Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Advanced Topics in
Communication Networks (Fall 2019)

Group Project Report

pForest: In-Network Inference with Random Forests

Authors:
Adrien Prost
Benno Schneeberger
Roberto Rossini

Advisor: Roland Meier

Supervisor: Prof. Dr. Laurent Vanbever

Submitted: Dec 16, 2019
10 Pages

Abstract

As the field of machine learning continues to strive, interesting applications are discovered regularly. As it turns out, using only few packets statistics, one can easily train machine learning models to classify entire packet flows. More specifically, a model can be trained to detect flows corresponding to a Denial of Service attack. Furthermore, recent research has led to the emergence of programmable data planes, enabling a great deal of flexibility in the functionality of network switches. The paper pForest: In-Network Inference with Random Forests[1], proposes a new framework in which multiple random forest models trained on packet flow statistics can be implemented directly into the data plane using P4, allowing switches to detect malicious flows as early as possible after a flow has started. In this paper, we describe an actual, simplified, implementation of this framework.

Contents

1 Introduction

The paper *pForest: In-Network Inference with Random Forests*[1], proposes a framework in which random forest models can be implemented directly on the data plane using `P4`[2]. More specifically, the paper suggests to train multiple random forest models using as features statistics from the first n packets of a flow. This can allow packets to be classified early on if the classification certainty is high enough.

In order to perform this task on the data plane, switches must keep information about previous packets in memory, as well as information about the model. At first glance, implementing such a classifier on a switch would seem like an unrealistic task as programmable data planes have limited capabilities. However, the `P4` programming language [2] allows an efficient solution to this problem. Furthermore, due to the many programming restrictions imposed by this language, the framework describes a systematic way to perform classification using series of match-action tables which can be easily generated.

The key challenge is to design classification models that fit the constraints of programmable data planes (e.g., no floating points, no loops, and limited memory) while providing high accuracy.

In this project, we realized a small-scale version of the framework proposed in *pForest*. In a nutshell, we have performed the following tasks:

- Training of a random forest model using a subset of the large Intrusion Detection Evaluation data set (CICIDS2017)[3], which consists of many flows involving benign and up-to-date common attacks, using features extracted from the statistics of all packets in the flow.
- Implementation of the classifier in `P4`, employing the random forests match-action tables representing the decision trees of the resulting model.
- Implementation of a `Python` script that generates match-action table entries depending on the model resulting from the training.
- Implementation of a `Python` program which wraps everything up and generates the whole `P4` code for our project. This script builds and trains a random forest model and finally, if the model is good enough, it translates the model into a sequence of match-action `P4` tables with the corresponding table entries.

2 Background and related work

In this section, we summarize the key concepts of programmable data planes and `P4` language(§2.1) and we introduce random forest classifiers (§2.2).

2.1 Programmable data planes and P4 language

The data plane component of *pForest* is implemented in `P4`. A `P4` program describes the behaviour of a data plane in multiple building blocks:

- Parser: it parses the incoming packets, extracting useful information from them, such as the headers data, which will be used in the ingress pipeline;
- Match-Action pipeline: it consists of some processing logic combined with match-action tables which can be used to take various decisions for each packet, e.g. forwarding, dropping or selecting a specific outgoing interface; and
- Deparser: it is responsible for reassembling the dissected packet.

The P4 programming language provides various programming functionalities in order to enable some flexibility in the aforementioned steps. Additional information can be found in the P4 language specification [4].

2.2 Random Forests

In this project, we used random forests model to classify packets on the data plane. In this section, we explain briefly the way random forests classifiers are trained.

The basic building block of a random forest is the *decision tree*. "A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility."[5] A decision tree is a binary tree structure in which each internal node represents a threshold to which a feature will be compared, each branch represents the outcome of the comparison and each leaf represents a class label. When training a decision tree, the internal nodes threshold and feature are chosen in order for the tree to provide the most accurate estimates possible. A *random forest* consists of multiple decision trees. Each of these decision trees considers a random subset of features when learning its internal nodes values and it only has access to a random set of the training data points. This increases diversity in the forest, leading to more robust overall predictions. To predict labels, the random forest performs a majority voting on all decisions taken by the individual decision tree estimates.

In Figure 1 you can see a schematic representation of a random forest model.

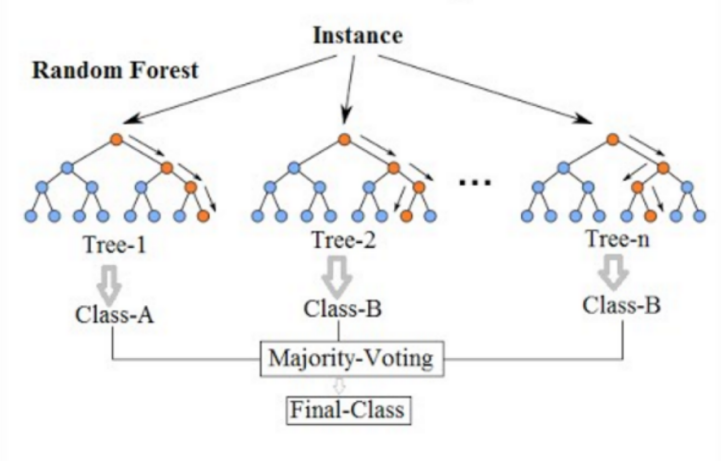

Figure 1: In the figure is shown a schematic representation of a random forest. Multiple trees come up with the prediction for a specific class, using just a subset of the available features. The final prediction performed by the random forest model is given by the 'majority voting' of all the tree estimates. [6]

3 Implementation

We implemented a small-scale version of the framework. Differently from the original system, our implementation uses only one random forest model, trained to classify the whole flow. Thus, the correct classification is expected at about the last packets of a flow. However, as we will discuss in Section 4, our adaptation can classify flows correctly even after few packets. Moreover, the code can easily be extended to support multiple models.

Our implementation is structured as a Python program which, given in input model's parameters, it trains a random forest classifier (§3.1), it generates the P4 code (§3.2) and it finally translates the model into a series of match-action tables and their corresponding table entries (§3.3).

3.1 Random Forest Classifier

In order to build and train our random forest model, we first analysed a subset of CICIDS2017 [3] (in particular, the *Tuesday-WorkingHours* subset) and then we extracted the features important to us. The features we considered are the same stateful features used in the paper:

- Packet inter-arrival time (IAT):
  - Minimum
  - Maximum
  - Average
- Packet's payload length:
  - Minimum
  - Maximum
  - Average
  - Total
- Number of packets
- TCP flag counts of the following flags:
  - SYN
  - ACK
  - PSH
  - FIN
  - RST
  - ECE
- Duration: Time since first packet

In order to simplify our classification, we used only 2 different labels for our flows:

- BENIGN: with this label we indicate the legitimate flows
- MALIGN: with this label we merged together all the evil flows present.

To easily extract those features, we relied on the following tools: Jupyter Notebook [7], pandas [8] and NumPy [9].

Subsequently, we trained our model to binary classify flows, based on the aforementioned features. After extracting the features mentioned above, we built our model (referring to [10]) using scikit-learn [11].

Before translating the random forest in a sequence of table entries that will be used in the `P4` tables, the trained model's performance is tested. To do so, we randomly selected 6 flows from the dataset - 4 'MALIGN' and 2 'BENIGN' - and we extracted the corresponding packets. We created a `.pcap` file for each flow, containing all its packets. The Scapy [12] library is used to manage these packets and extract the desired features. If the model classifies correctly at least 5 flows out of 6, it is translated. Otherwise, another model will be trained and subsequently tested until a good model will have been trained. Due to this phase, we are sure that the classifier used by the switch is solid and thus we can proceed.

3.2 P4 Implementation

In this section, we start by summarizing the whole classification (§3.2.1) and then we explain in detail how the `P4` code is organized (§3.2.2, §3.2.3, §3.2.4).

3.2.1 Summary

In order to classify packets, the switch keeps multiple registers to store the features for each flow it sees. Given a specific flow, when the first packet arrives, the flow's features are initialized according to the packet and then stored in the registers according to some indexes calculated by hashing the packet's flow ID. Depending on the features, the packet is classified using many match-action tables which represent the decision trees. Whenever a subsequent packet from the same flow arrives, the previously stored features are retrieved from the registers, updated to include the information from the new packet, and then the classification is done with the updated set of features. Depending on the classification output and its certainty, a decision can be taken or the packet can simply be updated with a special tag value reflecting the random forest's prediction.

The logic is summarized in the following flowchart 2

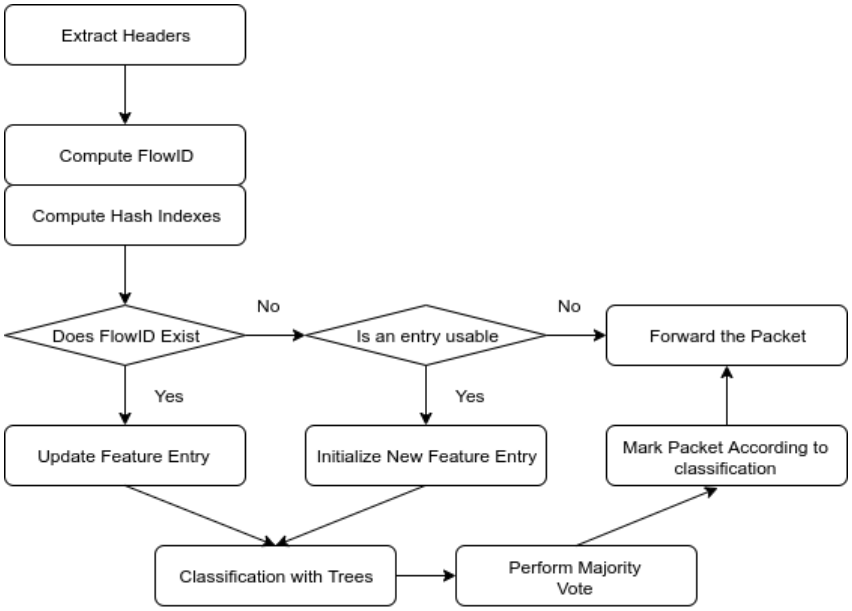

Figure 2: Flowchart of the data plane pipeline

3.2.2 Parser

The parser takes care of extracting various headers from the packet. In this case it will try to extract the following headers from the packet:

- Ethernet Header,
- IP Header, and
- TCP or UDP Header, according to transport protocol used.

These headers are extracted from the packet using the header definitions defined in the `headers.p4` file.

When the packet is about to be forwarded, the deparser reassembles it, attaching the updated headers.

Parser and deparser are implemented in the `parser.p4` file.

3.2.3 Headers

The `headers.p4` file contains all header definition for the headers mentioned above, as well as different structures that are stored in the user defined metadata field which are defined below:

- **Features Structure**: The structure `features_t` is responsible for storing all features used to classify the flow.
- **Flow Entry Structure**: The structure `flow_entry_t` represents a flow along with its features vector to be stored in registers. it contains:
  - 1 bit indicating if flow entry is used,
  - 32 bit flow ID which corresponds to the hash function's output of the 5-tuple *(srcIP, dstIP, srcPort, dstPort, Protocol)*,
  - 32 bit time-stamp indicating when this flow was last seen, and
  - `features_t` structure of the current packet.
- **Classification Variables**: The structure `classification_variables_t` contains the variables used in the classification process to navigate through the decision trees of the forest and store the results when a leaf is attained. For each tree, there are four variables stored:
  - The node that the tree is currently using in the classification process,
  - 1 bit indicating whether the feature compared in that node is bigger than the corresponding threshold or not,
  - The label predicted by the tree, and
  - The certainty of the prediction made by the tree.

Finally, we store the following in the user defined `metadata` structure:

- Indexes for registers (hash outputs),
- Bits indicating the status of register entries,
- Flow entry corresponding to the current packet's flow,
- The packet's time stamp, and
- Bit indicating if a valid register has been found for the current packet's flow.

3.2.4 Ingress

In this section we detail how the features are stored (§3.2.4.1), how they are calculated (§3.2.4.2) and how the classification task is implemented (§3.2.4.3).

3.2.4.1 Feature Storage

The ingress pipeline maintains `N` register buckets, containing `K` registers which can store a flow entry structure corresponding to a specific flow. After the parsing of the packet, its flow ID (32 bits) is computed using the `crc32` hash algorithm available in the *simple switch* architecture. [13]. Additionally, `N` different hashes are calculated using the `crc32_custom` algorithm (this allow us to specify the polynomials used in the calculations) and then the results are taken modulo `K` and stored in the metadata as `index_table0,index_table1,..,index_tableN-1`. These `N` values represent indexes for each register buckets in which the features of the flow are stored. After calculating these indexes, each register bucket is read at its corresponding index, and the flow ID stored at this index is compared to the flow ID of the current packet. If the flow IDs match it means the same flow has already been seen and the features stored at this entry are updated. If no matching flow ID is found at these indexes and at least one register is usable [1], a new set of features is initialized for this flow and stored in the register. The following algorithm is equivalent to our `P4` code to perform this step efficiently:

1. Initialize `N` 1-bit variables `usable_0,usable_1,...,usable_N` to 0.
2. Initialize a 1-bit variable `found_entry` to 0.
3. For each register bucket `i`: If `found_entry` is equal to 0:
   (a) Read the register at index `hash_i` containing a flow entry.
   (b) Check if the flow entry is marked as 'used' (first bit of the entry).
   (c) If it is not used, then set `usable_i` to 1 and go on checking the other buckets.
   (d) If it is used, check if the flow ID stored matches the packet's flow ID.
   (e) If the flow ID do not match, then:
      i. If the difference between the current timestamp and the previous timestamp stored in the register is bigger than a predefined threshold, set `usable_i` to 1.
      ii. Go on checking the other buckets.
   (f) If the flow ID matches, then set the variable `found_entry` to 1 and update the features stored in the entry.
4. For each variable `usable_i`: If `found_entry` is equal to 0 and `usable_i` is equal to 1:
   (a) Set the variable `found_entry` to 1.
   (b) Initialize the features for this flow and store them in the register.

3.2.4.2 Feature Initialization and Updating

When the first packet from a flow enters the switch, the flow's features are initialized the following way:

- The packet inter-arrival time (IAT) features are all initialized to 0, the minimum value is set to the current IAT at the second packet.
- The packet length features are all initialized to the length of the payload of the first packet.
- The number of packets is initialized to 1.

[1] A register is usable if either it is empty or the timestamp of the last update is too old (depending on a predefined threshold)

- The TCP flags are initialized to 1 or 0, depending if the flag was present in the first packet or not.
- The duration of the flow is initialized to 0.

Whenever subsequent packets from a given flow enter the switch, the features are updated the following way:

- All features that represent a minimum or maximum statistic is updated depending on the current packet's IAT and payload length.
- Due to the lack of floating point numbers and the division operator, performing the average in `P4` is not trivial. We approximated averages using the moving average technique. The moving average is calculated with the following formula:

$$new\_average = \frac{old\_average + new\_value}{2}. \tag{1}$$

  In `P4` this value is calculated by right shifting the sum between the old average and the new value of one position.
- The duration of the flow is incremented by the current packet's IAT.
- The packet count is incremented by 1.
- The TCP flag counts are incremented if the corresponding flag is present in the current packet.

3.2.4.3 Classification

In order to classify a packet in `P4` using the model previously trained, we proceed as follows.

The random forest is implemented in `P4` using match-action tables (one table per decision tree per level [1]). The tables match exactly on the current node number and on whether the feature compared in that node is bigger or smaller. They have two different actions: one for the internal nodes and one for the leaves. More precisely, for the X^{th} tree of the forest, the tables match on `meta.classification_variables.current_node_treeX` and `meta.classification_variables.bigger_than_threshold_treeX`, which are stored in the `metadata` field. If there is a match, one of two following action is called:

- `action_node_tree_X`: it compares the corresponding feature with the threshold and set the values `current_node` and `bigger_than_threshold` in order to match on the next match-action table.
- `action_leaf_tree_X`: it stores the label and the certainty of the leaf, respectively in `meta.classification_variables.label_treeX` and `meta.classification_variables.certainty_treeX`.

The classification in the apply block of the `P4` code works as described below:

1. Choose the appropriate model for classification. In our case, as our implementation uses one model for any flow length, the model chosen is always the same (see [1] for more details about using multiple models).
2. Classify the flow using the match-action tables. It can be done by applying all the tables of each tree one after the other, until the leaves are reached.
3. When each tree of the forest has classified the flow, every `label_treeX` and `certainty_treeX` variables are read and majority voting is performed in order to get the resulting label. Finally the total certainty of the prediction is computed.
4. The flow is classified if the total certainty has attained a given threshold.

3.3 P4 Code Generation

Having a significant amount of table entries in the match-action tables, as well as a major part of our `P4` code depending on the model used, we felt the need of a way that allows us to generate efficiently and in a flexible manner all the code and the commands required. In order to achieve that, we developed a `Python` program - `generate_pforest.py` - which takes in input the wanted number of trees and their depth and then it trains the random forest model. Unless the model performs finely, the program discards it and trains a new one, until it finds a valid one. If it is the case, tables's entries and the whole `P4` code are generated and saved in the appropriate place. The different steps of the scripts are:

1. Training and Testing: As described in Section 3.1, until a satisfying accuracy is achieved.
2. Encoding: The random forest classifier is encoded in `table_add` entries as detailed in [1] and stored inside `s1-commands.txt` file. Some changes need to be performed to make the entries usable by the `P4` code (e.g. converting the certainties and thresholds in integer values).
3. Generating `P4`: The `P4` code is generated depending on the number of trees and the depth of the model used. There are multiple elements that depend on the model:
   - the classification variables stored in the metadata,
   - the match-action tables and their corresponding actions,
   - the apply chain of match-action tables, and
   - the majority voting and the computation of the certainty.

   Once all the `P4` has been generated, it is saved in the corresponding files: `pforest.p4`, `headers.p4` and `parser.p4`.

4 Evaluation

In this section, we present how we assessed our implementation. At first, we briefly present the network topology we used (§4.1). Subsequently, we show how we tested the system and the obtained results (§4.2).

4.1 Network Topology

In Figure 3 it is shown the topology we used to evaluate our implementation. This topology is described in our configuration file `p4app.json`, which is subsequently used by `p4run` to build and configure the virtual network [14]. As `assignment_strategy`, we used `l2`: this means that all the devices composing the topology are assumed to belong to the same subnetwork. Hosts get automatically assigned with IPs belonging to the same subnetwork (in our case, starting from `10.0.0.1/16`).

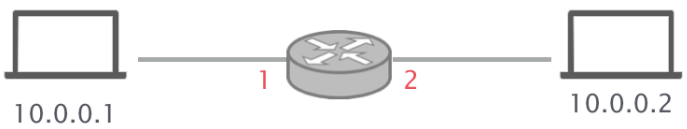

Figure 3: Topology we used during the evaluation. From left to right: `h1`, `s1`, `h2`

In order to make it simple, we implemented a forwarding policy which makes the switch, `s1`, forwarding each packet it receives to `h2` (except the packets coming through `h2`: those are forwarded to h1).

4.2 Testing and Final Results

In order to test our implementation, we used `p4run` [14], Wireshark [15] and `tcpreplay` [16].

Hereafter, the procedure to evaluate our project is presented:

1. Run `generate_pforest.py` in order to generate the whole `P4` program and the related files.
2. Run `p4run`: it creates a virtual network and it compiles the `P4` program, finally loading it into the switch.
3. Run the controller (`controller.py`) in order to initialise custom hash functions and reset the registers.
4. Done with all this preparations, get a terminal for each host (using `mx`).
   (a) In the terminal corresponding to `h2`, Wireshark and start capturing packets on the `h2-eth0` interface.
   (b) In the terminal corresponding to `h1`, run `tcpreplay` specifying as input interface `h1-eth0` and providing the path to the .pcap file to be used.
5. Our *pForest* implementation saves its classifications by overwriting the packet's source MAC addresses. Thus, in order to check if a flow was correctly classified we can just check the captured packets in Wireshark. In case of a 'BENIGN' label, `c0:01:c0:01:c0:01` is written. Otherwise, in case of a 'MALIGN' label `de:ad:de:ad:c0:de` is written.

To test our implementation, we used 6 flows - 4 'MALIGN' and 2 'BENIGN'. We randomly selected these flows from [3]'s subset *Tuesday-WorkingHours*. In the following table are reported some interesting results. These results were obtained using a random forest model with 5 trees of 5 maximum depth and 70% certainty threshold.

FlowID	First	Label	Final	Label	Packets	Real Label
IP1-IP2-49114-22-6	1	BENIGN	47	MALIGN	55	MALIGN
IP1-IP2-50386-22-6	3	BENIGN	47	MALIGN	55	MALIGN
IP1-IP2-53020-21-6	4	MALIGN	24	MALIGN	28	MALIGN
IP1-IP2-55418-21-6	4	MALIGN	24	MALIGN	28	MALIGN
IP3-IP4–53-44694-17	1	BENIGN	1	BENIGN	4	BENIGN
IP3-IP5-53-51801-17	1	BENIGN	1	BENIGN	4	BENIGN

Table 1: In this table are shown the classification results over the test flows using our implementation of *pForest* that uses a random forest model with 5 trees with a depth of 5 each and using a certainty threshold of 70%. The first column refers to the 5-tuple of the flow. The next two columns represent the first packet classified by the system and how it was classified. The next two columns are similar but they represent the packet starting from which the system 'finally' classify the flow. The last two columns represent the total number of packet present in the flow and the real label.

As expected, after most of packets of a flow go through the switch, we could see that the classification was delineated. This is due to the fact that we used a single model trained with data from whole flows, instead of multiple context-dependent models. However, it can be easily seen that the classification in the end is always correct. Furthermore, it is interesting to note that, in most flows, the system is able to correctly classify the system early on. This shows this framework's potential, which can be further developed implementing context-dependent models.

5 Conclusion

During this project, we studied the framework proposed in the paper *pForest: In-Network Inference with Random Forests* and we implemented a simplified version of it. We only employed one random forest model, which we trained ourselves with features representing statistics of an entire flow, instead of all the different context-dependent models used in the paper. Therefore our implementation does not detect flows early on, and starts classifying packets with enough certainty only after most packets of a given flow have passed through the switch. However, extending our implementation to support context-dependent random forests would not require much effort, as the entirety of our code is generated by a python script and only few changes would be required.

From the results of our study, we can imagine that these kind of systems will have a wide usage once programmable switches will become more common. However, we have also realized that due to the restrictions of `P4`, implementing the classification can be very tedious and results in thousands of line of code to perform a simple task, which is undesirable for code review and testing. We saw the power of these models which are able to detect malicious traffic and can be implemented to behave accordingly in case of an attack (for example dropping all the packets from that specific flow).

References

[1] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pforest : In-network inference with random forests," tech. rep., 2019.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] Canadian Institute for Cybersecurity, "Intrusion Detection Evaluation Dataset (CICIDS2017),"

[4] The P4 Language Consortium, "P4-16 language specification,"

[5] Wikipedia, the free encyclopedia, "Decision tree,"

[6] W. Koehrsen, "Random forest simple explanation,"

[7] *Project Jupyter*. https://jupyter.org/.

[8] *Python Data Analysis Library*. https://pandas.pydata.org/.

[9] *NumPy*. https://numpy.org/.

[10] W. Koehrsen, "Random forest in python,"

[11] *scikit-learn*. https://scikit-learn.org/stable/.

[12] Philippe Biondi and the Scapy community, *Scapy project*. https://scapy.net/.

[13] A. Fingerhut, A. Bas, "BMv2 Simple Switch Target,"

[14] *p4-utils*. https://github.com/nsg-ethz/p4-utils#topology-description.

[15] *Wireshark*. https://www.wireshark.org/.

[16] *Tcpreplay*. https://tcpreplay.appneta.com/.

A Group organization

A.1 Adrien Prost

- `P4`: Implementation of the first part of the ingress related to feature storage and manipulation
- Controller set up
- Debugging
- Report

A.2 Benno Schneeberger

- Python code that encodes a random forest into table entries
- Classification related `P4` code (table, action, majortity voting, certainty) and python code that generates it.
- Debugging
- Report

A.3 Roberto Rossini

- Data set analysis, Random Forest model training and testing.
- `P4`: Features handling and serialization.
- Evaluation
- Debugging
- Report