Byte Embeddings for File Fragment Classification

Md Enamul Haque[a], Mehmet Engin Tozal[a]

[a]*School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, 70504, LA, USA*

Abstract

In digital forensics, file carving is the process of recovering files on a storage media in part or in whole without any file system information. An important problem in file carving is the identification of fragment types. Many fragment classification studies in the literature employ inflexible and indiscernible feature selection methods such as different statistics of byte frequency distributions. Moreover, assessing the strengths and weaknesses of some approaches is difficult as they are specific to certain file types such as graphics. In this paper, we propose a novel feature generation model using byte embeddings (`Byte2Vec`) which map fragments to dense vector representations. The proposed model extends the *word2vec* and *doc2vec* document embedding models to bytes and fragments, respectively. We use `Byte2Vec` for feature extraction and *k*-Nearest Neighbors (*k*NN) for classification. We present effectiveness of `Byte2Vec+kNN` in file fragment classification using a publicly available digital forensics dataset and a random web search dataset. Our experimental results show that `Byte2Vec+`*k*`NN` reaches an accuracy rate of 72% along with 74% precision and 72% recall. Compared to the other feature extraction techniques such as n-gram, byte distributions, byte statistics, byte distances, and sparse dictionaries for byte n-gram along with different classifiers, `Byte2Vec+`*k*`NN` achieves an absolute improvement of 3% and 12% in accuracy and precision, respectively.

1. Introduction

Digital forensics is a branch of forensic science dealing with the collection and analysis of information affiliated with digital devices. In its most common form, digital forensics involves the analysis of data kept on storage media in electronic devices such as computers, phones, tablets, and cameras. Digital storage devices including hard drives, flash drives, or SD cards are used for recording and retaining digital information. Often, the recorded information such as email communications, pictures, videos and confidential documents are used in legal proceedings and criminal investigations.

A storage device stores information as blocks of raw data without any particular organization or access control. A disk-block (or sector) is the smallest physical storage unit on a storage device with a typical size of 512 or 4096 bytes. A file system on the other hand, facilitates the organization, management, storage, and retrieval of information. File systems abstract the entire storage area as file-blocks with typical size of 4096 bytes. More importantly, they store raw data in terms of files and organize these files into folders. A file system oversees all in-use and available blocks on storage devices; manages meta information, e.g., owner, size, access rights and creation time, about files; and keeps track of the blocks holding the actual content of the files. File systems use the first several sectors of a storage device to keep information about the overall storage space, files and their organization. They use the remaining blocks to store the actual content of the files. Figure 1 shows a high-level outline of an ordinary storage medium. In the figure, the boot sector contains the instructions to boot the device. The super block contains information about the file system itself. The file system data structures keep information about files and their data blocks. Lastly, the data blocks hold the actual content of the files. Note that the data blocks constituting the content of a file are not necessarily contiguous. In the figure, the first two blocks of file *A.pdf* is followed by an unused block and three blocks belonging to another file, *B.png*.

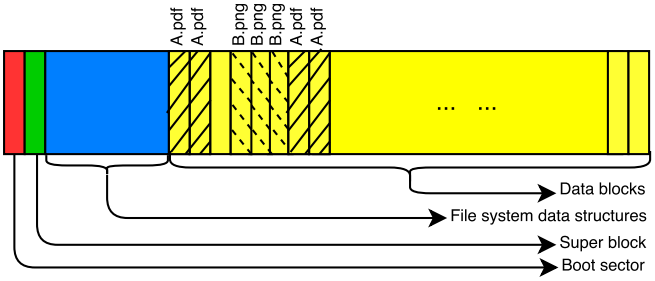


Figure 1. The high-level structure of an ordinary storage medium

File carving is the process of recovering files on a storage media without the file system information in digital forensics [1, 2, 3]. File system information may be unavailable due to disk damages or disk format operations. However, the most common cases take place when one or more deleted files are needed to be recovered. Note that when a file is deleted the file system deletes the internal record of the file and returns its data blocks back to the pool of available blocks. Typically, the file system does not zero-out the returned data blocks because their content will be overwritten when they are reclaimed later.

A file system stores and retrieves the content of a file as

Email addresses: `enamul@louisiana.edu` (Md Enamul Haque), `metozal@louisiana.edu` (Mehmet Engin Tozal)

https://doi.org/10.1016/j.future.2021.09.019

blocks of raw bytes without any reference to the type of the content. However, files are encoded in standard formats to achieve portability among different platforms and applications. For example, *.png* files store bitmap images using lossless compression or *.pdf* files encapsulate document layouts along with images, texts, and fonts to display it. Therefore, an essential step in file carving is the identification of block/fragment types. Note that, the term "fragment type" instead of "block type" is more common in digital forensics domain. Once the fragment types are identified, the next step involves ordering and merging the fragments to reconstruct the original file(s) in part or in whole [4].

Most of the earlier file fragment classification research focus on different statistical methods and machine learning algorithms for feature generation and classification, respectively [5, 6, 7, 8]. In some other works, researchers use file header, footer, magic numbers and MIME types as features to classify fragments [9, 10, 11, 12]. Although, much progress have been reported for statistical approaches, reproduction of the observed results is quite challenging due to the use of non-standard datasets [6, 13]. Additionally, it is difficult to assess the strengths and weaknesses of some approaches, because they do not generalize across different file types [14, 15, 16, 17].

In this work, we propose a novel feature generation model using byte embeddings (`Byte2Vec`) which maps fragments to dense vector representations. The proposed technique extends the idea of continuous vector representations of words (*word2vec*), which is widely used in natural language processing. *Word2vec* generates vector representations of words in high-dimensional space using continuous bag of words [18] and Skip-gram [19] models. `Byte2Vec` on the other hand, generates vector representations of bytes in file fragments using the Skip-gram model. For each byte b in a fragment, `Byte2Vec` generates a vector, $\mathbf{b}$, based on the surrounding bytes of b in the fragment. To generate the vector, $\mathbf{b}$, we set up a neural network which maximizes the log likelihood of the neighboring bytes of b in the fragment. One can think of the vector $\mathbf{b}$ as the enclosing context of byte b. Once we generate the continuous vector representations of the bytes for different fragment types, we aggregate them to obtain a corpus model for each file type. Next, we generate features to train a k-Nearest Neighbors (kNN) classifier by averaging the vector representations of the bytes in our training dataset. Finally given a fragment, we identify its type using the classification model generated by the kNN classifier.

The main contribution of this paper is the `Byte2Vec` corpus model which extends the *word2vec* and *doc2vec* to bytes and file fragments, respectively. Unlike some of the previous methods, `Byte2Vec` works for different block sizes and supports fragments of any type. Moreover, the addition of a new file type does not require the reconstruction of any existing corpus models. To support reproducibility, we use a publicly available dataset obtained from Digital Corpora to train, test and compare our approach. We use `Byte2Vec` to extract features from fragments and k Nearest Neighbor algorithm to classify them, i.e., `Byte2Vec+`k`NN`. Our experimental results show that `Byte2Vec+`k`NN` reaches an accuracy rate of 72% along with 74% precision and 72% recall. Compared to the other feature extraction techniques such as n-gram, byte distributions, byte statistics, byte distances, and sparse dictionaries for byte n-gram along with different classifiers, `Byte2Vec` achieves an absolute improvement of 3% and 12% in accuracy and precision, respectively.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the proposed model, `Byte2Vec`. Section 4 presents experimental results of `Byte2Vec` model along with the dataset collection, creation, and sampling. This section also presents comparative analysis of results obtained from `Byte2Vec` and some well-known feature extraction and classification techniques in digital forensics. Finally, section 5 concludes our work.

2. Related Work

Several works in the literature propose solutions to the file and fragment type classification problem.

McDaniel and Heydari developed an approach to generate file "fingerprints" [20]. They use Byte Frequency Analysis (BFA), Byte Frequency Cross-correlation (BFC), and File Header/Trailer (FHT) algorithms for file type detection.

Roussev and Quates [21] introduced a deflate classifier, *zsniff*, for parsing *zlib/deflate* encoded data. The *zlib/deflate* encoding consists of a sequence of compressed blocks each comprising of a 3-bits header, Huffman tables, and compressed data. The proposed tool can distinguish among deflate-coded text/markup, deflate-coded image data (*.png*), and deflate-coded portable executables.

Axelsson [5] applied normalized compression distance (NCD) instead of standard Euclidean distance as the distance metric to be used in k-NN classifier for file fragment classification. To calculate NCD between two fragments, one needs to first append the fragments together and then find the complexity. Then the complexity is normalized using min-max function. The proposed method performs significantly slower compared to other fragment classification methods due to high computational complexity of NCD. On the other hand the classification performance is not significant for most of the file types.

Gopal et al. reported four types of problematic cases in file type classification including files with missing signature bytes, random bytes, and isolated segments [22]. The main focus of their work is on the performance comparisons of statistical classifiers and COTS (Commercial Off the Shelf) solutions. Experiments, performed on *RealisticDC* corpus [23], also shows that SVM and kNN outperforms all COTS solutions.

Conti et al. [6, 13] presented a statistical analysis of 14,000 low-level binary fragments along with the presentation and evaluation of a classifier for identifying 14 primitive binary fragment types. The statistical features include Shannon entropy, Hamming weight, Chi squared goodness of fit measure, and arithmetic mean which proved useful for analyzing low-level binary data without relying on possibly non-existent or untrustworthy metadata.

Xu et al. [24] proposed gray-scale image pixels based solution for file fragment classification. The idea is to construct

features from binary fragments by linear transformation. A well known image feature, *GIST*, is employed with kNN classifier to classify the fragments. Their experimental results show that the classification accuracy varies between 39% and 55%.

Veenman used histogram, entropy, and Kolmogorov complexity as features to classify disc images [8]. Support vector machine with unigram and bi-gram features are used to classify file fragments in [4]. However, the general approaches for file fragment classification, e.g., header-footer classification, statistical analysis, and machine learning algorithms, are considered flawed in [23].

Li et al. showed that collection of 1-gram binary distributions, *file-prints*, are distinct for different files [14]. They use K-means clustering [25] on the *file-prints* to generate models for each file type on eight different data sets. Although the experimental results show promising accuracy, the approach is not applicable to most of the file fragments due to the use of header information as prefix.

Calhoun and Coles performed Fisher analysis on four types of graphics files (*.jpeg*, *.gif*, *.bmp*, and *.pdf*) for classification [15]. Shannon entropy and frequency of ASCII codes are considered as features to achieve 88.3% classification accuracy. One limitation of this method is that it is applicable to only graphic types.

Karresand et al. proposed a classification method named OSCAR [16] using a similar centroid idea presented in [14]. They extend their previous work [17] by introducing rate-of-change (RoC) which computes the difference of the ASCII values between two consecutive bytes. However, their method performs well only on *.jpg* file types.

Recently, Wang et al. [26] proposed an approach for file fragment classification that uses sparse coding of different dimensions and uni/bigram as feature set. The method learns sparse dictionaries for n-grams, continuous sequences of bytes, of different sizes with respect to file fragments. Later, these dictionaries are used for new file fragments to generate features and classify them using Support Vector Machine (SVM).

In this study, we propose a novel approach for file fragment classification with the introduction of continuous vector representations of bytes, `Byte2Vec`. Unlike most other approaches, our model works with high entropy file fragments and generalizes the feature extraction process. Roussev and Garfinkel argued that existing methods do not work well for high entropy fragments because there is no discernible pattern to exploit [23]. The proposed approach on the other hand, does not have such a limitation for either low or high entropy fragments. Although there are different file types, our approach can efficiently differentiate their fragment patterns except a few. As the original bytes are transformed into a lower dimensional vector space, the patterns become distinguishable enough for classification.

3. The `Byte2Vec` Model

In the following, we first describe *Skip-Gram*, *CBOW* [18, 19] and *Paragraph Vector* models [27] from which `Byte2Vec` is extended. Next we introduce the theory of `Byte2Vec` along with its implementation. Finally, we briefly present k-Nearest Neighbors as it is the classifier used with `Byte2Vec`. We do this to accentuate the fact that the underlying vector representation model used for feature generation in `Byte2Vec` is highly transferable to solve problems in other domains.

Both continuous bag of words and skip-gram models, together called as *word2vec*, were proposed as an efficient neural language model to learn embeddings (or vector representations) for words. The *word2vec* model creates dense vector representations of words which carries semantic meanings and are useful in a wide range of applications from sentiment analysis [28, 29, 30] to on-line product recommendations [31, 32, 33]. The most useful aspect of the word embeddings is that the vectors interpret the semantic meanings of words. For example, words with similar/opposite connotations are inclined to have similar/opposite vectors considering the cosine distance measure. Here is an example from the original papers [18, 34] that shows the idea: $v_{queen} - v_{woman} + v_{man} \simeq v_{king}$, where $v_{queen}, v_{woman}, v_{man}$ and v_{king} are the word vectors for queen, woman, man, and king respectively. The working principle of the two models (CBOW and Skip-gram) are opposite of one another. In CBOW model, a word is predicted given a context as an input which can be a single or multiple words. In Skip-gram model however, the context is predicted given a word as an input.

Paragraph vector model [27] is an unsupervised learning framework that learns distributed representations of texts of any length such as sentences, paragraphs, and documents. It is also known as *doc2vec* which is an extension of *word2vec* for learning embedding vectors for documents. The *doc2vec* model creates unique vector representations for paragraphs with the help of *word2vec* model. The word vectors in a paragraph are averaged and/or concatenated along with unique document identifiers to obtain the paragraph vector. We used the coordinate-wise average method to combine the byte embeddings.

The primary goal to build `Byte2Vec` is to create byte embeddings (or vector representations) for fragments which can be used as features during classification processes. Note that `Byte2Vec` generates a corpus model for all file types with respect to different vector lengths and these corpus models can be used with different classifiers. We used kNN as default classifier in our experiments. In addition, new corpus models can be generated with new file types and vector lengths independent of the existing ones.

3.1. `Byte2Vec` Theory

We use the Skip-gram algorithm to generate corpus models for different file types. The Skip-gram algorithm (*word2vec*) requires the labeled file fragments to construct the corpus model using a neural network. On the other hand, the vector aggregation concept from *doc2vec* is used to generate the final feature representation from the output of the hidden layer of the Skip-gram model.

Figure 2 shows the architecture to create a vector representation of a byte with respect to its surrounding bytes with window size c. The two-layer layer neural network consists of a hidden and an output layer. The hidden layer generates a k-dimensional

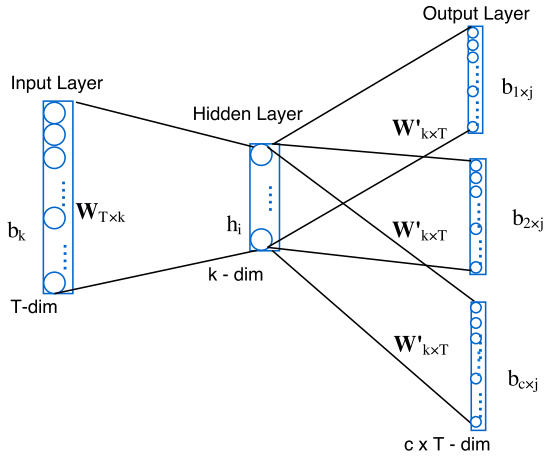


Figure 2. Skip-gram model predicting context bytes given an input byte. This architecture provides a vector of probabilities for all the context bytes of length c.

vector representation of byte b_i. The output layer generates the probabilities for all the bytes present in the window c to be the context of the input byte b_i. As our goal is to generate the vector representations of bytes, we are more interested in the output of the hidden layer. The input vector is transformed into "one-hot" representation that refers to a vector of length T consisting of all zeros and a one at the i-th position. T refers to the vocabulary size, and the i-th position is one for the i-th byte in the vocabulary. As a result the i-th row of the hidden layer will be selected, which is the vector representation of the i-th input byte. Once Skip-gram model is trained on the whole corpus of bytes, the vector representations are obtained from the output of the hidden layer for the corresponding bytes. Note that the vector length k refers to the feature size which is an important parameter for our `Byte2Vec` corpus model. For a fragment of size $|F|$, the `Byte2Vec` corpus model generates a feature vector $\boldsymbol{v}$ of size k by applying vector averaging using Equation 1.

$$\boldsymbol{v} = \frac{1}{|F|}\sum_{i=1}^{|F|} \texttt{Byte2Vec}[b_i] \tag{1}$$

The training objective of the Skip-gram model is to find the byte representations that are useful for predicting the context bytes in a fragment or a file. More formally, given a stream of training bytes $b_1, b_2, b_3, \ldots, b_L$, the objective is to maximize the average log probability:

$$\frac{1}{L}\sum_{t=1}^{L}\sum_{-c\le j\le c, j\neq 0} \log p(b_{t+j}|b_t) \tag{2}$$

where c is the size of the training context or window which is a function of the center byte b_t and L is the number of input bytes. The training time to build the models grow with the increase in context size. The basic Skip-gram formulation defines $p(b_{t+j}|b_t)$ using softmax function:

$$p(b_O|b_I) = \frac{\exp\left({v'_{bO}}^{\top} v_{bI}\right)}{\sum_{b=1}^{T}\exp\left({v'_b}^{\top} v_{bI}\right)} \tag{3}$$

where v_b and v'_b are the input and output vector representations of byte b, and T is the vocabulary size. In our training models each of the fragment types has a maximum of 256 unique bytes. Computing $\nabla \log p(b_{t+j}|b_t)$ (gradient) in Equation 2 of basic Skip-gram model is computationally expensive. However, in our approach, the computational cost is limited due to the smaller vocabulary size, 256. In the original Skip-gram model, a hierarchical softmax and negative sampling are used to minimize the cost.

3.2. `Byte2Vec` Implementation

In this part we present the details of our `Byte2Vec` approach using Algorithms 1 and 2. Note that, two types of training inputs are mentioned in the algorithms. First, corpus training files are needed to generate `Byte2Vec` corpus models. Second, training and test fragments are required to generate feature matrices from `Byte2Vec` corpus models as input for the kNN classifier. During feature matrix generation, file types are known only for the training fragments.

Algorithm 1 explains how `Byte2Vec` corpus models are generated for different user defined vector lengths. We use a publicly available dataset containing 1 million documents, collected from `govdocs1` [35]. Selected files are treated as input to the algorithm. The files are then fragmented and sampled using our "fragment generation" process which is described in Section 4.1. Vocabulary and models are initialized to NULL during the initialization phase of the algorithm. Minimum (η) and maximum (λ) vector lengths are also initialized at this stage. Note that, we increase vector length by the minimum vector length (η) at each time step. One can easily use a customized parameter for this during the corpus training phase. Once the fragments are generated, lines 2-4 collect all the fragments to build the vocabulary required for training Skip-gram model. Lines 5-8 generate corpus models from the vocabulary for different vector lengths that are initialized at the initialization step. Line 9 returns all the different `Byte2Vec` corpus models based on the Skip-gram model.

Algorithm 1 `Byte2Vec` corpus model generation

Input: input files ▷ corpus training files
Output: models ▷ corpus models
Initialization : vocabulary[] = ∅, model[] = ∅, η, λ
1: generate fragments from *input files* using Algorithm 3.
2: **for** each fragment **do**
3: vocabulary = vocabulary ∪ fragment
4: **end for**
5: **for** $i = \eta$ to λ **do**
6: models[i] = build skip_gram model on the vocabulary using Equations 2 and 3.
7: $\lambda = \lambda + \eta$
8: **end for**
9: **return** *models*

Algorithm 2 generates a feature matrix from the training fragments. The number of the fragments (n), feature length (k), feature vector ($\boldsymbol{v}$), and feature matrix ($\boldsymbol{V}$) are initialized during the

initialization phase. We use $\mathbf{0}_{n,k+1}$ to refer a matrix with n rows and $k+1$ columns having all the elements initialized to zero. $\boldsymbol{b}$ refers to a byte vector of dimension 4096, i.e., a fragment. $\boldsymbol{v}$ defines a k-dimensional vector of zeros ($\mathbf{0}_k$).
Lines 1-12 generate the feature matrix of dimension $n \times (k+1)$. Line 2 converts the i^{th} fragment into a byte array. Line 3 finds out the corresponding file type of the i^{th} fragment. It is to be noted that the test fragment types cannot be directly known from this step. Line 4 is used to select the `Byte2Vec` corpus model. Line 5 calculates the fragment length. Lines 6-8 create a feature vector for each byte of the i^{th} fragment and aggregate them into vector $\boldsymbol{v}$. $m[\boldsymbol{b}[j]]$ at line 7 refers to the k-dimensional vector representation of the j-th byte of the i-th fragment. Line 9 averages the aggregated vector representation of the i^{th} fragment. Line 10 copies the i^{th} vector of length k to i^{th} row of matrix $\boldsymbol{V}$. $\boldsymbol{V}_{i,k+1}$ position is filled by the fragment type collected at line 3. This process is repeated for all the fragments available in the training and test dataset. The only difference for the test dataset is that we remove the $k+1$-th column. At the end, the algorithm returns a feature matrix $\boldsymbol{V}$.

Algorithm 2 Feature matrix generation

Input: input fragments ▷ training or test fragments
Output: feature matrix, $\boldsymbol{V}$
Initialization: n, k, $\boldsymbol{V} = \mathbf{0}_{n,k+1}$, $\boldsymbol{v} = \mathbf{0}_k$
1: **for** $i = 1$ to n **do**
2: $\boldsymbol{b} = i^{th}$ fragment
3: $t = i^{th}$ file type ▷ known for training fragments only
4: m = `Byte2Vec` corpus model
5: l = length of $\boldsymbol{b}$
6: **for** $j = 1$ to l **do**
7: $\boldsymbol{v} = \boldsymbol{v} + m[\boldsymbol{b}[j]]$
8: **end for**
9: $\boldsymbol{v} = \boldsymbol{v} \,/\, l$
10: $\boldsymbol{V}_{i,1:k} = \boldsymbol{v}$
11: $\boldsymbol{V}_{i,k+1} = t$
12: **end for**
13: **return** $\boldsymbol{V}$

The low dimensional vector generation architecture is also depicted in Figure 3. The figure shows a toy example fragment with six bytes $b_1, b_2, \ldots, b_6$. For each of the bytes b_i, the corresponding vector v_i of length k is generated using the `Byte2Vec` corpus model. The vectors are averaged to get the final feature vector of dimension k for all the six bytes present in the fragment. For this particular example, we explain how the averaging is performed using coordinate wise calculation. As each byte has feature vector length k, the vector representation of byte b_1 is $v_{b_1} = [v_{11}, v_{12}, \ldots, v_{1k}]$. Similarly for byte b_6, we have $v_{b_6} = [v_{61}, v_{62}, \ldots, v_{6k}]$. Thus the coordinate-wise average of these six bytes become: $\frac{1}{6}\sum_{i=1}^{6}[v_{i1}, v_{i2}, \ldots, v_{ik}]$. It should also be mentioned that the original paper [19] refers to concatenation as another method of aggregation.

3.3. `Byte2Vec` Classification

We consider `Byte2Vec` in the following supervised classification setting. We create z labeled fragments $\boldsymbol{V} = (\mathbb{X}^z, \mathbb{Y}^z)$ with

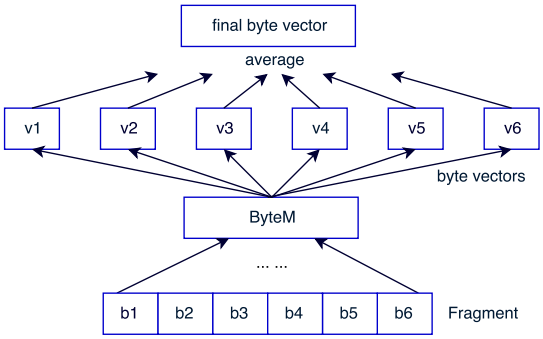


Figure 3. A framework for learning byte vectors. This diagram is shown for an example fragment with six bytes $(b_1, b_2, b_3, \ldots, b_6)$ and their corresponding vectors $(v_{b_1}, \ldots, v_{b_6})$ of length k.

the help of `Byte2Vec` corpus model. The fragments are then randomly split into 70% training ($\boldsymbol{V}^{train} = (\mathbb{X}^{trn}, \mathbb{Y})$) and 30% test data ($\boldsymbol{V}^{test} = (\mathbb{X}^{tst}, \hat{\mathbb{Y}})$) with 10-fold cross validation. While building `Byte2Vec` corpus models using Algorithm 1, we examine 63 different file types. As a result $\mathbb{Y} \in \{1, 2, 3, \ldots, 63\}$, where each number in the set corresponds to a specific file type. This setting makes our file fragment classification task a multi-class classification problem.

In the following, we present a brief introduction to k-Nearest Neighbors (kNN) classifier which we adopted to solve our multi-class classification problem. kNN is nonparametric and falls in the supervised learning family of algorithms. This means that we are given a labeled dataset consisting of training observations $(\mathbb{X}^{trn}, \mathbb{Y})$ and would like to capture the relationship between $\mathbb{X}^{trn}$ and $\mathbb{Y}$. More formally, our goal is to learn a function $h : \mathbb{X} \rightarrow \mathbb{Y}$ so that given an unseen observation x, $h(x)$ can confidently predict the corresponding output y. We employ a basic kNN with 1-hop neighborhood as a parameter. Increasing the value of k from 1 to 4 only adds computational complexity, but does not provide significant improvement in the evaluation metrics. Therefore, we chose $k = 3$ to reduce computational complexity. Additionally, we tried standard SVM with linear kernel to classify the data. However, that did not perform well as it tries to make a universal hyperplane from the whole data at the time of classification. On the other hand, kNN uses 1 hop neighbor to see which fragment types are closer to the existing fragments in the transformed space.

4. Experiments

In this section we first introduce our dataset. Next, we present an empirical analysis of `Byte2Vec`. Then, we compare our approach with several well-known classification techniques in digital forensics literature. Next, we provide an approximate timing analysis on feature generation and classification process. Finally, we present a discussion on classification performance with respect to file types and their general categories.

4.1. Dataset

Most of the earlier file fragment classification studies are evaluated using private datasets. Additionally, researchers used

copyrighted datasets which are difficult to collect. Consequently, the reproduction of their results may be quite challenging. Considering the issue, we use a publicly available dataset containing 1 million documents, collected from `govdocs1` [35] which is also suggested in [36, 37]. This directory consists of 1000 folders, each containing approximately 1000 files of 63 different file types. Each file in the corpus is presented as a numbered file with a file extension, e.g., *0000001.jpg*. Table A.5 in Appendix Appendix A demonstrates the frequency distribution of files and corresponding fragments. The table shows that *.pdf* is the most frequent file type and *.icns* is the least frequent. The file and fragment frequencies are imbalanced with respect to the file types. Since unconditional evaluations of imbalanced classifications might be misleading, we applied under-sampling to achieve a clear assessment [38]. We use another `random` dataset collected from Google search for 9 different file types with keywords as *.docx, .gz, .jpg, .pdf, .png, .pptx, .swf, .xls, .xlsx*. We collect 10 files of each category and create a separate set of fragments. In this case, we randomly select different sample sizes ranging from 500 to 2500 with increment of 500 as interval for the experiments. Note that the sample size includes fragments from all 9 different file types. On the contrary, we selected fragments with respect to type-wise samples from `govdocs1` data due to higher file counts. In the following we detail the fragment creation and sampling procedures to support reproducibility in future studies.

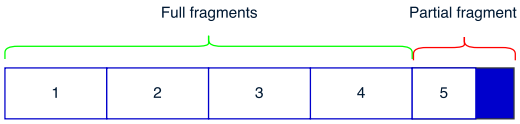


Figure 4. Concept of full and partial fragment creation using a toy example with four full and one partial fragment. Each block is considered as a fragment.

Algorithm 3 Fragment creation

Input: F ▷ set of files
Output: f ▷ set of fragments
Initialization: $q = 4096$ ▷ fragment size
1: **for** $i = 1$ to $|F|$ **do**
2: $m = i$-th file size
3: $d = \lceil \frac{m}{q} \rceil$ ▷ number of allocated fragments
4: **if** $m \mod q \neq 0$ **then**
5: $e = (q * d) - m$
6: $lb = m - (d - 1) * q$ ▷ last fragment data length
7: $f = f \cup$ generate $d - 1$ full fragments of size q
8: $f = f \cup$ generate 1 partial fragment of size $q = e + lb$
9: **else**
10: $f = f \cup$ generate d full fragments of size q
11: **end if**
12: **end for**
13: **return** f

Fragment creation. Algorithm 3 details the fragment creation process. The algorithm expects a set of files, F, as input and returns a set of fragments, f, as output. At lines 1-12 the algorithm traverses the set of files and generates the fragments for each file. Line 2 collects the file size in bytes into variable m. Line 3 computes the fragment count of the i^{th} file by dividing the file size by fragment size q. We set the fragment size to 4096 bytes because it is the default value in modern file systems. Line 4-11 generates both full and partial fragments. If file size is completely divisible by q, then the algorithm generates all full fragments by slicing the file into the fragments of equal size $q = 4096$. Otherwise, it creates $d - 1$ full and 1 partial fragments. To create a partial fragment the algorithm takes random bytes of length e from another file or fragment and append it to the last bytes of length lb of the file being processed. We included the partial fragment from random file types to see how the performance gets impacted. Our intuition is that the random bytes can not provide contextual byte similarities to the existing fragment type. For example, including random *xlsx* bytes at the end of a pdf fragment means it will only add contextual bytes for *xlsx*. However, in the best case scenario, random *pdf* bytes are appended at the end of a *pdf* file which makes the classification relatively simpler. Figure 4 presents an example file consisting of four full fragments and one partial fragment. The area shown in white represent the bytes belonging to the file. The black region corresponds to the bytes that are randomly selected from another file or fragment. Note that most of the previous works disregarded the partial fragments in classification to avoid experimental complexity [15, 6, 39].

Sampling. Due to the large and uneven number of fragments, we pick a set of random samples. A few file types such as *.icns*, *.mac* and *.lnk* have frequency counts as close to zero compared to some other high frequency files such as *.pdf*, *.png* and *.csv*. We consider each fragment type as a strata and apply stratified sampling with equal allocation to correct potential disputes related to imbalanced classification.

We made samples for both corpus model generation and classification to conduct the experiments. While building `Byte2Vec` corpus models, we considered maximum of 4000 samples from each fragment category to ensure involvement of all 63 file types. However, the model generation is not limited to the 63 file types available in the `govdocs1` directory. We can always update the models whenever new file types come into account.

Table 1. The number of file types for different balanced sample sizes on `govdocs1` dataset.

Sample size	500	1000	1500	2000	2500
File type count	42	40	37	35	35

Table 1 shows the sample size and the number of fragment types taken to generate the training and test data for the classification procedure. Note that, we omit the files with lower frequencies while generating the training and test data to avoid any classification bias. For example, sample size 1000 refers to a set of randomly selected fragments where every type has exactly 1000 instances. This process enabled us to generate a

large dataset of file fragments with an equal number of file fragments from each file type. The type count decreases due to the imbalanced fragment count. Table 1 also shows that the file types decreases with the increase in sample size. If we continue to increase the samples per type then total types will likely to decrease. We take different samples to show how increasing and decreasing the file types affect the overall performance of the fragment classification process. Please see Appendix Appendix A for a detailed description of `govdocs1` data statistics.

Corpus model building. It is to be noted that once we create the set of fragments we do not refer to the files later. The corpus model is generated by randomly selecting maximum of 4000 fragments. Also note that, we needed at least a single fragment of each file type to build the whole corpus model. However, during the testing phase of classification, we performed undersampling to acquire subsets of data fragments. The training and test dataset is sampled from the same population used for the model. Although random sampling may introduce some overlapping fragments, the training and test dataset do not overlap. The overlap for building the corpus models does not affect our results because it averages the byte embeddings for the fragments of different file types. In summary, we followed the steps below for the experimental evaluation.

- Model generation: we build corpus models from full file fragment samples of file types where sample size does not exceed 4000 fragments.
- Training and testing: we create a set of sample data fragments of 500,1000,1500,2000, and 2500 for valid file types. Here valid refers to the types that have at least the desired number of sample fragments. We perform 10-fold cross validation for every sampled dataset with train:test ≃ 7:3 ratio.

Reproducing Byte2Vec. All data and code used in this paper, as well as the models and sampled data fragments are available in this link (`https://github.com/enamul-haque/Byte2Vec/`) to help with the reproduction of the results.

4.2. Empirical Analysis of Byte2Vec

In the following, we first use `Byte2Vec` to generate corpus models for different file types. We demonstrate the time it takes to generate the models using different vector lengths. Then, we use the corpus models to generate a feature matrix for fragment classification using *k*NN. We present the empirical results demonstrating the accuracy, precision and recall with respect to different vector lengths as well.

Figure 5 shows `Byte2Vec` model generation time in hours for varying vector lengths starting from 5 to 100 with intervals of 5. We used window size of 5 during model generation (corpus training) step that reflects the context bytes at distance 5 for an input byte. In addition, we disregarded bytes with zero frequency and employed four parallel threads for faster model generation. The most computation sensitive part (off-line), corpus model building, is run on a multi-core server with 32 physical processing cores, 512 GB of RAM, and a clock rate of 2.5 GHz per core. In the figure, model generation time decreases between vector lengths of 5 and 10 and then, it increases monotonically. The decrease observed in the beginning is related to the initial loading of data and model parameters. The model generation time approximately presents a linear increase as the vector length grows. Note that the state-of-the-art methods for file fragment classification do not have any precomputed model building step. Therefore, we are unable to compare this aspect.

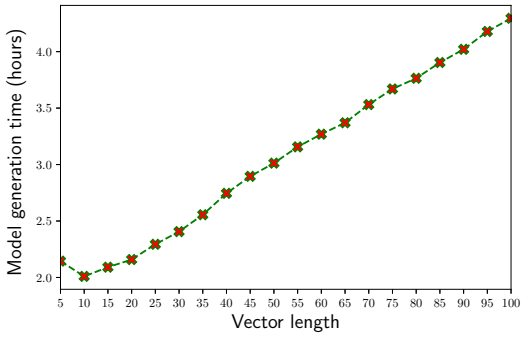


Figure 5. `Byte2Vec` corpus model generation time for varying vector lengths for `govdocs1` dataset.

Our experimental results involve the classification of fragments using `Byte2Vec` generated features. Among five different samples, we chose the highest sampled (2500) dataset to explain the results while achieving an unbiased classification. We use *k*-Nearest Neighbors (*k*NN) with $k = 3$ for all classification steps. Additionally, we perform 10-fold cross validation for every sampled dataset with train:test ≃ 7:3 ratio. The train and test data never overlap to ensure bias-free classification.

4.2.1. Accuracy

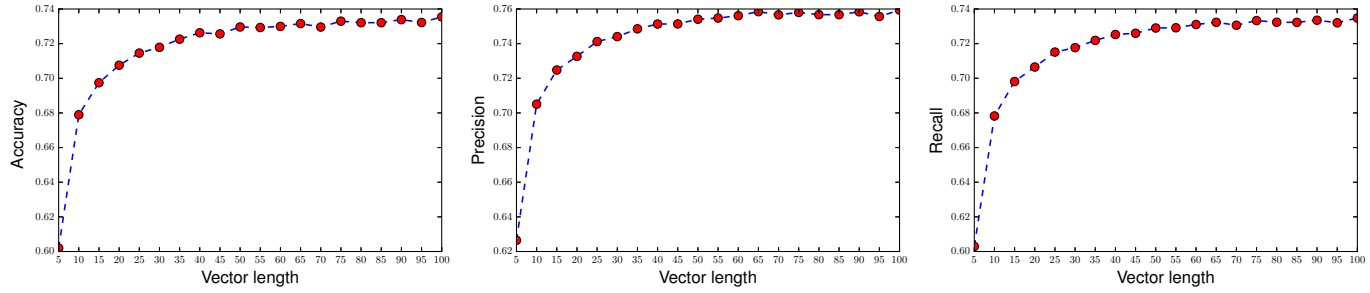
Accuracy of a classifier is straightforward to measure. Equation 4 defines standard accuracy measure of a classifier where $\hat{y}$ and y denote actual and predicted class labels, $1(.)$ refers to an indicator function, and n denotes the total number of the instances.

$$Accuracy(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} 1(\hat{y}_i = y_i) \tag{4}$$

Figure 6a and 7a shows the classification accuracy of `Byte2Vec` feature generation model on `govdocs1` and `random` dataset, respectively. The accuracy significantly improves for larger vector lengths up to 60 in Figure 6a. After 60 however, our results show very small improvements in accuracy. Maximum accuracy (0.73) is achieved with vector/feature length of 60 and above. Note that the model generation time also increases as the vector length increases. Similar pattern is also observed in Figure 7a using our corpus model while the pattern is not as smooth as Figure 6a.

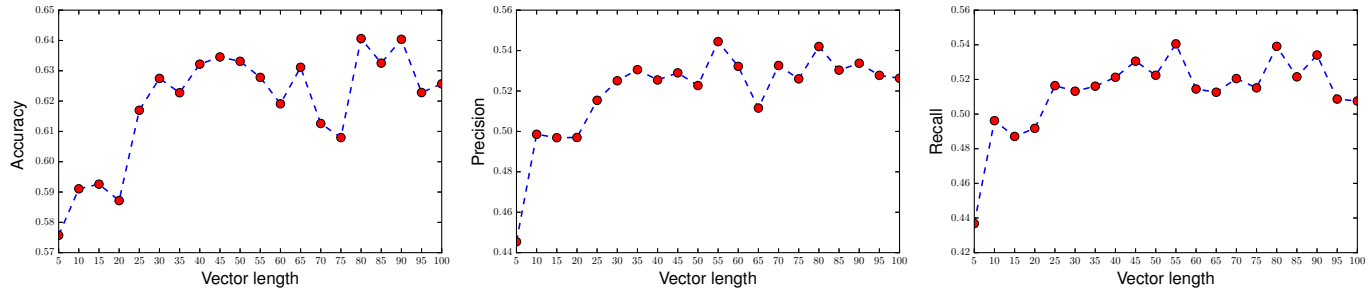
4.2.2. Precision

In order to support the accuracy of our model, we show the precision scores in Figure 6b and 7b for `govdocs1` and `random` datasets. Precision scores measure the precision across all labels, i.e., the number of times any class was predicted correctly



(a) Accuracy (b) Precision (c) Recall

Figure 6. Classification (a) accuracy, (b) precision, and (c) recall for `govdocs1` dataset with sample size of 2500 of each file type.



(a) Accuracy (b) Precision (c) Recall

Figure 7. Classification (a) accuracy, (b) precision, and (c) recall for `random` dataset from Google search with sample size of 2500 of each file type.

(true positives) normalized by the number of the data points. We use "macro" precision score as the dataset has balanced labels. Equation 5 shows the normalization function $P(.)$ with actual and predicted output as parameters.

$$Precision = \frac{1}{|L|} \sum_{l \in L} P(y_l, \hat{y}_l) \tag{5}$$

where L denotes set of labels ($|L| = 63$) and $P(y_l, \hat{y}_l) = |y_l \cap \hat{y}_l| / |y_l|$. The $|y_l \cap \hat{y}_l|$ in the numerator denotes the number of correct predictions. Figure 6b shows that the precision scores are also consistent with the accuracy scores. The figure confirms the robustness of our method with a maximum precision of 0.75 for vector length of 60 and above. Figure 7b shows that the maximum precision of 0.54 is achieved for vector length of 60 and above. We observe performance degradation as our training data is very low in case of random fragments compared to `govdocs1` dataset.

4.2.3. Recall

We use "macro" recall as another evaluation metric due to the balanced labels, again. Recall scores show the ratio of correctly predicted positive observations to all observations in a particular class. Equation 6 computes the recall

$$Recall = \frac{1}{|L|} \sum_{l \in L} R(y_l, \hat{y}_l) \tag{6}$$

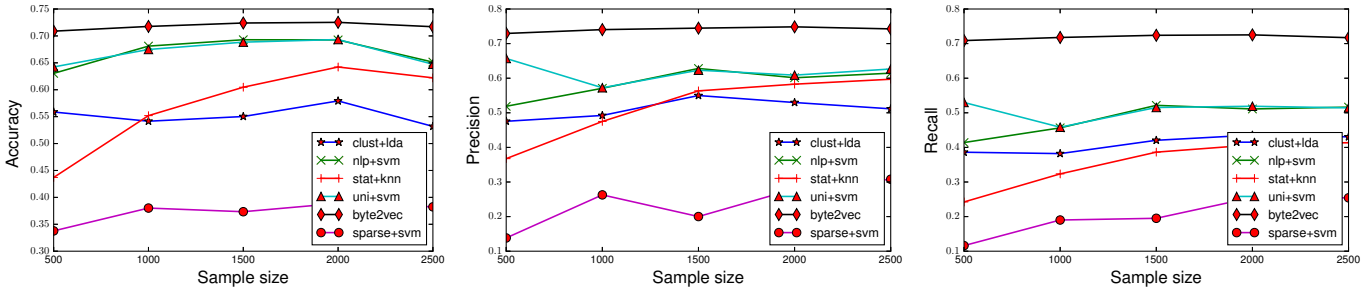
where L denotes set of labels ($|L| = 63$) and $R(y_l, \hat{y}_l) = |y_l \cap \hat{y}_l| / |\hat{y}_l|$. Figure 6c and 7c present recall scores for varying vector lengths for `govdocs1` and `random` datasets. In Figure 6c, our recall scores also conform with the precision and accuracy scores with a maximum recall of 0.73 for vector length of 60 and above. In Figure 7c, however, we observe reduced performance for lower experimental samples.

Considering the accuracy, recall, precision and model generation time with respect to the vector length we decided to use 60 as our empirical vector length. Vector length of 60 achieves very good scores in all performance metrics. Moreover, going beyond 60 does not present a significant amount of gain (Figure 6 and 7).

4.3. Comparative Analysis

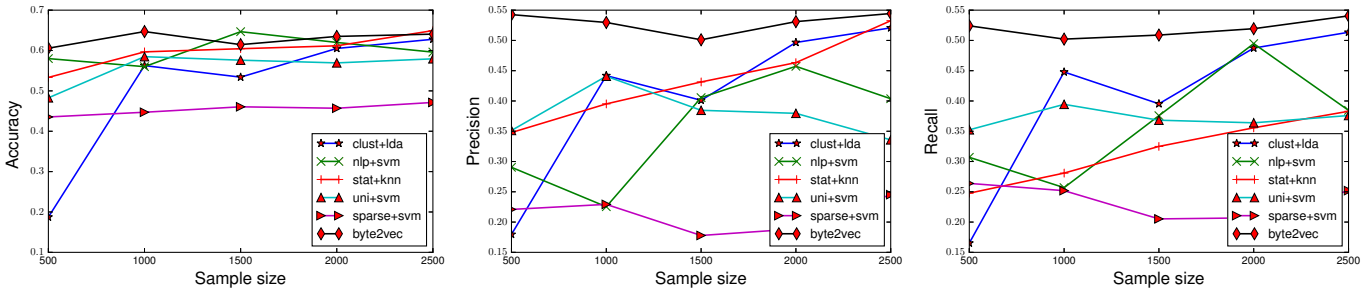
In this part we compare the `Byte2Vec+kNN` model with several works in the literature including Unigram+SVM [7], NCD+kNN [5], Cluster features+LDA [8], Stat+kNN [6], NLP+SVM [4], and SparseCoding+SVM [26]. Note that we represent each work using "features + classifier" pair to avoid any elongated naming convention.

Table 2 and 3 demonstrate average accuracy, precision, and recall scores for all methods with respect to different sample sizes. Note that we show multiple sample sizes to demonstrate the sensitivity of these approaches under varying sample sizes for `govdocs1` and `random` datasets, respectively. The table shows partial performance scores for **NCD+kNN** because the technique takes several days to complete the classification for larger sample sizes. Lastly, Figure 8 and 9 present the plots of the performance scores in Table 2 and 3, respectively, to provide the reader with visual insight.



(a) Accuracy (b) Precision (c) Recall

Figure 8. Summary of (a) accuracy, (b) precision, and (c) recall scores presented in Table 2 for `govdocs1` dataset.



(a) Accuracy (b) Precision (c) Recall

Figure 9. Summary of (a) accuracy, (b) precision, and (c) recall scores presented in Table 3 for `random` dataset.

The first method, **Unigram+SVM**, uses unigram features and SVM classifier [7]. The maximum accuracy and precision achieved through this method is 69% and 62%, respectively on `govdocs1` dataset. On the other hand, we achieved a maximum of 58% accuracy on `random` dataset. Using unigram counts as features is similar to using byte frequency distribution or histogram of bytes. One major drawback of this technique is that different file types can have similar frequency distributions which may cause increased mis-classifications.

The second method, **NCD+*k*NN**, uses normalized compression distance (NCD) and k-nearest neighbor (kNN) classifier [5]. The NCD of two files X and Y is defined as

$$NCD(x,y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \tag{7}$$

where $C(x)$ is the compression distance of file x and $C(xy)$ refers to the compression distance of two combined files or fragments. While implementing this process, we noticed that the calculation of distance measure takes long. The complexity of NCD compression for n samples is $O(n^2)$. Therefore, kNN classifier takes longer to generate k neighbors set. As a result this technique is not suitable for larger datasets. We had to stop our experiment for this technique after running it for more than 72 hours. Hence, we report the results of sample size 500 for this particular technique on `govdocs1` data. The maximum accuracy, precision, and recall achieved through this method is 42%, 40%, and 31%, respectively. However, we were able to run this technique on our smaller `random` dataset that achieved a maximum of 42% accuracy.

The third comparison method, **Cluster features+LDA**, uses several statistical features collected from disk clusters and Fisher classifier [40] for classification. We use linear discriminant analysis (LDA) which is a generalization of Fisher classifier as the classification algorithm for reproducing the method [41]. The authors use unigram (histogram of byte values), entropy, and Kolmogorov complexity as cluster features.

Entropy [42] is used to calculate the lower bound of the number of bits required to encode a string of symbols based on their frequency. The authors use the entropy as a feature due to the randomness of a byte frequency for each 256 values of different file fragments. This feature is also used in Veenman's statistical approach [8]. For a random variable, X with n outcomes $(x_1, \ldots, x_n)$, the Shannon entropy, $H(X)$, is defined as

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \tag{8}$$

where X is the byte-frequency content of a file fragment, n=256, and $p(x_i)$ is the probability of byte x_i.

Given a string x, Kolmogorov complexity [43] refers to the length of the smallest function or program that generates x.

$$C(x) = \min_{p}\{|p| : f(p) = x\} \tag{9}$$

where $C(x)$ denotes complexity, $f(p)$ is the function that regenerates the string x. $C(x)$ computes the information content of string x. The maximum accuracy, precision, and recall achieved through this method on `govdocs1` is 58%, 55%, and 44%, re-

Table 2. Performance comparison of `Byte2Vec` with the state-of-the-art techniques on `govdocs1` dataset. All the results are rounded to two digits after decimal point. >3d indicates the computation did not finish after 3 days.

Method	Sample size	Accuracy	Precision	Recall
Unigram + SVM [7]	500	0.63	0.52	0.41
	1000	0.67	0.57	0.46
	1500	0.69	0.62	0.52
	2000	0.69	0.60	0.52
	2500	0.65	0.63	0.51
NCD + *k*NN [5]	500	0.42	0.40	0.31
	1000	>3d	>3d	>3d
	1500	>3d	>3d	>3d
	2000	>3d	>3d	>3d
	2500	>3d	>3d	>3d
Cluster features + LDA [8]	500	0.56	0.48	0.39
	1000	0.54	0.49	0.38
	1500	0.55	0.55	0.42
	2000	0.58	0.53	0.44
	2500	0.53	0.51	0.43
Stat + *k*NN [6]	500	0.44	0.37	0.24
	1000	0.55	0.48	0.32
	1500	0.60	0.56	0.39
	2000	0.64	0.58	0.40
	2500	0.62	0.60	0.41
NLP + SVM [4]	500	0.63	0.52	0.41
	1000	0.68	0.57	0.46
	1500	0.69	0.62	0.52
	2000	0.69	0.60	0.51
	2500	0.65	0.61	0.52
Sparse coding + SVM [26]	500	0.33	0.14	0.12
	1000	0.38	0.26	0.19
	1500	0.37	0.20	0.20
	2000	0.38	0.28	0.26
	2500	0.38	0.30	0.25
`Byte2Vec` + *k*NN	500	**0.71**	**0.73**	**0.71**
	1000	**0.72**	**0.74**	**0.72**
	1500	**0.72**	**0.74**	**0.72**
	2000	**0.73**	**0.75**	**0.73**
	2500	**0.72**	**0.74**	**0.72**

Table 3. Performance comparison of `Byte2Vec` with the state-of-the-art techniques on `random` dataset. All the results are rounded to two digits after decimal point. The "Sample size" column denotes total sample fragments used for the experiment, instead of denoting sample size for each file type.

Method	Sample size	Accuracy	Precision	Recall
Unigram + SVM [7]	500	0.48	0.35	0.35
	1000	0.58	0.44	0.39
	1500	0.57	0.38	0.37
	2000	0.57	0.38	0.36
	2500	0.58	0.34	0.38
NCD + *k*NN [5]	500	0.42	0.27	0.22
	1000	0.34	0.19	0.25
	1500	0.42	0.16	0.20
	2000	0.38	0.16	0.16
	2500	0.33	0.19	0.18
Cluster features + LDA [8]	500	0.18	0.18	0.17
	1000	0.56	0.44	0.45
	1500	0.53	0.40	0.40
	2000	0.60	0.49	0.49
	2500	0.62	0.52	0.51
Stat + *k*NN [6]	500	0.53	0.35	0.25
	1000	0.60	0.40	0.28
	1500	0.60	0.43	0.32
	2000	0.61	0.46	0.35
	2500	0.64	0.53	0.38
NLP + SVM [4]	500	0.58	0.29	0.30
	1000	0.56	0.22	0.25
	1500	**0.65**	0.40	0.37
	2000	0.62	0.45	0.49
	2500	0.60	0.40	0.38
Sparse coding + SVM [26]	500	0.44	0.22	0.26
	1000	0.45	0.23	0.25
	1500	0.46	0.18	0.21
	2000	0.46	0.19	0.21
	2500	0.47	0.25	0.25
`Byte2Vec` + *k*NN	500	**0.61**	**0.54**	**0.52**
	1000	**0.65**	**0.53**	**0.50**
	1500	0.62	**0.50**	**0.51**
	2000	**0.63**	**0.53**	**0.52**
	2500	**0.64**	**0.54**	**0.54**

spectively. On the other hand, the method achieved maximum of 62% accuracy on `random` dataset.

The fourth method, **Stat+*k*NN** uses a different set of statistical measures including Shannon entropy, Chi square, Hamming weight, and arithmetic mean and *k*NN classifier [6]. The *k*NN algorithm is used with Euclidean distance and its neighbor selection parameter, *k*, is set to 10.

Mean byte value refers to the arithmetic mean of the bytes in a fragment. They use Chi square (χ^2) Goodness of fit measure (Equation 10) to compare the byte distributions with a uniform random distribution.

$$\chi^2 = \sum_{i=0}^{n} \frac{(\text{observed - expected})^2}{\text{expected}} \tag{10}$$

Hamming weight is defined by the fraction of the total number of ones divided by the total number of bits [6]. We converted each fragment to a byte array of 4096 bytes and counted the total number of one bits and divided that by 4096×8=32768. The maximum accuracy, precision, and recall achieved through this method on `govdocs1` is 64%, 60%, and 41%, respectively. On the other hand, the method achieved maximum of 64% accuracy on `random` dataset.

The fifth comparison method, **NLP+SVM**, uses several statistical and natural language processing (NLP) features and SVM classifier [4]. The statistical feature set includes histogram of byte values (i.e. unigram counts), histogram of bigram counts, Shannon entropy of the bigram counts, Hamming weight, mean byte value, and compressed length of file fragments. The NLP features include average distance between consecutive bytes and longest contiguous streak of repeating bytes. Average distance between consecutive bytes within a fragment is defined using Equation 11 in [4].

$$C = \sum_{i=0}^{4094} \frac{|byte_i - byte_{i+1}|}{4096} \tag{11}$$

Longest contiguous streak of repeating bytes represents a sequence of bytes that have the highest length within a fragment. The maximum accuracy, precision, and recall achieved through this method on `govdocs1` is 69%, 62%, and 52%, respectively. On the other hand, the method achieved maximum of 65% accuracy with 40% precision on `random` dataset. Note that our approach also achieved 62% accuracy in that particular experimental setting (sample size of 1500 in total) with 57% precision and 54% recall.

The sixth and final method, **SparseCoding+SVM**, uses sparse coding of different dimensions and uni/bigram as feature set for the classification purpose. The method only uses 18 file types from `govdocs1` dataset. The method selects 64 randomly sampled n-grams from each file fragment to use during dictionary learning phase. We replicated the method using *scikit-learn* dictionary learning library with all the file types from `govdocs1` and found differences in empirical results. In addition, the study shows that Office Open XML (OOXML) or Microsoft Open XML (MOX) file types such as *docx, pptx,* and *xlsx* files exhibit very poor performance. However, we observe better performances in OOXML file types in our proposed approach compared to **SparseCoding+SVM**. The biggest drawback of this approach is the amount of time it takes to extract the feature set from a given file fragment due to expensive computation during the dictionary learning. The maximum accuracy, precision, and recall achieved through this method on `govdocs1` is 38%, 30%, and 26%, respectively. On the other hand, the method achieved maximum of 47% accuracy with 25% precision and 26% recall on `random` dataset.

`Byte2Vec` generates a corpus model for all file types with respect to different vector lengths and these corpus models can be used with different classifiers. We used *k*NN as default classifier in our experiments. We explain the effectiveness of `Byte2Vec` using sample confusion matrices in Table 4 and Figure 10 that show evaluation from sample of 2500 fragments of each file types and vector lengths of 60. Note that, we share all confusion matrices in our code repository for comprehensive comparison as well.

4.4. Discussions

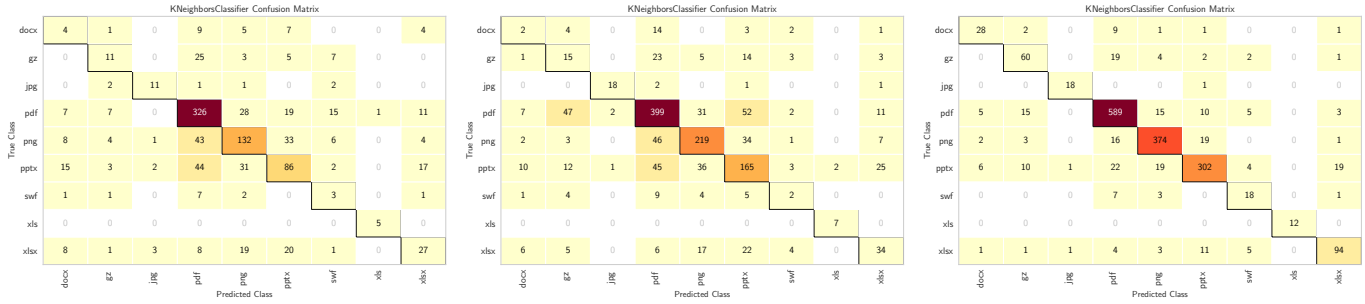
In this part, we discuss per file type classification performance with respect to the internal file structure. We present confusion matrices from both datasets to explain the misclassification of individual file types using Table 4 and Figure 10 for `govdocs1` and `random` datasets, respectively.

From the first confusion matrix presented in Table 4, we notice that fragments with distinguishable patterns such as bitmap tags for *bmp*, the commas in *csv*, and tags in *xml* are the most easily classified file types. Overall, we notice that *bmp, csv, eps, gif, html, xml, xls,* and *txt* fragments are classified with more than 85% accuracy. On the contrary, we observe moderate prediction accuracy for *doc, docx, dwf, jpg, kmz, png, ppt,* and *zip* file fragments. The content of a *doc* file is stored either using plain text or binary format. The confusion matrix of the table shows that most of the *doc* files are misclassified as other container file types such as *png, pdf,* and *ppt* in the range between 3% and 5%. We also observe similar behavior from *docx* file fragments with respect to misclassification of file types. *docx* files are encoded in *xml* format consisting of a *zip* archive file containing *xml* and binaries. *docx* fragments are mismatched with *doc, dwf, gz, kmz, pdf, pps, ppt* and *zip* in the range between 4% and 8%. The maximum mismatch occurred with *kmz* file type which is an *xml* notation for expressing geographic annotation and visualization within Internet-based 2-D maps and 3-D Earth browsers. Design Web Format (*dwf*) is a secure and highly compressed file format developed by Autodesk Inc. [1] for efficient and fast distribution of rich media data such as 3-D CAD (Computer Aided Design) drawings. As a result, the misclassification of *dwf* files mostly occur with compressed file formats such as *gz, pdf,* and *zip* and media files such as *kmz, png,* and *swf*. The misclassification rate for *dwf* fragments ranges between 4% (*pdf*) and 16% (*kmz*). The other file types such as *jpg, kmz, png, ppt,* and *zip* exhibit less than 50% accuracy due to mismatch with each other in the range

[1] https://www.autodesk.com/

Table 4. Confusion matrix of *k*NN classifier on `govdocs1` dataset. The height of the table has been increased to augment the visual clarity for the readers.

data_type	bmp	csv	dbase3	doc	docx	dwf	eps	f	fits	gif	gz	html	jpg	kml	kmz	log	pdf	png	pps	ppt	pptx	ps	rtf	sgml	sql	squeak	swf	text	tmp	txt	unk	wp	xls	xml	zip
bmp	**0.97**	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
csv	0.0	**0.99**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dbase3	0.0	0.0	**0.93**	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0
doc	0.01	0.0	0.0	**0.39**	0.03	0.05	0.0	0.0	0.0	0.01	0.04	0.0	0.04	0.0	0.06	0.0	0.03	0.03	0.05	0.05	0.01	0.0	0.01	0.03	0.0	0.0	0.03	0.01	0.02	0.01	0.0	0.03	0.02	0.0	0.05
docx	0.0	0.0	0.0	0.04	**0.47**	0.04	0.0	0.0	0.0	0.01	0.04	0.0	0.03	0.0	0.08	0.0	0.04	0.04	0.04	0.04	0.02	0.0	0.0	0.02	0.0	0.0	0.03	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.06
dwf	0.0	0.0	0.0	0.06	0.05	**0.25**	0.0	0.0	0.0	0.01	0.06	0.0	0.02	0.0	0.16	0.0	0.04	0.06	0.02	0.06	0.02	0.0	0.0	0.04	0.0	0.0	0.07	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.06
eps	0.0	0.0	0.0	0.0	0.0	0.0	**0.98**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
f	0.0	0.0	0.02	0.0	0.0	0.0	0.0	**0.88**	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.01	0.03	0.0	0.0	0.0	0.0
fits	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	**0.93**	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gif	0.0	0.0	0.0	0.01	0.01	0.02	0.0	0.0	0.01	**0.74**	0.01	0.0	0.01	0.0	0.01	0.0	0.04	0.01	0.01	0.02	0.03	0.0	0.0	0.01	0.0	0.0	0.02	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01
gz	0.0	0.0	0.0	0.05	0.03	0.06	0.0	0.0	0.0	0.01	**0.54**	0.0	0.02	0.0	0.05	0.0	0.02	0.05	0.04	0.02	0.02	0.0	0.0	0.02	0.0	0.0	0.02	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.04
html	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.01	0.0	0.01	0.0	**0.89**	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.01	0.0
jpg	0.0	0.0	0.0	0.05	0.03	0.04	0.0	0.0	0.0	0.03	0.03	0.0	**0.43**	0.0	0.04	0.0	0.03	0.03	0.04	0.08	0.02	0.0	0.0	0.03	0.0	0.0	0.03	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.08
kml	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.99**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
kmz	0.0	0.0	0.0	0.03	0.05	0.07	0.0	0.0	0.0	0.01	0.04	0.0	0.01	0.0	**0.49**	0.0	0.04	0.05	0.02	0.03	0.01	0.0	0.0	0.02	0.0	0.0	0.08	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.04
log	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	**0.92**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.05	0.01	0.0	0.0	0.0	0.0	0.0
pdf	0.0	0.0	0.01	0.02	0.03	0.06	0.0	0.0	0.0	0.02	0.03	0.01	0.03	0.0	0.05	0.0	**0.52**	0.03	0.02	0.03	0.01	0.0	0.0	0.02	0.0	0.0	0.03	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.04
png	0.0	0.0	0.0	0.04	0.05	0.07	0.0	0.0	0.01	0.01	0.05	0.0	0.02	0.0	0.1	0.0	0.06	**0.32**	0.02	0.05	0.03	0.0	0.0	0.04	0.0	0.0	0.07	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.05
pps	0.0	0.0	0.0	0.04	0.02	0.02	0.0	0.0	0.01	0.0	0.02	0.0	0.03	0.0	0.03	0.0	0.01	0.03	**0.64**	0.04	0.02	0.0	0.0	0.01	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.04
ppt	0.0	0.0	0.0	0.05	0.04	0.04	0.0	0.0	0.01	0.01	0.04	0.0	0.07	0.0	0.08	0.0	0.04	0.04	0.08	**0.30**	0.03	0.0	0.0	0.03	0.0	0.0	0.04	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.05
pptx	0.0	0.0	0.0	0.01	0.03	0.03	0.0	0.0	0.0	0.02	0.02	0.0	0.03	0.0	0.04	0.0	0.01	0.04	0.03	0.02	**0.62**	0.0	0.0	0.03	0.0	0.0	0.03	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02
ps	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.96**	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rtf	0.0	0.0	0.0	0.01	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.96**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sgml	0.0	0.0	0.0	0.03	0.03	0.03	0.0	0.0	0.0	0.0	0.01	0.0	0.01	0.0	0.06	0.01	0.02	0.01	0.01	0.04	0.01	0.0	0.0	**0.63**	0.0	0.0	0.02	0.01	0.0	0.01	0.01	0.0	0.0	0.0	0.03
sql	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.98**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
squeak	0.01	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.93**	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0
swf	0.0	0.0	0.0	0.02	0.03	0.06	0.0	0.0	0.01	0.01	0.02	0.0	0.01	0.0	0.1	0.0	0.03	0.03	0.01	0.03	0.01	0.0	0.0	0.02	0.0	0.0	**0.59**	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.02
text	0.0	0.0	0.0	0.01	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.91**	0.0	0.0	0.01	0.01	0.0	0.0	0.0
tmp	0.0	0.0	0.0	0.02	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.06	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.86**	0.0	0.01	0.01	0.0	0.0	0.0
txt	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	**0.90**	0.03	0.0	0.0	0.0	0.0
unk	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.02	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.03	**0.86**	0.01	0.0	0.0	0.0
wp	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.01	**0.90**	0.02	0.0	0.0
xls	0.0	0.0	0.0	0.04	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.01	0.01	0.0	0.0	0.0	0.01	0.0	0.0	0.01	0.0	0.01	0.0	0.01	0.02	**0.86**	0.0	0.0
xml	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.96**	0.0
zip	0.0	0.0	0.0	0.05	0.03	0.06	0.0	0.0	0.0	0.01	0.04	0.0	0.06	0.0	0.07	0.0	0.03	0.03	0.02	0.05	0.03	0.0	0.0	0.02	0.0	0.0	0.04	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.46**



(a) 1500 (b) 2000 (c) 2500

Figure 10. Sample confusion matrix from `random` dataset experiment using vector lengths of 60 on sample data size of (a) 1500, (b) 2000, and (c) 2500.

from 1% to 8%. All these file types have common structures such as high compression, *xml* encoding, and media embeddings. Therefore, they are moderately mixed with each other during the classification process. Another important aspect is that we included partial fragments in the test data which can increase the misclassification rate. A typical scenario for partial data case is that the randomly included bytes can dominate the overall bytes in the newly constructed partial fragment. For instance, a partial fragment of *pdf* can contain majority of its bytes from a random *zip* fragment.

Figure 10 demonstrates confusion matrices on a smaller dataset that is collected from random web search using file types as keywords. We present three sample results that used vector length 60 during feature generation process using `Byte2Vec` model. We observe higher misclassification of OOXML (*docx, pptx, xlsx*) and high-entropy (*gz*) fragments with small data sample. Although the errors are widely spread over all file types for high-entropy fragments, we notice significant bias toward the *gz* file type with file types such as *pdf* and *png*. However, we notice significant improvement in Figure 10c on both types of fragments with the increase in sample size. Therefore, the results from Figure 10 indicate that experimenting on larger training data results in more favorable classification performance.

In general, we notice that the most significant improvements when using the `Byte2Vec` model as feature generator are for file types with common byte sequences such as *csv, text, xls,* and *xml*. For high-entropy file types such as *gz*, we notice that they are mixed with each other at a moderate rate. Regarding container file types such as *pdf* and *pptx*, we observe that they are misclassified with image and compressed files. For instance, a *pdf* file is very likely to be embedded with different image formats such as *jpg, png,* and *bmp*. Regarding OOXML file types, we observe that our feature generation method performs very well with larger training data. All things considered, the results indicate that our approach is able to capture the inherent file structure on a broader scale, due to the transformation of original features into a vector space, and improves the overall classification performance.

5. Conclusions

File carving is the process of recovering files on a storage media in part or in whole without any file system information. An important problem in file carving is the identification of fragment types. Many fragment classification studies in the literature employ inflexible or indiscernible feature selection methods such as different statistics of byte frequency distributions. Moreover, assessing the strengths and weaknesses of some approaches is difficult because they do not generalize across different file types.

In this paper, we propose a novel feature generation model, `Byte2Vec`, to identify file fragment types using byte embeddings, i.e., vector representations. The proposed model extends the *word2vec* and *doc2vec* models to bytes and fragments, respectively. Unlike some of the previous methods, `Byte2Vec` works for different block sizes and supports fragments of any type. Moreover, the addition of a new file type does not require the reconstruction of any existing corpus models. We used `Byte2Vec` for feature extraction from fragments and k Nearest Neighbors for classification.

Our experimental results show that `Byte2Vec`+k`NN` reaches an accuracy rate of 72% along with 74% precision and 72% recall. Compared to the other feature extraction techniques such as n-gram, byte distributions, byte statistics, byte distances, and sparse dictionary learning features along with different classifiers, our approach achieves an absolute improvement of 3% and 12% in accuracy and precision, respectively.

Above all, we demonstrate that `Byte2Vec` can be posed as a feature extraction technique and that classifying fragments without looking at the inherent structure can be made tractable and accurate. Our proposed formulation is general and offers a potentially different mode of thinking about file fragment classification problem in digital forensics and security area. It is also important to note that `Byte2Vec` model is highly transferable to other domains.

Appendix A. Govdocs1 data statistics

We present the frequency distribution of all files and fragments with respect to file types from all the files collected from digital corpora in Table A.5.

Table A.5. Frequency distribution of all files and fragments with respect to file types from all the files collected from digital corpora.

File type	File Count	Fragment Count
.123	2	56
.bin	1	1
.bmp	72	7878
.chp	2	10
.csv	18360	856331
.data	3	218
.dbase3	2601	6307
.doc	76616	7528792
.docx	163	8340
.dwf	299	10850
.eps	5191	722399
.exported	3	42
.f	602	11486
.fits	182	83395
.fm	25	850
.g3	2	64
.gif	36302	756618
.gls	60	94
.gz	13725	2215529
.hlp	659	1495
.html	214568	3286673
.icns	1	1
.ileaf	4	210
.java	292	1970
.jpg	109233	9192380
.js	2	6
.kml	993	39363
.kmz	943	69242
.lnk	2	2
.log	9976	871937
.mac	2	2
.odp	2	299
.pdf	231232	33230061
.png	4125	277581
.pps	1619	925252
.ppt	49702	31236286
.pptx	215	143597
.ps	22015	7040370
.pst	1	3
.pub	55	227
.py	1	61
.rtf	1125	117175
.sgml	62	5555
.sql	462	30908
.squeak	1	3170
.swf	3476	476535
.sys	7	7
.tex	163	1390
.text	839	190140
.tmp	180	3683
.troff	110	1067
.ttf	10	198
.txt	78285	12240966
.unk	5186	374462
.vrml	1	83
.wk1	7	816
.wk3	1	29
.wp	364	10835
.xbm	8	78
.xls	62635	7248727
.xlsx	37	1635
.xml	33458	2130210
.zip	10	3945

References

[1] V. L. Thing, T.-W. Chua, M.-L. Cheong, Design of a digital forensics evidence reconstruction system for complex and obscure fragmented file carving, in: Computational Intelligence and Security (CIS), 2011 Seventh International Conference on, IEEE, 2011, pp. 793–797.

[2] A. Hadi, et al., Reviewing and evaluating existing file carving techniques for jpeg files, in: Cybersecurity and Cyberforensics Conference (CCC), 2016, IEEE, 2016, pp. 55–59.

[3] G. G. Richard III, V. Roussev, Next-generation digital forensics, Communications of the ACM 49 (2) (2006) 76–80.

[4] S. Fitzgerald, G. Mathews, C. Morris, O. Zhulyn, Using nlp techniques for file fragment classification, Digital Investigation 9 (2012) S44–S49.

[5] S. Axelsson, The normalised compression distance as a file fragment classifier, digital investigation 7 (2010) S24–S31.

[6] G. Conti, S. Bratus, A. Shubina, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg, R. Perez-Alemany, Automated mapping of large binary objects using primitive fragment type classification, digital investigation 7 (2010) S3–S12.

[7] Q. Li, A. Ong, P. Suganthan, V. Thing, A novel support vector machine approach to high entropy data fragment classification, in: Proc. South African Information Security Multi-Conf.(SAISMC), 2011, pp. 236–247.

[8] C. J. Veenman, Statistical disk cluster classification for file carving, in: Information Assurance and Security, 2007. IAS 2007. Third International Symposium on, IEEE, 2007, pp. 393–398.

[9] P. Penrose, R. Macfarlane, W. J. Buchanan, Approaches to the classification of high entropy file fragments, Digital Investigation 10 (4) (2013) 372–384.

[10] W. Qiu, R. Zhu, J. Guo, X. Tang, B. Liu, Z. Huang, A new approach to multimedia files carving, in: Bioinformatics and Bioengineering (BIBE), 2014 IEEE International Conference on, IEEE, 2014, pp. 105–110.

[11] E. Uzun, H. T. Sencar, Carving orphaned jpeg file fragments, IEEE Transactions on Information Forensics and Security 10 (8) (2015) 1549–1563.

[12] K. Nguyen, D. Tran, W. Ma, D. Sharma, A proposed approach to compound file fragment identification, in: International Conference on Network and System Security, Springer, 2014, pp. 493–500.

[13] G. Conti, S. Bratus, A. Shubina, A. Lichtenberg, R. Ragsdale, R. Perez-Alemany, B. Sangster, M. Supan, A visual study of primitive binary fragment types, White Paper, Black Hat USA.

[14] W.-J. Li, K. Wang, S. J. Stolfo, B. Herzog, Fileprints: Identifying file types by n-gram analysis, in: Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC, IEEE, 2005, pp. 64–71.

[15] W. C. Calhoun, D. Coles, Predicting the types of file fragments, digital investigation 5 (2008) S14–S20.

[16] M. Karresand, N. Shahmehri, Oscar—file type identification of binary data in disk clusters and ram pages, Security and privacy in dynamic environments (2006) 413–424.

[17] M. Karresand, N. Shahmehri, File type identification of data fragments by their binary structure, in: Proceedings of the IEEE Information Assurance Workshop, 2006, pp. 140–147.

[18] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781.

[19] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in neural information processing systems, 2013, pp. 3111–3119.

[20] M. McDaniel, M. H. Heydari, Content based file type detection algorithms, in: System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on, IEEE, 2003, pp. 10–pp.

[21] V. Roussev, C. Quates, File fragment encoding classification—an empirical approach, Digital Investigation 10 (2013) S69–S77.

[22] S. Gopal, Y. Yang, K. Salomatin, J. Carbonell, Statistical learning for file-type identification, in: Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on, Vol. 1, IEEE, 2011, pp. 68–73.

[23] V. Roussev, S. L. Garfinkel, File fragment classification-the case for specialized approaches, in: Systematic Approaches to Digital Forensic Engineering, 2009. SADFE'09. Fourth International IEEE Workshop on, IEEE, 2009, pp. 3–14.

[24] T. Xu, M. Xu, Y. Ren, J. Xu, H. Zhang, N. Zheng, A file fragment classification method based on grayscale image., JCP 9 (8) (2014) 1863–1870.

[25] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, A. Y. Wu, An efficient k-means clustering algorithm: Analysis and implementation, Pattern Analysis and Machine Intelligence, IEEE Transactions on 24 (7) (2002) 881–892.

[26] F. Wang, T.-T. Quach, J. Wheeler, J. B. Aimone, C. D. James, Sparse coding for n-gram feature extraction and training for file fragment classification, IEEE Transactions on Information Forensics and Security 13 (10) (2018) 2553–2562.

[27] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31st International Conference on Machine Learning (ICML-14), 2014, pp. 1188–1196.

[28] S. Poria, E. Cambria, A. Gelbukh, Deep convolutional neural network textual features and multiple kernel learning for utterance-level multimodal sentiment analysis, in: Proceedings of the 2015 conference on empirical methods in natural language processing, 2015, pp. 2539–2544.

[29] A. Severyn, A. Moschitti, Twitter sentiment analysis with deep convolutional neural networks, in: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 2015, pp. 959–962.

[30] C. dos Santos, M. Gatti, Deep convolutional neural networks for sentiment analysis of short texts, in: Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, 2014, pp. 69–78.

[31] W. X. Zhao, S. Li, Y. He, E. Y. Chang, J.-R. Wen, X. Li, Connecting social media to e-commerce: Cold-start product recommendation using microblogging information, IEEE Transactions on Knowledge and Data Engineering 28 (5) (2016) 1147–1159.

[32] F. Vasile, E. Smirnova, A. Conneau, Meta-prod2vec: Product embeddings using side-information for recommendation, in: Proceedings of the 10th ACM Conference on Recommender Systems, ACM, 2016, pp. 225–232.

[33] M. E. Haque, M. E. Tozal, A. Islam, Helpfulness prediction of online product reviews, in: Proceedings of the 2018 ACM Symposium on Document Engineering, ACM, 2018.

[34] T. Mikolov, W.-t. Yih, G. Zweig, Linguistic regularities in continuous space word representations., in: hlt-Naacl, Vol. 13, 2013, pp. 746–751.

[35] Digital Corpora, Govdocs1, http://downloads.digitalcorpora.org/corpora/files/govdocs1/.

[36] S. Garfinkel, P. Farrell, V. Roussev, G. Dinolt, Bringing science to digital forensics with standardized forensic corpora, digital investigation 6 (2009) S2–S11.

[37] C. Grajeda, F. Breitinger, I. Baggili, Availability of datasets for digital forensics–and what is missing, Digital Investigation 22 (2017) S94–S105.

[38] X.-Y. Liu, J. Wu, Z.-H. Zhou, Exploratory undersampling for class-imbalance learning, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 39 (2) (2009) 539–550.

[39] Q. Li, A. Y. Ong, P. N. Suganthan, V. L. Thing, A novel support vector machine approach to high entropy data fragment classification., in: SAISMC, 2010, pp. 236–247.

[40] R. O. Duda, P. E. Hart, D. G. Stork, Pattern classification, John Wiley & Sons, 2012.

[41] M. Welling, Fisher linear discriminant analysis, Department of Computer Science, University of Toronto 3 (1).

[42] C. E. Shannon, W. Weaver, A mathematical theory of communication (1948).

[43] O. Watanabe, Kolmogorov complexity and computational complexity, Springer, 1992.