MLCS – fall 2024
Assignment 4

Instructor: Hans P. Reiser

Submission Deadline: Tuesday, Oct 13, 2024 – 23:59

3 Malware classification with API call traces

In this assignment, we will be using API call traces for classifying executable programs of different kinds of malware. For each sample of malware (or benign program), such traces can be recorded, for example, with the Cuckoo sandbox (to minimize the risk of collateral damage, we'll not use actual executable malware samples, but pre-recorded traces instead).

Dataset: `api-call-traces.tgz` on canvas (at A4).
Each line in `api_trace.csv` contains one trace of a program as comma separated list of numbers (representing different Windows API calls). For each line, a corresponding line with a classification can be found in the file `apt_trace_labels.txt`.

3.1 Simple data preprocessing and splitting (2pt)

In a first step, our objective is to use basic classifiers designed to operate on fixed-size feature vectors. To achieve this, we will convert the variable-sized API call traces into feature vectors by employing a straightforward 'bag-of-words' methodology. In this approach, each feature signifies the presence of an API call within the trace, regardless of its position or order.

Your first task is to implement the necessary preprocessing steps (reading the files, converting the traces into feature vectors, and dividing the dataset into a training set and a test set.)

It might be useful to do one further preprocessing step: Remove repeated API calls (i.e. replace a sequence of N identical calls with a single call).

3.2 Train and test simple classifiers (4pt)

Your next objective is to apply four distinct simple classifiers to the preprocessed data to evaluate their ability to effectively distinguish between various classes based on the bag-of-words representation. You have the flexibility to choose from a range of classifiers, including but not limited to, naïve Bayes, logistic regression, K-nearest neighbors (KNN), decision tree, and random forest.

For each classifier, generate a graphical representation of the confusion matrix (using a seaborn heatmap), and compare the suitability of all classifiers for this task.

3.3 Export your best model for an independent classifier (4pt)

We have a (secret) test set (based on an extended version of the data set above, with exactly the same labels) that will be used for evaluating the model you consider your best. For this part, you have to do the following

- Extend your code from the previous part such that it persists the trained model to a file (see, for example, `https://scikit-learn.org/stable/model_persistence.html`) – make sure to include your persisted model in your submission.
- Implement a separate Python program `classify-trace.py` that takes a file name as command line parameter. Lines in the file are equivalent to lines in your data set. Your program shall classify each line (=trace of a single program) and output a label (just the label, one line of output for each line of input).

Scores will be awarded for (1pt) for any solution that does perform better than a random classifier, (2pt) for any solution that performs better than the worst classifier in the sample solution, (3pt) for any solution that is either better than the (non-optimized) best classifier in the sample solution for 3.2 or better than the average score of all submissions, whatever is lower, and (4pt) for the three best solutions (and any solution that has a score within 0.02 of the third best solution).

Evaluation metric is the average (non-weighted) F1-score over all classes.

Submission

Submit the following files: Documentation (PDF file), Python code (two separate files for part 3.2 and 3.3), file with your persistent model (or any files that you need to run your classifier).

After unpacking your submitted solution, it must be possible to run your classifier on the command line, in the folder where it was unpacked, using
`./classify-trace.py` *`filename`*

Additional (optional) part on next page...

3.4 Optional enhancements (bonus points – up to 2pt)

Implement one additional approach (you may included that approach in your solution for 3.3 if you are convinced that it performs better). Two suggestions (you may choose one of them, or use another approach as long as it is substantially different to the basic approaches used in 3.2).

3.4.1 LSTM

Long Short-Term Memory (LSTM) networks have become popular in recent years, emerging as a powerful machine learning approach. These neural networks possess a good ability to process sequential data, making them particularly well-suited for handling streams of words and text, and might be suitable for processing streams of API calls.

A typical simple LSTM architecture comprises three key components. The initial layer transforms input features, such as words, into numerical feature vectors. These vectors are then processed sequentially by the LSTM node, which maintains a series of internal states to capture temporal dependencies. Finally, the LSTM's output is fed into an output layer, where it is translated into labels when the network is utilized for classification tasks.

If you take this approach, feel free to get help from sources like ChatGPT, and focus on experimenting with parameters. Some hints:

- Start with a one-hot encoding of each API call (i.e. map each call number to a feature vector (as many elements as different API calls, around 400)
- Use a (trainable) embedding layer as your first layer that transforms the large one-hot encoded input vector into a shorter feature vector (size = "dimension" of the embedding).
- The second layer is a single LSTM layer. One important parameter is the number of hidden states (lstm_units in TensorFlow).
- The last layer is maps the LSTM layer output to the output classes. It is a "Dense" layer that has as many elements (and outputs) as there are different classes (labels). The highest output defines the output label of the classification.

3.4.2 N-Grams

Instead of using single API calls, you can use sequences of N consecutive API calls. Note that this will cause a huge amount of possibilities (if there are 400 API calls, in theory there can be up to 400^7 different 7-grams. Many combinations will not exist in the data, so the actual number will be lower.)

You could limit the dictionary size to N-grams that appear more frequently in the data set, but most likely you still would like to use something in the magnitude of 10K to 100K N-grams to obtain a good classifier. Note that not all classifiers will work well with such a large feature space.