Partitioning Based N-Gram Feature Selection for Malware Classification

Weiwei Hu and Ying Tan(✉)

Key Laboratory of Machine Perception (MOE),
Department of Machine Intelligence,
School of Electronics Engineering and Computer Science,
Peking University, Beijing 100871, China
{weiwei.hu,ytan}@pku.edu.cn

Abstract. Byte level N-Gram is one of the most used feature extraction algorithms for malware classification because of its good performance and robustness. However, the N-Gram feature selection for a large dataset consumes huge time and space resources due to the large amount of different N-Grams. This paper proposes a partitioning based algorithm for large scale feature selection which efficiently resolves the original problem into in-memory solutions without heavy IO load. The partitioning process adopts an efficient implementation to convert the original interactional dataset to unrelated data partitions. Such data independence enables the effectiveness of the in-memory solutions and the parallelism on different partitions. The proposed algorithm was implemented on Apache Spark, and experimental results show that it is able to select features in a very short period of time which is nearly three times faster than the comparison MapReduce approach.

Keywords: Malware classification · Feature selection · Data partitioning · Apache Spark

1 Introduction

With the rapid popularization of the Internet, malware has become the main threat to computer security. Hundreds of millions of new malware samples are created every year [2]. How to detect and classify such a large amount of new malware has become a challenging issue in computer security.

Many researchers have proposed to use machine learning to detect and classify new malware in recent years. The malware detection task trains a classifier such as support vector machine [12] and random forest [6] to classify executable files into benign or malware, while the malware classification task uses a classifier to classify executable files into families. It can be seen that malware detection is just a special case of malware classification. Therefore, we just use the term malware classification to refer to both tasks hereinafter.

The machine learning based malware classification approaches proposed so far mainly differ on the way to extract malware features. Schultz et al. [9] proposed to use DLLs, APIs, strings and byte sequences as malware features. Kolter

Y. Tan and Y. Shi (Eds.): DMBD 2016, LNCS 9714, pp. 187–195, 2016.
DOI: 10.1007/978-3-319-40973-3_18

et al. [4,5] selected a certain number of byte level N-Grams using information gain [17] to construct malware features. Tabish et al. [13] divided binary files into blocks and calculated 13 statistical features on each block. Each block was classified as a normal block or a potentially malicious block, and the correlation module was used to combine the classification results of different blocks from a single binary file. Shafiq et al. [10,11] regarded the key fields of the PE structure [7] as features.

Among the proposed feature extraction algorithms, N-Gram shows good experimental performance and is more robust than other algorithms. Therefore, there are many derived feature extraction algorithms from N-Gram. Wang et al. proposed the immune concentration of N-Grams to construct a low dimensional feature [15,16]. Zhang et al. used the class-wise information gain to select malicious N-Grams to detect infected executables [18].

The value N for N-Gram is usually set to 4, which is able to keep a balance between final classification performance and the robustness. A smaller N will decrease the classification performance heavily while a larger N can be easily obfuscated by the insertion of irrelevant machine instructions.

However, the feature selection of N-Gram is very expensive in real world applications. Most feature selection algorithms need to count the numbers of N-Grams in all classes, while there are a huge number of different N-Grams when the dataset is large. The main memory of an ordinary computer usually does not have enough space to store the variable (e.g. a hash map which stores the counts of all N-Grams for each class) used to collect the counts. Kolter et al. [4] adopted a disk-based implementation to handle this problem but they said that their implementation was very slow.

This paper proposes a partitioning based N-Gram feature selection algorithm which divided the huge number of N-Grams into several partitions and uses an in-memory implementation to select the most informative features.

We will introduce the N-Gram feature selection algorithm in Sect. 2 and explain the proposed partitioning based algorithm in Sect. 3. Sections 4 and 5 will give the experimental results and the conclusion respectively.

2 N-Gram Feature Selection for Malware Classification

In the feature selection process of malware classification, each executable sample is broken into N-Grams using an overlapping sliding window of N bytes. The window slides from the beginning of a binary file to the end with a step of one byte. At each step the binary content in the window is regarded as an N-Gram feature. Duplicated N-Grams in a sample will be removed.

After generating N-Grams for all of the samples in the training set a feature selection metric score is calculated for each unique N-Gram and the top K N-Grams with the largest metric scores will be selected as the final features, where K is a pre-specified variable.

Information gain is the most used feature selection metric, which is defined as Formula 1 [5].

$$IG(f) = \sum_{v_f \in \{0,1\}} \sum_{c \in C} P(v_f, c) \frac{P(v_f, c)}{P(v_f) P(c)} . \tag{1}$$

In this formula f represents the feature and v_f represents the feature value. $v_f = 0$ means the feature f is not in a sample, while $v_f = 1$ means the feature f appears in a file sample. C is the set of all classes. $P(v_f, c)$ is the probability that the class of a sample is c and its feature value for f is v_f. $P(v_f)$ represents the probability that a sample's feature value for f is v_f and $P(c)$ represents the probability that a sample's class is c.

Formula 1 needs to calculate three probabilities (i.e. $P(v_f, c)$, $P(v_f)$ and $P(c)$) in advance. $P(c)$ can be easily obtained by calculating the fraction of samples with class c in the training set. If we have calculated $P(v_f, c)$, $P(v_f)$ can be derived as the sum of $P(v_f, c)$ over all classes, as shown in Formula 2.

$$P(v_f) = \sum_{c \in C} P(v_f, c). \tag{2}$$

The remaining work is to calculate $P(v_f, c)$. We can obviously get Formula 3.

$$P(v_f = 0, c) + P(v_f = 1, c) = P(v_f, c) . \tag{3}$$

Therefore we only need to calculate $P(v_f = 1, c)$ first and then derive $P(v_f = 0, c)$ using Formula 4.

$$P(v_f = 0, c) = P(v_f, c) - P(v_f = 1, c) . \tag{4}$$

$P(v_f = 1, c)$ can be calculated using Formula 5.

$$P(v_f = 1, c) = \frac{T(v_f = 1, c)}{T}. \tag{5}$$

T in Formula 5 is the total number of samples in the training set and $T(v_f = 1, c)$ is the number of samples in the training set whose feature values for f are all 1 and classes are all c.

When applied to N-Gram feature selection for malware classification, $T(v_f = 1, c)$ can be written as $T(g, c)$, which represents the number of an N-Gram g in class c, provided that the duplicated N-Grams in a single executable file are removed.

However, the number of possible N-Grams is exponential to the length of N-Gram N. For example when $N = 4$ there will be 2^{32} (i.e. about 4.3 billion) possible N-Grams. For a large training set a large fraction of the possible N-Grams will appear. If there are 100 classes and we use a 32-bit integer to store the number of an N-Gram, we will need at most 1.6 TB space to store the numbers, even not including the additional space used by the data structure (e.g. a hash table) to store and organize the N-Grams, which could be in the same order of magnitude as the space used by the numbers.

When traversing the training set to calculate $T(g, c)$, the whole variable that stores $T(g, c)$ for all of the N-Grams and all of the classes should be kept in main

memory because the variable will be accessed frequently and randomly, while an ordinary computer with a limited memory size is not able to store such a large variable in main memory.

To handle this problem, Kolter et al. [4] claimed that they implemented a disk-based approach which consumed a great deal of time and space, but they didn't give the detailed description of their implementation.

A straightforward solution is to split the training set into small subsets and calculate the numbers of N-Grams for each subset. The result for each subset is written to a file on the disk. After all the subsets are traversed the files for different subsets are merged into a single one. This solution is workable but it needs a lot of IO operations which are very inefficient.

Another possible solution is to use MapReduce [3,8] which is designed for distributed systems. Most MapReduce tutorials begin with a word count example, while calculating the number of N-Grams is just a special case of the word count problem; we need to take count of N-Grams for all of the classes. However, the shuffle operation of MapReduce is very time-consuming which will heavily decrease the time efficiency of feature selection.

In the next section we will propose an efficient N-Gram feature selection algorithm based on partitioning.

3 Partitioning Based N-Gram Feature Selection

The process of partitioning based N-Gram feature selection is shown in Algorithm 1.

Algorithm 1. Partitioning Based N-Gram Feature Selection

Input: the training set.
Input: K: the number of N-Gram features to be selected.
Output: K N-Gram features with the largest information gains.

1: Determine the number of partitions P according to the training data size and the main memory size available in a computer.
2: Create P lists which are stored in the external memory, namely $L_0, L_1, ..., L_{P-1}$.
3: **for all** executable sample in the training set **do**
4: $c = the\ class\ of\ the\ sample$
5: **for all** unique N-Gram g generated from the sample **do**
6: $L_{hash(g)\%P}.add(< g, c >)$.
7: **end for**
8: **end for**
9: the feature set $F = \varnothing$
10: **for all** $i \in \{0, 1, ..., P-1\}$ **do**
11: Use an in-memory feature selection algorithm to select top K features from L_i.
12: Add the selected N-Gram features to F.
13: **end for**
14: **return** the top K features in the set F.

First of all, the number of partitions should be determined so that each partition can be processed using an in-memory algorithm. Let the total size of the training data be S and the main memory size available be M. The average data size of each partition is S/P.

The actual memory occupied by the in-memory algorithm used below is approximatively proportional to the size of data in a partition. Let the proportionality coefficient be λ. Then we have $\lambda * S/P \leq M$, that is $P \geq \lambda * S/M$.

The coefficient λ depends on the programming language and the data structure used in the program. We need to estimate its value according to the detailed implementation. If λ is underestimated, there may be not enough main memory space for the program and the program may crash. Therefore, we should select a slightly large value for λ.

Then, P lists are created to store the temporary data. The temporary data is also too large to be stored in main memory. Therefore, the data are actually stored in external memory such as the disk. A cached implementation of the list in external memory will significantly reduce the amount of IO operations. When adding data to the list, the new data are stored into a buffer in main memory first. Once the buffer is full all the data in it will be written to external memory and the buffer is set to empty again.

Next, each executable file in the training set is broken into N-Grams and duplicated N-Grams in a file are removed. For each N-Gram a hash function is calculated based on its binary content. The remainder of Euclidean division of the hash value and P determines which list the N-Gram goes to. The pair of the N-Gram and the class of the executable file is added to the list.

For the cached implementation of the list in external memory, a smaller P will consume less cache space for all the lists in main memory. From the above analysis we know that the condition for selecting P is $P \geq \lambda * S/M$. Therefore, we can just choose $P = \lambda * S/M$.

This partitioning process will make the same N-Gram across the whole training set to be partitioned into the same list. Therefore, the information gain of the N-Gram can be calculated within the list.

After that, for each list an in-memory feature selection algorithm is used to select top N-Grams. The numbers of all N-Grams in the list is counted for each class and the information gains are calculated using Formulas 1-5. The K N-Grams with largest information gain is selected. The P lists produce $P * K$ N-Grams in total.

Finally, the top K N-Grams are selected from the $P * K$ N-Grams.

The time complexity of Algorithm 1 is proportional to the size of training data. The partitioning process reads the whole training set linearly and writes the pairs of N-Grams and the classes to external memory. We estimate the total size of data in external memory here without considering the data compression and indexing. For $N = 4$, after removing duplicated N-Grams the number of N-Grams will reduced to about one half according to our experiment. Therefore, the external memory space occupied by the contents of N-Grams is about 2 (i.e. $4 * 0.5$) times larger than the training data size. If the number of classes is not

larger than 128 we can use one byte to store the class value for each N-Gram, and the external memory space occupied by the classes of N-Grams will be about the same as the training data size. If the number of classes is between 129 and 65536 the class value will need two bytes and the external space will be about 2 times larger than the training data size. Therefore, the total size of data in external memory is about 3 or 4 times larger than the training data size, depending on the number of classes. The in-memory feature selection algorithm will read the data from external memory again to calculate information gain. In conclusion the data size of IO operations is about 7 or 9 times larger than the training data size. Such amount of IO operations is acceptable in the real world application.

Algorithm 1 can be easily parallelized in a very efficient approach. The most time-consuming processes of Algorithm 1 are partitioning N-Grams from all files into different lists and calculating information gain for each list. The partitioning process for each file is independent, and the calculations of information gain for different lists are also independent. Therefore, these two processes can both be distributed to different CPU cores or even different computers.

4 Experiments

The proposed algorithm was implemented on Apache Spark [1] using Python. Apache Spark has a built-in data partitioning routine which makes the proposed algorithm very easy to be implemented. What's more, Apache Spark can distribute the computation tasks to different CPU cores and different computers, enabling the parallelization of the proposed algorithm automatically.

We used two workers for the Apache Spark cluster, each with one CPU core and 5 GB memory.

The dataset we used contains 11058 executables, including 5563 benign files and 5495 malicious files from the VX Heaven virus collection [14]. The total size of the dataset is 5.50 GB. We did a two-class malware detection task on this dataset.

When selecting the number of partitions using the formula $P = \lambda * S/M$, λ was estimated as about 100. At last we chose 100 as the number of partitions.

We also implemented a comparison algorithm based on the famous word count example using MapReduce given in the Apache Spark website[1], which is shown as follows:

```
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
```

The function of this piece code is to calculate the numbers of all words from a given text file. Each line of the file is mapped to several words and the same word will be reduced to get the count. We extend this algorithm to calculate the number of N-Grams for all of the classes and then calculate the information gains.

[1] http://spark.apache.org/examples.html.

Shafiq et al. [11] used the N-Gram feature as a comparison algorithm of their proposed PE-Miner. In their article they claimed they had developed an optimized implementation of N-Gram which is more efficient. We also took their implementation as one of our comparison algorithms.

The time consumption of different implementations is shown in Table 1.

Table 1. Feature selection time for partitioning based algorithm, MapReduce based extended word count algorithm and the implementation of Shafiq et al.

Algorithm	#Samples	Hardware and platform	Time
Partitioning	11058	3.0 GHz+3.3 GHz, Spark, Python	3.4 h
MapReduce	11058	3.0 GHz+3.3 GHz, Spark, Python	10.1 h
Shafiq et al.	2895	2.19 GHz, C++, STL	25.3 h

As we can see in the table, partitioning based algorithm is almost three times faster than MapReduce based algorithm. MapReduce needs to shuffle the huge amount of N-Grams, while shuffle is very expensive.

Both partitioning based algorithm and MapReduce based algorithm have two stages in the Spark jobs. The time consumption of each task for the two algorithms is shown in Fig. 1.

For partitioning based algorithm, stage 1 partitions the N-Grams, while stage 2 counts the N-Grams and calculates information gain to select top N-Grams. The two stages takes 1.6 h and 1.8 h respectively. The most expensive operation of the two stages is IO, and the IO data sizes of the two stages are close. Therefore, the time consumptions of this two stages are close. The calculation of information involve some expensive logarithm operations in stage 2, so that the time consumption of stage 2 is slightly higher than that of stage 1.

For MapReduce based extended word count algorithm, stage 1 uses MapReduce to obtain the numbers of N-Grams in each class while stage 2 maps the counts of an N-Gram in all the classes to information gain. Stage 1 takes 6.8 h which is twice as the whole time consumption of partitioning based algorithm. The most expensive operation of stage 1 is shuffle. We can see that shuffle will

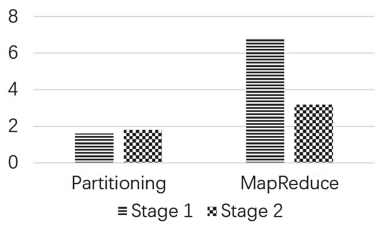

Fig. 1. The time (in hour) of different stages for partitioning based algorithm and MapReduce based extended word count algorithm

significantly decline the time efficiency of the whole algorithm. Stage 2 of this algorithm takes 3.2 h while stage 2 of partitioning based algorithm only takes 1.8 h. The superiority of partitioning based algorithm is achieved by using the concentrative in-memory feature selection for the whole partition.

The last row of Table 1 gives the estimated result of Shafiq et al.'s implementation. They used many categories of executables in their experiments. The average number of executables in each category is 2895. They reported that the average feature selection time for a file of their implementation is 31.5 s. Therefore, we estimated that the total feature selection time for 2895 files is 25.3 h.

The size of dataset used by partitioning based algorithm is almost 4 times larger than that used by Shafiq et al. We used two CPU cores and the CPUs' frequencies are higher than theirs. Besides, they adopted a faster C++ implementation. We can estimate that our computation capability is about 4 times stronger than theirs. Considering the dataset size and the computation capability, the numeric difference between the time consumption of Shafiq et al.'s implementation and the partitioning based algorithm in Table 1 is able to roughly reflect the difference between the time efficiencies of the two algorithms. Shafiq et al.'s implementation took more than one day to select features, while partitioning based algorithm only took 3.4 h, which is more than 7 times faster.

5 Conclusions

Byte level N-Gram is a well-known feature extraction algorithm for malware classification which is able to extract relevant features for malware and is very robust. However, the feature selection of N-Gram suffers from heavy time and space load because the large number of features generally cannot be stored in main memory. This paper proposed a partitioning based approach to accelerate the feature selection process. The proposed approach divides N-Gram features into independent partitions and uses the in-memory feature selection algorithm for each partition. Experimental results showed that the partitioning based algorithm is very efficient and is superior to a MapReduce based implementation and the optimized implementation by Shafiq et al.

Acknowledgments. This work was supported by the Natural Science Foundation of China (NSFC) under grant no. 61375119 and the Beijing Natural Science Foundation under grant no. 4162029, and partially supported by National Key Basic Research Development Plan (973 Plan) Project of China under grant no. 2015CB352302.

References

1. Apache: Apache spark (2016). http://spark.apache.org/
2. AV-TEST: Malware statistics & trends report (2016). http://www.av-test.org/en/statistics/malware/
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

4. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
5. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 470–478. ACM (2004)
6. Liaw, A., Wiener, M.: Classification and regression by randomforest. R News **2**(3), 18–22 (2002)
7. Pietrek, M.: Peering inside the pe: A tour of the win32 portable executable file format (1994). https://msdn.microsoft.com/en-us/library/ms809762.aspx
8. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, vol. 1. Cambridge University Press, Cambridge (2012)
9. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: 2001 IEEE Symposium on Security and Privacy. Proceedings, S&P 2001, pp. 38–49. IEEE (2001)
10. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: A framework for efficient mining of structural information to detect zero-day malicious portable executables. Technical report. Citeseer (2009)
11. Shafiq, M.Z., Tabish, S.M., Mirza, F., Farooq, M.: PE-Miner: mining structural information to detect malicious executables in realtime. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 121–141. Springer, Heidelberg (2009)
12. Suykens, J.A., Vandewalle, J.: Least squares support vector machine classifiers. Neural Process. Lett. **9**(3), 293–300 (1999)
13. Tabish, S.M., Shafiq, M.Z., Farooq, M.: Malware detection using statistical analysis of byte-level file content. In: Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, pp. 23–31. ACM (2009)
14. VX-Heaven: Virus collection (vx heaven) (2016). https://vxheaven.org/vl.php
15. Wang, W., Zhang, P., Tan, Y., He, X.: Animmune local concentration based virus detection approach. J. Zhejiang Univ. Sci. C **12**(6), 443–454 (2011)
16. Wang, W., Zhang, P., Tan, Y.: An immune concentration based virus detection approach using particle swarm optimization. In: Tan, Y., Shi, Y., Tan, K.C. (eds.) ICSI 2010, Part I. LNCS, vol. 6145, pp. 347–354. Springer, Heidelberg (2010)
17. Yang, Y., Pedersen, J.O.: A comparative study on feature selection in text categorization. In: ICML, vol. 97, pp. 412–420 (1997)
18. Zhang, P., Tan, Y.: Class-wise information gain. In: 2013 International Conference on Information Science and Technology (ICIST), pp. 972–978. IEEE (2013)