Toolkit for machine learning experiments with decision forests on network data

Lukáš Sahula

2018-01-12

Abstract

The aim of the project is to implement a toolkit for malware classification that will be used in the followup Bachelor's thesis for machine learning experiments upon datasets with missing values. Given the size and form of the data taken from network proxy logs, the toolkit has to be able to properly load potentially large datasets, train a classifier and evaluate its results. This thesis describes the form and relevance of the network datasets, gives an introduction to malware classification and specifies the measures that are evaluated by the toolkit. A large portion of the thesis focuses on the toolkit implementation and the problems surfacing during the process.

Záměrem tohoto projektu je implementovat knihovnu pro klasifikaci malwaru, která bude využita v následující bakalářské práci pro experimenty na datasetech s chybějícími hodnotami ve strojovém učení. Vzhledem k velikosti a formě dat získaných ze síťových proxy logů musí knihovna umět správně nahrávat potenciálně velká data, natrénovat klasifikátor a vyhodnotit jeho výsledky. Práce popisuje formu a relevanci těchto síťových dat, podává stručný úvod do klasifikace malwaru a specifikuje vyhodnocované veličiny. Velká část práce se zabývá samotnou implementací jednotlivých modulů knihovny a problémy, které se v průběhu objevily.

Table of contents

Introduction

This thesis focuses on the implementation of a machine learning toolkit for malware classification. The purpose of this toolkit is to be able to load variously sized datasets previously extracted from network proxy logs, preprocess these datasets, use them to fit a classifier and evaluate the classifier's performance. Because the toolkit has three arguably different jobs to do - loading, training a classifier, classification and evaluation of the datasets - it is separated into three independent modules. Each module is designed to be working as a standalone piece of program and to leave open the possibility of applying it to a problem other than malware classification with minor changes.

The thesis consists of several sections. At the beginning, there is a brief introduction to machine learning and malware classification, followed by a short explanation of the network datasets. The next chapter gives a summary of the measures evaluated in the classification results. The chapter after that delves into the implementation process and breaks down the three modules of the toolkit one by one. The last section is dedicated to the random forest classifier algorithm that will have to be implemented from scratch in order to provide more possibilities for future experimentation.

1 Malware classification

Classification is the process of assigning a label (or a class) from a set of categories to a new object based on its *features*. An example could be a classification algorithm that predicts whether a patient's tumor is benign or malign based on its size and the age of the patient. In the context of this project, an object is a single record from a dataset of network proxy logs carrying information about a specific network request passing through the proxy. The algorithms doing the classification are called *classifiers*.

Classification is a form of machine learning that usually follows the *supervised learning* paradigm. This means that the classifier learns from a previously classified set of records called the *training dataset*. The classifier learns from this training dataset and is then able to predict the class of a new previously unseen object with some level of accuracy. To test the performance of the classifier, usually a different set of labeled records called the *testing dataset* is used. It is important to separate the testing data from the training data to see if the classifier can perform adequately on previously unseen observations.

Since there are multiple classes of malware in the mentioned network datasets, the classification process is called *multiclass classification*. The classifier does not simply state if the observation is malware or non-malware, it also states which class of malware the observation belongs to. From the set of classifier algorithms, the *random forest classifier* is well-suited for this malware classification problem. It handles multiclass classification, evaluates the data relatively fast, is not heavily affected by *imbalanced datasets*[1] (more on that later), runs efficiently on large amounts of data, and gives estimates of what feature variables are important in the classification[2].

2 Network data

The datasets extracted from network proxy logs used in this project are in the form of zipped csv files, where each file contains proxy logs of a single user. They are divided into *negative* samples and *positive* samples. Negative samples form the majority of the data. Those are the ones that were previously labeled as benign (non-malware). Positive samples were previously classified as some specific class of malware. When there are significant differences in counts of the classes of the dataset, which is the case here, it is called an *imbalanced dataset*.

The samples are further diversified by the day they were taken. This provides the opportunity to observe the differences in performance of a classifier trained on data from one day and tested on data from a week later, compared to the performance of the same classifier tested on data from a month or a year later.

The *feature vector* of these samples contains 55 features but a significant amount of the records is missing some of its feature values. To deal with this missing values problem is the goal of the following bachelor's thesis. These *missing values* are simply replaced by a constant outside of the interval of that feature's values for this project's purposes, since it does not affect the random forest classifier strongly. Other popular and easy to implement choices are to replace it with the mean of that feature value. Apart from the label and the feature vector, each record contains a few metadata entries, like the timestamp of the record or the user initiating the request, that are utilised further in the evaluation process.

3 Evaluation measures

There is a plenty of measures that can be used to determine the classifier's performance. The most commonly observed is perhaps the *classification error*[1]. The problem with classification error is that it does not say much when the dataset is imbalanced, as is the situation with the network data. For example if we had a dataset where there are 10,000 samples and only 1 of them belonged to a positive class, the classifier could classify all samples as negative and it would achieve a classification error of 0.01 %. In this project we are interested in two different measures, *precision* and *recall*. In order to compute them, we also need the *confusion matrix*.

3.1 Confusion matrix

In binary classification, we can label the two classes as positive and negative. *Confusion matrix* divides the classification results into the following categories:

TP: True positives. The number of positive objects the classifier labeled as positive.

FP: False positives. The number of negative objects the classifier labeled as positive.

TN: True negatives. The number of negative objects the classifier labeled as negative.

FN: False negatives. The number of positive objects the classifier labeled as negative.

These categories are then used to compute other, more interesting measures.

In multiclass classification, the *confusion matrix* gets more complicated. Consider a dataset with 3 different classes A, B and C. If the classifier takes a sample that belongs to class A and labels it as B, then it counts as FN for class A, but as FP for class B.

This gets even more confusing with the network datasets. The samples are either negative and thus belonging to one specific class, or positive, which means one of multiple positive classes. For that reason, samples belonging to the negative classes that are classified as negative do not count as TP. Similarly, when a positive sample is labeled as positive but a different class,

it should not be counted as FN, rather as TPish. As for TNs, it does not make much sense to distinguish them from TPs, since they basically are TPs for the negative class. This is a way of reducing the multiclass problem to a binary problem. In the context of this project, a *one-vs-all* confusion matrix, which evaluates one class at a time, is computed. Furthermore, TNs do not get counted and the TPs of the negative class are optionally filtered out in the process.

3.2 Precision

Precision, or *positive predictive value* is defined as:

$$Precision = \frac{TP}{TP + FP}$$

Precision is the fraction of positive objects among all objects that the classifier labeled as positive. It can also be interpreted as the probability that a positively labeled object is truly positive. A precision score of 1.0 means that every object labeled as positive by the classifier is truly positive. It does not say, however, anything about the classifier's ability to recognize all truly positive instances. In multiclass environment, precision of every class varies because it is computed separately.

3.3 Recall

Recall, or *sensitivity* is defined as:

$$Recall = \frac{TP}{TP + FN}$$

Recall is the fraction of positive objects that the classifier labeled as positive among all truly positive objects. It can be interpreted as the probability that a truly positive object is labeled as positive. A recall score of 1.0 means that every truly positive object is labeled as positive by the classifier. This could be achieved simply by labeling all objects as positive, recall does not say anything about the number of false positives. Recall of every class varies among all classes in multiclass environment.

4 Toolkit implementation

This chapter takes a detailed look at the toolkit's implementation process and the three modules it consists of. The programming language chosen for this task is Python[3] because of its advantages and convenience when it comes to fast prototyping and also because of the number of existing libraries for machine learning and data analysis. To name a few of those used in this project, there is *pandas, numpy and scikit-learn*[4, 5, 6].

Other notable technology utilised is *git* hosted at *Bitbucket* because of its option to create private repositories and its free plan allowing the use of continuous integration[7].

The whole implementation runs in accord with the *test-driven development* process, meaning that there is a test written before any new functionality gets implemented.

For each of the toolkit modules, there is a list of all public functions and an explanation of their purpose and the problems that arose during the implementation.

4.1 Loading tool

This module contains mainly functions to load training and testing data. Apart from that, there are also some data preprocessing functions like data quantization, that did not fit into any of the other modules. A function to load a csv with classifications for the purpose of evaluation is also in this module.

4.1.1 Load training data

In order to train a classifier on the network datasets, the first issue is to load the csv files containing the training data into a data structure. That is a straightforward task, but working with relatively large datasets comes with a problem regarding memory. It is practically impossible to load the data from the whole day into memory even on high-end machines. As was mentioned earlier, the network datasets are imbalanced, the amount of negative samples is multiple times larger than the amount of positives. This fact provides an option to take a random sample of the negatives and train the classifier with this sample. Since each csv file is specific for a different user, it is not

a safe practice to take a random sample of the files. To avoid this potential information loss, the better way is to go through each file, take a sample from it and concatenate it to the data structure.

4.1.2 Load testing data

The testing dataset, however, should not be sampled at all. Luckily, the classifier does not require all the testing data at once. It is possible to send it to the classifier part by part, which can be done in Python simply by turning the function into a *generator function*[3] and instead of returning the data it *yields* it. That way it is possible to iterate through this function in a for loop and call the prediction function of the classifier on each iteration.

4.1.3 Quantize data

The feature values of every feature vector in the network datasets span across large amount of distinct values and to increase the classifier's performance, it is helpful to *quantize* the values into a set number of bins (intervals like in histograms). This is done by computing a set number of quantiles and then replacing all the values with the quantile value. After some testing, the classifier had the best results with 16 bins. The bins get computed only once from the training dataset and then they are applied to the testing dataset as well.

4.1.4 Load classifications

After the classifier is done with the classification, the classification tool saves all its predictions into a csv file with columns containing the true labels, the predicted labels and, optionally, the metadata values mentioned in the previous chapter. This produces a large csv file that has to be processed by the evaluation tool part by part. Like the LoadTestingData function, this function is also a generator function that reads the classifications in parts of a specific number of lines.

4.2 Classification tool

This module works as a superstructure to the classifier that is passed as a parameter to the constructor. It encapsulates the classifiers functions and simplifies the classification process into two functions, one to fit the classifier and the other to use it to predict the classes of the testing data and save the results into a file.

4.2.1 Train classifier

This function passes the training data into the classifier's fit function. It relies on the classifier implementing this function, but sice all relevant classifiers do (namely the classifiers in the *scikit*) library, this function should work no matter what classifier is passed to the classification tool upon construction.

4.2.2 Save predictions

This function reads the testing data chunk by chunk and calls the classifiers predict function upon every chunk. After predicting each chunk, it then saves the predicted class, the true class and the metadata into a new csv file.

4.3 Evaluation tool

The purpose of this module is to create the confusion matrix and compute precision and recall for every class from the classifier's output.

4.3.1 Compute stats

This function computes the confusion matrix and saves the counts of TPs, FPs and FNs for each class in a dictionary.

4.3.2 Compute aggregated stats

This function essentialy does the same as the function above, with one difference. It takes an extra parameter specifying one column of the metadata by which the stats should be aggregated. This could be for instance the column identifying a user. If a user is associated with a true positive record of some class already, it does not make sense for him to have a false positive or false negative record of that same class. To add to that, the aggregation removes duplicities, as in if a user has a true positive record of some class, it remains there only once.

4.3.3 Compute precision

Computes precision for the given class from the confusion matrix according to the math formula. If there are no TPs and no FPs, this function return NaN.

4.3.4 Compute recall

Computes recall for the given class from the confusion matrix according to the math formula. If there are no TPs and no FNs, this function return NaN.

4.3.5 Get average precision

Iterates over all known classes and computes the average precision. Optionally, it is possible to choose whether or not to include the negative class in the computation.

4.3.6 Get average recall

Iterates over all known classes and computes the average recall. Optionally, it is possible to choose whether or not to include the negative class in the computation.

4.3.7 Get stats count

For the given list of classes and the aggregated/unaggregated stats, this function counts the amount of TPs, FPs and FNs.

5 Runner

The runner is the part of the program, where all the modules come together and form a pipeline that loads the data, trains the classifier and evaluates its result. In the process of evaluation, the runner writes into an output file the average precision and recall as well as the individual statistics for each class. It also gives information about the number of classes with precision or recall above some specific levels. After this pipeline finishes, the runner encapsulates the trained classifier along with some additional data, such as the bins for quantization, and serializes it into a file for future use. By passing an argument into the runner's execution function it is possible to specify whether a new classifier should be trained, or an already trained one should be used.

6 Future work

This section briefly explains what work needs to be done in the immediate future before the work on the bachelor's thesis can begin.

6.1 Random forest classifier

As mentioned before in the malware classification chapter, of the algorithms from the classification school, *Random Forest Classifier* is most suited for classification of large datasets with missing values[2]. To test the toolkit in this project an implementation from the *scikit-learn framework*[6] was used. But in order to conduct larger experiments, it is imperative that the inner structures of the algorithm can be manipulated. Thus an implementation of this algorithm from scratch is needed.

Conclusion

The purpose of this project was to implement a set of tools for malware classification. This toolkit was supposed to properly load data extracted from network proxy logs, preprocess them for better performance, fit them to a classifier and evaluate the classifier's output. Based on this requirements, the toolkit was divided into three separate modules - loading tool, classification tool and evaluation tool. All three modules work independently and can be later used to other problems as well.

The biggest problem to overcome during the implementation process was the size of the network datasets. The datasets used for testing purposes were much smaller and when it came to actually use the original data, many of the functions stopped working due to running into memory errors. These functions had to be rewritten into generator functions in order to process the data without storing them in memory all at once.

Reference

[1] Jan Brabec, *Decision Forests in the Task of Semi-Supervised Learning*, Czech Technical University in Prague. Computing and Information Centre, 2017-01-29

[2] Leo Breiman, *Manual–Setting Up, Using And Understanding Random Forests*, University of California, Berkeley. Department of Statistics, Version 4.0, Available at www.stat.berkeley.edu/~breiman/ as of 2018-01-12.

[3] Python Software Foundation, *Python Language Reference*, Version 3.6, Available at www.python.org as of 2018-01-12.

[4] Wes McKinney, *pandas: a python data analysis library*, 2008, Available at www.pandas.pydata.org as of 2018-01-12.

[5] Travis Oliphant, *NumPy*, 2005, Available at www.numpy.org as of 2018-01-12.

[6] Pedregosa, F. and Varoquaux, G. and Gramfor, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, Volume 12, 2011, Available at www.scikit-learn.org as of 2018-01-12.

[7] Atlassian Corporation Plc, *Bitbucket*, Available at www.bitbucket.org as of 2018-01-12.