect to
its CLI from your host operating system using this command:
  simple_switch_CLI --thrift-port <switch thrift port>

To view a switch log, run this command from your host OS:
  tail -f ... /test_environment/logs/<switchname>.log

To view the switch output pcap, check the pcap files in ...
 ↪  /targets/bmv2/software/model_test/test_environment/pcaps:
 for example run:  sudo tcpdump -xxx -r s1-eth1.pcap

To view the P4Runtime requests sent to the switch, check the
corresponding txt file in ... /test_environment/logs:
 for example run:  cat ... /logs/s1-p4runtime-requests.txt

mininet> Predicted load table time ... (0.037s)

===============================================================
= Two steps to exit Planter:                                   =
= 1. Input 'exit', press 'return'.                             =
= 2. Press ctrl + c.                                           =
================================================================
Test the switch model:
Switch model 45th prediction: ... acc to sklearn: 1.0 ...

               precision    recall  f1-score   support

           0      1.0000    1.0000    1.0000        13
           1      0.9500    0.9500    0.9500        20
           2      0.9167    0.9167    0.9167        12

    accuracy                          0.9556        45
   macro avg      0.9556    0.9556    0.9556        45
weighted avg      0.9556    0.9556    0.9556        45

======================== Test Finished ========================
mininet>
Process <load data> cost 0.0073s
Process <train model> cost 0.0154s
Process <convert model> cost 0.0182s
Process <python-based test> cost 0.0779s
```

This output matrix gives the inference performance of the generated in-network random forest model on a BMv2 software switch. The three output matrixes (inference results) should be similar in the test. **Test finished!**

5 Planter Workflow

Before diving into Planter's modules, this section shows its components and workflows, especially in its back end.

5.1 Back-end

The back end is the core of the Planter framework. It can be used separately from the front end (not activated by default). Its workflow, shown in Figure 1, has seven steps. In the first two steps, Planter loads a dataset ❶ and trains it ❷. The model is mapped to P4 ❸ using the selected architecture and target. Generated P4 code is compiled ❹ and loaded to the target's data plane ❺. In step ❻, table entries and registers are loaded through the control plane. Last, in step ❼, an auto-generated functionality test is run on the target.

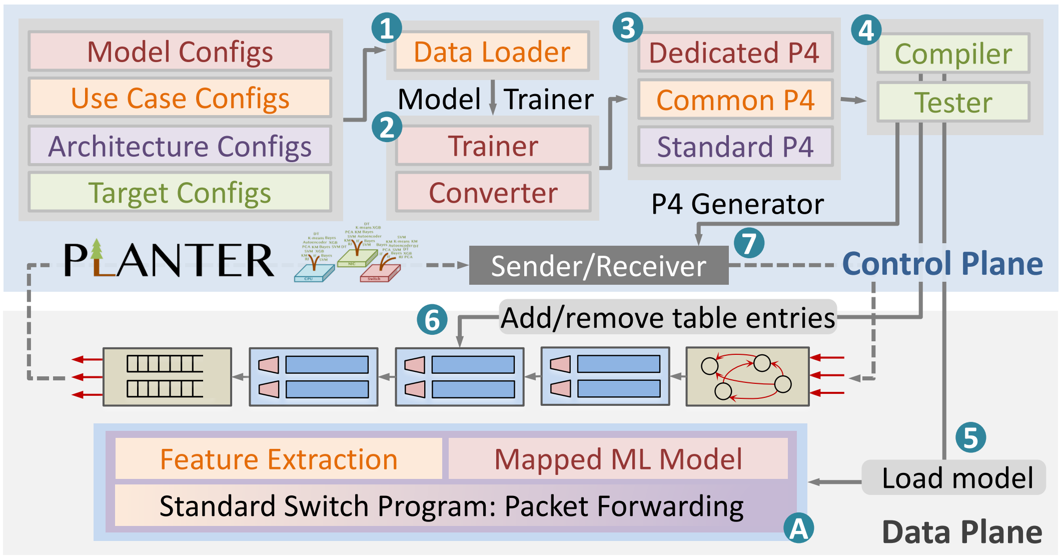

Figure 1: The Planter framework (back-end) [23].

Configurations. Planter's configurations are stored in a file. It can be manually prepared or entered through an interactive CLI (using `-m` or when Planter find there is a missing). When no configuration is missing, the one-click in-network ML (training, algorithm mapping, deployment, and testing) can be realised. The configuration is in `Planter_config.json` file, under the `src/config` folder.

Load Data. Planter's data loader loads datasets for training purposes. It is use-case specific, based on used features and data format. All loaded data are cached in the same format. The load data file is in `<name>_dataset.py` file, under the `src/load_data` folder.

Model Trainer & Converter. ML Training is conducted by the Model Trainer, which drives a standard ML training framework on a host. Trained models are mapped to the M/A pipeline in the Model Converter. A software test is generated to test the validity of the mapped model. The above process is done inside a `table_generator.py` file, under the `src/models` folder, within each `<model_name>/type_n` variation folder.

P4 Generator The P4 Generator has three parts. The main program, including architecture-specific code, is the Standard P4 Generator located under `src/architectures/` folder in `p4_generator.py` file. This file manages and calls all the functions required to generate code for a switch model. These functions are included in the Common P4 Generator and Dedicated P4 Generator. The Common P4 Generatorwrites use case-specific P4 code, such as bespoke feature extraction. It is under `common_p4.py` file (located under the use-case folder `src/use_cases/<use_case_name>`. The Dedicated P4 Generator creates ML model-related P4 code, which is in `dedicate_p4.py` file under the model folder `./src/models/<model_name>/<type_n>`.

Compiler & Tester The Model Compiler & Tester deploy and test the generated model in a P4 platform. The Model Compiler generates bash scripts to compile, load, and run mapped ML models (by using multiple sessions). This process is realised in under `/src/targets/<target_name>/<test_name>` folder in a file named `run_model.py`. Similarly, the Tester generates testing scripts and runs the functionality test on the selected target. The model tester is realised in `test_model.py` file under the same folder.

5.2 Front-end

Planter provides a front-end optimiser (Figure 2) that automates hyperparameter tuning, and fixes compilation failures. It applies Bayesian Optimisation, to optimise parameters based on a predefined objective function. Use command `-o` to activate the Planter's front end. **This front end is recommended only when the user understands the Planter's back end.**

6 Simple Test Guide

This section provides a concise overview of module selection and resulting outputs for creating customised testing procedures. For comprehensive information on Planter-supported modules (such as models, platforms, use

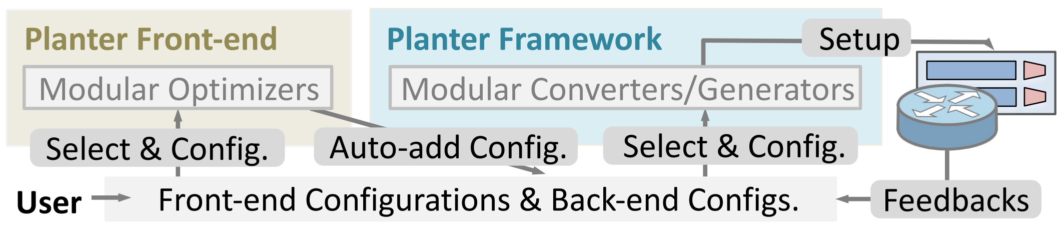

Figure 2: The Planter framework's front-end (not activated by default) [23].

cases, etc.), please refer to Section 7. Instructions for modifying the Planter framework and incorporating your own modules can be found in Section 8.

6.1 Performance Mode

To test the Planter in performance mode, configure the model by using the following Table 1. The table shows what mapping (`EB`, `LB`, or `DM`) can be used in each type of model when choosing the use case `performance`. As manual optimization is applied, the mappings supported by performance mode consume fewer stages but only match the performance use case.

Model	SVM	KM	NN	NB	DT	RF	XGB	IF	KNN	PCA	AE
EB		✓			✓	✓	✓	✓	✓		
LB	✓	✓		✓						✓	✓
DM			✓		✓	✓					

Table 1: Supported variations when `performance_mode` is applied.

When using other variations, especially named by `numbers`, choose other use cases such as `standard_classification`. **This is for avoiding module mismatch**. For other Planter-supported modules (e.g., models, platforms, use cases, and others), please refer to Section 7.

6.2 Output Matrix

Generally, each test will have three classification performance matrix outputs. Each matrix has a similar structure, for example:

```
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        13
           1     0.9500    0.9500    0.9500        20
           2     0.9167    0.9167    0.9167        12

    accuracy                         0.9556        45
   macro avg     0.9556    0.9556    0.9556        45
weighted avg     0.9556    0.9556    0.9556        45
```

The introduction to these three output matrices is shown below.

Matrix 1. The first matrix is the classification performance of the scikit-learn standard model in Python (i.e., on a host).

Matrix 3. The second matrix is the classification performance of the generated table entries using Python-simulated pipeline logic (i.e., theoretical).

Matrix 3. The third matrix is the classification result of the generated P4 on either software/hardware programmable network devices or its emulation environment (i.e., actual).

6.3 Hardware Test

The hardware test is highly platform-dependent. For guidance on conducting tests on particular hardware, it is advisable to reach out to the respective hardware vendors. Detailed information about Planter's hardware test configuration can be found in [23] and its artifacts [25].

7 Planter Supports

This section lists part of the Planter-supported modules.

7.1 Algorithm Mappings

The paper introduced three common methodologies for algorithm mapping [23], which are Direct Map (DM), Lookup Based (LB) and Encode Based

(EB). Under each methodology, each ML model can be mapped to a data plane in multiple ways. This subsection introduces Planter's currently supported in-network ML algorithms and their variations. The recommended variation for each model is marked with [*legacy*].

Mappings marked as EB, DM or LB are suitable only for the performance use case. In these cases, stage allocation is better with manual optimization.

7.1.1 Decision Tree (DT)

Planter-supported decision tree models are under `./src/models/DT`. The implemented variations up to now are listed below.

Type 1: In this encode-based mapping, feature tables can share stages. This model uses only exact matches in feature tables and the decision table (without a default action). This is an original implementation in IIsy's workshop paper [11]. For IIsy baseline [21] (with arXiv [20]), please reference to Type EB.

Type 2: This mapping is similar to Type 1, but uses the default action to reduce the number of entries in M/A tables.

Type 3: This mapping is based on Type 2. This method also uses ternary matches to reduce the number of entries in feature tables.

Type 4 [*legacy*]: This mapping is very similar to Type 3. This method also uses LPM matches (instead of ternary) to reduce the number of entries in feature tables.

Type 5: This mapping is very similar to Type 1. It uses LPM match to reduce the number of table entries in both feature and tree tables.

Type Depth Based BMv2 Only: This mapping is based on direct mapping methodology. It is based on SwitchTree's [8] (and pForest's [1]) design.

Type EB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 3. The stage consumption is manually optimised. It can be treated as the baseline decision tree model of Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]).

Type DM [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type Depth Based BMv2 Only. The stage consumption is manually optimised.

Type 1 xsa: This variation supports xsa architecture (AMD Alveo U280 FPGA over Open-NIC).

7.1.2 Random Forest (RF)

Planter-supported random forest models are under `./src/models/RF`. The implemented variations up to now are listed below.

Type 1: In this encode-based mapping, feature tables can share one stage and code tables can share the other stage. This mapping uses exact match tables without a default action.

Type 2: This mapping is similar to Type 1, but uses the default action to reduce the number of entries in M/A tables.

Type 3: This mapping is based on Type 2. This method also uses ternary matches to reduce the number of entries in feature tables.

Type 4 [*legacy*]: This mapping is very similar to Type 3. This method also uses LPM matches (instead of ternary) to reduce the number of entries in feature tables.

Type 5: This mapping is very similar to Type 1. It uses LPM match to reduce the number of table entries in both feature, tree, and decision tables.

Type hybrid: This mapping supports the hybrid use case [21]. It runs a small model on a programmable network device and a large model on the backend. This mapping allows different trees to use different features.

Type Depth Based: This mapping is based on direct mapping methodology. It can be treated as an adjusted SwitchTree's [8] (and pForest's [1]) design to run on TNA.

Type Depth Based BMv2 Only: It is based on SwitchTree's [8] (and pForest's [1]) design.

Type EB: Select only in conjunction with the performance use case. This mapping is based on Type 3. The stage consumption is manually optimised. It can be treated as the baseline random forest model of Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]).

Type EB_auto [*legacy*]: Same as the Type EB (support updated CLI).

Type DM [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type Depth Based BMv2 Only. The stage consumption is manually optimised for Tofino.

Type DM BMv2 Only: The most standard SwitchTree's [8] method (not optimised for Tofino).

Type hybrid EB: Same as Type hybrid and select only in conjunction with the performance use case.

Type 1 xsa: This variation supports xsa architecture (AMD Alveo U280 FPGA over Open-NIC).

7.1.3 XGBoost (XGB)

Planter-supported XGBoost models are under `./src/models/XGB`. The implemented variations up to now are listed below.

Type 1: This mapping is very similar to Type 3 RF. This method also uses ternary matches to reduce the number of entries in feature tables. It also uses the default action to reduce the number of entries in M/A tables.

Type 2: This mapping is very similar to Type 1. This method also uses LPM matches (instead of ternary) to reduce the number of entries in feature tables.

Type 3 [*legacy*]: This mapping is very similar to Type 2. It uses LPM match to reduce the number of table entries in both feature, tree, and decision tables.

Type hybrid: This mapping supports the hybrid use case [21]. It runs a small model on a programmable network device and a large model on the backend. This mapping allows different trees to use different features.

Type EB: Select only in conjunction with the performance use case. This mapping is based on Type 1. The stage consumption is manually optimised. It can be treated as the baseline XGBoost model of Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]).

Type EB_auto [*legacy*]: Very similar to Type EB (support updated CLI). More than CLI, this variation adds an option to trade-off accuracy and memory consumption.

Type 1 xsa: This variation supports xsa architecture (AMD Alveo U280 FPGA over Open-NIC).

7.1.4 Support Vector Machine (SVM)

Planter-supported SVM models are under `./src/models/SVM`. The implemented variations up to now are listed below.

Type 1: This SVM model is mapped using a lookup-based approach. It is an original implementation of IIsy's SVM [23].

Type LB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1. The stage consumption is manually optimised. It can be treated as the baseline SVM model from Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]). **Please be aware that this code serves as an illustrative example rather than representing the most optimized mapping. We possess versions that strive for maximal parallelism in both the placement of feature tables and addition operations. This principle is applicable across all mappings.**

7.1.5 Naive Bayes (NB)

Planter-supported naïve Bayes models are under `./src/models/Bayes`. The implemented variations up to now are listed below.

Type 1: This mapping employs lookup-based methodology, using *log*() operation to change Bayes from comparing 'multiplied components' to 'added components'. This change can achieve a better accuracy result and save memory compared to IIsy's workshop paper [11].

Type 2 [*legacy*]: This mapping is similar to the Type 1. In this mapping, after training, $P(x_i|y)$ is mapped to a predefined range. It is then converted to int and used for $P(X|y)$ lookups, where $X = x_1, x_2, ..., x_n$.

Type 3: This mapping is very similar to the Type 1. The difference is that this mapping allows all feature tables to share stages. It saves stages but requires more logic operations.

Type LB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1-3. The stage consumption is manually optimised. It can be treated as the baseline nïve Bayes model of

Planter [23] (arXiv [22]).

Type LB Bernoulli [*legacy*]: Select only in conjunction with the performance use case. Different from Type 1-3 and Type LB using traditional GaussianNB, this variation applies BernoulliNB. It can also be treated as the baseline nïve Bayes model of Planter [23] (arXiv [22]).

7.1.6 K-means (KM)

Planter-supported k-means models are under `./src/models/KM`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping is based on lookup-based approach. In this mapping, M/A tables are used to store an intermediate result (L2 distance) of each feature component. The next operation in the pipeline is logic-based, adding up all the components. Next, there is a comparison and labelling of the packet. This is the reproduction of the K-means model in IIsy [21] (arXiv [20]).

Type Clustream: This mapping is based on Clustream's [4] design. This solution is classified as an encode-based solution. The input features should be preprocessed on the host before in-band classification.

Type LB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1. It can be treated as the baseline k-means model from Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]).

Type EB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type Clustream. The stage consumption is manually optimised.

7.1.7 K-nearest Neighbors (KNN)

Planter-supported k-means models are under `./src/models/KNN`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping solution is similar to k-means' Type EB and can be classified as an encode-based solution. The input features should be preprocessed on the host before in-band classification.

Type EB [*legacy*]: Select only in conjunction with the performance use case.

This mapping is based on Type 1. It can be treated as the baseline k-nearest neighbours from Planter [23] (arXiv [22]) and IIsy [21] (arXiv [20]).

7.1.8 Autoencoder (AE)

Planter-supported AE models are under `./src/models/Autoencoder`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping is a newly designed Autoencoder model based on the lookup-based methodology.

Type LB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1. It can be treated as the baseline autoencoder model from Planter [23] (arXiv [22]).

7.1.9 Principle Component Analysis (PCA)

Planter-supported PCA models are under `./src/models/PCA`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping is a newly designed PCA model based on the lookup-based methodology.

Type LB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1. It can be treated as the baseline PCA model from Planter [23] (arXiv [22]).

7.1.10 Isolation Forest (IF)

Planter-supported isolation forest models are under `./src/models/IF`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping is a newly designed isolation forest model based on the encode-based methodology. The implemented isolation forest is based on the original paper [9]. This variation uses LPM matches to reduce the number of entries in feature tables.

Type 2: This mapping is the simplified version of scikit-learn. The result consumes fewer table entries but is different from the original paper [9]. This variation also uses LPM matches to reduce the number of entries.

Type EB: (Same as Type 1) Select only in conjunction with the performance use case. The stage consumption in this mode is manually optimised.

Type Simplified EB [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 2. It can be treated as the baseline isolation forest model from Planter [23] (arXiv [22]).

7.1.11 Neural Network (NN)

Planter-supported neural network models are under `./src/models/NN`. The implemented variations up to now are listed below.

Type 1 [*legacy*]: This mapping is based on Qin's work [10]. This variation currently uses MLP instead of XNOR Net.

Type 2: This mapping is very similar to Type 1 and is under development.

Type DM [*legacy*]: Select only in conjunction with the performance use case. This mapping is based on Type 1. It can be treated as the baseline neural network model from Planter [23] (arXiv [22]).

7.1.12 Q-learning

Using the direct mapping methodology proposed by Planter, Q-learning can be realised on programmable network devices using two approaches [17]. Its implementation can be found in [18].

7.2 P4 Architectures

This subsection lists part of Planter-supported architectures to date. The codes are located under the `./src/architectures/` folder.

TNA: The Tofino Native Architecture (TNA) is an architecture designed for Intel Tofino hardware and emulation targets. One example is the Tofino Switch. TNA-related code is under `src/architectures/tna`.

v1model: The v1model was originally designed to support $P4_{14}$ and later to support $P4_{16}$. It is used in BMv2 and its software targets. v1model-related code is under `src/architectures/v1model`.

PSA: The Portable Switch Architecture (PSA) is more realistic compared with v1model. Although the PSA has workable demos, it is still under development.

PSA-related code is under `src/architectures/psa`.

XSA: The xsa is for AMD Alveo U280 FPGA over Open-NIC. XSA-related code is under `src/architectures/xsa`.

Spectrum: The spectrum is for NVIDIA Spectrum. Spectrum-related code is under `src/architectures/spectrum`. The call to the spectrum architecture is subject to the confidentiality agreement. Releasing these lines of code is pending approval. For more information, please e-mail changgang.zheng@eng.ox.ac.uk.

7.3 P4 Targets

This subsection outlines part of the Planter-supported targets. The combination of P4 target and P4 architecture forms a P4 Platform. Therefore, each chosen target module must collaborate with the selected architecture module. Recommended modules for use are indicated by [*legacy*].

Tofino Switch: Tofino represents a programmable switch ASIC designed for efficient packet switching with high throughput and low latency. When choosing this target, the `tna` architecture should be chosen. The emulation and testing platform (`bf-sde-x.y.z`) for the Intel (Barefoot) Tofino switches is subject to a non-disclosure agreement (NDA). To utilise this platform, individuals need to become members of the Intel® Connectivity Research Program. For further details, please reach out to Intel directly. Tofino-related code is under `src/targets/tofino*` folder, with similar but different 7 module options. The call to the Intel/NVIDIA compiler is subject to the confidentiality agreement. Releasing these lines of code is pending approval. For more information, please e-mail changgang.zheng@eng.ox.ac.uk.

1. Tofino_old_SDE_version: A module support for SDE versions earlier than bf-sde-9.7.0. It includes various compiler and tester modes, such as: compile, hardware, software, software_ASCII, and software_UDP. **Software, hardware, and compile modes can apply to most current use cases. The same principle applies to all target modules.**

2. Tofino: A module support for SDE versions later than bf-sde-9.7.0. It includes various compiler and tester modes, such as: compile, compile_auto, compile_for_opt_mode, hardware, software, software_ASCII, and software_UDP.

3. Tofino2: A module support for Tofino's second-generation programmable switch design. It includes various compiler and tester modes, such as: compile, software, software_ASCII, and software_UDP.

AMD Alveo U280 FPGA: AMD Alveo U280 FPGA over Open-NIC is a P4 target. With this target currently implemented model variations such as DT Type1 xsa and RF Type1 xsa can be chosen. At the same time, when choosing this target, the `xsa` architecture should be chosen. AMD Alveo U280 FPGA-related code is under `src/targets/alveo_u280` folder, with 1 module option.

1. alveo_u280: A module support MD Alveo U280 FPGA over Open-NIC. It includes various compiler and tester modes, such as: compile, hardware, and behaviour.

BMv2: The behavioral model version 2 (BMv2) is a P4 behavioural model of a software switch. BMv2 versions 1.15.0 and 1.14.0 are tested and proofed to be functional. The BMv2-related code is under `src/targets/bmv2*` folder, with similar but different 2 module options.

1. bmv2: A module supports BMv2 on CPU. It includes various compiler and tester modes, such as: compile, hardware, and behaviour.
2. bmv2_DPU: A module supports BMv2 on NVIDIA BlueField2 DPU. It includes various compiler and tester modes, such as: compile, hardware, and behaviour.

NVIDIA Spectrum The NVIDIA Spectrum is a type of high-performance programmable ethernet switch. This target only supports P4 codes in NVIDIA's specific P4 architecture. When using this target, the Planter should run under administration permission. The current NVIDIA Spectrum target only supports the compilation of P4 files. The Spectrum-related code is under `src/targets/NVIDIA_Spectrum` folder, with 1 module option.

1. NVIDIA_Spectrum: A module support NVIDIA Spectrum. It includes a compile mode.

NVIDIA BlueField2 DPU NVIDIA BlueField DPUs has two modes. It can be configured and operated via [Link]. This platform has a CPU, allowing BMv2 to run on it. This target module is implemented by bmv2_DPU. For better performance, the module using P4-OvS (instead of BMv2) is under development.

P4Pi (BMv2) This target module is implemented by bmv2.

T4P4S: The P4Pi platform can generate DPDK-based software switches by using the T4P4S compiler. It supports P4 codes in PSA and v1model. T4P4S front end uses p4c reference compiler and basic end compiler generates target-agnostic switch code. Planter can generate P4 files for both BMv2 and T4P4S compilers on P4Pi. The T4P4S-related code is under `src/targets/t4p4s` folder, with 1 module option.

1. t4p4s: A module support T4P4S (P4Pi). It includes a compile mode.

7.4 Use Cases

This subsection outlines part of the use cases implemented in Planter. Each use case corresponds to a specific scenario. The extraction of features within the data plane occurs within these use case modules, while the packets to be injected (both original and extracted for training) are determined by the chosen load data module. Therefore, each chosen use case module must collaborate with the selected load data module. Use cases recommended for use are indicated by [*legacy*].

Standard Classification [*legacy*]: This use case is for functionality verification. The sender sends packets containing `Ether|Planter` headers, which the switch will parse, infer, and send the inference back straight to the sender. Its related code is under `src/use_cases/standard_classification`.

Performance [*legacy*]: This mode is a template that shows the performance. Select only type Type EB, LB, or DM when choosing this mode. The stage consumption in this mode is manually optimised. This use case is very similar to the Standard Classification use case. Its related code is under `src/use_cases/performance`.

Anomaly Detection: This is an anomaly detection use case. In this case, the IP 5-tuples will be used as input features, and the Planter will be able to create P4 files for parsing and classifying IP 5-tuples. Its related code is under `src/use_cases/anomaly_detection`.

Standard UDP: In this use case, the input and final results are encapsulated in UDP. The sender sends packets containing `Ether|IP|UDP|...|<binary data>` headers. The switch will parse, infer, and send results back to the sender. Its related code is under `src/use_cases/standard_UDP`.

Standard ASCII: This use case is based on stadard UDP module. On top of it, this use case can parse unencrypted text files and do the inference. Its related code is under `src/use_cases/standard_ASCII`.

Performance ASCII: This combines the performance mode and the standard ASCII mode. Its related code is under `src/use_cases/performance_ASCII`.

Performance Snake: This combines the performance mode and snake test (cost 3 extra stages). The snake test is used to test the throughput. Its related code is under `src/use_cases/performance_ASCII`.

Standard Classification for XSA: This supports AMD Alveo U280 FPGA over Open-NIC. It is based on stadard classification module. Its related code is under `src/use_cases/standard_classification_with_xsa_support`.

7.5 Datasets

This subsection outlines part of the Planter-supported datasets.

UNSW: The UNSW dataset is a traditional anomaly detection dataset that contains 9 typical network attacks.

KDD: The KDD dataset is a legend network intrusion detection dataset.

CICIDS: The CICIDS dataset is a network intrusion detection dataset with abstract behaviour of 25 users and 7 different attacks.

AWID3: The AWID3 is a network intrusion detection dataset with 13 different attacks. The data is collected from 16 different physical devices and VMs.

Requet: The Requet dataset is a Quality of Experience (QoE) metric detection dataset collected from video streaming services to give the records of video playback information.

Jane Street Market: The Jane Street Markets dataset is a high-frequency trading dataset with 500 days of historical data, which contains 130 features.

Iris: The Iris dataset is one of the best-known datasets to be found in the pattern recognition literature. The dataset contains four features for three different types of Iris flowers. This dataset is mainly used for demos.

8 Adding New Modules and Designs

Planter design allows users to easily add new models, architectures, targets, use cases, and datasets. For example, users can create a load data file in the `./src/load_data` directory to employ a custom dataset. This file imports and preprocesses the raw data according to user requirements, adhering to a standardised interface. Detailed instructions on integrating new modules and their standardised interfaces are provided in this section.

8.1 Adding Algorithm Mapping

To add your own in-network ML algorithm, there are two things to be considered, which are P4 file generation and M/A table entries generation. The main procedure can be generalised into creating a `<model_name>` folder under the directory ./src/models. Then, in the model folder, create a variation folder, e.g., `Type_n`. In the `Type_n` folder, create `dedicated_p4.py` file for model training and conversion, and `table_generator.py` for generating the ML model-related tables. The following is a detailed explanation.

8.1.1 The `table_generator.py` file

In `table_generator.py`, there are two compulsory functions to be written, `run_model(*)` & `test_tables(*)`:

The **`run_model(*)`** function:

1. The overview of `run_model(*)` function:

```
def run_model(train_X, train_y, test_X, test_y,
↪   used_features):
    # 1. Model Training
    # 2. Model Testing
    # 3. Model Conversion (M/A table entries generation).
    return sklearn_test_y
```

2. The input of `run_model(*)` function:

```
train_X # data frame/ndarray/list
        # training data
        # generate from <dataset_name>_dataset.py

train_y # data frame/ndarray/list
        # the training labels
        # generate from <dataset_name>_dataset.py

test_X # data frame/ndarray/list
       # testing data
       # generate from <dataset_name>_dataset.py

test_y # data frame/ndarray/list
       # testing labels
       # generate from <dataset_name>_dataset.py

used_features # int,
              # number of used features
              # input in Planter.py
```

3. The output (return) of `run_model(*)` function:

```
sklearn_test_y # data frame/ndarray/list
               # sklearn (baseline) inference result
               ↪  (output labels)
```

The **`test_model(*)`** function:

1. The overview of `test_tables(*)` function:

```
def test_tables(sklearn_test_y, test_X, test_y):
    # 1. Test the generated M/A table entries.
    return
```

2. The input of `test_tables(*)` function:

```
sklearn_test_y # data frame/ndarray/list
               # sklearn (baseline) inference result
                ↪  (output labels)
               # generated by funtion test_tables(*)

test_X # data frame/ndarray/list
       # testing data
       # generate from <dataset_name>_dataset.py

test_y # data frame/ndarray/list
       # testing labels
       # generate from <dataset_name>_dataset.py
```

8.1.2 The dedicated_p4.py file

In dedicated_p4.py, we can define functions to write M/A pipeline logic into P4 files. This file requires some key functions:

The **load_config(*)** function:

1. The overview of load_config(*) function:

```
def load_config(fname):
    # 1. load the config files.
     return config, Planter_config
```

2. The input of load_config(*) function:

```
fname # str
      # config file directory
      # generate from p4_generator.py
```

3. The output (return) of load_config(*) function:

```
config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']

Planter_config # dict
               # Planter's standard configs
```

The **add_model_intro(*)** function:

1. The overview of `add_model_intro(*)` function:

```
def add_model_intro(fname, config):
    # 1. Write intro (header in comment format) in P4
     ↪  files.
    with open(fname, 'a') as intro:
       intro.write("// ...\n")
    return
```

2. The input of `add_model_intro(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **separate_metadata(*)** function:

1. The overview of `separate_metadata(*)` function:

```
def separate_metadata(fname, config):
    # 1. Write model related meta data.
    with open(fname, 'a') as file:
        file.write("...\n")
    return
```

2. The input of `separate_metadata(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
       ↪  config']
       # generated by function load_config(*)
```

The **`separate_logics(*)`** function:

1. The overview of `separate_logics(*)` function:

```
def separate_logics(fname, config):
    # 1. Write model related apply() logic in ingress
    ↪  pipeline.
    with open(fname, 'a') as file:
        file.write("...\n")
    return
```

2. The input of `separate_logics(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
       ↪  config']
       # generated by function load_config(*)
```

The `separate_tables(*)` function:

1. The overview of `separate_tables(*)` function:

```
def separate_tables(fname, config):
    # 1. Write model related tables, actions, and
     ↪  definitions in ingress.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of `separate_tables(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The `create_load_tables(*)` function:

1. The overview of `create_load_tables(*)` function:

```
def create_load_tables(fname, fjson, config,
 ↪  Planter_config, file_name):
    # 1. Prepare load data files for bfrt.
    #    varies from different targets
    return
```

2. The input of `create_load_tables(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py
```

```
fjson # str
      # M/A table file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)

Planter_config # dict
               # Planter's standard configs
               # generate from anywhere

file_name # str
          # P4 file name
          # generate from p4_generator.py
```

8.2 Adding P4 Architecture

To add new P4 architecture, we need to add a file `p4_generator.py` under the directory `./src/architectures/<architecture_name>` (Standard P4 in Figure 1). File `p4_generator.py` is the key component in the P4 generation process which is able to drive the model-related file `dedicated_p4.py` and the use case-related `filecommon_p4.py` to collaboratively generate overall the P4 file.

8.2.1 The `table_generator.py` file

In `p4_generator.py`, the key function is `main(*)`, which will be called by `Planter.py`. The `main(*)` is able to call functions in `dedicated_p4.py` and `common_p4.py`:

Import functions:

```
from dedicated_p4 import *
from common_p4 import *
```

The **`main(*)`** function:

1. The overview of `main(*)` function:

   ```
   def def main():
       # 1. Load configuration from file
       config, Planter_config = load_config(config_file)
       # 2. Write an introduction in your P4 file header
       add_model_intro(p4_file, config)
       # 3. Call functions in ```p4_generator.py```,
        ↪  ```dedicated_p4.py``` and ```common_p4.py```
       #    to generate your P4 file
       return
   ```

8.3 Adding P4 Target

The Planter supports the tests on the software (e.g. BMv2) and hardware (e.g. Tofino) switch target with the following options:

- Compile: generate P4 code for compilation.
- Software: generate P4 code for compilation and generate simple test scripts for accuracy results under software environment.
- Hardware: generate P4 code for compilation and generate simple test scripts for accuracy results under hardware environment.

New modules can be added for targets, such as `alveo_u280/behavioral` for FPGA, and for specific use cases' testing workflows, like `bmv2/Linnet` for finance use cases [6] (removed from this release). This subsection mainly uses the option of software as an example.

Test environment (`model_test` folder & `utils` folder): Software target like the BMv2 has the emulated software environment. To support the tests on the software target, for this example, the test environment needs to be added to the model_test folder and the corresponding building packages need to be added to the utils folder. In this environment, p4c compiles the P4 code into a JSON representation to be consumed by the software switch. The switch usually runs in a topology simulated with the Mininet tool. In this case, the whole environment (BMv2, p4c, Mininet) to compile the code, simulate the network topology and run the code can be put under the model_test folder.

8.3.1 The **run_model.py** file

The main function of the run_model.py code is to load the P4 code (generated by Planter under the ./P4/folder) to the test environment. This is implemented by generating a bash file to 1) copy the P4 code to the test environment, and 2) run the compilation.

The **file_names** function: This function reads the working directory of the P4 file and test environment.

1. The overview of file_names(*) function:

```
def file_names(Planter_config):
    return work_root, model_test_root, file_name,
     ↪  test_file_name
```

2. The input of file_names(*) function:

```
Planter_config # dict
               # P4 generator's configs -
                ↪  Planter_config['p4 config']
              # loaded from json file in the main function
```

3. The output of file_names(*) function:

```
work_root # working directory of Planter
model_test_root # software target test environment
file_name # P4 file name defined in the format:
          # model_usecase_dataset
```

```
test_file_name # test file name in the format:
               # test_switch_model_TargetDevice_TargetType
```

The **add_make_run_model** function: This function generates the bash script to copy the P4 code from `./P4/` to the test environment for compiling and running. The commands generated in the script need to adapt to the test environment and follow the compiling and running steps of the software target.

1. The overview of `add_make_run_model(*)` function:

```
def add_make_run_model(fname, config):
    with open(fname, 'w') as command:
        command.write("#!/bin/bash\n")
    ...
```

2. The input of `add_make_run_model(*)` function:

```
fname # str
      # File name of the bash script to be generated in
       ↪  this function
      # Defined as '/src/scripts/make_run_model.sh'

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # loaded from json file in the main function
```

8.3.2 The `test_model.py` file

This file provides a test script to run the simple test with the test data on the software target.

The **write_common_test_classification** function: This function generates a Python file to create the simple test packets with the Planter header to carry the test data. The packets are input to the target switch. After the switch gives a classification result to the packets, a classification report will summarize the classification accuracy. This applies to the models like DT/RF for classification tasks.

1. The overview of `write_common_test_classification(*)` function:

```
def write_common_test_classification(fname,
 ↪  Planter_config):
    with open(fname, 'a') as test:
        test.write("import json\n")
                    ...
```

2. The input of `write_common_test_classification(*)` function:

```
fname # str
      # File name of the test Python script to be generated
       ↪  in this function
      # Defined as '/src/test/test_switch_model_targetNam
      # e_targetType.py'

Planter_config # dict
               # P4 generator's configs -
                ↪  Planter_config['p4 config']
               # loaded from json file in the main function
```

The **`write_common_test_dimension_reduction`** function: Similar to the previous `write_common_test_classification` function, this function also generates a python file to create the simple test packets with the Planter header to carry the test data. The difference is that this test script is created for models (e.g. PCA/AE) doing the dimension reduction tasks. After the switch gives an analysis result to the packets, the comparison will be presented in Pearson's correlation value.

The **`generate_test_file`** function: This function calls one of the previous two functions to generate the test script.

1. The overview of `generate_test_file(*)` function:

```
def generate_test_file(config_file):
    ... # select the test script to generate
    if Planter_config['test config']['type of test'] ==
     ↪  'dimension_reduction':
        write_common_test_dimension_reduction(test_file,
         ↪  Planter_config)
    elif Planter_config['test config']['type of test'] ==
     ↪  'classification':
        write_common_test_classification(test_file,
         ↪  Planter_config)
    return test_file
```

2. The input of `generate_test_file(*)` function:

```
config_file # dict
            # P4 generator's configs - Planter_config['p4
             ↪  config']
            # loaded from json file in main function
```

`Add a hardware target?` Different from the software target with the software test environment, the hardware target requires compiling and testing on the hardware device. For targets like the Tofino switch, the software development environment (SDE) is provided for program testing and debugging. In this case, to enable Planter to support your own hardware target, the SDE environment information needs to be added to the following functions to generate the scripts for running and testing. The hardware implementation and tests are based on the hardware device. The functions defined in `run_model.py` and `test_model.py` files are similar to the descriptions above [25].

8.4 Adding Use Case

To add your own use case, several things should be considered.

- Where to generate use case-specific P4 codes?
- What is the input data?

- How to generate use case-specific testing procedures?

Among these, only the first one, case-specific P4 codes, is compulsory. To realise it, we need to create file `common_p4.py` under the created folder `<use_case_name>` under the directory `./src/use_cases`. The second and third ones are optional and depend on whether we want to utilise Planter to test our design. For the two optional changes, the preprocessing logic in the M/A pipeline will make training data different from testing data. Therefore, both input data in the file `<name>_dataset.py` under the directory `./src/load_data` and the testing procedure in the file `test_model.py` under the directory `./src/targets/<target_name>/<test_name>` need to be changed.

If this is still too complex, a good example is to see the difference between software and `software_ASCII` under `src/targets/Tofino`.

8.4.1 The `common_p4.py` file

In `common_p4.py`, we can define functions to write use case-related parsing and M/A pipeline logic into P4 files. This file requires some key functions:

The **`common_headers(*)`** function:

1. The overview of `common_headers(*)` function:

```
def common_headers(fname, config):
    # 1. Write use case related headers.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of `common_headers(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **common_metadata(*)** function:

1. The overview of common_metadata(*) function:

```
def common_metadata(fname, config):
    # 1. Write use case related meta data.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of common_metadata(*) function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **common_parser(*)** function:

1. The overview of common_parser(*) function:

```
def common_parser(fname, config):
    # 1. Write use case related parsers.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of common_parser(*) function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **common_tables(*)** function:

1. The overview of common_tables(*) function:

```
def common_tables(fname, config):
    # 1. Write use case related tables, actions, and
     ↪  definitions in ingress.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of common_tables(*) function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **common_feature_extraction(*)** function:

1. The overview of common_feature_extraction(*) function:

```
def common_feature_extraction(fname, config):
    # 1. Write use case related apply() logic (and feature
     ↪  extraction) in ingress pipeline.
    #    before the execution of in-network ML model.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of common_feature_extraction(*) function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

The **common_logics(*)** function:

1. The overview of common_logics(*) function:

```
def common_logics(fname, config):
    # 1. Write use case related apply() logic in ingress
     ↪  pipeline
    #    after the execution of in-network ML model.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of common_logics(*) function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
        ↪  config']
       # generated by function load_config(*)
```

8.4.2 The `test_model.py` file

In `test_model.py`, the packet header and testing data need to be customised based on the use case before testing. Not all the functions in the `test_model.py` need to be updated. Based on the reference tester `src/targets/<target_name>/software/test_model.py`, only function `write_common_test_<model_type>(*)` need to be updated. In `write_common_test_<model_type>(*)`, we need to make sure the header is use case customised (aligned to what is generated by `common_p4.py`) and focus on how to generate raw data without preprocessing (different from `test_x`).

The **`write_common_test_<model_type>(*)`** function:

1. The overview of `write_common_test_<model_type>(*)` function:

```
def write_common_test_<model_type>(fname, config):
    # 1. use case related packet formulation (including
     ↪  use case customized packets)
    # 2. load the proper input data
     ↪  from'/src/temp/Test_Data.json'.
    # 3. send the generated packets, receive packets, and
     ↪  calculate the accuracy.
    with open(fname, 'a') as file:
       file.write("...\n")
    return
```

2. The input of `write_common_test_<model_type>(*)` function:

```
fname # str
      # P4 file directory
      # generate from p4_generator.py

config # dict
       # P4 generator's configs - Planter_config['p4
       ↪  config']
       # generated by function load_config(*)
```

3. By using function `write_common_test_<model_type>(*)`, files like `src/test/test_switch_model_Tofino_software.py` can be generated.

8.4.3 The `<name>_dataset.py` file

This file includes the preprocessing of the dataset. To prepare the training and learning on a new dataset, the dataset loading and preprocessing steps are necessary. With the .csv dataset files saved in a directory, several preprocessing steps need to be added in this load data file `name_data.py`, such as a) reading the dataset, b) replacing/removing the N/A values, c) encoding & selecting the features, and d) splitting the training/testing sets. The file includes one main function described below.

The **`load_data(*)`** function:

1. The overview of `load_data(*)` function:

```
def load_data(num_features, data_dir):
    # 1. Load the data
    # 2. Save original data Test_Data['original_test_X']
    #    to '/src/temp/Test_Data.json' and the number of
    #    original input features to Planter_config['data
    #    config']['number of original features']
    # 3. Preprocess the original data to simulate P4
    #    feature extraction and generate train_X, train_y,
    #    test_X, test_y
    return train_X, train_y, test_X, test_y, used_features
```

2. The input of `load_data(*)` function:

```
used_features # int,
              # number of used features
              # input in Planter.py

data_dir # str
         # data file directory
         # generate from Planter.py
```

3. The output of `load_data(*)` function:

```
train_X # data frame
        # preprocessed training data

train_y # ndarray/list
        # the training labels

test_X # data frame
       # preprocessed testing data

test_y # ndarray/list
       # testing labels

used_features # list
              # list of feature names
```

8.5 Add Dataset (Quick Start)

To use your own dataset, a load data file `name_data.py` should be created under `./src/load_data` folder. The load data file is responsible for loading the source data and preprocessing it based on your needs. The returned dataset should follow the same standard: return `X_train`, `y_train`, `X_test`, `y_test`, `used_features`. **The load data is only part of the use case (Section 8.4). However, beginning with a quick start (for in-network ML inference**

performance verification) with only the data set (ignore the feature extraction and use the existing use case, e.g., standard classification using the Planter header) is not a bad choice.

9 Version History

This section provides a list of all released versions of Planter.

Ver. 0.0.0 - May 3rd, 2024: The initial internal release of the Planter framework for code review.

Ver. 0.0.1 - May 6th, 2024: The internal release of the Planter framework for code review.

Ver. 0.1.0 - June 10th, 2024: The first release version of the Planter framework. Calls to commercial compilers are not included, subject to confidentiality agreements. Releasing these lines of code is pending approval.

10 History and Applications

This section introduces the history and applications of Planter.

10.1 Planter Project's History

The poster, arXiv, and SIGCOMM CCR version of Planter:

- Planter: Rapid Prototyping of In-Network Machine Learning Inference, ACM SIGCOMM Computer Communication Review, 2024 [23].
- Automating In-Network Machine Learning, arXiv, 2022 [22].
- Planter: Seeding Trees Within Switches, ACM SIGCOMM'21 Poster and Demo Sessions, 2021 [24].

10.2 Planter Applications

We are also excited to introduce several works that apply the Planter framework. If your work uses Planter, please kindly email us if possible (E-mail:

changgangzheng@qq.com). We will include your latest publication or project in the Planter's applications list.

- SmartEdge - Design of Dynamic and Secure Swarm Networking, D4.1 Design of Dynamic & Secure Swarm Networking, EU Semantic Low-code Programming Tools for Edge Intelligence horizon project, 2024 [2].
- IIsy: Hybrid In-Network Classification Using Programmable Switches, IEEE/ACM Transactions on Networking, 2024 [21].
- E-Commerce Bot Traffic: In-Network Impact, Detection, and Mitigation, 27th Conference on Innovation in Clouds, Internet and Networks, 2024 [5].
- In-Network Machine Learning Using Programmable Network Devices: A Survey, IEEE Communications Surveys and Tutorials, 2023 [16].
- DINC: Toward Distributed In-Network Computing, ACM CoNEXT'23 & Proceedings of the ACM on Networking, 2023 [19].
- Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways, IEEE Internet of Things Journal, 2023 [13].
- QCMP: Load Balancing via In-Network Reinforcement Learning, ACM SIGCOMM Workshop on Future of Internet Routing & Addressing, 2023 [17].
- Advanced Threat Defense with In-Network Traffic Analysis for IoT Gateways, MobiUK, 2023 [12].
- Federated Learning-Based In-Network Traffic Analysis on IoT Edge, Security for IoT Networks and Devices in 6G, IFIP Networking 2023 [14].
- LOBIN: In-Network Machine Learning for Limit Order Books, IEEE 24rd International Conference on High Performance Switching and Routing (HPSR), 2023 [7].
- Linnet: Limit Order Books Within Switches, ACM SIGCOMM'22 Poster and Demo Sessions, 2022 [6].
- P4Pir: In-Network Analysis for Smart IoT Gateways, ACM SIGCOMM'22 Poster and Demo Sessions, 2022 [15].

11 Acknowledgements

The following people contributed to this project: Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In addition, Peng Qian contributed to the repository. This work was partly funded by VMware and the EU Horizon SMARTEDGE (101092908, UKRI 10056403). We acknowledge support from Intel and NVIDIA.

References

[1] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. *CoRR*, abs/1909.05680, 2019.

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

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

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

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

[6] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. Linnet: Limit Order Books Within Switches. In *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, pages 37–39, 2022.

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

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

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

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

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

[12] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. Advanced Threat Defense with In-Network Traffic Analysis for IoT Gateways. 2023.

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

[14] Mingyuan Zang, Changgang Zheng, Tomasz Koziak, Noa Zilberman, and Lars Dittmann. Federated Learning-Based In-Network Traffic Analysis on IoT Edge. 2023.

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

[16] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials*, pages 1–1, 2023.

[17] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. QCMP: Load Balancing via In-Network Reinforcement Learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pages 35–40, 2023.

[18] Changgang Zheng and Benjamin Rienecker et al. QCMP's GitHub Repository. https://github.com/In-Network-Machine-Learning/QCMP, 2023.

[19] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. DINC: Toward Distributed In-Network Computing. In *Proceedings of ACM CoNEXT'23*, 2023.

[20] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification, 2022.

[21] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Hybrid In-Network Classification Using Programmable Switches, 2024.

[22] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Automating In-Network Machine Learning, 2022.

[23] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *ACM SIGCOMM Computer Communication Review*, 2024.

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

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