Project 2: Model Engineering

Using Feed Forward Neural Network (FFNN), Recurrent Neural Network (RNN) and Graph Neural Network (GNN)

1 Objective

The goal of this project is to learn how to perform *model engineering*. Students will use different architectures and preprocessing techniques to model the same problem and data.

Specifically, students will:

- Learn how to handle *categorical data*.
- Learn how to preprocess the data based on the chosen Neural Network architecture.
- Experiment with different architectures (FFNN, RNN, and GNN) using different hyper-parameters, optimizers and architectures.
- Engineer a simple *baseline solution* to systematically evaluate the effects of increasingly complex architectures and choices.

2 Submission Rules

- Groups consist of 3 students.
- Each group must submit a zip file containing the following:
  - A report (maximum 10 pages) describing the approach, experiments and results, including tables and plots.
  - The Juypyter notebook(s) and code (e.g., libraries with classes and functions written by you) to solve the tasks.
    - **Best practice**: Add comments and headers (Markdown) sections to understand what you are doing. They will i) help you tomorrow to understand what you did today and ii) help us to interpret your solution correctly.
    - **The notebook needs to be executed**: code and results must be visible so that we can interpret what you have done and what the results look like.
    - **Must run**: the code must work if we need to run it again.
    - **Submission format**: Include the notebook file (`.ipynb`) and an HTML export for easier review.
- Each group must upload the zip file to the teaching portal via Moodle before the deadline.

3 Dataset: Malware Analysis Datasets: API Call Sequences

This dataset is part of research on malware detection and classification using Deep Learning. The original dataset contains 42,797 sequences of API call labelled as *malware* and 1,079 API call sequences labelled as *goodware*. Each API call sequence contains **up to** 100 non-repeated consecutive API calls associated with the parent process, extracted from the 'calls' elements of Cuckoo Sandbox reports[1].

This is an example of API call:

```
api_call = ['RegOpenKeyExA', 'NtOpenKey', 'NtQueryValueKey', 'NtClose',
'NtOpenKey', 'NtQueryValueKey', 'NtClose', 'NtQueryAttributesFile',
'LoadStringA', 'NtAllocateVirtualMemory', 'LoadStringA', 'LdrGetDllHandle']
```

More information about the dataset can be found here:

- **Link to the Kaggle Dataset**: `https://www.kaggle.com/datasets/ang3loliveira/malware-analysis-datasets-api-call-sequences/data`
- **Link to the Paper**: Oliveira, Angelo; Sassi, Renato José (2019), "Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks.", TechRxiv. Preprint. at `https://doi.org/10.36227/techrxiv.10043099.v1`

We pre-processed the dataset for you, and release it in a `json` format. Also, we already divided the original dataset into *train* and *test* partitions. Each line of the dataset describes an API call sequence and is composed of:

- **api_call_sequence**: the sequence of API calls. Notice that each API call is a string.
- **is_malware**: the class label. It can be 0 (Goodware) or 1 (Malware).

4 Tasks

Students are required to design distinct Machine Learning pipelines utilizing Feed Forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs), and Graph Neural Networks (GNNs). In addition, they will implement a simple frequency-based baseline to analyze the impact of progressively adding architectural components. Remember:

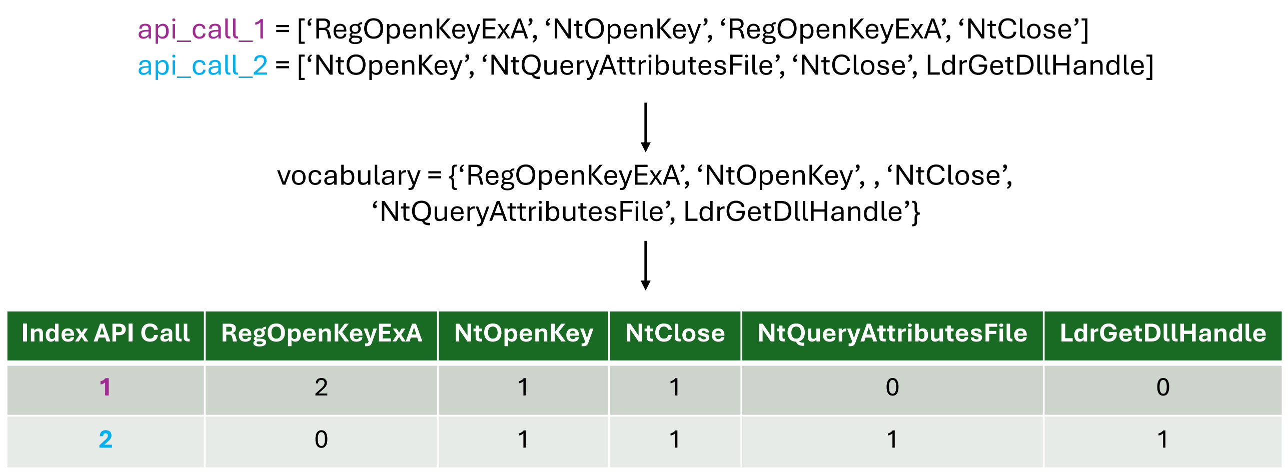
- **A key objective** of this lab is for students to understand whether a model has converged successfully. Keep in mind that not all models may converge equally well – RNNs, for instance, can be particularly challenging to train. However, **DO NOT** stop after your first attempt. Demonstrate that you made a ***reasonable*** effort to achieve convergence by experimenting with different learning rates, loss functions, training strategies, and other relevant parameters.
- This lab is **GPU-demanding** – especially from Task 3. We suggest that you i) debug your code on CPU and ii) when you are ready, run your model on the GPU.

4.1 Task 1: Frequency-based baseline

In Machine Learning problems, it is always good practice to compare against baseline solutions. Typically, one baseline involves a simple approach that helps determine whether simple choices and assumptions can already address the problem - before progressing to potentially more complex architectures like RNNs or GNNs.

In this context, a suitable baseline is a **frequency-based approach**. Specifically:

[1]More info at: `https://cuckoo.readthedocs.io/en/latest/introduction/`



api_call_1 = [‘RegOpenKeyExA’, ‘NtOpenKey’, ‘RegOpenKeyExA’, ‘NtClose’]
api_call_2 = [‘NtOpenKey’, ‘NtQueryAttributesFile’, ‘NtClose’, LdrGetDllHandle]

$\downarrow$

vocabulary = {‘RegOpenKeyExA’, ‘NtOpenKey’, , ‘NtClose’, ‘NtQueryAttributesFile’, LdrGetDllHandle’}

$\downarrow$

Index API Call	RegOpenKeyExA	NtOpenKey	NtClose	NtQueryAttributesFile	LdrGetDllHandle
1	2	1	1	0	0
2	0	1	1	1	1

Figure 1: Example of Frequency-Based Baseline, given a dataset of 2 API calls

- Extract the vocabulary from your input dataset - that is, the **set of all the API calls** appearing in it.
- Use this vocabulary as the **feature set**: for each row in the input dataset, count the **number of times** (frequency) each vocabulary term occurs.
- Generate the **output dataframe**: it should contain the same number of rows as the input dataset, with one column per vocabulary term.

We provide an example of the algorithm in Figure 1.
Now, try to apply the algorithm here! **Extract the vocabularies** from the provided train and test datasets and try answering the following questions:

- **Q:** How many unique API calls does the training set contain? How many the test set?
- **Q:** Are there any API calls that appear only in the test set (but not in the training set)? If yes, how many? And which one are they?

Now, try creating the **new frequency-based dataframes** - one for partition.

- **Q:** Can you use the test vocabulary to build the new test dataframe? If not, how do you handle API calls in the test set that do not exist in the training vocabulary?
- **Q:** One issue of this frequency-based approach is that it creates *sparse vectors* (i.e., vectors with many zeros per row): how many non-zero elements per row do you have on average in the training set? How many in the test set[2]? What is the ratio with respect to the number of elements per row?
- **Q:** The original API sequences were **ordered**. Is it still the case now (i.e., in the frequency-based dataframe, do you still know which API came first)? Why?

Finally, **feed the frequency-based datasets to a classifier**. Choose any classifier here (shallow non-neural; shallow neural; deep neural).
Q: Report how you chose the hyperparameters of your classifier, and the final performance on the test set.
Q: Is the final performance good - even ignoring the order of API calls and handling very sparse vectors?

[2]Count the number of non-zero elements per row, and then compute the average

4.2 Task 2: Feed Forward Neural Network (FFNN)

Now, move on to more complicated solutions.

With the frequency-based approach, we completely ignored the number of API calls per sequence. Therefore, obtain some statistics on the number of processes called per sample:

- **Q:** Do you have the same number of API calls for each sequence? If not, is the distribution of the number of API calls per sequence in the training set the same as the test one?
- **Q:** The first neural approach we learned involves **FFNN**. Can a FFNN handle a variable number of elements? If not, why?

Regardless of the answers above, let us say that you really want to use a simple FFNN to solve the problem. To do so, you need to make sure that each sequence has the **same number of API calls** (i.e., that each row turns into a fixed-size vector).

- **Q:** How to estimate a *fixed-size* candidate? What partition do you use to estimate it?
- **Q:** Given the estimate of the previous point, what technique could you use to obtain the same number of API calls per sequence?
- **Q:** Suppose that at test time you have more API calls than the fixed-size you estimated. What do you do with the API calls exceeding such fixed-size?

One last problem before running the FFNN: The API calls are **strings**, and hence you are handling **categorical features**. As you learned during lectures, you need to find a **mapping** from the categorical features to a numerical space the network can process. Use two approaches:

- **Sequential identifiers**: map each API call to an arbitrary sequential ID. Remember to use consistent IDs (if you say that `RegOpenKeyExA` is 1, then it is always 1!).
- **Learnable Embeddings**: let the model autonomously learn a good mapping. In such a case, play with different embedding sizes.

Q: Use a FFNN in both cases. Report how you selected the hyper-parameters of your final model, and justify your choices.

Q: Can you obtain the same results for the two alternatives (sequential identifiers and learnable embeddings)? If not, why?

4.3 Task 3: Recursive Neural Network (RNN)

FFNN cannot model well sequences. Hence, try using a Recursive Neural Network (RNN) to model your problem.

- **Q:** With RNNs, do you still have to pad your data? If yes, how?
- **Q:** Do you have to truncate the testing sequences? Justify your answer with your understanding of why it is/it is not the case.
- **Q:** Is the RNN padding - if any - more *memory efficient* with respect to the FFNN's one? Why?
- **Q:** Start with a simple one-directional RNN. Is your network fast as the FFNN? If not, where do you think that time-overhead comes from?
- **Q:** Train and test three variations of networks[3]:

[3]Remember: you will probably need to use a small LR here - e.g., 0.00001

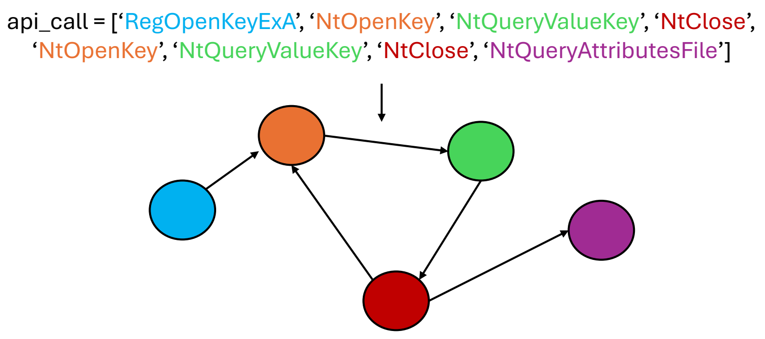

api_call = ['RegOpenKeyExA', 'NtOpenKey', 'NtQueryValueKey', 'NtClose', 'NtOpenKey', 'NtQueryValueKey', 'NtClose', 'NtQueryAttributesFile']

Figure 2: How to convert an API call into a graph - Schema.

  - Simple one-directional RNN
  - Bi-Directional RNN
  - LSTM (choose whether you want it one-directional or bi-directional)

- **Q:** Is the RNNs training as stable as the FFNN's one?
- **Q:** How does your model's performance compare to the simple frequency baseline, given that you now account for the sequence of API calls and use a significantly more complex network?

4.4 Task 4: Graph Neural Network (GNN)

Another way of modeling the sequence of API is through a Graph Neural Network (GNN). Observe the example of Figure 2: each API call becomes a node in the graph, and two nodes are connected if the user moves from one API call to the other - notice that you can have loops and that edges are **not weighted**.

Translate all your sequences into graphs as you learned during the lectures.

- **Q:** Do you still have to pad your data? If yes, how?
- **Q:** Do you have to truncate the testing sequences? Justify your answer with your understanding of why it is/it is not the case.
- **Q:** What is the advantage of modeling your problem with a GNN with respect to an RNN in this scenario? However, what do you lose?
- **Q:** Finally train and tune variations of GNN considering different message and aggregation functions and architectures:
  - Simple GCN
  - GraphSAGE
  - and GAT
- **Q:** How does each model perform with respect to the previous architectures? Can you beat the baseline? :P