Identifying API Access Behavior Anomalies

Remy LeWinter and Rahul Suryadevara

Abstract

APIs can expose components of internal business logic which attackers may try to exploit. Attacks on APIs can have different API access patterns than normal API use. We evaluate various machine learning approaches to binary classification of observed behaviors as either normal or outlier. Our dataset consists of a supervised training set with 1,674 records and an unsupervised testing set of 33,837 records, with 9 features representing API access patterns. We use stratified 5-fold cross-validation with successive halving to tune hyperparameters and evaluate performance of various support vector classifiers, logistic regression models, and Ridge classifier. API access uniqueness was most commonly found to be the most important feature, and all models except the Ridge classifier had perfect training, validation, and ROC scores.

1. Introduction

As companies transition to an agile methodology, microservices are becoming increasingly popular over the traditional monolithic architecture. While a monolithic software application is built as a single self-contained artifact where all functions are managed in one place, microservices are loosely coupled services which each run their own processes and perform specific functions (Figure 1). This transition is motivated by the better scalability, fault tolerance, and integration offered by microservice components. A bug in one microservice does not affect the entire application, nor does scaling involve updating the entire application [1]. The granularity of microservices also allows for faster software delivery and reduced downtime, which result in higher customer satisfaction [2].

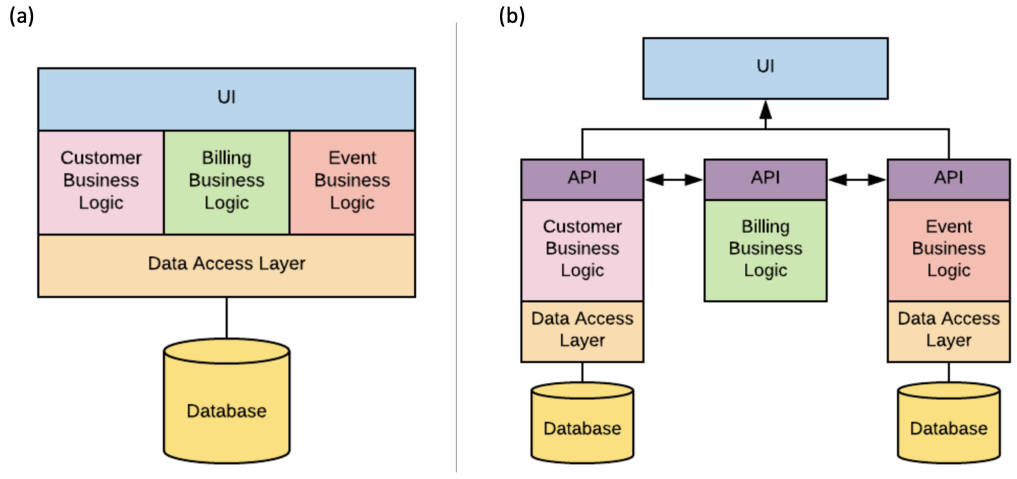


Figure 1: Application architecture as a (a) monolith and (b) microservices. UI = User Interface; API = Application Programming Interface.

Application Programming Interfaces (APIs) are a fundamental part of maintaining the granularity of microservices. An API is the software interface that allows communication between the microservices and the distribution of the microservice to users. Without APIs, the microservices become disconnected and can no longer be used by the customers or developers [3]. Common examples of APIs include weather snippets, integrated Google Maps, Pay with PayPal, Facebook app login, etc. Due to the convenient pipelining of data between microservice platforms and to users, cybercriminals have increasingly been targeting APIs. API vulnerabilities are rooted in the fact that developers assume users only interact with the API through the user interface. They are typically configured in a way for the user to make a request and pass the permissions to the API. However, when an attacker bypasses the user authentication process, the API which has unrestricted access becomes exposed, granting the attacker access to the organization's data [4]. In the interest of securing customer privacy and preventing downtime and financial impact to businesses providing essential services, it is imperative to secure APIs by identifying anomalies in API access behavior. API call graphs are a way to monitor API connections as they log various usage statistics [5].

The goal of our project was to evaluate the performance of various machine learning algorithms in classifying API access patterns, captured as behavior metrics, as either normal (non-malicious) or outlier (potentially malicious) behavior. While outlier API access patterns may not necessarily be attacks, they significantly narrow the list of API access instances that a company must further investigate for potential attacks. We approached this by training two linear support vector classifiers (SVC) with a liblinear and stochastic gradient descent (SGD) solver respectively, two logistic regression models with a liblinear solver and SGD solver respectively, and a Ridge regression model with a stochastic average gradient descent solver.

2. Technical Approach

In our dataset, API call graphs are transformed into a set of numerical and categorical features. We encode categorical features with integer labels and center and scale numerical features. Using a common random seed across all steps, we then use stratified 5-fold cross-validation to tune hyperparameters on all models. About 34% of the records in the supervised set are classified as outlier, so with stratified 5-fold cross-validation the expected number of records in the smallest class for each validation set is still about 114. We generate candidate parameters in logspace ranging an order of magnitude in each direction from the default values for each hyperparameter. To perform cross-validation, we use a successive halving algorithm, which runs through multiple iterations and eliminates a proportion of candidate parameters while increasing the amount of data available for training and validation at each step such that the last step uses the entire training and validation set. The developers of the successive halving algorithm report that it "achieves comparable test accuracies an order of magnitude faster than the uniform strategy."[6] We visualize performance and candidate elimination at each step of successive halving.

We chose models with all linear kernels to get a thorough evaluation of performance of some of the most common linear classifiers on API activity classification, as these models are known to perform well on binary classification tasks. On all models except Ridge regression, which does not accept a penalty term, we use the L2 norm for consistency. We use hinge loss for both SVC models. Linear SVC and logistic regression are both run with a liblinear coordinate descent solver, and with a SGD solver. Ridge regression is run with a stochastic average gradient descent solver.

After optimal parameter candidates are identified, we use a separate training/validation split to compute the area under the receiver operating characteristic

(AUROC) scores for comparison. We then train a model with optimal parameters on the entire training set and make predictions on the unlabeled test set, which could hypothetically be deployed for flagging suspicious activity. We analyze the importance scores of features for each of the final models.

3. Experimental Results

3.1 Dataset

The API security: Access behavior anomaly dataset used in this project is public and was retrieved from Kaggle [7]. The API access patterns captured in this dataset come from financial institutions and E-commerce groups and therefore reflect real-world behavior. The data source is presented in the form of a supervised dataset with 11 columns and 1,674 rows, and an unlabeled dataset with 12 columns and 33,837 rows. The former was used as the training dataset and the latter as the testing dataset. The datasets shared 9 features which were used in the training of our models: time interval between API accesses in a user session, ratio of distinct APIs in a user session to the total number of API calls made in that session, total number of API calls made in a session by a user (on average), duration of a user session within an observation window, type of IP the user came from, number of user sessions each with a different session ID, number of users generating the same type of API call sequences, number of unique APIs, and source of data. The set of 9 features encompass an observed behavior, which is classified as either an outlier or normal behavior (classification feature of training dataset). Of the 1,674 supervised records, 568 records (34%) were classified as outlier, so a proportionally sampled 20% validation set would still contain about 114 records in the smallest class.

3.2 Preprocessing

Prior to training, both datasets were processed to remove duplicates and missing values (which resulted in 1,674 and 33,837 rows for training and testing datasets, respectively). In addition to the 9 features, each dataset had an ID column which was removed prior to training. The training dataset also had a classification column which was involved in upcoming preprocessing steps but saved as a separate dataset later. Feature correlation was visualized using a heatmap in the training dataset, indicating that API access uniqueness had the strongest positive correlation to the outcome (Figure 2).

The non-categorical features of the training and testing datasets were normalized using the StandardScaler function from scikit learn, which removed the mean and scaled data to unit variance. The categorical features (classification for the training dataset only, and IP type and source for both datasets) were encoded with integer labels using the LabelEncoder function from scikit learn. The training dataset was saved as a feature dataset (x) and a target dataset (y). The testing dataset was saved as a feature dataset (x). Additional training and validation (features and target) datasets were saved to be used in computing ROC scores for the models.

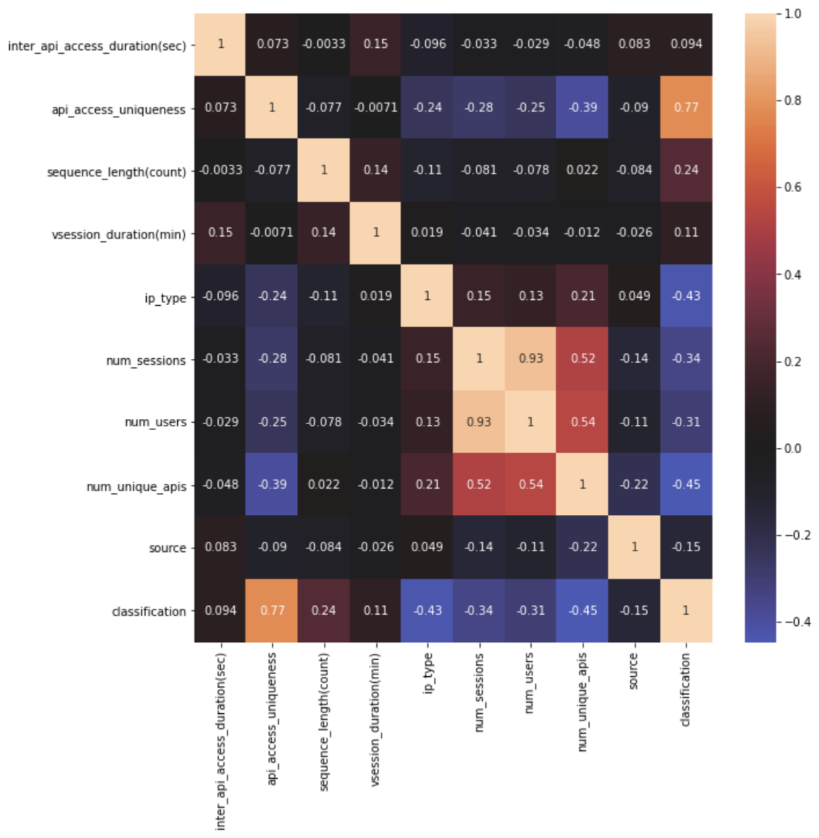


Figure 2: Heatmap of feature correlations in supervised (training) dataset where blue shows a negative correlation and orange shows a positive correlation.

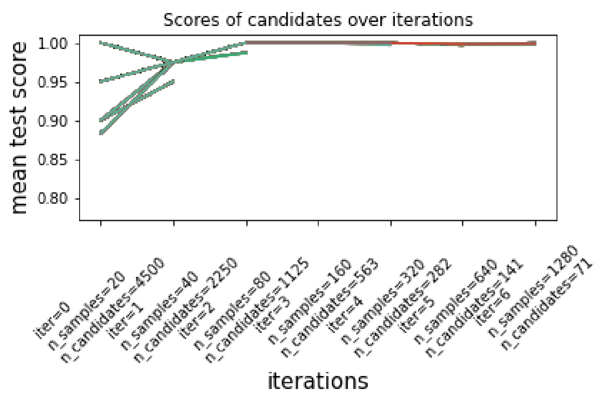
3.3 Models

3.3.1 Linear SVC

For the linear SVC model with liblinear solver, a Linear SVC model was initialized with hinge loss and L2 penalty, as well as a set random state. The hinge loss function provides maximum margin classification and is largely used for support vector machines (SVMs) [8]. The L2 penalty shrinks coefficients by the same factor without eliminating them, and is the standard used in SVC.

A stratified K-fold cross-validator with 5 splits (80% train, 20% validation) was supplied to a halving grid search cross validation algorithm with a specified factor of 2 (i.e. select half the candidates for subsequent iterations). The optimal parameters were C = 1.389, intercept scaling = 0.719, and tolerance = 0.000297. The iterations of the search are summarized in Figure 4. Using these optimal parameters, the model was fit on the separate training and testing datasets to compute the ROC score. The purpose of creating a separate set of training and testing data, equivalent to the ones used to train the model, was to have also have a validation dataset necessary to compute true and false positive values. We did not have a separate validation dataset for the model because the stratified K-fold cross-validator internally created (5) validation sets for cross validation. Both the ROC, training, and validation scores were 1.0. The model fit on the (non-ROC) training data had a training score of 1.0.

Finally, the coefficients of the features used in the model were ranked by their magnitude (Figure 5). All features had non-zero coefficients (with API access uniqueness ranking the highest), so the model did not need to be rerun with an updated feature set.



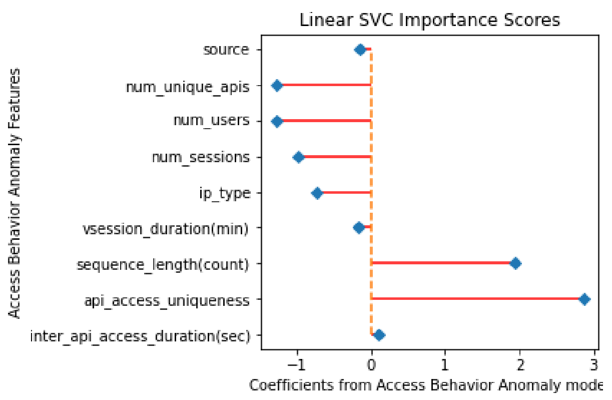
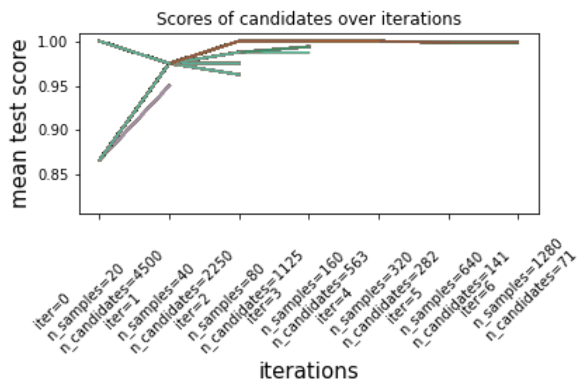


Figure 4: Linear SVC Candidate Hyperparameter Scores in Successive Halving

Figure 5: Linear SVC Feature Importance Scores

3.3.2 Logistic Regression

For the logistic regression model with liblinear solver, a logistic regression model was initialized with an L2 penalty and set random state. A stratified K-fold cross-validator (5 splits) was input into a halving grid search cross validation algorithm just as it was with the linear SVC model (Figure 6). We identified the optimal parameters as C = 7.197, intercept scaling = 1.931, and tolerance = 0.00127. The ROC score was computed on the model fit on the optimal parameters, and was found to be 1.0 (same training and validation scores). The (non-ROC) training score for the dataset used to train the model was 1.0. Just as with the linear SVC model, API access uniqueness had the highest importance score (Figure 7). Feature importance scores were very similar to those found in the linear SVC model.



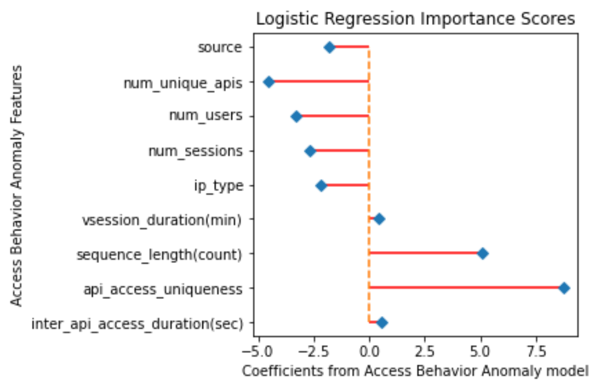


Figure 6: Logistic Regression Candidate Hyperparameter Scores in Successive Halving

Figure 7: Logistic Regression Feature Importance Scores

3.3.3 Ridge Regression Classification

The Ridge Classifier was initialized with a SAGA solver and set random state. The SAGA solver is an improved and unbiased version of the SAG (stochastic average gradient descent) solver. The same stratified K-fold cross-validator (5 splits) as in the previous two models was input into the halving grid search cross validation algorithm (Figure 8). The optimal parameters were alpha = 1.0 and tolerance = 0.000127. After fitting the model on the optimal parameters, the ROC score was calculated to be 0.939. The training and validation scores on the ROC datasets were 0.951 and 0.958, respectively. The model fit on the (non-ROC) training data had a training score of 0.952. Interestingly, this model had the only non-perfect training and validation scores. From the importance score plot, it can be seen that IP type was the most important feature, followed by API access uniqueness (unlike most other models evaluated) (Figure 9).

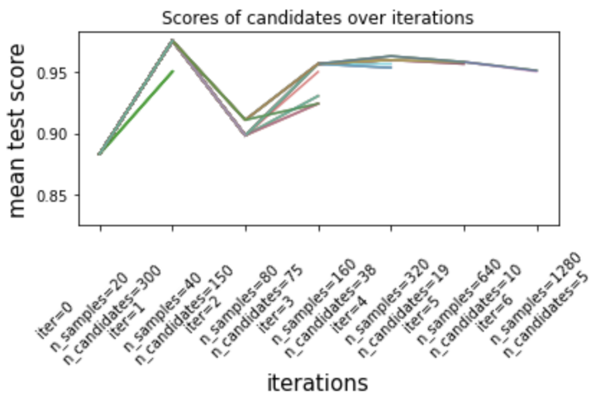


Figure 8: Ridge Regression Candidate Hyperparameter Scores in Successive Halving

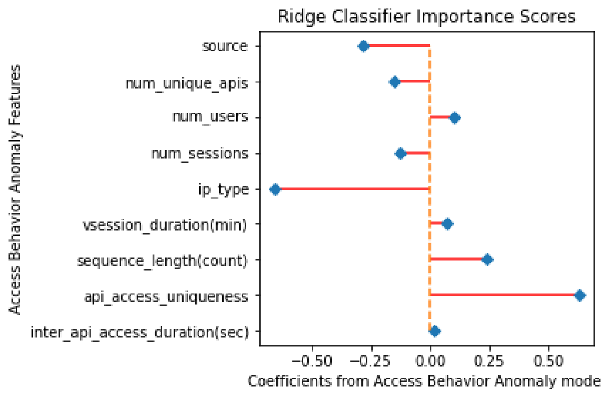


Figure 9: Ridge Regression Feature Importance Scores

3.3.4 Linear SVC SGD

This model was initialized identically to the liblinear (coordinate descent) linear SVC model, using a hinge loss function with L2 penalty, only with an SGD solver instead. Using the same cross-validation method, we identified optimal parameters as alpha = 0.0000264 and tolerance = 0.0000695. The iterations of the search are summarized in Figure 10. The ROC, training, and validation scores were 1.0. The model fit on the (non-ROC) training data had a training score of 1.0. Interestingly, feature importance scores (Figure 11) were markedly different from the pattern seen in every other model. This is the only model in which inter API access duration was the most important feature.

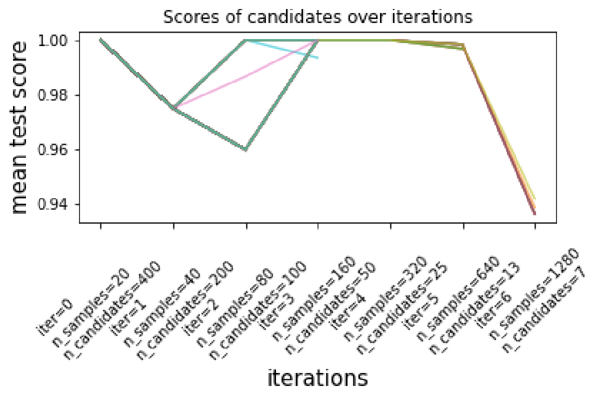


Figure 10: Linear SVC SGD Candidate Hyperparameter Scores in Successive Halving

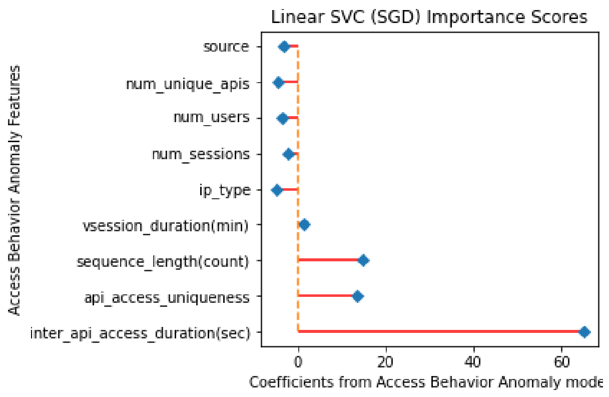


Figure 11: Linear SVC SGD Feature Importance Scores

3.3.5 Logistic Regression SGD

This model was initialized identically to the coordinate descent linear SVC model, using an L2 penalty, only with an SGD solver instead. We identified optimal parameters as alpha = 0.0000546 and tolerance = 0.0000428. The iterations of the search are summarized in Figure 12. The ROC, training, and validation scores were 1.0. The feature importance scores followed a pattern similar to that of the first two models, and the highest importance score was API access uniqueness (Figure 13).

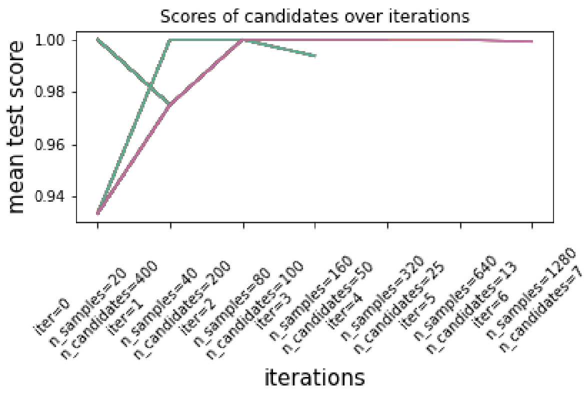


Figure 12: Logistic Regression SGD Candidate Hyperparameter Scores in Successive Halving

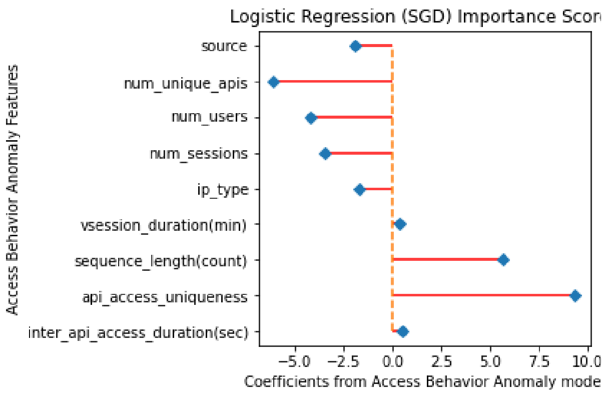


Figure 13: Logistic Regression SGD Feature Importance Scores

3.4 Conclusions

While such high accuracy can sometimes be a symptom of overfitting, we believe the high scores across all models reflect the high quality of the dataset. The simplicity of the classification problem makes it likely that models with optimized hyperparameters would achieve perfect or near-perfect scores on relatively small, well-prepared validation sets. It is also reassuring that in the Ridge regression model and when dialing in options for successive halving cross-validation on all models, any runs in which training scores differentiated from validation scores had higher scores on validation sets than training sets.

4. Contributions

Rahul Suryadevara was primarily responsible for preprocessing the datasets (supervised training and unlabeled testing), literature review, and generating ROC and importance scores. Rahul imported the datasets and prepared them for training by filtering data, encoding features, normalizing, and generating feature correlations. Remy LeWinter generated the structure for all 5 models starting with cross validation, hyperparameter tuning, and finally model fitting and prediction, along with training and validation scores. Rahul completed and ran the SGD solver models while Remy completed and ran the other liblinear and SAGA solver models, and assisted with creating separate ROC datasets. Both authors worked jointly on code review, writing the report, and analyzing the final results.

References

1. Gnatyk, R. (2018, October 3). *Microservices vs Monolith: Which Architecture Is the Best Choice for Your Business?* N-IX. https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
2. Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016, December). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 318-325). IEEE.
3. SentinelOne. (2018, September 25). *API vs. Microservices: A Microservice is More Than Just an API*. SentinelOne. https://www.sentinelone.com/blog/api-vs-microservices/
4. Akamai Technologies. (2021, October 27). *Akamai Finds API Vulnerabilities to be a High-Stakes Game for Companies and Individuals Worldwide*. Akamai Technologies. https://www.akamai.com/newsroom/press-release/akamai-finds-api-vulnerabilities-to-be-a-high-stakes-game-for-companies-and-individuals-worldwide
5. Parvez, F., Vijay, L., Manoj, G., & Vinod, P. (2012, October). Mining control flow graph as API call-grams to detect portable executable malware. 130-137. 10.1145/2388576.2388594.
6. Jamieson, K. & Talwalkar, A. (2016). Non-stochastic Best Arm Identification and Hyperparameter Optimization. *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, in *Proceedings of Machine Learning Research* 51:240-248. https://proceedings.mlr.press/v51/jamieson16.html.
7. https://www.kaggle.com/tangodelta/api-access-behaviour-anomaly-dataset
8. Rosasco, L., De Vito, E., Caponnetto, A., Piana, M., & Verri, A. (2004). Are loss functions all the same?. *Neural computation*, *16*(5), 1063–1076. https://doi.org/10.1162/089976604773135104