Information Retrieval for Malware Analysis

Charles Nicholas
CSEE Department
UMBC
`nicholas@umbc.edu`

DRAFT version of March 28, 2016

Abstract

Several concepts from information Retrieval (IR) have proven to be useful in the context of malware analysis. Malware specimens can be and are studied "in the small", i.e. at the individual level. Malware specimens can also be studied "in the large", where analysis is done on many specimens, possibly millions of them or more. From the IR perspective, though, such a collection of malware specimens is a corpus of documents, where concepts like term frequency and latent semantics can be applied. [1]

1 Introduction

Malware Analysis is emerging as a field of academic study, with journals, conferences, and textbooks such as [16].

We distinguish between malware analysis in the small, i.e. detailed analysis of an individual malware specimen, and malware analysis in the large. Governments, anti-virus vendors, and cyber-security consultants, certainly among others, maintain large collections of malware specimens. The number of specimens in such collections numbers in the hundred of millions, according to some reports[6].

[1]This paper is a draft Statement of Work prepared for the US Government. It may be shared within government for official purposes, but otherwise we request that distribution be limited except by the lead author's consent.

We assume that the reader has some familiarity with malware and malware analysis, and of course everybody knows how to use search engines on the web for search. For a comprehensive introduction to information retrieval as a subject within computer science, the book Modern Information Retrieval [3] is a great place to start.

To cite a blog post or other URL, such as this post from Cryptography Engineering. [2]

2 Malware Specimens as Documents

Rather than trying to give a formal definition of ”document”, we accept that books, articles, and web pages serve as examples of forms of communications, to which some idea of duration may apply. So an email message is a document, but an ordinary UDP packet is not. Documents can be regarded as messages that have some basis in human language, even if the document is not human-readable. So computer programs written in C are documents, as are the corresponding executable binaries. A set of related documents is referred to as a *corpus*, and the plural form is *corpora*.

This broad understanding of ”document” raises the question of how documents in general, and malware specimens in particular, are to be represented. Many answers to this question have been proposed, but it seems fair to say that the ”bag of words” model has been the most widely used. In the bag of words model, a document is represented as a frequency distribution of words and their frequencies. We can immediately use the more general concept of ”term” rather than ”word”, and then a document d can be represented as a vector of the form $[(t_1, tf_1), (t_2, tf_2), ..., (t_n, tf_m)]$ where each of the m distinct terms t_i is associated with its frequency tf_i.

2.1 Term-Document Matrix

Representing a document as a list of ordered pairs, as shown above, is cumbersome and inefficient. If the set of terms in a corpus is known in advance, and this is a big if sometimes, then we can represent a document as a vector $[tf_1, tf_2, ..., tf_n]$. A corpus of n documents can then be represented as an mxn matrix T where each row in T corresponds to one of the m distinct terms t_i,

[2] `http://blog.cryptographyengineering.com/2016/03/attack-of-week-drown.html`

each column in T corresponds to one of the n documents d_j in the corpus, and each entry $T_{i,j}$ is the number of times term t_i occurs in document d_j. Such a matrix is known as a *term-document matrix*, or TDM.

$$\begin{bmatrix} t_{1,1} & \dots & t_{1,n} \\ \vdots & t_{i,j} & \vdots \\ t_{m,1} & \dots & t_{m,n} \end{bmatrix}$$

Some observations may be made regarding terms and the Bag of Words model of document representation:

- Terms may be ordinary English words, and this would be appropriate when working with a corpus of documents written in English.
- If the documents are programs written in C, then the possible terms may include the keywords and identifiers and other lexical elements of C.
- If the documents are executable binaries, as is often but not always the case with malware, then one could attempt to parse the document to identify machine instructions, for example, and use those as terms. Another option is to use n-grams. An n-gramis a sequence of n characters, where n is fixed at a small integer such as 4 or 5. If the document(s) in question are binary, as in executable files or images, then n-gramsoffer some important advantages[7]. But the larger the value of n, the more n-gramsthere may be.
- Not all terms are of equal importance in understanding a document's content or importance. Terms may be omitted if they are so rare as to be meaningless, or so common as to offer no help in distinguishing one document from another.
- The Bag of Words model for document representation does not preserve word order. In the definition of term-document matrix (or TDM) given above, we used raw term frequencies. It is more common, in practice, to use some expression derived from raw frequency as the value of a given entry $T_{i,j}$. There are a number of these term-weighting schemes, and the advantages and disadvantages of these schemes have been studied deeply over the years[3].

The number of terms in a corpus tends to increase as the number of documents in the corpus increases. But the size of a vocabulary grows in a rate roughly proportional to the square root of the size of the corpus. This follows from Heaps' Law, which says that the number of distinct terms in a document of length n is Kn^{β}, where, at least in English, one expects K to be between 10 and 100, and β to be between 0.4 and 0.6. (See [3], or [3]) For example, suppose we have a corpus of length one million total terms, and of those 50,000 are distinct. That suggests values of $V = 50$ and $\beta = 0.5$. Were that corpus to be 100 times larger, i.e. 100 million terms, Heaps' Law predicts that the number of distinct terms would increase by only a factor of 10.

We now introduce an example malware corpus. The VX Heaven corpus includes many thousands of malware specimens, grouped into folders by malware type. There are, for example, banking Trojans, rootkits, keyloggers, and many other varieties. The data takes up about 48 gigabytes, compressed. For more on VX Heaven, see `www.vxheaven.org`. [4]

Research Questions: Does Heaps' Law hold for collections of executable binaries? What values of K and β would fit the VX Heaven corpus? Do the same values hold for other malware corpora?

Since the documents in this corpus are executable binaries, using n-gramsas terms is convenient. Again, since these are binaries, the bytes can and do take on all the values between 0 and 255. Using $n = 4$ we have 2^32 or roughly four billion possible n-grams. This would be the number of rows in the TDM. The number of malware specimens in the VX Heaven corpus is a little above 270,000. So the number of entries in the TDM is on the order of $2^{32} \times 2^{18}$ or approximately 2^{50}. This is a big matrix, but fortunately most of the entries are zero: only a small fraction of the possible 4-grams are likely to appear in any one of the documents. Furthermore, some n-gramscould be removed from the term set, on the basis of appearing in a few, or even just one, of the documents. The ratio of the number of non-zero entries in a matrix to the total number of entries is referred to as the matrix's *sparsity.* For the VX Heaven corpus, and using $n = 4$, a sparsity factor of 1% or less would be expected. Many programming languages have facilities for dealing with sparse matrices efficiently, and Python is among those languages.

[3]`https://en.wikipedia.org/wiki/Heaps\%27_law`

[4]The author is making a copy of VX Heaven available. These are live malware specimens, to be used at your own risk. Cat the files, and pipe them through bunzip. `https://www.dropbox.com/sh/3bn3bevl8zolmxj/AACyRQ83vfbgq7dHphiy9k2ra?dl=0`

Programming Exercise: Create a Python script to build a frequency distribution of n-grams. For a given value of n, have the script write the terms and their frequencies to, say, a CSV file. This script could be tested on a variety of input data files, including its own source code. A programmer working with the VX Heaven corpus might find it helpful to have subsets of the corpus, say a random selection of 10% of the documents and 1% of the documents, for testing of this and other scripts.

2.2 Operations on the TDM

Term-weighting can be used to remove terms that don't matter, and keep some terms from being too important. The tf.idf term weighting scheme is well-known and suitable for many if not most applications. Keeping track of the number of documents in which a given term occurs is not difficult.

The vector product between columns of the TDM is a way to compare them - the famous cosine similarity metric. The cosine similarity formula divides each of the input vectors by its own length, to avoid any document skewing the calculation simply because it's longer. Columns in the TDM can be normalized for length when the TDM is built, simplifying the cosine calculation.

It may be worth the trouble to build a similarity matrix S, in which entry $S_{i,j}$ is the similarity between documents i and j. In the case of VX Heaven, for example, the resulting matrix is of size 270,000 by 270,000, and this matrix is *dense* as opposed to sparse. The similarity matrix helps answer questions like which documents are most similar to each other?

The TDM can be subjected to matrix decomposition algorithms. SVD has been worked with a lot. The patterns of terms that occur together, or not, can give insight. And one can draw pictures. The non-negative matrix factorization, or NMF, is faster (at least on average) than SVD. [10]

The TDM itself is expensive to build, but it can be added to without too much trouble. But if tf.idf is being used, or any term weighting scheme that relies on document frequency, adding a document may require substantial recalculation.

Being a sparse matrix, the TDM can be stored using a hash table. Work has been done on speeding up such hash functions, since they dominate the cost of storing and accessing entries in the TDM.

2.3 Entropy

The entropy of a document, or any string of characters, is a measure of the randomness found in that string. Using 8-bit ASCII as an example, there are 256 possible characters, and a perfectly random string of arbitrary length would have each character appearing about 1/256 of the time. Strings in human (and computer) languages, though, are anything but random, and their entropy value would be less as a result. A commonly used formulation, known as Shannon's Entropy, is: [5]

$$H(x) = -\sum_{1}^{n} p(x_i) log_b p(x_i) \tag{1}$$

where each of the observed character probabilities $p(x_i)$ is the number of times character x_i occurs in the string divided by the total number of characters in the string. It is convenient to use logs of base 2 if, as in the case of 8-bit ASCII, the character set has 256 characters. The length of the string does not matter. For example, take a string that consists of twenty characters, half of which are 'x' and the other half 'y'. Then we have:

$$\begin{aligned} H(X) &= -[(0.5log_2 0.5) + (0.5log_2 0.5)] \\ H(X) &= -[(-0.5) + (-0.5)] \\ H(X) &= -[-1] = 1 \end{aligned}$$

Where most of the possible characters x_i did not occur at all, giving an observed probability of zero and having no effect on the sum in Equation 1. A similar calculation shows that a string of length 20, with four distinct characters each occurring exactly five times, has an entropy of 2. Indeed, given a sufficiently long string, and a perfectly random sequence of characters, the maximum Shannon Entropy value is 8.

It may come as a surprise to some, but executable binaries have entropy values that aren't too much higher than those common in ordinary text documents. Lyda found that files containing ordinary source code, executable binaries, and packed binaries could be identified as such by entropy calculations alone [12]. Sorokin extended this work to polymorphic malware [17].

Research Question: Has Sorokin's work been extended? Are Android binaries distinguishable from x86 binaries using Lyda's approach?

[5] `http://www.shannonentropy.netmark.pl`

2.4 Compression

The reader is likely aware that files can be compressed in order to save disk space. (Data structures in memory can also be compressed, and there are reasons to do so.) Popular compressors include zip, gzip, and bzip2.

Compression algorithms work by finding substrings within the input string (which may be a file on disk, or a buffer in memory) that occur repeatedly. These substrings cane be replaced with shorter strings in the compressed output, which includes a dictionary structure that shows the mapping between the shorter and longer string forms. This allows the compressed output to be decompressed, recreating the input.[19]

We next present an example pf a compression algorithm. Quoting, if I may, from Wikipedia, "Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978." LZW reads the input data in a single pass. As the input is read, the leading characters are matched against a dictionary that grows over time. The longest matching string is found, and a (shorter) code that represents that matching string is emitted as output. The matching string, plus the character that follows it in the input, is added to the dictionary. (That string cannot already be in the dictionary, since it would have matched the input stream.) The dictionary is initialized with all the possible strings of length one. When the input string is exhausted, the dictionary that maps strings to the codes that represent them are written to the output, preferably at the *front* of the output, allowing the compressed string to be decompressed in a single pass.

Compression algorithms may suffer in terms of space and time complexity if these dictionaries grow too large. To address this, such algorithms may break the input stream in chunks, of say a megabyte or less, and reset the dictionary at the end of each chunk. [19]

When there are many repeated substrings, the input file in more compressible than if such substrings are few. This implies that strings of low entropy are more compressible than strings with high entropy, and in fact compressors will often refuse to compress an input that is *not* compressible. This may happen if, for example, the file is already compressed - as is the case for PDF files.

2.5 Compression-Based Similarity

We have already introduced the bag of words representation of documents, and the cosine similarity function that arises from that representation. If we regard documents simply as strings, then the compressibility of a document in the presence of another, as it were, is a measure of the similarity between the two documents.

Li formalized this notion as the Normalized Compression Distance, or NCD, which involves taking two strings a and b to be compared, and calculating the lengths of three objects: $C(a)$, $C(b)$ and $C(ab)$ where C is a compression function and ab is the concatenation of a and b.[11] The formula is:

$$NCD = \frac{|C(ab)| - min(|C(a)|, |C(b)|)}{|C(a)| + |C(b)|} \tag{2}$$

Where $|C(x)|$ is the length of the object that results from string x being compressed. NCD has been studied in the context of malware.[18]

If the malware has been packed or even encrypted, NCD is not helpful since files with high entropy aren't suitable for comparison with NCD.

Research Questions: Popular compressors like bzip2 were not designed for use in NCD. What would an ideal compression algorithm for NCD be like? Can such an algorithm be implemented with tolerable if not great time and space performance? Would it matter, in an application like clustering? The recent paper by Borbely [4] should be studied, as well as perhaps Nicholas and Stout [14].

3 Clustering

According to Raschka,"Clustering is an exploratory data analysis technique that allows us to organize a pile of information into meaningful subgroups (clusters) without having any prior knowledge of their group memberships." [15] Since no prior knowledge is required, clustering can be regarded as an unsupervised machine learning technique. There are many excellent surveys of clustering, with Aggarwal and Zhai among the most recent[1].

There are lots of clustering algorithms. The k-meansfamily of algorithms is probably the most well known and the most studied. In general, the k-meansalgorithm works as follows: First, choose k. This may be done with the help of cluster analysis on a small subset of the whole data set. Then select k seed values to serve as centroids (or mediods) of the k proto-clusters. Again, sampling can help with this selection. The algorithm then proceeds with a number of passes through the data. In each pass, each data point is assigned to the proto-cluster surrounding the seed value closest to that data point. After each pass, the centroids are recalculated, based on the points assigned to each proto-cluster. If a pass takes place and no points have moved from one cluster to another, the process has converged and the algorithm terminates. However, it is possible for an oscillation condition to arise, where one or more points alternate from one cluster to another on each pass. To force the algorithm to terminate, one can limit the number of times a given point is allowed to shift, or one can limit the number of passes to some appropriate m. With such limits in place, the time complexity for k-meansis $O(n)$.

The k-meansalgorithm(s) are sensitive to the similarity metric used, and the initial partitions.[9, 2]

We must not forget about hierarchical clustering (HC), which deals explicitly with clusters and the sub-clusters that may exist within them. The output of a hierarchical clustering algorithm is sometimes rendered as a dendrogram, or tree diagram, as shown in Figure 1.

The agglomerative (bottom-up) HC algorithm works as follows. Consider a corpus of n documents. First, a similarity matrix is built that indicates the similarity between documents i and j for all such pairs of documents. The similarity matrix can be kept in upper triangular form, if (as is usual but not always the case) the similarity function is symmetric[3]. The algorithm makes repeated passes over the corpus, until some stopping criterion, such as no two objects are similar enough to warrant merging, is satisfied. Each pass requires the similarity matrix to be scanned for the most similar pair of objects, namely points or sub-clusters, so in the naive version of HC each pass requires $O(n^2)$ time. Two (or more) objects are merged into one on each pass, so a total of n-1 passes are required, yielding $O(n^3)$ for the overall time complexity. The similarity matrix does grow smaller as the algorithm progresses. As each pair of objects are merged, their respective entries can be removed from the similarity matrix, but entries for the newly created, merged object need to be added. The HC algorithm accounts for ties by merging three or more objects with equal similarity into a single cluster. If the similarity matrix is sorted by decreasing similarity score, using a priority queue for example, then the asymptotic complexity goes down to $O(n^2 \lg n)$. The priority queue will need to be updated after each pass, to account for

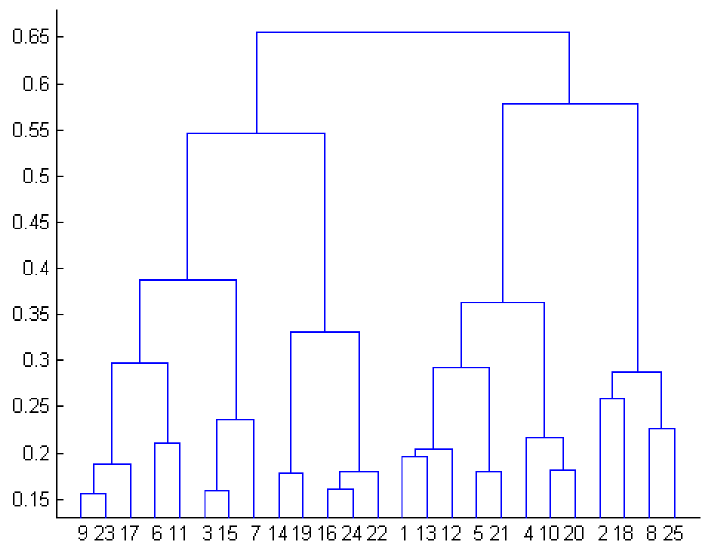

Figure 1: This dendrogram is from `www.mathworks.com`.

the merger of objects, but this can be done in linear time.

Ed can maybe give me a reference for an HC algorithm of $O(n^2)$ time.

If the similarity function preserves triangularity, then we might gain some speed by using $sim(a, b)+sim(b, c)$ as an upper bound for $sim(a, c)$. This lets us defer or avoid some similarity computations. Even so, HC is intrinsically slower than k-means, and requires $O(n^2)$ space, due to the need to refer to the similarity matrix. Unlike k-means, HC is not subject to oscillation. The complexity of HC is discussed at length in Chapter 17 of [13].

4 Evaluation of Clustering Algorithms

In the previous section we (briefly!) summarized the time and space complexity of two of the most common clustering algorithms. Apart from computational complexity, the evaluation of clustering algorithms in terms of how well they "cluster" is not yet well understood. To be a little more formal, we refer to the "effectiveness" of a clustering algorithm, as well as to its complexity. Some studies of clustering effectiveness use what IR researchers would call a "scored corpus", which in this context is a set of documents and a list of the cluster(s) to which each document belongs. Such corpora tend to be small, by modern standards, since human expert judgment is needed to do the scoring[8]. **Any follow on to Halkidi?** For example, the RCV1 corpus is (or has been) widely used for the evaluation of clustering algorithms.

Another approach to measuring clustering effectiveness considers each of the clusters in terms of its entropy. We refer to a *partition* as the output of a clustering algorithm such as k-means, where the partition is bidirectional mapping from clusters to the document(s) contained therein. A partition is which the clusters are tight, in the sense of minimizing total cluster entropy, would be superior to a partition in which the clusters are not as tight. (Some of our previous work on k-meansused cluster entropy as the objective function in an optimization [9].) This relationship between clustering, entropy, and compression was explored by Cilibrasi and Vitanyi [5], among others.

Research Questions:

- Is VX Heaven, with its roughly twenty categories of malware divided into sub-directories, suitable for evaluation of malware clustering? Is there a newer corpus that we should use instead?
- Has there been any follow on to Cilibrasi and Vitanyi?
- Cilibrasi notwithstanding, is NCD suitable for malware clustering? Or any other forms of compression-based similarity as in Nicholas and Stout?
- Does the choice of similarity function make much difference in k-means? Previous work by Kogan *et al* would say yes. How about NCD versus dzd or whatever in a malware corpus?

We noted above that hierarchical clustering was more expensive than k-meansin terms of computational complexity.

RQ: Has anybody used HC in evaluation of k-means?

Suppose we assume that HC gives accurate clusters, whatever that means. We absorb the expense of running HC, and then use each level of the resulting dendrogram, with 2 clusters, or 3, or 4, as ground truth for evaluating a k-meansalgorithm, varying k accordingly. A k-meansalgorithm that did well for several values of k might be a good choice in general.

If we run HC on a subset of the data, we can get reasonable seed values for each value of k. We can still evaluate the cluster quality by checking that those points in the HC sample set are indeed assigned to the right clusters by k-means.

5 Some Research Issues

To summarize the research issues mentioned in this white paper:

- Are there any worthy test corpora for evaluating malware clustering algorithms? Do entropy properties and Heaps' Law hold for VX Heaven and other collections? Are the given categories in VX Heaven valid, in the sense that random pairs of files from *within* a category tend to be more similar than pairs of files from *between* categories? Are there duplicates or near-duplicates?
- Are there compression-based measures of similarity suitable for malware clustering on a large scale? Does the similarity function matter that much, for malware clustering on a large scale?
- Can hierarchical clustering on all or part of a data set be used to evaluate k-means, for different values of k, on that data set?

6 Budget Rationale

Should the work described in this paper be approved, our first step will be to recruit qualified graduate and/or undergraduate students as research assistants. Being the faculty advisor for the UMBC Cyberdefense Team, and the instructor for the malware course mentioned earlier, the author is aware of several students who may be willing and able to work on such a project. The proposed budget calls for two or three undergraduates to be used during the summer of 2016 and academic year 2016-17 as their availability permits.

Benefits for GRAs include health insurance and tuition remission.

Sufficient lab space is available, but we have requested equipment funds to help establish accounts with a few of the important PaaS vendors. Software expenses may include purchase of network analysis and simulation tools.

Travel to relevant domestic conferences is anticipated.

The proposed work will be kept at the unclassified level, unless specific arrangements are made to do otherwise. There are various outlets for unclassified papers, and classified papers maybe be submitted to appropriate venues, such as the Malware Technical Exchange Meeting.

References

[1] Charu C Aggarwal and Chengxiang Zhai. A survey of text clustering algorithms. In Charu C. Aggarwal and ChengXiang Zhai, editors, *Mining Text Data*, chapter 4, pages 77–128. Springer US, 2012.

[2] David Arthur and Sergei Vassilvitskii. k-means ++ : The advantages of careful seeding. In *Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 2011.

[4] Rebecca Schuller Borbely. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques*, 2015.

[5] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 4 2005.

[6] Symantec Corp. Internet Security Threat Report. Technical Report April, 2011.

[7] Marc Damashek. Gauging similarity with n-grams. *Science*, 267(5199):843–848, 1995.

[8] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. Clustering algorithms and validity measures. In L Kerschberg and M Kafatos, editors, *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management SSDBM 2001*, pages 3–22. IEEE Comput. Soc, 2001.

[9] Jacob Kogan, Marc Teboulle, and Charles Nicholas. Data driven similarity measures for k-means like clustering algorithms. *Information Retrieval*, 8:331–349, 2005.

[10] Daniel Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(21 October):788–791, 1999.

[11] M. Li, X. Chen, X. Li, B. Ma, and P.M.B. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 12 2004.

[12] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *Security & Privacy, IEEE*, 5(2):40–45, 2007.

[13] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[14] Charles Nicholas and Kevin Stout. Experience with compression-based distance metrics for malware. In *Malware Technical Exchange Meeting*, El Segundo, CA, August 14-16 2012. poster session.

[15] Sebastian Raschka. *Python Machine Learning*. Packt Publishing, 2015.

[16] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. no starch press, 2012.

[17] Ivan Sorokin. Comparing files using structural entropy. *Journal in Computer Virology*, 7(4):259–265, 6 2011.

[18] Stephanie Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320, 2007.

[19] Terry A Welch. A technique for high-performance data compression. *Computer*, 17(June):8–19, 1984.