Project 2: Model Engineering

Scursatone Matteo - s332153, Lorenzato Fabio - s332186, Milani Alessandro - s332136
Politecnico di Torino

21-04-2024

Contents

Some Considerations about the Dataset and Methodologies

Before starting the report, we need to specify some key points regarding the dataset and the applied methodologies:

- The **field md5hash was ignored** during preprocessing because was a unique identifier for each sample and did not contribute to the analysis of API call sequences as, for the nature of the hash, it does not provide meaningful information about the sequence itself.
- In the analysis we **focused on the minority class** (goodware) as, given the imbalance of the dataset, the majority class (malware) always got metrics close to 1.00 that also lead to **misleading accurancy and averages performance** evaluation.
- All the results and classification reports presented in this report are based on test set performance.

1 Task 1: BoW Approaches

To begin the analysis of the dataset using a Bag-of-Words approach, we treated each sequence of API calls as a single sentence, where each API represents a word. More concretely, we join the list of APIs into a single "string" for each entry in the dataset.

After applying the **CountVectorizer**, we obtained a matrix with 30,713 rows and 253 columns. Each row is a sparse vector representing the word counts for a single entry, and the 253 columns correspond to the vocabulary extracted from the training dataset. However, this process results in the loss of the original order of API calls, which may affect downstream analysis for this type of dataset.

When transforming the test dataset, we used the vocabulary derived from the training set. As a result, five API calls in the test set were **out-of-vocabulary** (OOV)—they were not present in the training data ('obtainuseragentstring', 'controlservice', 'ntdeletekey', 'wsasocketa', 'wsarecv'). Due to the default behavior of **CountVectorizer**, these OOV terms were ignored during vectorization.

1.1 Model Training

For the model training phase, we chose a simple
`LogisticRegression` classifier with a maximum of 1000 iterations. All other parameters were kept at their default values, except for the `random_state`, which was set to ensure reproducibility.
This initial baseline classifier yielded the following results on the test dataset:

Table 1: Classification Report – Logistic Regression (Baseline)

	Precision	**Recall**	**F1-score**
Goodware	0.65	0.28	0.40
Malware	0.98	1.00	0.99
Accuracy			0.98
Macro avg	0.82	0.64	0.69
Weighted avg	0.97	0.98	0.97

The classifier performed very well on the majority class (Malware), achieving a recall of 1.00. However, it struggled significantly with the minority class (Goodware), achieving a recall of only 0.28. While the overall accuracy and weighted averages appear high, these metrics are misleading due to the extreme class imbalance in the dataset.

- **C**: loguniform(1e-4, 1e2)
- **penalty**: ['l1', 'l2']
- **max_iter**: [500, 1000, 2000, 3000, 5000]
- **class_weight**: [None, balanced]

As results, we obtained the parameters present in Table 2.
That lead to the following results 3:
The optimized model still performing quite poorly on the minority class (label 0), achieving a recall of 0.34, which is an improvement over the baseline but still far from satisfactory. The overall accuracy remains high due to the class imbalance, and the weighted averages reflect this trend.

Table 2: Best Hyperparameters for Logistic Regression

Hyperparameter	**Value**
C	89.80
penalty	l2
max_iter	3000
class_weight	None

Table 3: Classification Report - Logistic Regression (Optimized)

	Precision	**Recall**	**F1-score**
Goodware	0.63	0.34	0.44
Malware	0.98	1.00	0.99
Accuracy			0.98
Macro avg	0.81	0.67	0.72
Weighted avg	0.98	0.98	0.98

2 Task 2: Feed Forward Neural Network

A peculiarity of the FFNN models is the **requirement of fixed-length inputs**. To address this, we start by analyzing the distribution of the length of the sequences in the training set. The histogram in Figure 1 shows a distribution of lengths ranging from approximately 60 to 100, rather than a single fixed length. In particular, the test set contains sequences between 60 and 90, while the training set contains sequences between 70 and 100.

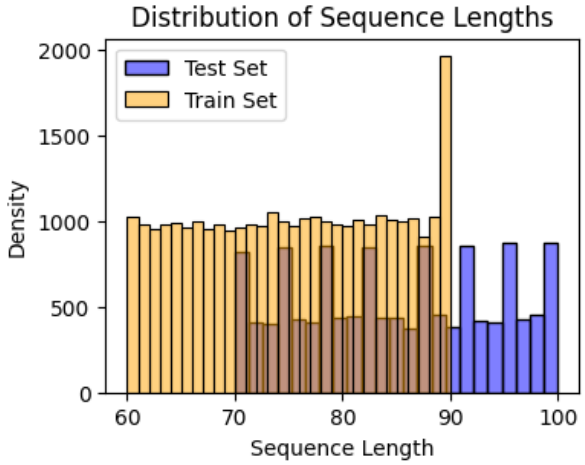

Figure 1: Histogram of Sequence Lengths in the Training Set

This situation need to be addressed, as the FFNN models require fixed-length inputs. Because an FFNN's architecture is defined by layers of neurons, where each connection has a specific weight. The first linear layer is designed to accept an **input vector of a fixed size**.

To address this variability, we apply a **combination of padding and truncation**. For keeping some distinction between training and test set, we decided to use only the data from the training set to determine the maximum length of the sequences.

We then create a dedicated function (*pad_truncate_sequences*) to pad or truncate the sequences to this maximum length (that for the current case is 90). The padding is done with a <PAD>tokens, and the truncation simply cuts off any excess length beyond the maximum.

For how we decided to handle the max length, this function will pad the sequences in the training set to 90, while for the test set we will get some padded sequences to 90 and some truncated sequences to 90.

2.1 Models Training - Sequential IDs vs. Learnable Embeddings

Initially, two approaches were tested: using sequential integer identifiers (each unique API call is assigned an **integer ID**) and using learnable embeddings (convert the ID in a **trainable dense vector**). This first execution is based on 2 models, both with 4 layers (input, hidden, hidden, output), relu activation for the hidden ones and 2 dropout layers with a 0.3 rate.

As optimized we chosen a common Adam with a learning rate of 0.001, as criterion BCEWithLogitsLoss[1] and to end we also retrain them with a weighted loss function to address the class imbalance.

Table 4: FFNNs Results for Goodware class

Model	Precision	Recall	F1-score
Sequential IDs	0.693	0.380	0.488
Embeddings	0.816	0.697	0.752
Sequential IDs (Weighted)	0.692	0.389	0.498
Embeddings (Weighted)	0.747	0.657	0.699

The comparison on table 4 clearly showed that the FFNN with Embeddings performed significantly better, achieving an F1-score of 0.752 for the minority class compared to 0.488 for the model with sequential IDs.

As clearly noted in the results, the two encoding methods not guarantee the same level of performance and results and also the loss curves during training exhibited different behaviors, with the embedding-based model showing more stable and longer convergence.

2.2 Optimization

Given the superior performance of the embedding-based model, it was selected for hyperparameter optimization using Optuna[2]:

- **embedding_dim**: [16, 32, 64, 128]
- **hidden_size**: [64, 128, 256]
- **dropout_rate**: [0.1, 0.5]
- **learning_rate**: [1e-5, 1e-2] (log-uniform)
- **batch_size**: [32, 64, 128]
- **num_layers**: [1, 2, 3, 4]

The optimization process was configured to run for 30 trials with the objective of maximizing the F1-score for the minority class (goodware). The best hyperparameters found were:

- **Embedding Dimension**: 32
- **Hidden Size**: 256
- **Dropout Rate**: 0.417
- **Learning Rate**: 0.002
- **Batch Size**: 32
- **Number of Layers**: 1

Table 5: Classification Report - FFNN (Optimized)

	Precision	Recall	F1-score
Goodware	0.85	0.69	0.76
Malware	0.99	1.00	0.99
Accuracy			0.99
Macro avg	0.92	0.85	0.88
Weighted avg	0.99	0.99	0.99

[1] BCEWithLogitsLoss combines a Sigmoid layer and binary cross-entropy loss (BCELoss) into one function, improving numerical stability during training

[2] Compared to a Grid Search, Optuna enables efficient hyperparameter optimization by adopting state-of-the-art algorithms for sampling hyperparameters and pruning efficiently unpromising trials

3 Task 3: Recursive Neural Network

3.1 Padding management

Differently from the FFNN models, with RNN theoretically **we could avoid the need for padding** sequences to a fixed length, as RNNs can process variable-length sequences.

However, in practice, we still need to apply some padding. To be more precise we don't need to have all the sequences of the same length, but we need to ensure that **all sequences in a batch are padded to the same length** to guarantee efficient processing and the possibility to use GPU acceleration.

To correctly manage the padding, we can use PyTorch's *pad_sequence* function, which allows us to pad a list of variable-length sequences to the same length during the creation of the batches with a special token <PAD>, and *pack_padded_sequence* to inform the RNN about the actual lengths of the sequences and so avoid unnecessary computations during training on padded section of the sequence.

This kind of management of the padding sequence plus the ability of the RNNs to process variable-length sequences, **remove the necessity to truncate the testing sequences**, allowing us to use the full context of each sequence (to keep the coerence with the training, during the creation of the dataloader also the testing sequences are padded). The most likely outcome is that the models aren't going to perform very well on test entry with a length higher than the maximum length seen during training (but also an eventually truncation of the sequences could lead to a loss of information).

3.2 Memory characteristics

In addition to padding management, it's crucial to consider the **memory characteristics of RNNs** compared to FFNNs. RNNs process inputs sequentially and **maintain a hidden state** that is updated at each time step, which enables them to reuse memory across timesteps and maintain a form of temporal memory. This sequential processing makes RNNs more efficient and expressive when dealing with sequences in a first phase, as they do not require flattening or heavy padding of input data like FFNNs do. However, this sequential nature can lead to higher memory consumption for very long sequences.

3.3 RNN speed

We observed that training a simple mono-directional RNN (processing data left-to-right) is significantly slower than training a Feed Forward Neural Network (FFNN). Specifically, the **RNN requires about 9 times more time** per epoch than the FFNN (9.9 s/epoch vs 1.1 s/epoch) this difference in training time can be attributed to the sequential nature of RNNs, which makes them inherently slower to train compared to the architecture of FFNNs, the need to do more memory intense work to manage the hidden state, the unpack of padded elements and the calculation of Backpropagation Through Time (BPTT).

3.4 RNN architectures comparison

As done with the FFNNs, we also compared different RNN architectures:

- **Simple RNN**: monodirectional processing of the data - left to right
- **Bi-directional RNN**: Get information from both left and right context
- **LSTM**: Long Short-Term Memory networks, designed to better manage information for long periods with forget and memory gates

All the model use a CrossEntropy loss function, the Adam optimizer with a learning rate of 0.001 and have a 2-layer architecture plus a dropout layer with a 0.3 dropout rate:

Table 6: Comparison of RNN Models (Minority Class Metrics)

Model	**Precision**	**Recall**	**F1-Score**
Simple RNN	0.781	0.549	0.645
Bi-directional RNN	0.867	0.685	0.766
Bi-directional LSTM	0.903	0.719	0.801
Simple RNN (Weighted)	0.289	0.806	0.425
Bi-directional RNN (Weighted)	0.363	0.784	0.497
Bi-directional LSTM (Weighted)	0.471	0.864	0.610

From the results, we can observe that the LSTM model consistently outperforms the other RNN architectures across all metrics, indicating its superior ability to capture long-range dependencies in the data. The Bi-RNN also shows improved performance compared to the Simple RNN, highlighting the benefits of bidirectional processing in sequence modeling.

This gain in performance for Bi-RNN and LSTM arrive at a cost of a slower training time (about 9 s/epoch for the simple RNN, about 13 for the Bi-RNN and about 15 for the LSTM), which is expected due to the increased complexity of these models. But we can also notice how the best epoch is reached much earlier within the 50 training epochs as opposed to the simple RNN that reach the best epoch only at the 39th epoch and with a more unstable process. So we can say that the LSTM model is more stable and converges faster than the other two models and with a current suppression mechanism the training time also could be reduced.

This trend is further reinforced when considering the weighted versions of the models, where the LSTM (Weighted) still maintains a competitive edge, particularly in recall, suggesting it is better at identifying true positives in the minority class but with heavy loss on the precision.

3.5 Optimization

Given the previous results, we aim to optimize the Bi-directional LSTM model, as it demonstrates the best performance on the minority class. As done for the FFNN, we use Optuna for hyperparameter optimization. The hyperparameters we decided to optimize are:

- **embedding_dim**: [16, 32, 64, 128]
- **hidden_dim**: [64, 128, 256]
- **dropout_rate**: [0.1, 0.5]
- **learning_rate**: [1e-5, 1e-2] (log-uniform)
- **batch_size**: [32, 64, 128]
- **num_layers**: [1, 2, 3, 4]

After 30 trials, we found the best hyperparameters for the Bi-directional LSTM model to be:

- **embedding_dim**: 128
- **hidden_dim**: 64
- **dropout_rate**: 0.338
- **learning_rate**: 0.005
- **batch_size**: 64
- **num_layers**: 4

Table 7: Classification Report - Bi-directional LSTM (Optimized)

	Precision	**Recall**	**F1-score**
Goodware	0.91	0.64	0.75
Malware	0.99	1.00	0.99
Accuracy			0.99
Macro avg	0.95	0.82	0.87
Weighted avg	0.99	0.99	0.99

4 Task 4: Graph Neural Network

4.1 Padding and truncation

The GNN, compared to FFNN and RNN **does not require any kind of padding or truncation** for both training and test set, as each sequence is treated as a graph, where each API call is a node and the connections between them are edges (to do this work is needed an extra library, such as PyTorch Geometric). This allows the GNN to naturally handle variable-length sequences without the need for padding or truncation.

4.2 Strengths and Weaknesses of GNNs

Strengths: GNNs excel at modeling *structural* relationships rather than relying solely on sequence order. Unlike FFNNs, which ignore ordering entirely, and RNNs, which emphasize strict sequential flow, GNNs explicitly represent relational dependencies between elements (e.g., API calls) through their graph structure and message-passing mechanisms.

This makes them especially powerful in scenarios like malware detection, where the same API calls may occur in different orders but still indicate malicious behavior. GNNs can capture such order-invariant patterns while retaining contextual information from the graph.

Moreover, GNNs offer flexibility:

- No fixed input dimensionality is required (unlike FFNNs).
- No need for padding or truncation of sequences, which reduces information loss and memory overhead for variable-length inputs.

Trade-offs: GNNs come with higher computational and preprocessing costs. Converting sequential data into graph form introduces additional complexity, and the message-passing process increases training time.

They may also struggle to preserve fine-grained sequential information compared to RNNs. In cases where the exact order of elements is critical, relying on node proximity rather than strict sequence order can weaken predictive performance.

Graph Construction Process: To transform each sequence into a graph, API calls (after the one-hot encoding) are treated as nodes with the encoded API value as the only feature of the corresponding node, edges connect consecutive API calls, forming a chain of all the API calls.
To each resulting graph is assigned the binary label indicating whether it is *malware* or *benign* of the original sequence.

4.3 Models Training

4.3.1 GCN CPU vs GCN GPU

To analyze the **different execution time** of a GCN model (GNN models that use the entire adjacency matrix to propagate information) on CPU and GPU, we trained a simple GCN model with 2 *GCNConv* layers, using the Adam optimizer with a learning rate of 0.001 and CrossEntropy loss function. The training was performed for 20 epochs.

At the end of the training (Done of the Keggle platform with a 4 core CPU and a NVIDIA Tesla P100), we observed 115.25 Seconds for the CPU run and 68.93 Seconds for the GPU run, showing a **significant speedup when using the GPU of the 1.67x**. (results in table 8)

4.3.2 Other architectures

Other than GCNs, we also tested other GNN architectures, such as:

- **GraphSAGE**: samples a fixed-size set of neighbours instead of using the full adjacency matrix and use different aggregation strategies to update node representations
- **GAT**: uses an attention mechanism to weigh neighbours differently

For both these architecture we keep the same training procedure as for the GCN model and generic architecture (2 convolutional layers, some optimizer and loss function). We also save the time required for GraphSAGE and GAT training on GPU that was 63.15 seconds and 84.67 seconds respectively.

From the results in Table 8, we can observe that **GAT consistently outperforms** both GCN and GraphSAGE across all metrics for the minority class, achieving the highest F1-score of 0.645. The attention mechanism in GAT appears to be particularly effective for this task, allowing the model to focus on the most relevant neighboring nodes when making predictions.

The weighted versions of all models show improved recall but at the cost of significantly reduced precision.

4.4 Optimization

From the results of the first training, we can see that the GAT model performs better than the other two architectures, so we decided to optimize it with an Optuna hyperparameter optimization process. The hyperparameters we decided to optimize are:

Table 8: Comparison of GNN Models (Minority Class Metrics)

Model	**Precision**	**Recall**	**F1-Score**
GCN	0.743	0.330	0.457
GraphSAGE	0.760	0.343	0.472
GAT	0.764	0.559	0.645
GCN (Weighted)	0.171	0.806	0.282
GraphSAGE (Weighted)	0.180	0.824	0.296
GAT (Weighted)	0.179	0.830	0.295
GCN (GPU)	0.715	0.302	0.425
GCN (Weighted, GPU)	0.168	0.809	0.279

- **hidden_dim**: [32, 64, 128, 256]
- **num_layers**: [1, 4]
- **dropout_rate**: [0.1, 0.5]
- **learning_rate**: [1e-5, 1e-2] (log-uniform)
- **batch_size**: [32, 64, 128]

After 30 trials, we found the best hyperparameters to be:

- **hidden_dim**: 128
- **num_layers**: 4
- **dropout_rate**: 0.123
- **learning_rate**: 0.000169
- **batch_size**: 32

Table 9: Classification Report - GAT (Optimized)

	Precision	**Recall**	**F1-score**
Goodware	0.89	0.67	0.77
Malware	0.99	1.00	0.99
Accuracy			0.99
Macro avg	0.94	0.83	0.88
Weighted avg	0.99	0.99	0.99