# Malicious URL detection using Streaming Techniques.

**Chouliaras Andreas** ACHOULIARAS@INF.UTH.GR
**Gkountouvas Stylianos** SGKOUNTOUVAS@INF.UTH.GR
**Pappas Apostolos** APOPAPPAS@INF.UTH.GR

## Abstract

*Malicious URL Detection* was the main topic of our project for the class ECE417 "Machine Learning for Data Science and Analytics" back in the Spring of 2019. As a continuation of that, we decided that a real time approach using streaming techniques would be more suitable for the project of this course. We have stated before that malicious URLs leading to malicious websites, are a common and serious threat to cybersecurity. Instead of using Blacklists(which tend to lack the scalability and the overall ability to to detect newly generated malicious URLs), we use Machine Learning algorithms as our detection approach. In a real-world problem, the predictions and further training of the model should be done in real time. Thus, we approach this problem in a real time manner using tools such as Kafka and Zookeeper.

## 1. Introduction

The importance and the effect that the World Wide Web has, is continuously increasing. Unfortunately, the technological advancements come coupled with new sophisticated techniques to attack and scam users. Scam attacks include a certain type of websites that perform fraud by tricking users into revealing sensitive information which eventually lead to theft of money, identity, or even installing malware in the user's system. There are various ways of implementing an attack. Some of them are:

- SQL injections
- Explicit hacking attempts
- Phishing and Social Engineering
- Drive-by-Download
- Denial of Service(DoS)
- Distributed Denial of Service(DDoS)

It is common that the mentioned attacks are realized through spreading compromised URLs. These compromised URLs are called *Malicious*. Research has shown that almost one third of all websites(thus, their URLs as well) are malicious in nature. Unsuspecting users visit such web sites and become victims of various types of scams, including monetary loss, theft of private information, and malware installation (Hong, 2012). The most traditional way of dealing with such threats is using blacklists.

Blacklists are essentially databases that contain several URLs marked as malicious. However, new URLs are created everyday, making these databases somewhat obsolete. In particular, there are numerous cases where the attackers create and modify URLs that seem legitimate, fooling the unsuspecting user into the attack. The main problem with blacklists is their incapability in making predictions for new URLs other than the ones already in their database (Sinha et al., 2008).

In order to overcome this barrier, we can use and apply several Machine Learning techniques that fit on the training data, in this case several URLs, and make predictions about newly seen URLs (Sahoo et al., 2017).

Having the training dataset, we first need to preprocess it in order for our algorithm to be able to build the classifier model. There is a wide variety of Machine Learning algorithms we can choose from, like Logistic Regression, K-Nearest-Neighbors, Neural Networks, Support Vector Machines, Decision Trees and others. Yet not all of these support real-time training without the use of the full dataset. After selecting the appropriate algorithm, the next step is to train our model. This will fit our model on the training data and will be the foundation of each prediction in the future.

When predicting and classifying a URL, there are two types of misclassification that can be made: False Positives and False Negatives. A False Positive (or FP) is just a "false alarm". In our case, the model classifies a URL as malicious, while in reality it is not. A False Negative (or FN) in

University of Thessaly, ECE514 Course Project.

our problem, is the misclassification of a malicious URL as non-malicious. One can understand the negative impact of the second kind of misclassifications. Therefore our aim is to keep FNs as low as possible.

To take our previous work one step further, we will be using the Apache's Kafka API to achieve real time prediction and training using streaming data. This way we can update the prediction model with new data, keeping it up-to-date for greater reliability. We will simulate a client-server scenario where the client asks the server if the URL that is about to be used is malicious. The server then, after acquiring a desirable amount of new URLs, retrains the model including the new data and updates the prediction model in real-time.

### 1.1. Dataset and Preprocessing

The dataset used for our experimentations, can be found at www.kaggle.com. It contains a total of 420,464 URLs, 75,643 of which are malicious and 344,821 are safe. Each URL contains only the host name and the path, excluding the HTTP Protocol at the beginning, if it had any.

*Table 1.* Quantitative description of the dataset.

| DATASET | URLS |
|---|---|
| TOTAL URLS | 420,464 |
| MALICIOUS URLS | 75,643 |
| SAFE URLS | 344,821 |
| % OF MALICIOUS | 18 |
| % OF SAFE | 82 |

As it is, the dataset cannot be directly used by Machine Learning algorithms. Thus, there is the need to transform it into vectors and matrices, easily manipulated by such algorithms. For this purpose we test two different algorithms called *tf-idf* and *FastText* (Bojanowski et al., 2016) respectively.

#### 1.1.1. TF-IDF ALGORITHM

*Tf-idf*, short for "term frequency-inverse document frequency", is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. Term frequency (tf) is essentially the output of the Bag-of-Words model. For a specific document, it determines how important a word is by looking at how frequently it appears in it. For a word to be considered a signature word of a document, it shouldn't appear that often in the other documents. Thus, a signature word's document frequency must be low, meaning its inverse document frequency must be high. The idf is usually calculated as:

$$idf(W) = log \frac{\#(documents)}{\#(documents\ containing\ word\ W)}$$

The *tf-idf* is the product of these two frequencies. For a word to have high *tf-idf* in a document, it must appear a lot of times in said document and must be absent in the other documents. It must be a signature word of the document.

In Python, the *tf-idf* algorithm is implemented using the *TfidfVectorizer* class. After the tokenization we have a vocabulary of about 420,000 tokens.

#### 1.1.2. FASTTEXT ALGORITHM

FastText is an open source library created by Facebook's AI Research (FAIR) lab that allows users to learn text representations and text classifiers(Bojanowski et al., 2016). FastText uses a shallow neural network and supports supervised (classifications) and unsupervised (embedding) representations of words and sentences.

Before understanding how FastText models work, an understanding of how the Word2Vec models work is expected. Word2Vec is a shallow 2-layer neural network that takes as an input a large collection of words and produces a vector space, with each word being assigned a vector in this space. These word vectors are placed in the vector space such as word with similar meaning or context are placed close together. Word2Vec can utilize two model architectures to produce the embeddings. The first is called "continuous bag-of-words" (CBOW) and the second "continuous skip-gram".

The Word2Vec architecture is similar to that of an autoencoder. We take a big input vector and we compress it down to a smaller one, but instead of decompressing it back to the original input vector we can output probabilities of target words.

Word2Vec is able to capture multiple degrees of similarities between words. Furthermore semantic and syntactic patterns can be reproduced using vector arithmetic. For example operations on the vector representations of the words "Brother", "Man", "Woman" such as the following "Brother"-"Man" + "Woman" produces a result which is closest to the vector representation of "Sister" in the model.

The FastText models works simmilar to Word2Vec models, but with a big difference. The main principle behind FastText is that the morphological structure of a word carries important information about the meaning of the word, which is not taken into account by traditional word embeddings, which train a unique word embedding for every individual word.

The main difference is that FastText treats each word as the

aggregation of its subwords. Each word is represented as a bag of character n-grams in addition to the word itself. The final vector embedding is taken to be the sum of all vectors of its component n-ngrams. Furthermore FastText can obtain embeddings or out-of-vocabulary (OOV) words, words that did not exist when creating the model, by summing up vectors for its component n-grams.

We use this algorithm only in the scenario C, that it is introduced later, because it can handle out-of-vocabulary words better than the TF-IDF algorithm and can transform vectors faster, which helps us in the retraining process.

### 1.2. The Models Evaluated

Most of algorithms we used in our previous work were not designed to handle streams of data. More specifically: All the algorithms we used require the whole dataset in training and they can't support training with streaming data. The Naive Bayes Models we used, require the whole dataset for making predictions as well. The K-Nearest-Neighbors algorithm also requires the whole dataset for making predictions and furthermore it is very slow for big datasets. The sklearn package we previously used for the Multilayer Perceptron Classifier does not support data streams.

So we used the LightGBM, which is a gradient boosting framework that uses tree based learning algorithms. It utilizes ensemble techniques but with the capability to work with real time data streams. The algorithms we used are:

- GBDT (Gradient Boosting Decision Tree)
- DART (Dropouts meet Multiple Additive Regression Trees)
- GOSS (Gradient-based One-Side Sampling)

## 2. The Evaluation Metrics

Before presenting the results of our experimentation, we should first take a look at the metrics on the basis of which we compare the different Machine Learning algorithms we mentioned above.

Firstly, we use Accuracy, Precision and Recall to evaluate our models. Accuracy is, in its core, the ratio of number of correct predictions to the total number of input samples.

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ input\ samples}$$

Precision refers to the proportion of positive identifications that was actually correct and is defined as the number of true positives divided by the number of true positives plus the number of false positives.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

On the other hand, Recall refers to the proportion of actual positives that was identified correctly and is defined as the number of true positives divided by the number of true positives plus the number of false negatives.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

Although Accuracy seems like a good and comprehensible way to evaluate our model, it may very well lead us to misinterpreting the results of our algorithm. We should have in mind that we examine a problem where False Positives and False Negatives have a big impact. Thus, we use the $F_1$ score as another way to evaluate our models. $F_1$ score is the harmonic mean of Precision and Recall and it is suited to problems where the cost of False Positives and False Negatives are very different. Therefore, this score takes both false positives and false negatives into account.

$$F_1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Moreover, we use the Area Under Curve, also known as AUC, a metric which is connected to the Receiver Operating Characteristic Curve (ROC curve). AUC - ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents degree or measure of separability.

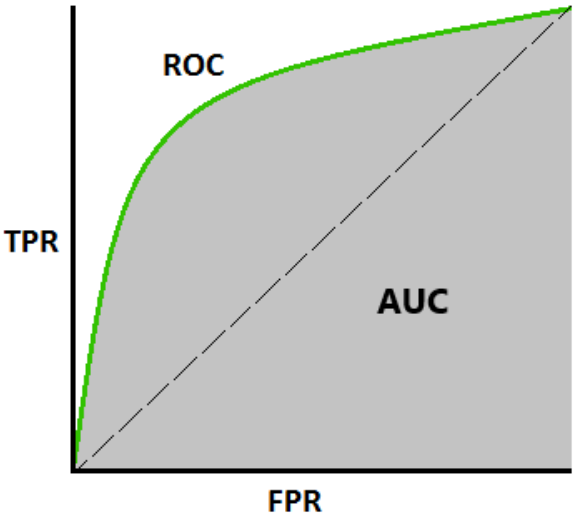



*Figure 1.* ROC curves and AUC.

The ROC curve is plotted with True Positive Rate against the False Positive Rate as can be seen at *Figure 1*.

So we know that the higher the AUC metric, the more successful a model is at a classification problem.

Last but not least we take into consideration the time needed to train a model and make predictions with it. It is very important to produce results as fast as possible in addition to good classification results, because training a slow model might make the predictions obsolete.

## 3. Apache Kafka

Apache Kafka is an open-source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation, written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka can connect to external systems (for data import/export) via Kafka Connect and provides Kafka Streams, a Java stream processing library. Kafka uses a binary TCP-based protocol that is optimized for efficiency and relies on a "message set" abstraction that naturally groups messages together to reduce the overhead of the network roundtrip.

### 3.1. Concepts and Architecture

As mentioned in Kafka's white paper (Kreps et al., 2011), a stream of messages of a particular type is defined by a *topic*, and a producer can publish messages to a topic. The published messages are then stored at a set of servers called *brokers*. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers. To subscribe to a topic, a consumer first creates one or more message streams for the topic. Each message stream provides an iterator interface over the continual stream of messages being produced. The consumer then iterates over every message in the stream and processes the payload of the message. It is important to have in mind that the message stream iterator never terminates. If there are currently no more messages to consume, the iterator blocks until new messages are published to the topic. Typically, a Kafka cluster consists of multiple brokers. Multiple producers and consumers can publish and retrieve messages at the same time.

### 3.2. The architecture of our experiment

In our case, we have a much easier schema to understand. We run our experiments in one computer using only one Kafka cluster having one broker. The broker has two available topics named *"app message"* and *"retrain topic"* respectively.There are three components in our experiment, each having a consumer and a producer interface. These are:

- The Service
- The Predictor
- The Trainer

We simulate a client-server scenario with The Service program working as a client and The Predictor, The Trainer along with the Kafka cluster all together as a server. In the following subsections, we will explain their functionality.

#### 3.2.1. The Service

This program works by creating two threads, the send and the receive threads. The send thread has the URLs that need to be tested. It also uses the producer interface when communicating with the the *app message* topic of the broker. The receive thread waits for the prediction from the server and it communicates with the same topic of the broker using the consumer interface.

Once again, it is important to remember that the stream will stay open, even if it is blocked.

#### 3.2.2. The Predictor

The predictor subscribes to both topics and interacts with each of them in a different way. After receiving the data from the *app message* topic, it is responsible of classifying it. Having the result of the classification, using its producer interface sends a message containing the prediction to the mentioned topic.

Every $N$ predictions, the Predictor decides that a retraining of the model should take place. Using producer interface again, it sends a message to the *retrain* topic containing the appropriate flag. After the retraining takes place, the Predictor will learn it through its consumer interface after seeing the proper flag from the broker. After that the producer loads the updated model and continues for another $N$ predictions.

#### 3.2.3. The Trainer

The sole purpose of its existence is to retrain the model when necessary. Thus, the Trainer only communicates through a stream with the *retrain* topic. When there is a message containing the retrain flag, it will inject the current batch of data to the original set by accessing the message as saved in the disk and converting it to a dataframe. Then it retrains the model for a small number of epochs and saves it in a file that the predictor uses to load the updated version of the model. Finally using the producer interface it sends a message to the topic indicating the training is complete.

## 4. The scenarios

Having described and implemented the above mentioned model, we decided to try out three different scenarios regarding real time prediction and training. One thing they

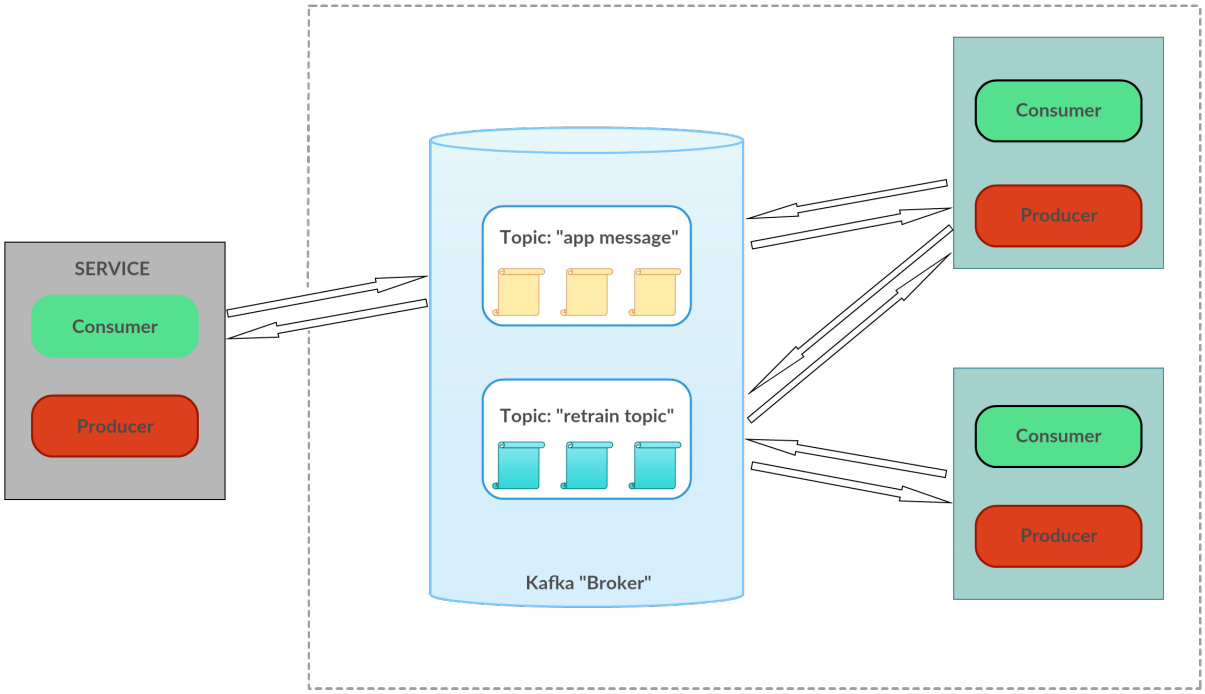



*Figure 2.* Our server-client model using Apache Kafka.

have in common is the initial training of the algorithm that takes place offline.

### 4.1. Scenario A - Real time prediction only

In this scenario, after the initialization, the service sends messages containing entries to be classified to our Kafka server. There, the predictor will perform the classification task and post them to the *"app message"* topic.

Essentially, we do not retrain our model. We use the firstly initialised one to classify incoming data from our stream. We expect this scenario to resemble the offline model presented in ECE417's report in terms of accuracy, recall and precision.

### 4.2. Scenario B - Retraining the model using new data injected to the full dataset

Here, the service sends messages containing entries to be classified to our Kafka server. There, the predictor will perform the classification task and post them to the *"app message"* topic. As mentioned earlier, every $N$ predictions, the predictor will send a retrain message to the *"retrain topic"*.

Now, the trainer has to retrain the model. It will fit the model to the full dataset plus the $N$ entries that were predicted earlier. Have in mind that this is a supervised classification problem. In the real world, given that there is the appropriate feedback, a company could blacklist certain newly-found URLs and update its dataset.

### 4.3. Scenario C - Retraining the model using new data only

The above model might be easy for one to conceive, yet is not straightly speaking realistic. We want our streaming machine learning algorithms to be able to adapt to newly seen entries without the use of the full dataset for training purposes.

The three algorithms contained in the LightGBM have the capability that we need in order to complete the training procedure in real-time. Note that we should be very careful since there is always the possibility of our model to overfit. That is a problem we actually faced. With the appropriate number of iterations and hyperparameter settings, we can achieve high accuracy without overfitting.

## 5. The Models

Ensemble based algorithms have been shown to achieve high accuracy for a number of machine learning tasks (Caruana and Niculescu-Mizil, 2006). In the following subsections, we briefly describe the functionality of each ensemble algorithm we use for this project. All these algorithms are included in the *LightGBM* framework. We test

each of them on each scenario and produce an "Accuracy over time" plot to evaluate their behavior.

### 5.1. The Gradient Boosting Decision Tree(GBDT)

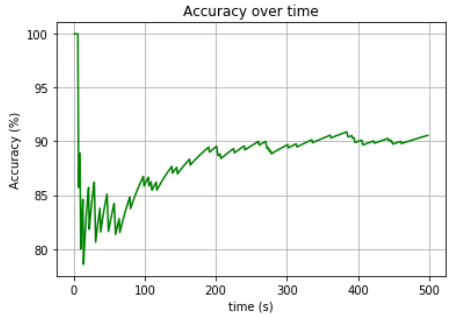
To fully understand what GBDT is, we should remind ourselves of terms like Gradient Descent and Boosting. The first is an optimization technique aiming to reduce the loss function by following the slope through a fixed step size.

Boosting is sequential ensembling technique where hard to classify instances are weighted more. This essentially means that subsequent learners will put more emphasis on learning miss-classified data instances. The final model will be a weighted average of n weak learners.

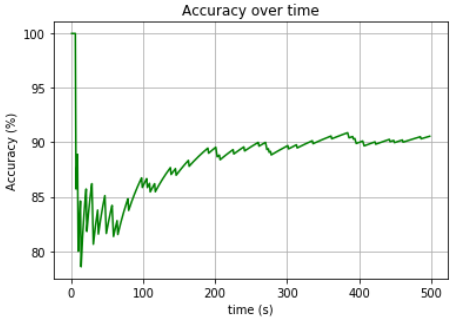
Thus, GBDT is an ensemble model of decision trees, which are trained in sequence. In each iteration, GBDT learns the decision trees by fitting the negative gradients (also known as residual errors). The main cost in GBDT lies in learning the decision trees, and the most time-consuming part in learning a decision tree is to find the best split points (Ke et al., 2017).

Split finding algorithms are used to find candidate splits. One of the most popular split finding algorithms is the *Pre-sorted* algorithm which enumerates all possible split points on pre-sorted values. This method is simple but highly inefficient in terms of computation power and memory. The second method is the Histogram based algorithm which buckets continuous features into discrete bins to construct feature histograms during training.

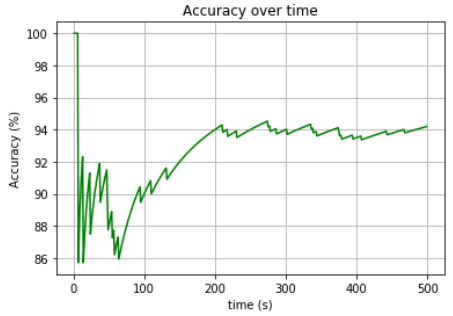
GBDT was the first of the algorithms that we examined. Testing the algorithms on big test sets proved very time-consuming. Thus, we chose to test the algorithm on a test set composed of 500, unknown to the algorithm, entries. Regarding the Accuracy value, we notice a fairly steep rise over time. (Fig. 3)




(a) Scenario A.




(b) Scenario B.




(c) Scenario C.

*Figure 3.* GBDT

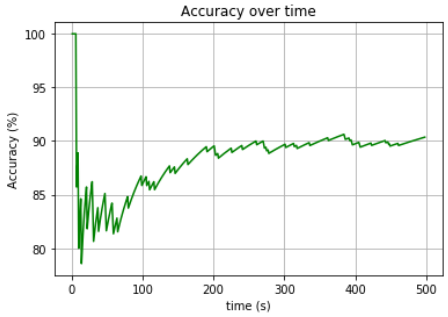
### 5.2. Gradient-based One-Side Sampling(GOSS)

GOSS (Gradient Based One Side Sampling) is a novel sampling method which down samples the instances on basis of gradients. As we know instances with small gradients are well trained (small training error) and those with large gradients are under trained. A naive approach to downsample is to discard instances with small gradients by solely focussing on instances with large gradients but this would alter the data distribution. In a nutshell GOSS retains instances with large gradients while performing random sampling on instances with small gradients

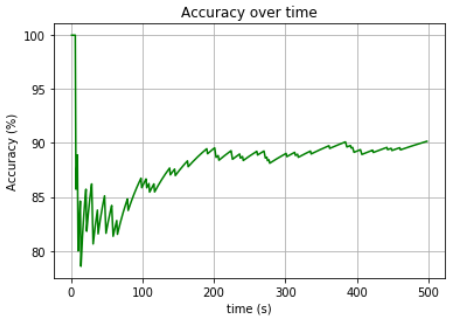
We run all three scenarios using the GOSS as our classifier having, once again, a test set of 500 entries. We can see that we have a less steep rise of the accuracy value over time than the one we had by using the GBDT algorithm (Fig. 4). Note that, once it stabilizes, the accuracy of GOSS at Scenario C is the highest we achieve in this project. The corresponding Recall value is the highest as well.

### 5.3. Dropouts meet Multiple Additive Regression Trees (DART)

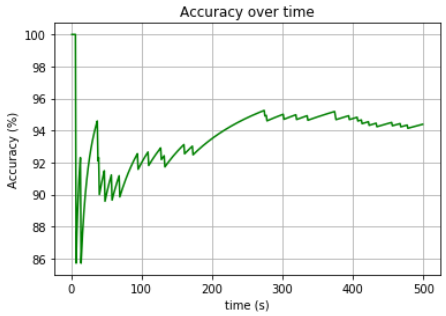
DART algorithm is built on the foundation set by MART. Multiple Additive Regression Trees (MART), an ensemble model of boosted regression trees, is known to deliver high prediction accuracy for diverse tasks, and it is widely used in practice. However, it suffers an issue which is called over-specialization, wherein trees added at later iterations tend to impact the prediction of only a few instances, and make negligible contribution towards the remaining instances. This negatively affects the performance of the model on unseen data, and also makes the model over-




(a) Scenario A.

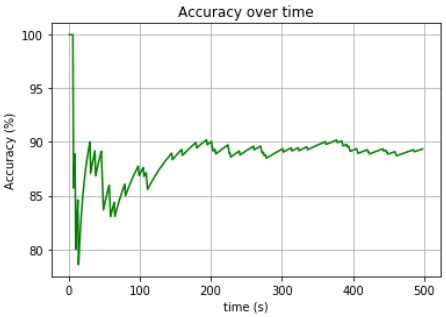



(b) Scenario B.

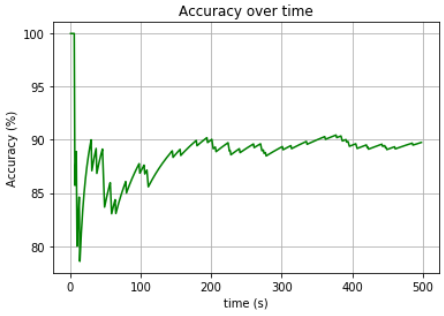



(c) Scenario C.

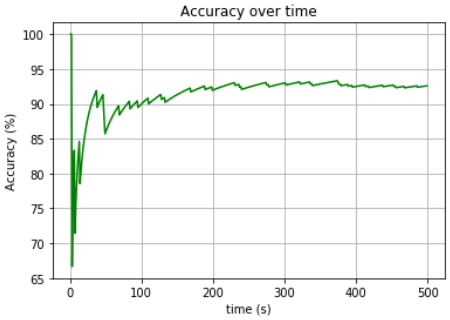
*Figure 4.* GOSS




(a) Scenario A.




(b) Scenario B.




(c) Scenario C.

*Figure 5.* DART

sensitive to the contributions of the few, initially added trees (Rashmi and Gilad-Bachrach, 2015).

The difference that DART has in comparison to MART is the dropout that it introduced. As its name mentions, DART algorithm drops trees in order to solve the over-fitting problem that the previous algorithm had. Because of that random dropout that takes place, we expect that training might be slower than GBDT because the random dropout prevents usage of the prediction buffer.

Finally, running the experiment using the DART algorithm, we notice that its Accuracy over time stabilizes rather quickly (Fig. 5). We may comment that it is the less effective algorithm of all tested ones.

## Concluding Remarks and Future Directions

Malicious URLs are a serious threat to cybersecurity. The detection of such URLs plays a critical role for many cybersecurity applications, and clearly machine learning approaches are a promising direction. In this project we tested three ensemble machine learning algorithms, capable of dealing with real time training scenarios.

The more realistic approach was Scenario C, where the online training was done using only the newly imported data. We found GOSS (Gradient-based One-Side Sampling) to be the best overall algorithm to use in this scenario since it achieves the highest values of both Accuracy and Recall (*Table 2*).

A lot more work could be done, testing even more algorithms and trying several hyperparameter optimization

*Table 2.* Summary of results.

| ALGORITHM | SCENARIO | ACCURACY % | PRECISION % | RECALL % | F1 SCORE % |
|---|---|---|---|---|---|
| GBDT | A | 90.6 | 95.7 | 92.6 | 94.1 |
| | B | 91 | 95.7 | 93.1 | 94.4 |
| | C | 94.2 | 97.7 | 95.1 | 96.4 |
| GOSS | A | 90.4 | 95.7 | 92.3 | 94 |
| | B | 90.2 | 95.6 | 92.1 | 93.8 |
| | C | 94.4 | 97.7 | 95.3 | 96.5 |
| DART | A | 89.3 | 94.4 | 92.3 | 93.4 |
| | B | 89.7 | 91.9 | 92.8 | 92.4 |
| | C | 92.6 | 97.2 | 93.6 | 95.3 |

techniques. Regarding the server-client model that we used, we could try it out on different computers rather than running it locally.

## Acknowledgements

We would like to give our sincere appreciation to the ECE 514: Problem Solving Environments and Data Science class and to our professor Elias Houstis for his guidance throughout the entire semester. We were able to overcome ourselves and push though what we thought was our limits.

## References

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

J. Hong. The state of phishing attacks. *Commun. ACM*, 55: 74–81, 01 2012. doi: 10.1145/2063176.2063197.

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

K. V. Rashmi and R. Gilad-Bachrach. Dart: Dropouts meet multiple additive regression trees. In *AISTATS*, pages 489–497, 2015.

D. Sahoo, C. Liu, and S. C. Hoi. Malicious url detection using machine learning: a survey. *arXiv preprint arXiv:1701.07179*, 2017.

S. Sinha, M. Bailey, and F. Jahanian. Shades of grey: On the effectiveness of reputation-based "blacklists". In *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pages 57–64, Oct 2008. doi: 10.1109/MALWARE.2008.4690858.

The dataset can be found at:
https://www.kaggle.com/antonyj453/urldataset