Named Entity Recognition (NER) Competition for Cybersecurity at IPSA

Britney Hong, Si-Thami Khaif
Professor : Atilla Alkan

ABSTRACT

For the course "Hand-on Machine Learning for Cybersecurity", this project focuses on developing a system that automatically recognizes domain-specific entities in the cybersecurity field. This project is part of a Named Entity Recognition (NER) competition, which aims to extract meaningful insights from cybersecurity reports using Natural Language Processing (NLP) techniques.

1. Introduction

Nowadays, the field of cybersecurity is evolving rapidly so the ability to accurately recognize and categorize entities within large volumes of data is crucial. NER systems play a important role in identifying and classifying entities such as IP addresses, file paths, and malware names, thereby enhancing the ability to detect and respond to threats effectively. Our project focuses on developing a sophisticated NER system tailored specifically for cybersecurity applications. The dataset used for this project is derived from SemEval-2018 Task 8, "Semantic extraction from cybersecurity reports using natural language processing (SecureNLP)". This dataset aims to identify and classify critical entities related to cybersecurity, such as malware, attacks, and malicious actors, in technical documents.

The project involves several stages, including data exploration and preprocessing, model selection and experimental settings, and results analysis. Through this project, we will apply machine learning techniques to enhance cybersecurity measures by automating the recognition of crucial entities in textual data. In this document, we will explain our project in a structured manner, starting with data exploration and preprocessing steps. We will then explain how we trained the model, and the model architecture we decided to create for the project. Finally, we will conclude with the prediction phase, showcasing the outcomes of our project.

2. Data Exploration and Pre-processing

The first step in this project is to familiarize with the data set, it is important to examine the data in detail to understand its structure and content. After analyzing the dataset, we will gain insights into the various entity types and their frequencies, which is crucial for effective model training.

The next step is to make to prepare the data for the training, so that the data is in optimal condition to learn and make accurate predictions. For that, we have to make tokenization, cleaning the data, remove any noise and converting the data into a format suitable for the model.

2.1. Loading Data

So first, we need to load raw data and then convert it into a suitable format for the training. JSONLines files are opened and read line by line. Each line is a JSON record containing crucial information such as tokens, NER tags and a unique identifier.

Then we need to extract the information. Tokens, NER tags and unique identifiers are extracted from each record. This information is essential for training a NER model, as it provides the text to be analyzed (tokens) and the target classes (NER labels) for each word in the text.

The extracted data are structured in the form of list of dictionaries, where each entry contains a token, its NER label and the identifier of the sentence to which it belongs. This enables the context of each word in the text to be preserved.

We decided to use Pandas DataFrames for efficient data manipulation and analysis. Indeed DataFrames offer a powerful and flexible data structure for manipulating and analyzing data. They enable transformations, filters and aggregations to be applied efficiently.

	index	tokens
0	1357	Stage
1	1357	3
2	1357	exports
3	1357	hundreds
4	1357	of
...	...	...
26121	1251	POST
26122	1251	URI
26123	1251	of
26124	1251	/bbs/info.asp
26125	1251	.

[26126 rows x 2 columns]

We have chosen to save our data in parquet files, as this has several advantages for our data volume. Indeed, it has better compression, more flexibility and therefore better performance.

2.2. Cleaning the Data

In this step, we need to analyze the data in order to find if there are any excess elements. But first It is important to understand the representation of each column :

1. **unique_id**: an integer
2. **token**: a string, the heart of our dataset
3. **ner_tag**: our label

	index	tokens	ner_tags
0	6422	Just	0
1	6422	1	0
2	6422	year	0
3	6422	later	0
...			

We've noticed that our dataset contains a number of unnecessary values that could be detrimental to the training of our model. For example, special characters and capital letters, which are irrelevant in this context. In this step, we will remove them to optimize data quality for training:

```
dimensions avant nettoyage :  (116594, 3)
dimensions avant nettoyage :  (25418, 3)
dimensions avant nettoyage :  (26126, 2)
```

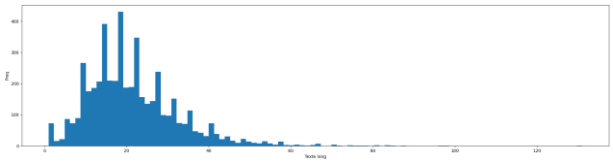
```
dimensions après nettoyage :  (104361, 3)
dimensions après nettoyage :  (22787, 3)
dimensions après nettoyage :  (23394, 2)
```

We can see that the dimension has been reduced.

We now select only the relevant values from our DataFrame, in order to train our model optimally, and we obtain :

index	tokens
1	Stage
1	3
1	exports
1	hundreds
1	of

Here, we try to analyze the data in order to identify possible outliners that could distort our model. We display a histogram of the text length to visualize the distribution, which helps us to identify outliners and potential problems with the data.



Now our DataFrames has been clean up by removing tokens that are punctuation marks or special characters, so we can improve the data quality for the model training.

3. Training Model

After finishing the pre-processing, the next step is the training model. Indeed, this step enables the model to learn from data, adjust its parameters to minimize errors, generalize its knowledge to new data, and finally be used for practical applications.

In our case of named entity recognition, this enables the model to automatically understand and identify important entities in texts.

3.1. Label encoding

To prepare the data for use by machine learning algorithms, we need to use a method called Label Encoding. The principle behind this technique is to convert categorical columns into numerical ones so that they can be fitted by machine learning models which only take numerical data.

index	tokens	ner_tags
1	Just	7
1	1	7
1	year	7
1	later	7
1	after	7

This ensures that all data is in a compatible format and optimized.

3.2. Tokenization

In this step, we will tokenize the data to segment it into smaller units (such as words or sentences), to facilitate processing by machine learning models.

First we need to initialize the tokenizer using the Keras Tokenizer to convert text into numerical sequences with specified filters to

eliminate certain characters, maintaining case sensitivity, limiting the vocabulary to the 2000 most frequent words, and managing out-of-vocabulary words.

The tokenizer is then fitted to the training, validation, and test sets using a function fit_on_texts, which builds the vocabulary.

The text is converted to numerical sequences where each token is transformed into an integer based on the learned vocabulary, ensuring each word is replaced by its index in the vocabulary.

Tokens that are empty after filtering are removed to ensure that only valid sequences remain in the training and validation datasets.

The data is then concatenated by index to form complete sequences of tokens and labels, ensuring each sequence represents a full sentence or logical unit. The concatenated tokens and labels are extracted for training, validation, and testing purposes.

```
                                                  tokens  \
index
1      1 7 645 4 589 129 6 1173 1 1 5 1 1 2 1 1 24 1 ...
2      36 1022 1 1 6 1 1 681 1761 1 682 1906 6 1 4 17...
3      387 2 1 4 308 1 46 38 1 13 308 38 30 18 1081 1...
4      166 31 1 2 55 269 388 16 1639 31 35 233 2 62 1...
5          10 1 1 1282 15 548 12 56 4 123 1082 14 469 44

                                                ner_tags
index
1      7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 ...
2      7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 2 7 1 4 ...
3      7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 ...
4                    7 7 7 7 7 7 7 7 7 2 7 1 2 5 5 5 5
5                          7 7 7 7 7 7 7 7 7 7 7 7 7
                                                  tokens  \
index
1      1 68 707 384 255 1124 47 1601 15 205 103 57 17...
2      36 578 118 76 4 105 301 111 71 38 13 6 1 4 481...
3      26 1760 20 1303 1387 7 9 577 1095 9 5 9 1 4 12...
4      1536 2 116 318 62 2 320 1 29 1 113 5 903 1116 ...
5      1 355 14 629 770 3 1016 6 1 1793 4 2 1 4 1852 ...

                                                ner_tags
index
...
2      128 279 1658 2 1 4 1 1 1287 1 11 1 3 1 1 143 3...
3      1536 151 4 2 976 287 25 195 1 7 362 236 6 136 ...
4      10 484 8 1212 14 6 136 4 1707 146 81 33 594 3 ...
5      9 215 20 1 5 158 1 34 1780 47 567 7 1302 5 54 ...
```

3.3. Converting Data for the Model Requirements

These datasets are now converted into TensorFlow RaggedTensors, which handle variable length sequences.

TensorFlow datasets are created from these constants ragged which facilitate batch processing and training.

Now, padding functions convert the ragged tensors into dense tensors with padding, which is necessary for training TensorFlow models.

The datasets are prepared by shuffling, batching, transforming for padding, and prefetching to optimize training performance.

So here, we can finally see how the input and output of the model looks like :

```
This is how the input and output to the model finally looks like:
(16, 42)
tf.Tensor(
[ 419  970    1  281    2   62    1 1130   35   23 1077  873    2   62
    1 1130  341  363  132  442  965    5    2  454  889    7  404    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0], shape=(42,), dtype=int32)
**********
(16, 42)
tf.Tensor(
[7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0], shape=(42,), dtype=int32)
```

4. Creating the Model

For our project we decided to use LSTM (Long Short-Term Memory). This model is a variation of Recurrent Neural Network (RNN), which its architecture is designed to address the limitations of traditional RNNs, particularly in learning long-term dependencies. RNNs, which have loops to allow information to persist, often struggle with problems like vanishing and exploding gradients. LSTM networks overcome these issues with a unique structure that enables them to remember information for extended periods. The key components of LSTMs include the cell state, which acts as a conveyor belt for information flow, and gates that control this flow.

In LSTM we have 3 different gates : the forget gate decides what information to discard, the input gate updates the cell state with new information, and the output gate determines the next hidden state.

Input Gate :

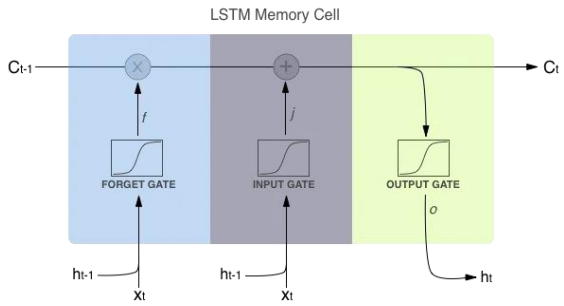
$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

Forget Gate :

$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$

Output Gate :

$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$



Now, from the below equation we can see that at any timestamp, our cell state knows that what it needs to forget from the previous state and what it needs to consider from the current timestamp.

Candidate cell state :

$$\widetilde{c_t} = tanh(w_c[h_{t-1}, x_t] + b_c)$$

Actual cell state :

$$c_t = f_t \times c_{t-1} + i_t \times \widetilde{c_t}$$

Hidden state update:

$$h_t = o_t \times \tanh(c^t)$$

Choosing an LSTM model is advantageous because it handles long-term dependencies, addresses the vanishing gradient problem, and is versatile across various tasks. LSTMs are particularly effective in natural language processing (NLP) tasks such as named entity recognition, as well as in time series prediction, speech recognition, and anomaly detection. The ability of LSTMs to process information bidirectionally (forwards and backwards) is beneficial for contexts where understanding both past and future information is essential.

In our project, the LSTM model is used to process sequences, likely for an NLP and sequence prediction task. This is ideal to handle sequential data, capturing context and dependencies between words.

Indeed, for the project our model includes both forward and backward LSTM layers to enhance context understanding, and combines outputs from both directions for predictions, ensuring comprehensive sequence comprehension. The extensive use of dropout layers in the model helps prevent overfitting, promoting better generalization to unseen data.

Furthermore, our model incorporates dense layers, which are used to transform the learned features into the desired output format. Dense layers apply learned weights to the input features and pass them through an activation function, enabling the model to learn complex representations and make accurate predictions. By combining dense layers on top of the LSTM layers, we can refine the output, ensuring that the final predictions are both precise and robust. This combination of LSTM and dense layers allows the model to effectively handle the intricacies of sequence prediction tasks in our project.

4.1. Forward Layer and Backward Layer

First we start by defining the hidden sizes for the LSTM and Dense layers, both set to 100. The process starts by allowing for flexible input sequence lengths, transforming input tokens into dense vectors suitable for LSTM processing.

The forward layer processes these embeddings through a series of steps designed to extract and refine features while preventing overfitting.

This involves a first layer of LSTM to handle sequential data, followed by dropout layers to avoid overfitting, and dense layers to enhance the network's learning capability.

The output from these layers is combined and further processed through a second LSTM layer, ensuring the model captures complex patterns in the data.

Like the forward layer, the backward layer is constructed to process sequences in reverse. This mirrors the structure of the forward layer but ensures that it processes the input data in the opposite direction, adding depth to the model's understanding by considering both past and future contexts.

4.2. Combining Outputs

The forward LSTM outputs are processed through dropout layers and combined with dense layer outputs using additive layers. By using a softmax layer, predictions are generated for each time step in the sequence.

After defining both forward and backward layers, the outputs from both layers are then combined. The final prediction is a weighted combination of the outputs from the forward and backward layers, ensuring a comprehensive understanding of the input data.

4.3. Training

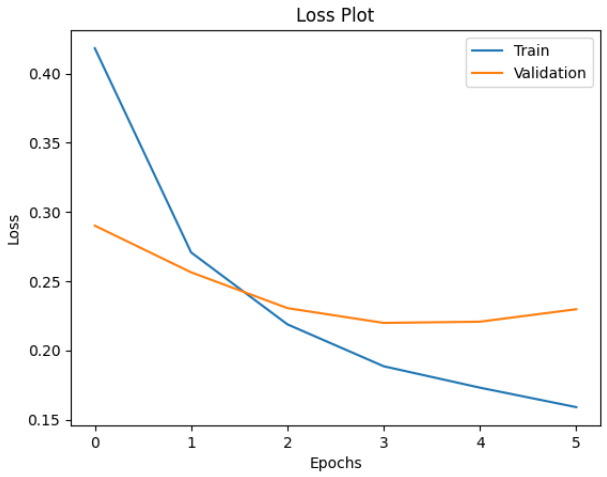
The model is then compiled using Adam optimizer with a learning rate of 5e-4 and sparse categorical cross-entropy loss. Early stopping and model checkpoint callbacks are used to prevent overfitting and save the best model.

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, None)	0	-
embedding (Embedding)	(None, None, 300)	600,000	input_layer[0][0]
not_equal (NotEqual)	(None, None)	0	input_layer[0][0]
lstm_2 (LSTM)	(None, None, 100)	160,400	embedding[0][0], not_equal[0][0]
dropout_3 (Dropout)	(None, None, 100)	0	lstm_2[0][0]
dense_1 (Dense)	(None, None, 100)	30,000	embedding[0][0]
lstm (LSTM)	(None, None, 100)	160,400	embedding[0][0], not_equal[0][0]
dense (Dense)	(None, None, 100)	30,000	embedding[0][0]
get_item (GetItem)	(None, None, 100)	0	dropout_3[0][0]
dropout_4 (Dropout)	(None, None, 100)	0	dense_1[0][0]
dropout (Dropout)	(None, None, 100)	0	lstm[0][0]
dropout_1 (Dropout)	(None, None, 100)	0	dense[0][0]
add_3 (Add)	(None, None, 100)	0	get_item[0][0], dropout_4[0][0]
add_1 (Add)	(None, None, 100)	0	dropout[0][0], dropout_1[0][0]
lstm_3 (LSTM)	(None, None, 100)	80,400	add_3[0][0]
lstm_1 (LSTM)	(None, None, 100)	80,400	add_1[0][0]

The model is trained on the provided data for 40 epochs, and the training process is monitored for loss and accuracy, we now plot these metrics to evaluate the performance and detect overfitting signs.

4.4. Performances Graphs

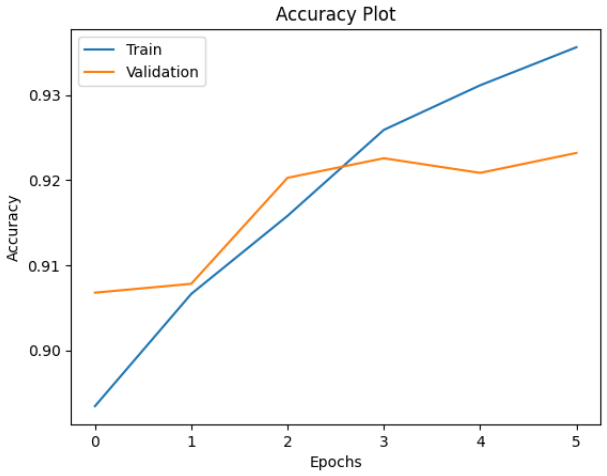
Here, we have the plots of the training and validation loss over epochs to monitor training progress and detect overfitting.



The loss of the training set is steadily decreasing, which is a good sign that the model is learning well on this set.

The loss of the commit set also decreases, and appears to stabilize but increase slightly at the end of the epochs. This indicate the onset of overfitting, where the model begins to memorize training data rather than learning.

Now, we have the plots of the training and validation accuracy over epochs to evaluate the model's performance.



The accuracy of the drive set is steadily increasing, confirming that the model is improving on the drive set.

The accuracy of the validation set also increases but seems to stabilize or fluctuate slightly at the end of the epochs. This indicate the observation of the loss curve, suggesting overfitting.

With the bidirectional LSTM's ability we can now capture temporal dependencies in both forward and backward directions, enhanced by dense layers and dropout for robust learning and generalization.

5. Testing Phase and Model Refinement

Once the model was completed, we had to send it to our professor to evaluate it on a separate dataset using the CoNLL-2000 shared task metric via the seqeval library, as required for the NER task. Our F1-score at the entity level was **0.023**, indicating that the model struggled to effectively identify the three key classes: Action, Modifier, and Entity. This poor performance highlighted two major issues. First, the initial model suffered from overfitting, failing to generalize and distinguish between different ner_tags. Secondly, our training dataset was highly imbalanced, with certain classes being overrepresented while others were significantly underrepresented.

labels	
O	95878
I-Entity	12644
B-Entity	4240
B-Action	1989
B-Modifier	1226
I-Action	518
I-Modifier	99

These challenges underscored the need for a more robust architecture and a better strategy to handle data imbalance.

To address these issues, we decided to change our model to **Sci BERT**. This model is a variant of BERT (Bidirectional Encoder Representations from Transformers) that has been pretrained on scientific texts. This specialization makes it well-suited for handling domain-specific tasks like ours, where understanding the nuances of technical language is critical.

SciBERT's architecture uses essential equations to effectively process and interpret sequences of scientific text. Each input token x_i is first converted into a dense vector e_i through an embedding layer, with positional encodings p_i added to integrate sequential context, resulting in :

$$e_i^{input} = e_i + p_i$$

This ensures that the model captures the order of tokens within the sequence.

The advantage of this model is its self-attention mechanism, which enables each token to dynamically focus on every other token in the sequence, as expressed by the equation:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Where Q, K, V are the query, key, and value matrices derived from the token embeddings, and d_k is the dimensionality of the keys. This mechanism enables the model to weigh the importance of different tokens relative to each other.

After fine-tuning the SciBERT model on our dataset, we observed a notable improvement in its performance.

To address the class imbalance in our dataset, we calculated specific weights for each class based on their relative frequency. Using the compute_class_weight function, we assigned higher weights to minority classes, such as Modifier and Entity, ensuring they were better represented during training. These weights were integrated into our SciBERT-based NER model through its configuration arguments. This approach prevented the model from overly favoring majority classes and improved its ability to recognize less frequent entities. Combined with other optimizations, such as continuous evaluation and early stopping, this strategy significantly enhanced the model's overall precision and F1-scores.

Initially, when we trained the model for 3 epochs, the Action class achieved the F1-score at **0.56**, demonstrating the effectiveness in identifying actions within the data. However, after further analysis of the global F1-score evolution across the training epochs revealed room for optimization. By evaluating the model on the validation set at each epoch, we noticed that performance continued to improve until around the 6th epoch, after which the performance stagnates. Using the model trained for 6 epochs, we achieved an even higher F1-score of **0.62** for the Action class on the test set. Similarly, the Modifier and Entity classes also showed marked improvements compared to earlier results, confirming the importance of careful tuning of training parameters to maximize the model's potential.

```
overall_precision
  0.6252342286071205
overall_recall
  0.6182828906732551
overall_f1
  0.6217391304347827
overall_accuracy
  0.9002281847509639
```

By switching our model to SciBERT and modify some parameters it enabled us to significant improvements in precision, recall, and F1-scores across all classes. This highlights the value of using domain-specific models and advanced transformer architectures for complex tasks like NER in specialized datasets.

However, to further enhance of our model's robustness, we can explore an another approach. This involves combining predictions from multiple high-performing models to determine the final prediction for each token through majority voting. For instance, if three models predict different tags for a token (e.g., B-Action, I-Action, and B-Action), the final prediction would be B-Action based on the majority vote.

This ensemble method could leverage the strengths of individual models, potentially improving overall performance and handling nuanced cases more effectively. Despite time constraints, the full implementation of this strategy, it presents an interesting perspective for future work and a valuable direction to explore in further research.

6. Conclusion

In this project, our objective is to build a robust NER system designed for cybersecurity applications. Initially, we implemented a bidirectional LSTM model, which captured contextual dependencies in both forward and backward directions, improving the understanding of each token's context within a sequence. Dense layers enhanced the model's ability to learn complex features, while dropout layers helped mitigate overfitting, ensuring better performance on unseen data.

However, the tests revealed limitations in the initial approach for accurately identifying key entities. To overcome this, we switched to SciBERT, a domain-specific transformer-based model. This change led to significant improvements in precision, recall, and F1-scores across all entity classes, highlighting the importance of using advanced architectures for specialized tasks.

These achievements represent a critical step forward in our project, showcasing the effectiveness of combining domain knowledge with machine learning techniques. The outcomes provide a solid foundation for further development and future advancements in NER systems tailored to cybersecurity.

Throughout the project, we employed various techniques to optimize the model and it could still be improved in the future especially with the suggested approach we mentioned earlier, which helped in maintaining the model's performance while avoiding overfitting.