SPARSE DESCRIPTORS FOR WHOLE GRAPH EMBEDDING AND DICTIONARY BASED FEATURE RANKING

by

Liyanpathige Kaveen Gayasara Liyanage

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 2024

DEDICATION

I dedicate this work to my mother and my family for their unwavering support and motivation.

ACKNOWLEDGEMENTS

I sincerely acknowledge the guidance, mentorship, and encouragement provided by Prof. Bradley Whitaker. I would also like to acknowledge Reese Pearsall for co-developing the experiment setup and Bryan Portillo for conducting experiments. Additionally, I would also like to acknowledge the preliminary work done by Veronika Strnadova-Neeley, Daniel Laden, David Opitz, Andrew Rippy, and Shayla Sharma. I also would like to thank Dr. Derek Reimanis and Prof. Clemente Izurieta for their guidance and supervision. I extend my gratitude to Prof. Reinhold and Prof. Snider for their advice and for being on the committee.

I am grateful to Hoplite Industries Inc. for providing malware files. Part of this research was conducted with the U.S. Department of Homeland Security (DHS) Science and Technology Directorate (S&T) under contract $70RSAT22CB0000005$. Any opinions contained herein are those of the authors and do not necessarily reflect those of DHS S&T.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

Table Page

LIST OF TABLES – CONTINUED

Table Page

LIST OF TABLES – CONTINUED

Table Page

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

Figure Page

ABSTRACT

Graph representation has gained wide popularity as a data representation method in many applications. Unfortunately, most data processing techniques cannot be applied directly to a graph structure. Therefore, graph embedding methods are frequently used to convert graphs to vectors. While such methods are essential in standard data processing pipelines, they often result in complicated, nonlinear, and high-dimensional mappings.

The goal of this dissertation is to utilize sparse dictionary learning techniques in the context of graph embedding. In contrast to traditional graph embedding methods, sparse representations are linear by design. This linearity also leads to intuition, since the building blocks of a sparse dictionary are directly related to the input space. Despite the potential advantages of sparse processing and the ubiquitousness of sparsity in other signal processing domains, its applications in graph embedding are not well studied.

This dissertation consists of three main tasks. First, a novel sparse graph descriptor algorithm is presented, inspired by the *Graph2Vec* graph embedding algorithm. Second, sparse representation-based feature ranking metrics are deployed to identify important subtree structures of the graphs that can be used to define a dictionary. The developed embedding algorithm and feature-ranking metrics are compared to existing graph embedding methods and feature-ranking algorithms on several typical benchmark graph datasets. Finally, these sparse representation-based techniques are applied to control flow graphs of binary files to detect malware, showing the utility of the developed algorithms.

1

INTRODUCTION

In recent years, the demand for representing complex data structures has risen. Graphs are considered a suitable solution to represent complex, relational, and hierarchical data structures [1]. Many applications use graphs as an intuitive way to organize data, such as social networks, e-commerce, and power grids [2]. However, many data processing techniques require that the inputs be represented as vectors [1]. To meet this requirement, graph representation learning or graph embedding is used to convert a graph model into a vector representation of low-dimensional space. Because of the challenging nature of this task, most graph embedding methods are not very interpretable [3].

Sparse representation is widely used in image processing domains and is known to capture latent properties of datasets and preserve these properties through linear relationships [4]. However, the sparse representation methods for whole graphs embedding are rare and limited in usage [5, 6]. Some limitations are that graphs have to be directed, have fixed topology, and cannot contain node features. This dissertation presents a novel generalized whole graph embedding method based on sparse representation to address these limitations.

Motivation

As part of research funded through the Department of Homeland Security (DHS) under the project "Cyber QR Ops: Improving the Quality and Resiliency of Critical Computing Infrastructure," graph analysis was studied in the cyber-security domain. The research focuses on exploring the viability of machine learning (ML) as an effective tool in identifying malware through graph analysis of control flow graphs (CFGs).

Several typical graph embedding methods were considered for representing the CFGs. Although they provided acceptable performance, they could not be used to analyze the input features of the graphs due to the non-linearity and complex relationship between input and embedded spaces. Part of the project is to identify sub-tree structures that lead to the separation of data classes. Hence, the possibility of utilizing sparse representation methods for graph embedding was considered. With malware detection as the main application, the advantages of having a linear and more intuitive embedding method were considered.

Goals, Objectives, and Tasks

The overall goal of this dissertation is to *incorporate the functional benefits of sparse representation methods for whole graph embedding.* The context of the goal is applications in malware detection; specifically, analyzing CFGs generated from binary files for the detection of malware.

In preliminary work, a sparse representation-based graph embedding method was introduced which combines Wiesfeiler-Lehman (WL) sub-tree kernels and K-means Singular Value Decomposition (KSVD). The new method is known as *WL+KSVD* [7]. Also, dictionary-based feature ranking metrics were introduced for analyzing input features namely *Dictionay Mapping* and *Dictionary Utilization* [8]. I am presenting the application of these two novel techniques for malware detection under two objectives:

1. Apply the sparse representation-based whole graph embedding method *WL+KSVD* [7] for embedding CFGs as sparse vectors for machine learning tasks.

2. Apply novel dictionary based feature ranking metrics, namely *dictionary mapping*, and *dictionary utilization* [8], for identifying important sub-tree patterns of Malware CFGs.

The outcomes of this dissertation are a framework and an analysis of CFG sub-tree structures. Such a framework will help identify problematic, suspicious, and vulnerable

binary file patterns. Another outcome would be a publicly available *Python* package consisting of a *WL+KSVD* graph embedding method and dictionary-based feature ranking metrics. The following tasks are conducted to achieve the above objectives and outcomes.

1. Develop a sub-tree pattern-based sparse graph embedding method
2. Create a framework for identifying important sub-tree patterns
3. Apply the developed methods to malware detection

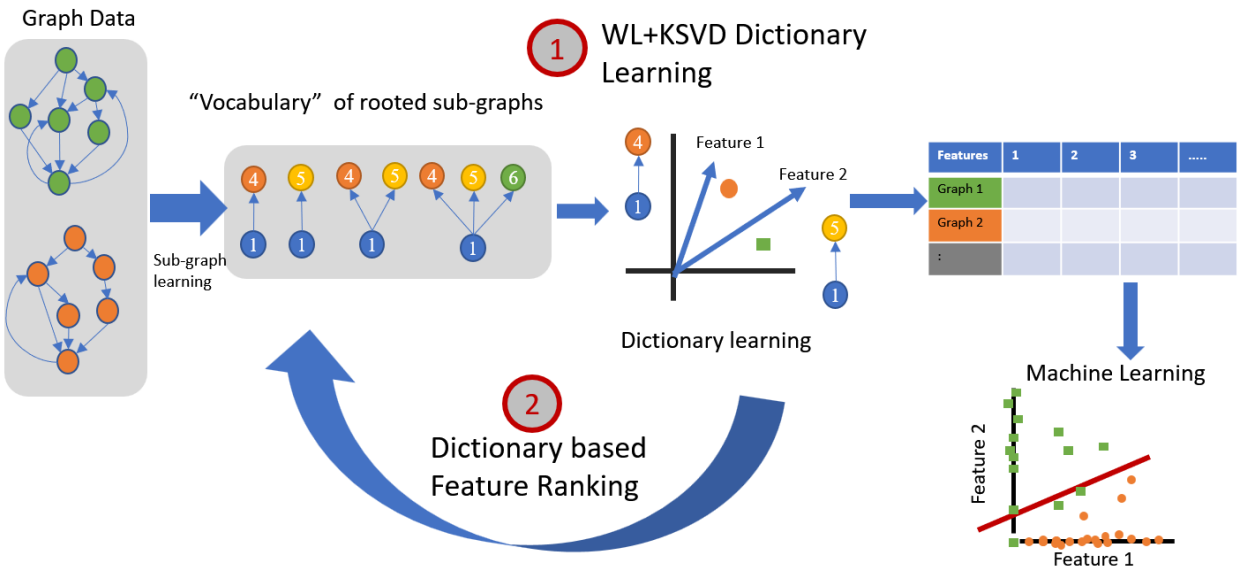


Figure 1.1: The objectives overview

The rest of the dissertation is organized as follows. Chapter 2 provides background information about graph embedding techniques, sparse representation, feature ranking, and malware. Chapter 3 discusses the novel Sparse Graph Descriptor method and the experiments. Chapter 4 presents the dictionary-based feature ranking metrics and its experiments. Chapter 5 discusses the work on analyzing CFGs through graph embedding unsupervised clustering and supervised classification. Finally, Chapter 6 provides a summary of the dissertation with discussion.

4

BACKGROUND

This chapter provides background information about graph embedding, Sparse Representation, Feature ranking, and Malware analysis.

Graphs Embedding

Graph data is usually highly dimensional and defined in a non-euclidean form. Hence, typical processing methods defined on Euclidean spaces cannot be used on graph data. Graph embedding methods convert the graph data into a vector representation while trying to preserve original graph properties [3, 9]. Figure 2.1 from *Cui et al.* [10] provides an overview of the taxonomy used with graph embedding methods.

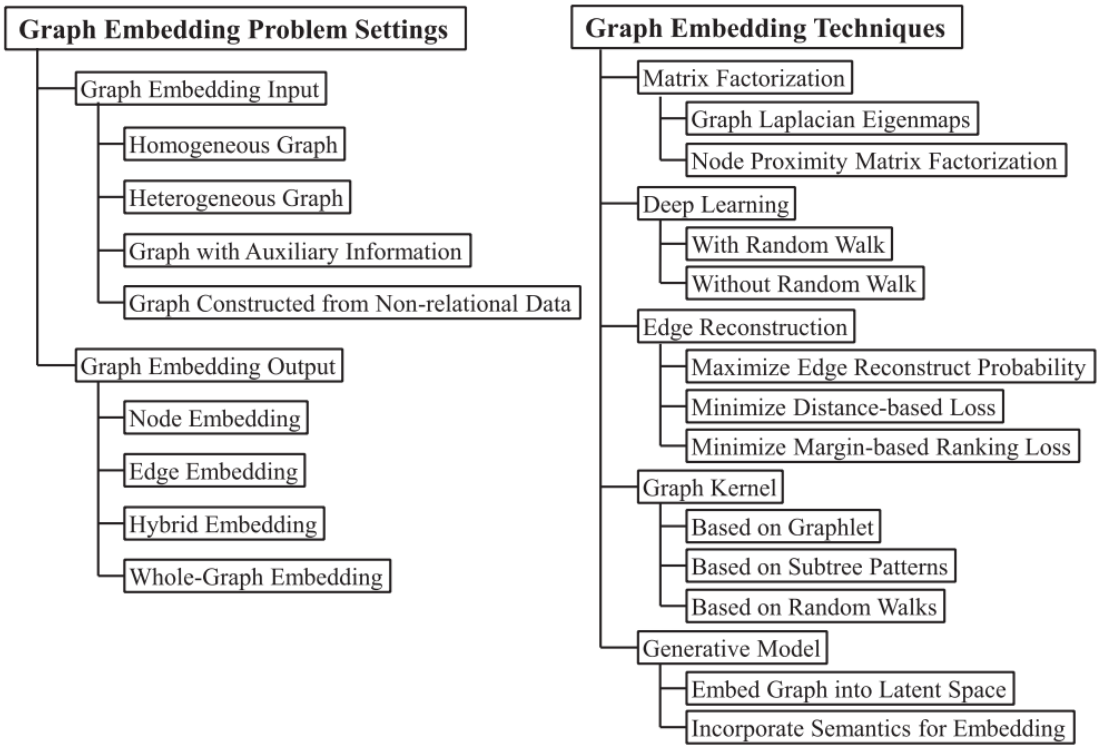


Figure 2.1: Graph embedding taxonomy (from Figure 4 of Cui *et al.*) [10]

Graph embedding methods can be classified as node embedding, edge embedding, hybrid embedding, and whole graph embedding. In literature, a distinction is made between graph representation learning and graph embedding [9, 1], where graph representation does not require the final vector to be low-dimensional. Figure 2.2 from *Cai et al.* [9] provides a visualization of the graph embedding output types.

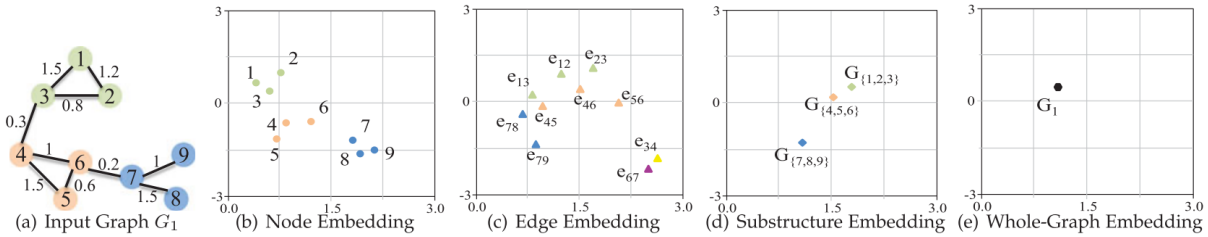


Figure 2.2: Graph embedding output types (from Figure 1 of Cai *et al.*) [9]

In this dissertation, we focus on whole graph embedding, where each entire graph is represented as a vector [11]. The vector representation can be used to compare graph similarity for important tasks, including classification and clustering. The main challenges in whole graph embedding are how to capture the properties of a whole graph and how to make a trade-off between expressiveness and efficiency [9]. Several methods have been proposed for whole graph embedding, including matrix factorization, deep learning, edge reconstruction, graph kernel, and generative models [9, 11].

Graph2Vec Graph Embedding

Several graph embedding methods can be used for vector representation purposes. In this work, we will be utilizing *Graph2Vec* [12] method. Typical implementations of the *Graph2Vec* are shown in figure 2.3, where the graphs are first converted into a word document using the Weisfeiler-Lehman graph kernel [13] and then uses *Doc2Vec* [14] to create a fixed length vector representation of the graph embedding.

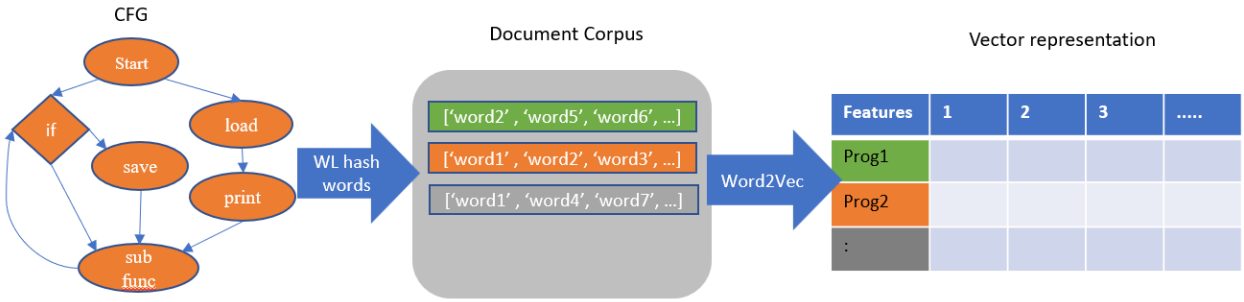


Figure 2.3: Graph2Vec implementation overview with WL hash words and Doc2Vec algorithms.

Weisfeiler-Lehman (WL) graph hash: The WL hash algorithm [13] is a simple algorithm to identify common sub-tree patterns in a set of graphs. The algorithm initially relabels the graph nodes according to the number of its neighbors. Then iteratively perform a breadth-first-search for neighboring nodes. At each iteration, the base node is relabeled by adding the information about its neighbors' labels. In each iteration, the appended relabel is converted to a fixed-length unique hash value. These techniques produce deterministic and unique relabeling for each node depending on the number of iterations.

It is possible to use different graph kernels for identifying sub-tree patterns. A recent extensive survey has listed out different graph kernels and their comparison [15]. In the survey, the authors provide a helpful "practitioner's guide", a recommendation pipeline for selecting a graph kernel according to the specific application. For the malware application considered here, the use of WL sub-tree kernel is justified through this pipeline. Since program CFGs do not contain edge information and typically form large graphs, WL kernels can be considered. For our study, we only consider the local structures of the graphs. Hence, WL sub-tree kernel is justified. The motivation for focusing on local structure only is to observe if any coding patterns would emerge as significant features for different kinds of malware.

Doc2Vec: *Doc2Vec* [14] is a technique used in the domain of natural language processing (NLP). Since the WL hash algorithm uniquely relabels nodes with information about their neighbors, the list of new node labels can be considered as a representation of the graph. If the labels are considered as words, each graph can be considered a *document* with a set of words. First, a vocabulary is constructed by using the most used words. Then *Word2Vec* [16] is used to measure the similarity. *Word2Vec* uses a *skip-gram* [17] approach using a shallow neural network to predict the closeness of the words. Finally, each *document* (graph) is represented as a vector in a low-dimensional vector space.

Sparse Representation and Dictionary Learning

There has been a growing interest in the search for sparse representations of signals in recent years. For example, in the field of computer vision, it can be reasonably assumed that image patches do not populate or sample the whole input domain [4]. Sparse coding is a representation learning method that aims to find a sparse representation of an n dimensional input signal $y_i \in \mathbb{R}^n$ in the form of a sparse linear combination, such that the reconstructed data is $\tilde{y_i} = \alpha_{i,1}d_1 + \alpha_{i,2}d_2 + \cdots + \alpha_{i,K}d_K$. Where $\alpha_i \in \mathbb{R}^K$ is the sparse vector and $d_i \in \mathbb{R}^n$ are the dictionary elements (atoms) of a Dictionary $\mathbf{D}$. Sparse representation algorithms optimize eq. (2.1) with an ℓ_0 regularization term:

$$\underset{D,\alpha}{\operatorname{argmin}} ||\mathbf{Y} - \mathbf{D}\alpha||_2^2 \;\; s.t. \;\; \forall i, ||\alpha_i||_0 \leq S, \tag{2.1}$$

where $\mathbf{Y} = [y_1, y_2, .., y_N] \in \mathbb{R}^{n \times N}$ denotes the N number of input signals, $\mathbf{D} = [d_1, d_2, \ldots, d_K] \in \mathbb{R}^{n \times K}$ is the learned dictionary of size K, $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_N] \in \mathbb{R}^{K \times N}$ is the sparse representation of the input signal, and S is the sparsity constraint of α_i (maximum number of non-zero elements). Usually, $K > n$, in which case the dictionary is called over-complete. If $K = n$ the dictionary is called complete and if $K < n$ it is called under-complete.

Figure 2.4 shows a visualization of the sparse representation matrices.

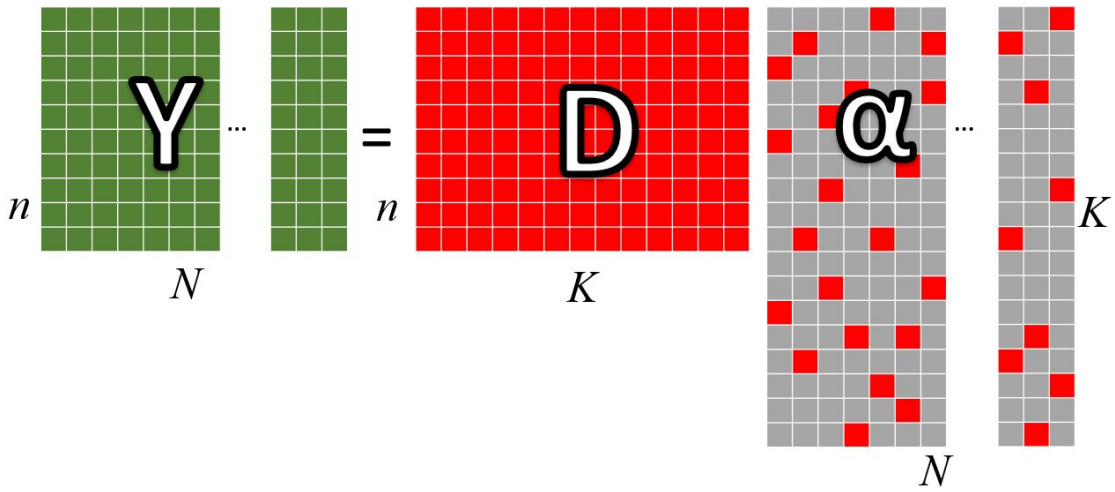
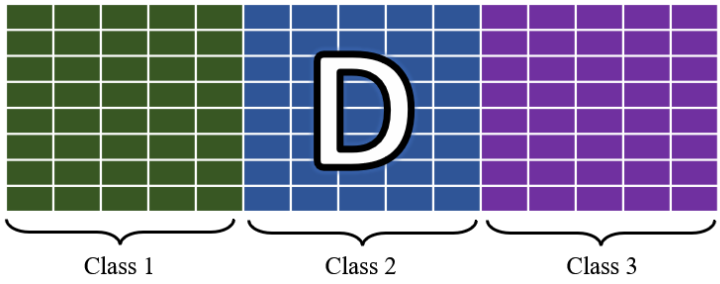


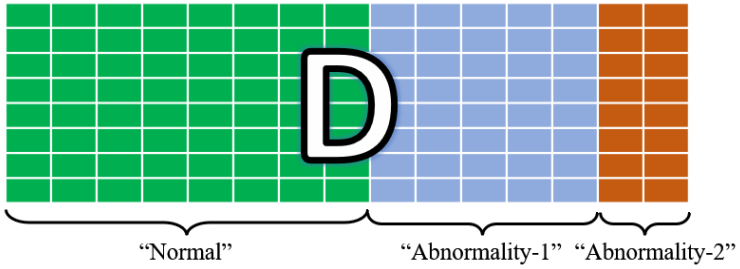
Figure 2.4: Visualization of sparse representation.

The problem stipulated by eq. (2.1) is NP-hard and does not have a solution, however, an approximate solution can be found through a greedy approach. Such an approximation can be calculated by alternating between the following two stages. First, sparse coding is to calculate α with a fixed over-complete dictionary D. Second, dictionary learning is performed to update D with a fixed α. K-means Singular Value Decomposition (KSVD) [18, 19] has emerged as an effective and popular algorithm for sparse representation tasks. KSVD first initializes a random dictionary. It then alternates between the two stages by utilizing Orthogonal Matching Pursuit (OMP) [20, 21] for the sparse coding and generalized k-means with Singular Value Decomposition (SVD) for the dictionary update. KSVD efficiently learns an over-complete dictionary and has been effectively utilized for tasks including de-noising, restoration, and classification. Several variations of the KSVD algorithm have been proposed to improve different aspects. In this dissertation, we will be focusing on the LC-KSVD and Frozen KSVD variants visualized in Fig. 2.5.

For classification tasks, in order to improve performance a more discriminatory



(a) Label consistent KSVD



(b) Frozen KSVD

Figure 2.5: Distribution of dictionary elements according to (a) Label Consistent KSVD (LC-KSVD), and (b) Frozen KSVD dictionary learning algorithms.

representation is required. Jiang *et al.* [22, 23] have presented a Label Consistent KSVD (LC-KSVD) algorithm as an extension of the KSVD framework, which is a supervised learning algorithm to learn a compact and discriminative dictionary. In LC-KSVD, class-specific dictionary elements are trained separately as initialization and then combined to learn a discriminative dictionary. A label consistent constraint called "discriminative sparse-code error", reconstruction error, and classification error terms are combined to structure a unified objective function to optimize the discriminated dictionary. Due to the class constraints in the sparse coding and dictionary update stages, the input data will be forced to be mapped to the dedicated dictionary atoms according to the label information. Consequently, in the sparse dictionary domain, a majority of the input signals will be projected to a subspace belonging to a certain class. Hence, a lower-order classifier can be trained for the classification.

Traditional dictionary learning models do not take into account the class imbalances of the training data. Hence the dictionary atoms can be biased towards the larger class. Therefore to address the class imbalances and the structure, a separate dictionary learning algorithm is also employed. Frozen dictionary learning modifies the dictionary learning process as a hierarchical structure to learn a dictionary that can effectively model imbalanced data sets [24]. In this algorithm, first, the dictionary learning step is carried out using the K-SVD algorithm on "normal" training data. Then the learned dictionary elements are frozen (held constant) and the dictionary is augmented with additional elements that are trained again on abnormalities. This process is repeated for all the remaining classes, by keeping the previously learned dictionaries frozen. The frozen elements of the dictionary represent the "normal" aspects of the data, hence the new elements (non-frozen) learn to represent the anomalous aspects of the data that are not present in the "normal" data. The frozen dictionary approach could be generally used and applied to the problems including data with or without abnormalities. An application of discriminative dictionary learning with frozen and LCKSVD for COVID-19 detection by analyzing using lung CT image can be found at Liyanage *et al.* [25].

Feature Ranking

Feature Ranking (FR) is an essential part of the machine learning pipeline to identify, reduce, remove, or craft features that benefit ML performance and reduce the cost of the operations. In general, FR methods evaluate features by looking at the amount of information they provide and ranking them accordingly so that the most relevant and complementary features can be used in ML training. There are three main categories of methods for FR algorithms: Filter methods (FM), Wrapper methods (WM), and Embedded methods (EM). The FR methods can be further classified as *myopic* and *non-myopic*. Whereas *myopic* methods only evaluate the feature by itself, *non-myopic* methods take

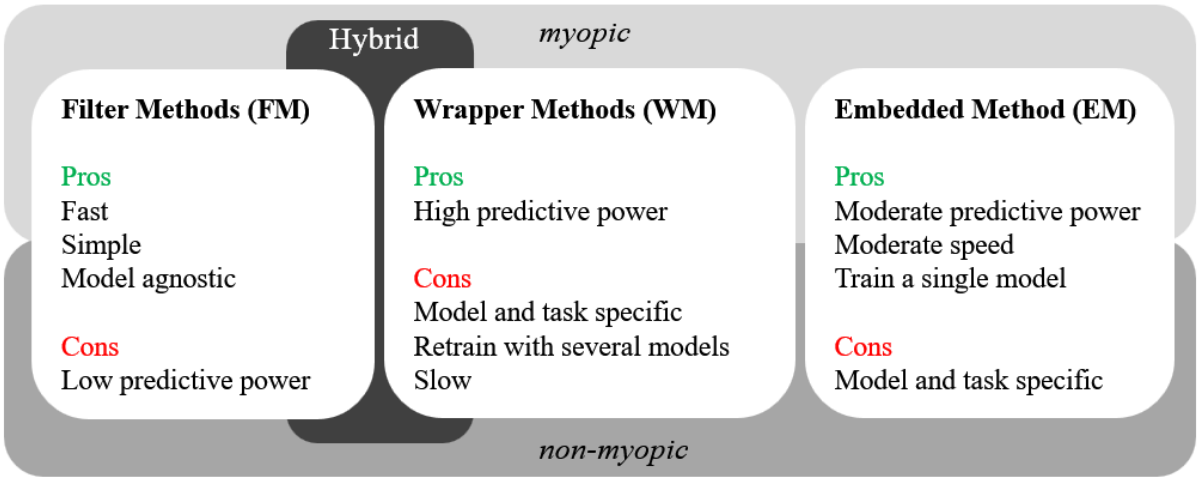


Figure 2.6: Classification of the FR methods and summary of pros and cons

into consideration the interrelationship between the features. Several surveys outline the current state-of-the-art in feature assessment techniques [26, 27, 28, 29]. A summary of each method's pros and cons is shown in Fig. 2.6.

In FM, intrinsic properties are evaluated to determine the relevance of the feature. FM methods are generally computationally inexpensive and do not depend on the ML model since the evaluation is done independently. In WM the feature ranking is tied to the ML model performance. This method is more computationally expensive than the FM as it needs to train multiple ML models to identify which features contribute more. WM is model- and task-specific and, as a result, gives better results. However, WM has to be performed again for a different task or a model. EM is similar to WM, however, it performs the feature selection while the ML model is trained. Therefore, time is saved by avoiding training multiple models by sacrificing performance compared to WM. EM is also model and task-specific.

The presented SR-based methods can be considered as *non-myopic* and a hybrid of FM and EM concepts. To calculate the two metrics, dictionary learning has to be carried out (like EM), which is more computationally expensive than the typical FM. However, these metrics

do not depend on any ML methods so these FR scores can be used in many applications, unlike EM methods. Furthermore, the presented metrics do not have to depend on any particular SR method. However, using a discriminatory dictionary learning and supervised (or semi-supervised) SR method would be able to improve the interpretability of the data. Since the atoms are a subspace of the original feature space when atoms are learned it takes into account the relationship between all the features hence, these metrics are *non-myopic*. Another main advantage is the ability to decompose the feature importance according to each class with supervised dictionary learning methods. Also, the learned dictionary is not wasted as the dictionary and the calculated sparse coefficients can be used for the training of classifiers in the next stages of the machine learning pipeline.

Relieff [30] and *mRMR* [31] are two commonly used FR algorithms. Both methods can be considered as *non-myopic* FM algorithms. Zhang et al. [32] implemented a novel SR-based feature assessment method called SRDA, where they employ both SR and information theory to identify dependencies and redundancies of the salient features. In this work, they evaluate the learned dictionary atoms (new features) for redundancy and complementary properties. In our work, we try to evaluate the original input features, not the derived sparse features. However, they provide some interesting frameworks for selecting candidate sparse features. In recent years several deep learning methods have been proposed that achieve high accuracy for data sets. However, deep learning methods lack an intuitive relationship between the learned features and the input layer. Also, they require large computational resources for training and testing. It can be seen that almost all methods that have been used are some form of deep architecture. The usage of deep networks is popular due to the high performance and ability to learn features automatically [33]. We would urge readers to get familiarized with our previous work [34] for a detailed discussion about the advantages of the SR methods concerning deep learning methods.

Malware

Malicious software, or malware, is a general term used to describe an unwanted, unauthorized computer program or script with the intent to cause some kind of damage or harm. System administrators are unaware of the presence and behavior of malware, and if they were to be aware, they would not permit such programs to run on the system [35]. Malware typically attempts to gain unauthorized access to some system to steal sensitive or financial information, disrupt services, or gain remote access for later use.

As thousands of new malware strains begin to surface each day, the ability to detect malware before it can cause harm has been an area of focus for many cybersecurity experts. Bazrafshanet. al [36] defines three distinct malware detection methodologies: Signature-based, Behavioral-based, and Heuristic-based. Signature-based detection is the most common method for detecting malware. This technique involves searching for a known digital footprint that has already been detected and recorded in the past. File hashes and byte strings that represent things such as function names, IP addresses, or coding structures are both commonly scanned for in static detection. Unfortunately, static-based detection methods fail to detect new forms of malware, or malware that obfuscates itself.

Behavior-based detection uses dynamic analysis for evaluating malware, which is the process of analyzing code or a script by executing it and observing its actions. Dynamic analysis is typically done in a sandbox or virtual environment so malware cannot cause any damage. Behavioral-based detection includes monitoring Windows registry activity, executing binary instructions, and detecting the presence of data in RAM during execution.

Lastly, heuristic-based detection leverages machine learning or data mining to make decisions. Features from malware are extracted and then used to train a machine-learning classifier. Extracted features include operating system API calls, opcode frequency, and program control flow. Such attributes of a program help capture the behavior of a program

and provide a set of traits for a machine learning algorithm to learn from.

Control Flow Graphs

Control Flow Graphs (CFGs) are one of the most common graph representations of code used for binary analysis. Several different graph representations are explored in this research, but the Control Flow Graph yielded the best results. The primary focus of a CFG is to illustrate the control flow, or order of executed statements, of a program. A node in this directed graph typically represents a basic block of code, which is a sequence of instructions that are executed sequentially with no jumps, or branches to other sections of code. The edges represent transitions from the basic block to the basic block through function calls, return statements, program branches, or looping. A binary can be converted into a CFG first by disassembling it into some intermediate low-level representation, such as an assembly language. The assembly language can then be analyzed and searched for program jumps and branch targets to determine the control flow. Several tools already exist for generating a control flow graph from a binary, such as Ghidra and anger [37].

15

GRAPH SPARSE DESCRIPTORS

The majority of the work reported on this section has been published as *Dictionary Learning on Graph Data with Weisfieler-Lehman sub-tree kernel and KSVD* by Liyanage et.al [7] in the *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2023*.

Introduction

Inspired by the Graph2Vec algorithm, we aim to modify the Doc2Vec model training portion of the Graph2Vec by incorporating unsupervised dictionary learning. We investigate the viability of using the sparse dictionary learning technique KSVD for graph data. We train the dictionary on Weisfeiler-Lehman graph sub-tree kernel features. Furthermore, we use graph-based labeled data sets to compare classification results with several existing graph embedding methods. Findings show that using the learned sparse coefficients as features for a supervised machine learning algorithm provides on-par classification results when compared to other graph embedding methods.

Problem

Graph2Vec is a popular neural network-based architecture for graph embedding [12]. Some advantages of Graph2Vec are that the model is trained in an unsupervised manner, the learned model is task agnostic, the algorithm is data-driven, and resulting vectors capture structural equivalences. Graph2Vec utilizes the non-linear Weisfeiler-Lehman (WL) kernel, [38], which is shown to outperform other linear kernels [13, 39]. WL kernel is used to rename the nodes using a hash value that represents a rooted sub-graph on the given node. These sets of node names are viewed as a set of words in a document. The techniques from the Natural language processing (NLP) domain are borrowed for learning an embedding. Doc2Vec is

based on Word2Vec [16], in which a feed-forward neural network (NNet) "SkipGram" model with negative sampling is used to learn a representation of word sequences [14]. Using the SkipGram model, the nodes with similar neighborhoods are embedded closer together [17]. The Graph2Vec is implemented in the "KarateClub" python package [40][1]. An overview of the implementation of the Grap2Vec is shown in Fig. 3.1, where a vocabulary of sub-tree structures is generated using a WL sub-tree kernel, and a Doc2Vec model is trained on the selected vocabulary.

Some disadvantages of Graph2Vec are the nonlinearity of the learned embedding and the generated sub-tree structures. Due to the nonlinearity, it is difficult to identify which sub-tree structures are contributing to the similarities and differences among graphs. Hence, we present a linear representation model to replace the Doc2Vec NNet architecture. Further, the SkipGram model is capable of embedding only a single node, rather than node combinations. In addition, the SkipGram model considers the neighborhood of the nodes, which depends on an arbitrary node numbering scheme that may not generalize between graphs in a given application.

Recently, explainable machine learning has gained popularity and tools are being developed to explain non-linear and complex neural networks [41]. SHAP (SHapley Additive exPlanations) is a popular tool that uses Shapley values to explain any machine learning model [42]. Such tools can be used to understand the relationship between the input features and the output. Although these tools perform well and provide a correlation between the input and the output, since the ML model is inherently nonlinear it is hard to establish a causation. Hence, For applications that prioritize intuition over performance, a linear model would be preferable. For example, in this thesis, we are looking at malware analysis where our goal is to identify important sub-tree structures. Hence a more intuitive model is preferable to an unintuitive high-performing model. If the goal was to detect malware in

[1] `https://karateclub.readthedocs.io/en/latest/`

real time, then a high-performing model is preferable even if it cannot be explained.

Solution

Sparse representation is a technique used to learn a dictionary that lies in the original feature domain and calculate a sparse representation using a linear combination of a few dictionary elements (atoms) [4]. The main advantages of using sparse representation are linearity and sparsity: the learned embedding consists of linear combinations of sub-tree structures; sparse representations allow using low-order classification models due to the low VC dimension [43]. Sparse representation was originally introduced in the signal and image processing domains, however recently it has been utilized in graph-related processes. Several methods have been proposed to represent graph signals on a fixed graph topology with sparse representations with theoretical guarantees [44].

The problem can be formally defined as: *A graph is represented as one sparse vector and two graphs with similar sub-structure are embedded to be closer. The graph can be directed, cyclic, and contain node features.*

Recent work by Matsuo et al. [5] develops a method to represent different network topologies with sparse representation. However, their work is still limited by requiring graphs to be undirected and requiring all topologies to have the same number of nodes. A recent article by Vincent-Cuaz et al. [45] has proposed an online Graph Dictionary Learning (GDL) algorithm that utilizes Gromov-Wessertein (GW) distance to find the relationship between nodes. They have provided a theoretical analysis and experimental results. Due to their online algorithm, it performs well with larger datasets without having memory issues. This work differs from GDL in the approach of the node structures are encoded and the similarity is measured between the nodes.

To address the shortcomings in sparse vector-based graph representations, we introduce a framework to incorporate WL sub-tree kernel with sparse representation methods

specifically aimed at machine learning classification tasks. Our framework allows sparse representation to be applied to graphs with different topologies and different numbers of nodes. In addition, the input graphs can be directed and can incorporate node features. An overview of the presented WL+KSVD pipeline is shown in Fig. 3.2. The presented method has the flexibility to swap different dictionary learning and graph kernel methods in the framework. The method is tested against several similar graph embedding methods with benchmark data sets. Finally, the Python implementation of the framework and the experiments are currently available on *Github* repository at `https://github.com/BMW-lab-MSU/WL-KSVD.git`.

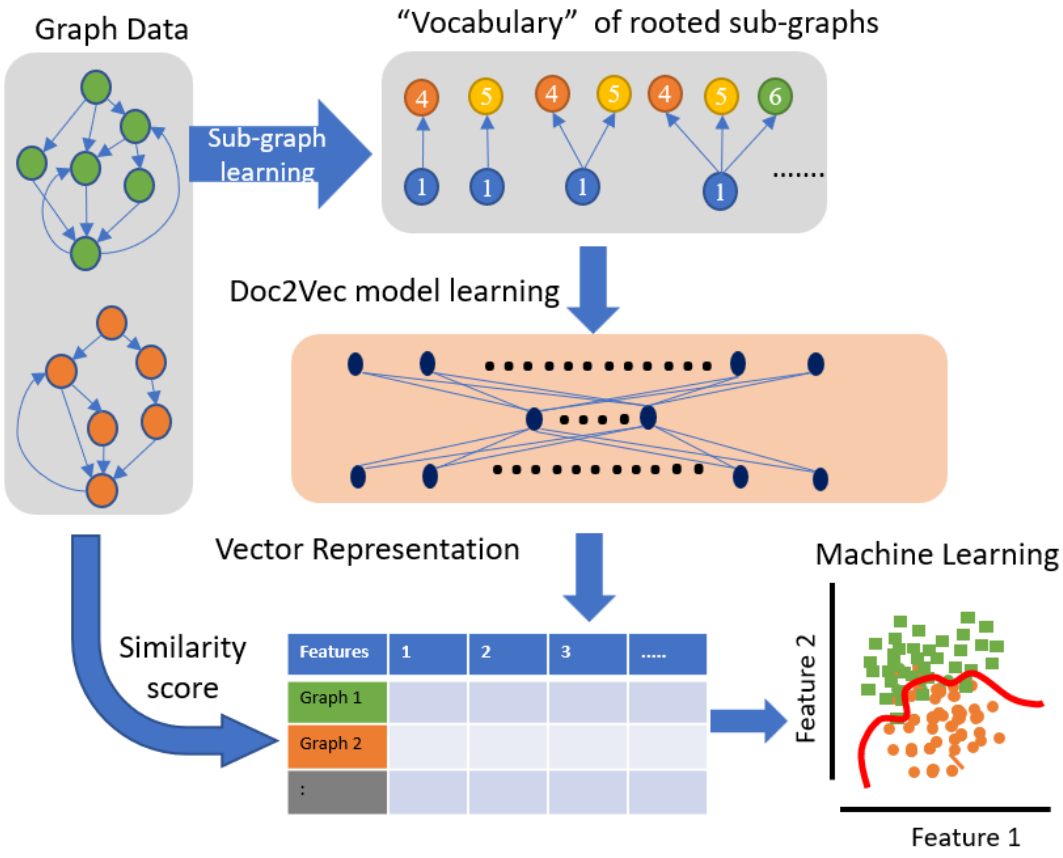


Figure 3.1: Graph2Vec pipeline overview

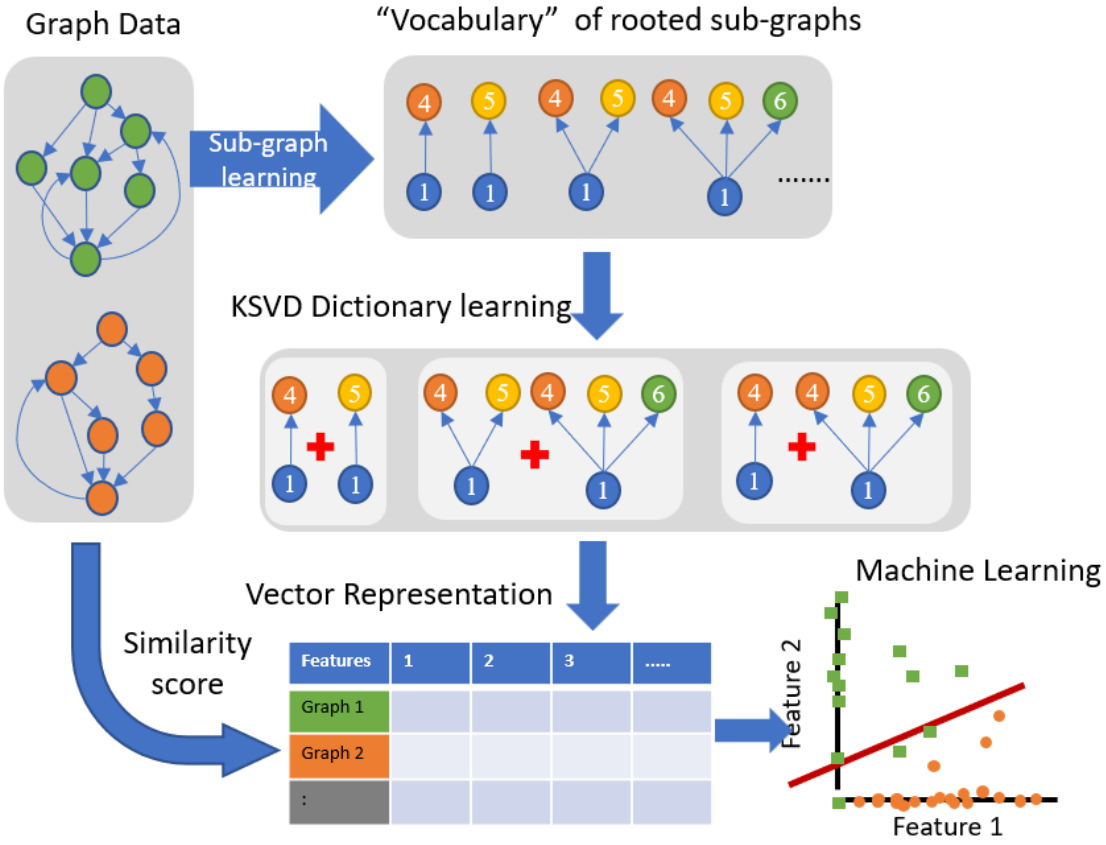


Figure 3.2: WL+KSVD pipeline overview

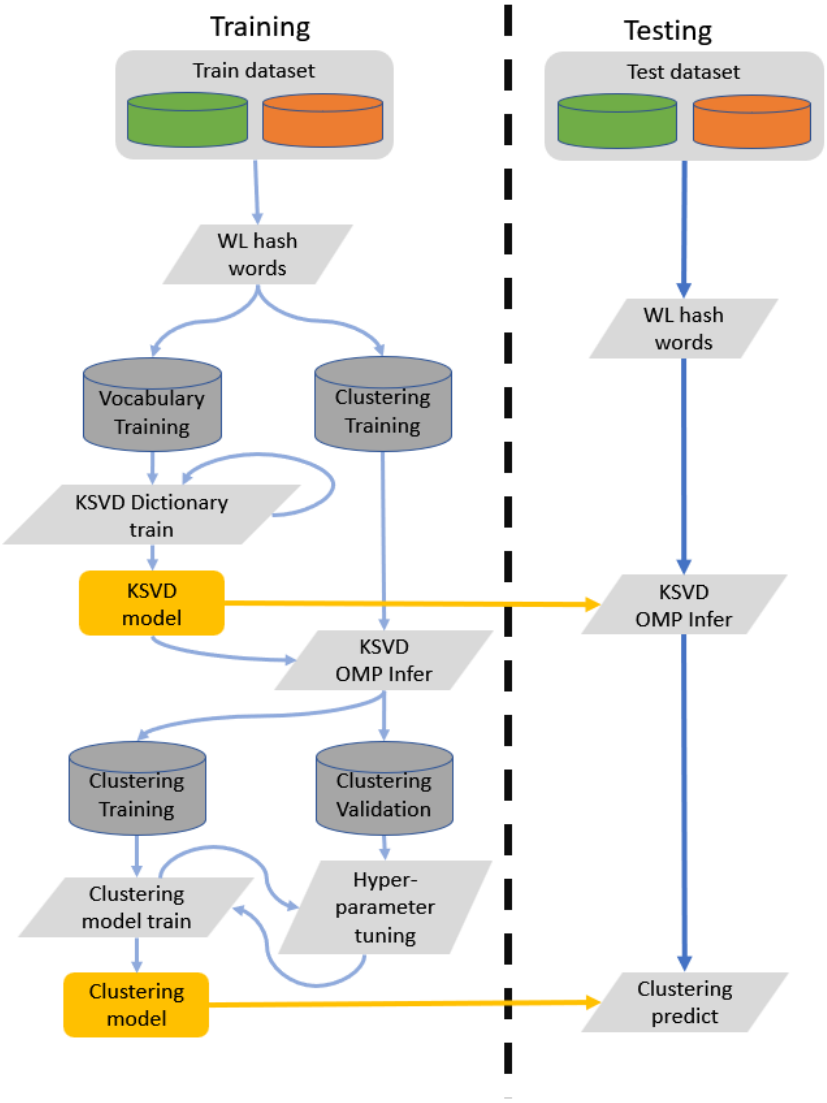


Figure 3.3: WL+KSVD method workflow

Methodology

An overview of the workflow of the presented method is shown in Fig. 3.3, where the training set is further divided in half into embedding training and classifier training to avoid overfitting.

Graph Notation and Definition

Let a graph be defined as $G = (V, E)$ which can be directed or undirected with unweighted edges, where node $v_i \in V$ and edge $e_{i,j} \in E$ connects v_i and v_j. Let the dataset be a set of N graphs with different topology and nodes $\mathbf{G} = [G_1, G_2, \ldots, G_N]$. Following the Graph2Vec algorithm, if node labels are not provided the nodes will be initialized with the degree of the node as its label. The degree of a node is a count of the number of edges the node receives and sends. We use the "NetworkX" python package as the graph data structure[2].

Weisfeiler-Lehman Sub-tree Kernel

WL sub-tree relabelling process (described in [13][3]) is used to relabel the nodes with a unique hash value for the rooted sub-tree structure. Note that the sub-tree structure learned is deterministic, so the same sub-tree structure in different graphs will have the same hash value. For each $G_i \in \mathbf{G}$, rooted sub-trees $sg^h_{i,j}$ are learned for each $v_j \in \mathbf{V}_i$, where i is the graph, j is the node and h is the WL rooted sub-tree depth. Now each graph is a set of hash words $G_i = [sg^h_{1,i}, sg^h_{2,i}, \ldots, sg^h_{l,i}]$, where l_i is the number of nodes in G_i.

[2] `https://networkx.org/`

[3] `https://github.com/benedekrozemberczki/karateclub/blob/master/karateclub/utils/treefeatures.py`

Vocabulary Creation

Using the Doc2Vec implementation in Gensim python package[4] a raw vocabulary is created using the unique set of sub-tree hash words sg across all the training graphs [14]. If the raw vocabulary is too large it can be trimmed according to a trim rule. In this work, we trim the vocabulary by selecting the K highest frequency sub-tree hash words. Other possible trimming rules are the highest likelihood, highest prior, etc.

Each graph G_i is then represented as the occurrences Y_i of the vocabulary elements, where $Y_i = [y_{i,1}, \ldots, y_{i,M}]$ and $y_{i,j}$ is the number of occurrences of vocabulary word j in graph i. Now the dataset can be represented as a collection of fixed-length vectors: $\mathbf{Y} = [Y_1, Y_2, \ldots, Y_M] \in \mathbb{R}^{M \times N}$.

Sparse Representation

Let $\mathbf{Y} = [Y_1, Y_2, \ldots, Y_N] \in \mathbb{R}^{M \times N}$ be a set of M input signals with fixed length M. Sparse representation attempts to represent the input signal as a linear combination of elements $d_i \in \mathbb{R}^M$ in a dictionary $\mathbf{D} = [d_1, d_2, \ldots, d_K] \in \mathbb{R}^{M \times K}$ while limiting the number of atoms used to S (sparsity). The sparse coefficient vector $\alpha = [\alpha_1, \alpha_2, \ldots \alpha_N] \in \mathbb{R}^{K \times N}$ will be the sparse representation with $|\alpha_i|_0 \leq T$, where $|.|_0$ operator counts the number of non-zero elements in the vector. The general form of the sparse representation can be formulated as:

$$\underset{\mathbf{D},\alpha}{\operatorname{argmin}} ||\mathbf{Y} - \mathbf{D}\alpha||_2^2 + ||\alpha||_0. \tag{3.1}$$

This equation is NP-hard, but an approximate solution can be provided using an iterative algorithm named K-means Singular Value Decomposition (KSVD) [18][5]. First, a dictionary $\mathbf{D}$ is fixed, and sparse vectors α are optimized using Orthogonal Matching Pursuit (OMP) [20]. Second, α is fixed and the dictionary $\mathbf{D}$ is updated with a generalized K-means

[4] `https://radimrehurek.com/gensim/`
[5] `https://github.com/nel215/ksvd`

algorithm. After many iterations, each graph is represented as a fixed-length *sparse* vector. In addition, using the trained dictionary, new graphs can be represented as sparse vectors.

Implementation

The initial implementation of the *WL+KSVD* methods is hosted at `https://github.com/BMW-lab-MSU/WL-KSVD` with the associated publication [7]. This implementation utilizes an approximate KSVD package from `https://github.com/nel215/ksvd`. It should be noted that the KSVD is known to have convergence issues and newer methods have better performance and convergence [46]. Since the initial implementation was mainly focused on the framework for using sparse coding for graph embedding. a simpler well-known method (KSVD) was chosen to demonstrate its viability.

To make the the presented method more flexible and robust a generalized implementation is created at `https://github.com/BMW-lab-MSU/Graph_Sparse_Coding`. This new package utilizes *scikit-learn* package and its dictionary learning algorithms. Some of the options available for dictionary learning [6] and sparse coding [7] are Least and Regression (lars) [47], Least Absolute Shrinkage and Selection operator (LASSO) [48, 49], and threshold methods. Methods like LASSO use ℓ_1 optimization rather than the ℓ_0 of KSVD which could lead to faster learning.

Experiment

The performance of the WL+KSVD should be compared to other graph embedding methods. The WL+KSD was tested on popular graph data. The graph data[8] sets used

[6] `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.DictionaryLearning.html`

[7] `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.SparseCoder.html`

[8] `https://chrsmrrs.github.io/datasets/`

in *Morris et al.* [2] shown in Table 3.1 is evaluated using WL+KSVD method. The results (accuracy) will be compared to G2V, GL2V, and SF graph embedding methods. A couple of large-scale datasets were considered, but due to the memory usage of the WL+KSVD method, the large-scale dataset testing was not finalized. The full experiment setup and results are hosted at `https://github.com/BMW-lab-MSU/Graph_Sparse_Coding`. The benchmark results for the datasets are published by Errica et al. [50] and the latest classifier performances on the datasets can be found at `https://paperswithcode.com/task/graph-classification`

Several publicly available benchmark datasets were chosen to compare the embedding performance, namely MUTAG (MU), PTC, PROTEINS (PROT), NCI1, NCI109 [51] [9]. The datasets were divided as $9 : 1$ for training and testing with random shuffling. Several graph embedding methods in the KarateClub library are compared: Graph2Vec [12], GL2vec[52], and SF [53]. The considered embedding dimensions are $N = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]$. Default settings for each embedding method are used to avoid bias in training. In the presented WL+KSVD method, the max size of the learned vocabulary is chosen as $K <= 10000$ and the sparsity is chosen roughly as 10% of the dimensions, $T = ceil(N/10)$.

Several classification methods from the *scikit-learn* python package are used to evaluate the performances of the embedding methods [54] [10]. The classifiers were used with default settings and without hyperparameter tuning for each method to avoid bias in training and for easy comparison. The classifiers and settings use the sci-kit learn example for easy recreation with 5 fold cross validation[11].

[9] `https://chrsmrrs.github.io/datasets/docs/home/`

[10] `https://scikit-learn.org/`

[11] `https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html`

Table 3.1: Graph data sets with various properties for evaluation. (N = Node, E = Edge, L = Lable, A = Attributes, + = positive, - = negative)

Dataset	Graphs	Classes	Avg. N.	Avg. E.	N. L.	E. L.	N. A.	E. A.
Samll scale								
Mutag	188	2	17.93	19.79	+	+	-	-
Enzymes	600	6	32.63	62.14	+	-	+	-
IMDB-Binary	100	2	19.77	96.53	-	-	-	-
IMDB-Multi	1500	3	13.00	65.94	-	-	-	-
NCI1	4110	2	29.87	32.30	+	-	-	-
NCI109	4127	2	29.68	32.13	+	-	-	-
PTC FM	349	2	14.11	14.48	+	+	-	-
PTC MR	344	2	14.29	14.69	+	+	-	-
Proteins	1113	2	39.06	72.86	+	-	+	-
Reddit-Binary	2000	2	429.63	497.75	-	-	-	-
Mid scale								
UACC257	39988	2	26.09	28.13	+	+	-	-
UACC257H	39988	2	46.68	48.71	+	+	-	-
OVCAR-8	40516	2	26.08	28.11	+	+	-	-
OVCAR-8H	40516	2	46.67	48.70	+	+	-	-

Results

Figure 3.4 shows the two-dimensional ($K = 2$) embedding of the MUTAG dataset with different graph embedding methods. (Note that using $K = 2$ dimensions does not result in optimal embeddings; this number is only chosen for demonstration.) The figure also shows

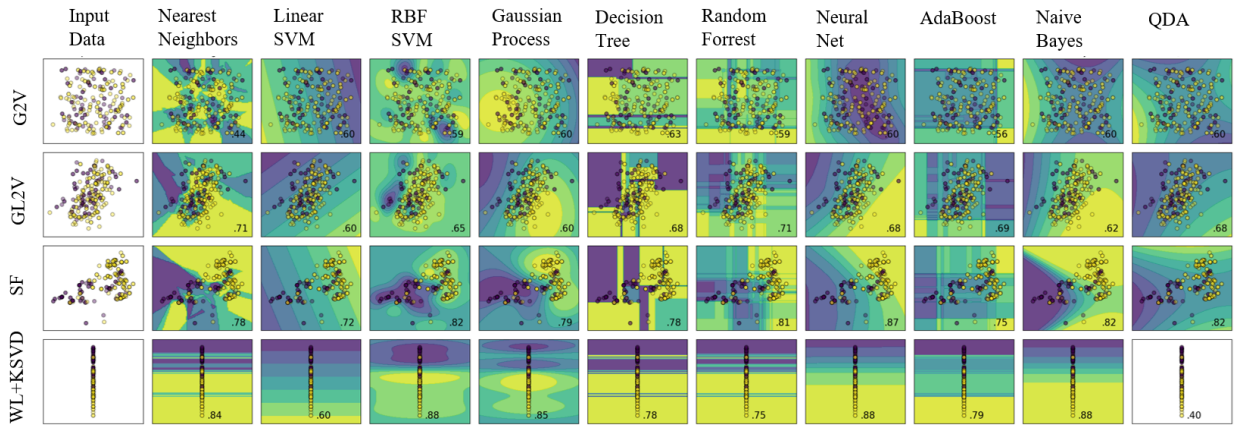


Figure 3.4: Classification decision boundaries for MUTAG dataset with 2-dimensional embedding for four embedding algorithms (Graph2Vec, GL2V, SF, and WL+KSVD). A sparse structure is seen in the WL+KSVD embedding (bottom row).

Table 3.2: Linear SVM validation accuracy for various embedding dimensions using the MUTAG dataset

Dimension	2	4	8	16	32	64	128	256	512	1024	2048
G2V	66.5	66.5	65.4	64.3	64.3	67.5	67.5	65.9	66.4	68.5	68.6
GL2V	67.0	67.6	71.8	73.4	75.0	76.6	76.0	73.3	76.5	74.9	76.6
SF	72.4	78.8	82.4	84.0	83.5	83.5	83.5	83.5	83.5	83.5	83.5
WL+KSVD	66.5	66.5	66.5	68.7	75.6	76.6	78.7	75.6	74.5	72.4	71.3

several classifier methods evaluated on the embedded data. It can be seen that the sparse coding limits the data to a sparse domain in the final vector space.

Table 3.2 shows the mean accuracy of the linear SVM clustering variation as a function of embedding dimension for each embedding method. It can be observed that the presented method performs on par with Graph2Vec and GL2vec. SF methods have higher performance over all the dimensions for the linear classifier. SF method stands for Spectral Features, which extracts the adjacency matrix and the Degree matrix from a graph and calculates

Table 3.3: Linear SVM validation accuracy for various embedding dimensions using the Yeast dataset

Dimension	2	4	8	16	32	64	128	256	512	1024	2048
G2V		66.5	65.4	64.3	64.3	67.5	67.5	65.9	66.4	68.5	68.6
GL2V		67.6	71.8	73.4	75.0	76.6	76.0	73.3	76.5	74.9	76.6
SF	72.4	78.8	82.4	84.0	83.5	83.5	83.5	83.5	83.5	83.5	83.5
WL+KSVD	66.5	66.5	66.5	68.7	75.6	76.6	78.7	75.6	74.5	72.4	71.3

the Laplacian. Then the lowest eigen-values are chosen as the embedding of the graph. This process takes both node and edge information for its calculation, hence performs well. However, this process cannot be used to identify the important sub-tree structures.

Table 3.6 shows the linear SVM test accuracy (mean and standard deviation) for the different datasets with an embedding dimension of 1024. The presented WL+KSVD method has on-par performance compared to others with relatively low standard deviation. It should be noted that the hyperparameters were not optimized for any method and executed with default settings. Hence the actual optimal results will be better[12]. However, this analysis provides a baseline comparison of the performance of the WL+KSVD method. Table 3.4 and 3.5 show the training time and the model size for each embedding method for each dataset. It can be seen that the presented WL+KSVD method has significantly worse time and memory requirements. This is partially due to the overcomplete nature of the dictionary and the complexity of the sparse representation algorithm.

Tests were conducted with different dictionary learning algorithms incorporated with the presented method to show the ability to customize. The KSVD dictionary learning algorithm in *WL+KSVD* is replaced with *lars* algorithm to build *WL+lars* method. The

[12] `https://github.com/BMW-lab-MSU/WL-KSVD.git`

scikit-learn classifiers considered with default parameters are Nearest Neighbours (NN), Linear SVM (lSVM), Gaussian SVM (GSVM), Decision Tree (DT), a simple Neural Network (NNet), AdaBoost (AB), and Naive Bayes (NB). Table 3.7 shows the mean classifier accuracies for the MUTAG dataset for $K = 1024$ dimensions with the modified *WL+lars* method. It can be seen that there is a performance improvement in *lSVM* classifier compared to *WL+KSVD*. Also, it can be observed that linear SVM (lSVM) and simple Neural Network (NNet) classifiers perform better with sparse data than others. Another reason is the non-optimized WL+KSVD pipeline which uses default settings and data structures. Online batch learning and usage of sparse data types would improve the performance of the WL+KSVD pipeline.

Table 3.4: The training time in seconds for each embedding for different datasets for the embedding dimension of K =1024.

Datasets	MU	PTC	PROT	NCI1	NCI109
G2V	0.043	0.044	0.282	0.704	0.728
GL2V	0.070	0.093	1.339	2.1076	1.941
SF	0.0811	0.130	1.910	2.205	2.294
WL+KSVD	10.35	25.71	233.646	258.42	226.723

Table 3.5: Embedding model size in MB for each embedding for different datasets for the embedding dimension of K =1024.

Datasets	MU	PTC	PROT	NCI1	NCI109
G2V	1.1	2.2	9.1	18.9	19.5
GL2V	2.0	3.7	21.5	31.5	31.2
SF	0.61	1.1	3.7	13.5	13.6
WL+KSVD	2.6	6.9	167.9	38.9	39.6

Table 3.6: Linear SVM test mean accuracy with $K = 1024$ embedding and standard deviation with 5-fold cross validation

Datasets	MU	PTC	PROT	NCI1	NCI109
G2V	68.55	55.23	67.30	59.30	56.46
	±10.03	±5.9	±0.87	±4.46	±2.82
GL2V	74.92	52.04	69.09	64.52	62.98
	±7.8	±6.5	±1.38	±1.99	±2.97
SF	83.47	57.59	70.98	61.90	61.96
	±4.15	±9.34	±1.00	±3.24	±2.40
WL+KSVD	72.38	54.36	64.60	64.16	62.93
	±3.20	±2.31	±2.00	±2.19	±0.24

Table 3.7: Classifier test mean accuracy with 5-fold cross-validation with standard deviation for MUTAG dataset for $K = 1024$ with embeddings

	NN	lSVM	GSVM	DT	RF	NNet	AB	NB
G2V	68.0	74.9	66.5	63.7	69.1	72.3	69.1	73.3
	± 13.4	± 11.1	± 0.9	± 11.2	± 3.0	± 9.6	± 5.7	± 9.4
GL2V	75.5	75.4	66.5	65.3	79.2	79.7	72.8	**77.5**
	± 4.7	± 12.1	± 0.9	± 9.7	± 4.8	± 7.7	± 7.2	± 8.6
SF	**83.9**	81.8	**73.8**	**77.1**	**82.9**	83.9	**78.7**	66.5
	± 9.1	± 7.8	± 7.5	± 8.9	± 9.1	± 7.1	± 5.7	± 0.9
WL+lars	57.8	**82.9**	66.5	63.2	67.0	**85.6**	69.7	71.3
	± 15.4	± 3.0	± 0.9	± 7.1	± 3.6	± 4.6	± 8.9	± 4.8

Conclusion

Sparse representation is a powerful tool in several signal-processing application domains because of its ability to extract inherent features. We aim to expand sparse representation tools into graph processing domains. The WL+KSVD framework is presented as an unsupervised whole graph embedding method. The input graphs can be directed or disconnected and may have node labels. Through benchmark datasets and several comparable graph embedding methods, it was shown that the presented method has on-par performance in classification tasks. While the WL+KSVD embedding method is non-reversible and some information is lost in the vocabulary trimming process, this is not an issue for ML tasks like classification and clustering. The framework is flexible in that the WL sub-tree kernel and the KSVD dictionary learning can be swapped with other graph kernels and dictionary learning methods.

Potential Limitations

The main limitation of the presented *WL+KSVD* is the performance. Since it creates an over-complete dictionary, time and memory usage is high. Hence, for large datasets or datasets with a large number of unique sub-tree structures the created vocabulary will be significantly large. Also, dictionary learning is currently performed atom-wise and in series. Hence testing time could be unfeasible. Further, due to the sparse nature of the representation, the computational cost might be expensive for functions that do not support sparse operations. Hence, The method does not scale properly to larger data sets and would be a bottleneck in testing large data sets. Also, in current vocabulary trimming, the most frequently used words are selected, leading to dense input features, therefore the WL+KSVD method would underperform due to the lack of sparsity. Hence, different trimming rules has to be explored.

Future Directions

In the case of infeasible evaluation time, several optimization strategies have been developed for efficient dictionary learning, such as batch update and online learning [55]. In case of memory issues, choosing supported sparse operations as much as possible would mitigate some issues.

DICTIONARY BASED FEATURE RANKING

The majority of this section is published as *Feature Analysis in Satellite Image Classification Using LC-KSVD and Frozen Dictionary Learning* by Liyanage et al. in the *2022 Intermountain Engineering, Technology and Computing (IETC)* conference [8].

Introduction

With the rise of data science and the popularity of deep networks, many researchers have started using machine learning (ML) for various applications. However, due to the complexity and the non-linearity of the ML models, researchers have difficulty interpreting model behaviors. It is important to have high performance as well as an understanding of which underlying features lead to these outcomes/performances. Feature ranking/selection (FR/FS) is an essential part of machine learning to identify, reduce or remove redundant and unnecessary features. We are introducing two FR metrics based on sparse representation. Sparse representation has gained popularity as a method with the ability to uncover hidden patterns in data. This proposal aims to justify the viability of the sparse representations' utility as a feature ranking metric. Preliminary results show on-par performance to the common FR metrics. However common FR metrics lack interpretability and are not able to give class-wise scores. Experiments were performed to show the classifier performance variation with suggested feature removal. A comparison of traditional FR methods and presented methods is conducted and an application in satellite image classification with different SR methods is also shown.

We are introducing two metrics for FR based on sparse representation. To our knowledge, sparse representation has not been used as an FR metric to rank features in the original input domain. Sparse representation reveals underlying patterns of the dataset in the sparse domain under certain assumptions. Due to the linearity and simplicity of

the sparse representation, we can map the sparse domain to the input domain and have an interpretable relationship between the two domains. We hypothesize that the sparse coefficients in the sparse domain can be used as an FR metric in the input domain. The presented metrics can give class-wise feature distribution as opposed to other common FR metrics. A public Python package is developed to deploy the presented FR metrics with working examples and experimental results. The code and the experimental results can be found at `https://github.com/BMW-lab-MSU/dictionary_feature_ranking` GitHub repository.

By identifying unnecessary features, researchers can save money, time, storage, and computational requirements in the ML procedure. Presented metrics will provide researchers with intuitive insight into model behavior from a different perspective. Combination with other FR methods will give researchers flexibility in their decisions to identify redundant features. These metrics are useful in machine learning applications related to vastly different disciplines. The application of the FR metrics is demonstrated with applications in image classification and computer malware detection. Especially, the metrics should be effective in any intrinsically sparse signals like audio, video, images, and natural signals.

Problem

As described in section 2, although many FR algorithms exist, there is no universal method that fits all the applications. Each FR method focuses on different aspects of the features depending on application requirements. The problems identified in common FR metrics and the focus to address are as follows.

1. Interpretability.

   Some of these measures are highly non-linear and complex, it is hard to have an intuitive understanding of the FR process. There is no direct/linear relationship between the features and the ML model behavior.

2. Class-wise feature relevance.

   Most FR methods cannot determine class-wise FR in the presence of multiple classes. Especially in FM, although FR can be applied to each class separately or as a whole, they cannot distinguish feature relevance for each class. The relationship between the classes is not considered when ranking is calculated for a single class.

3. Imbalance class data.

   Some traditional FR methods do not consider class distributions or class imbalances. This affects unfavorably small classes of the dataset as the larger classes will overwhelm the FR metric. This becomes a significant problem when you are trying to detect abnormalities/ outliers or have small sample size classes. For example, in a medical computational topology (CT) scan you might have rare occurrences of lesions. If you just apply FR metrics to find the best features to represent the CT images it will find the "best" of features to represent the majority of the images. Which could omit features that would be specific to represent the small number of lesions.

Solution

Sparse representation (SR) is a method of increasing the dimensions of the dataset to achieve a better representation of the data [4]. This process is linear and easier to understand. In SR, the process is divided into two steps. First, it learns a dictionary of subspaces of original feature space. These, subspaces or the dictionary elements are "atoms". Second, it tries to approximately represent each datum as a linear combination of those dictionary atoms. In practice, these two processes are iteratively optimizing each other until a stopping criterion is met. SR methods are effective in several under several assumptions. One is the original data is laying in a subspace of the original feature space This assumption seems to be true for applications like images and music analysis. The learned atoms reveal some information about the data distribution in the original feature space.

Hence, we are proposing a method that will use SR to identify important features for FR tasks. The *dictionary mapping*, and *dictionary utilization* are introduced as novel and simple metrics for FR tasks. These metrics evaluate the learned dictionary atoms and linearly transform them to the original feature space. This allows us to observe the atom distribution in the original feature space. These methods are *non-myopic* and a hybrid of FM and EM concepts. To calculate the metrics, dictionary learning has to be carried out (like EM), which is more computationally expensive than the typical FM. However, these metrics do not depend on any ML methods. Unlike, EM methods these FR scores can be used in many applications. Furthermore, the presented metrics do not depend on any particular dictionary learning method. However, using a discriminatory dictionary learning and (semi-) supervised SR method would be able to improve the interpretation ability of the data. Since the atoms are a subspace of the original feature space when atoms are learned it takes into account the relationship between all the features hence, these metrics are *non-myopic*. Figure4.1 summarizes the issues identified and the aspects of the SR that would help improve/address each of the issues about common FR metrics. It should be noted that not every FR metric has these issues, and might address one or more issues. However, with SR methods we could improve all of the issues at once. The goals of this section are as follows,

1. Develop two well-defined sparse representation-based robust FR metrics.

   To identify the operational capacity and limitations of the presented two SR-based FR metrics, namely, "dictionary mapping" and "dictionary utilization". The presented metrics will be tested on common datasets with different attributes.

2. Perform FR on class imbalanced data with better interpretability.

   K-means singular value decomposition (KSVD) [18] is an efficient algorithm for SR-based methods. However, it is an unsupervised dictionary learning method and cannot

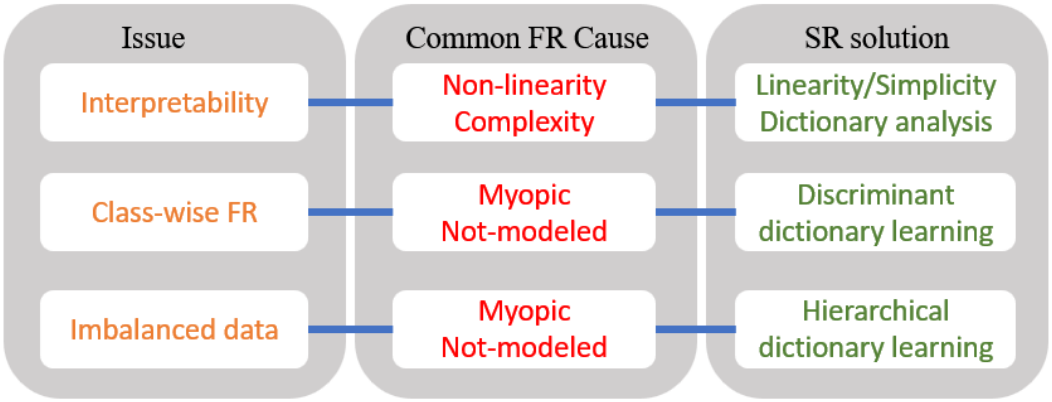


Figure 4.1: Issues with common FR methods, the common causes and the properties of SR which could improve the issues

incorporate label information. Hence, supervised dictionary learning algorithms will be tested with presented metrics to improve the feature interpretability. To incorporate class label information several variations of the KSVD algorithms exist. Two of the popular methods are label-consistent KSVD (LCKSVD) [22], and Frozen KSVD [24].

Methodology

Let $\mathbf{F} = [F_1, F_2, \dots F_d]$ be a set of d features F which are collected or curated. The goal is to rank the feature set $\mathbf{F}$ by evaluating a mean-removed training set of n samples: $\bar{\mathbf{Y}} = [y_1, y_2, \dots, y_n] \in \mathbb{R}^{d \times n}$. The set of p classes is defined as $\mathbf{C} = [C_1, C_2, \dots, C_p]$, where each of the y samples is assigned to a class C. In sparse representation the input sample is represented as a linear combination of dictionary elements D in an over-complete ($d \ll m$) dictionary $\mathbf{D} = [D_1, D_2, \dots, D_m] \in \mathbb{R}^{d \times m}$, where the number of dictionary elements used, s, is far less than the number of dictionary atoms: $s \ll m$. The set coefficients $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_n] \in \mathbb{R}^{m \times n}$ of the linear combination is called the sparse coefficients. Each coefficient vector contains s nonzero entries; the remaining $m - s$ entries are exactly zero. The general objective function used for calculating the sparse representation is given by (4.1)

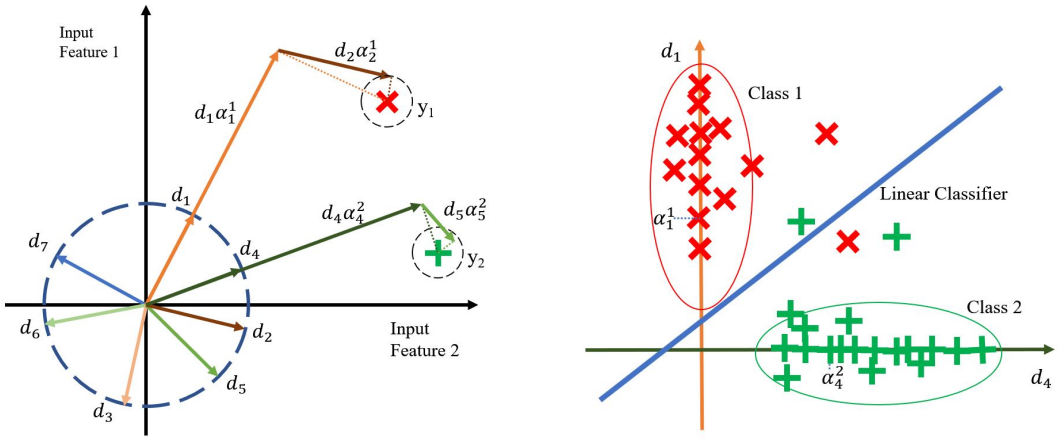


Figure 4.2: Vector representation of sparse representation. (a) Sparse representation of the input data with dictionary atoms. (b) Classifier in the dictionary atoms domain.

with an ℓ_0 constraint on sparsity.

$$\underset{\mathbf{D},\boldsymbol{\alpha}}{\operatorname{argmin}} ||\bar{\mathbf{Y}} - \mathbf{D}\boldsymbol{\alpha}||_F + ||\boldsymbol{\alpha}||_0 \tag{4.1}$$

The KSVD algorithm is an efficient iterative method that solves the objective function by, first fixing the $\mathbf{D}$ and optimizing the $\boldsymbol{\alpha}$ using orthogonal matching pursuit (OMP) [20]. Second, it fixes $\boldsymbol{\alpha}$ and then optimizes $\mathbf{D}$ with generalized K-means and singular value decomposition (SVD). Since the learned dictionary is over-complete, the spread of the dictionary atoms can give an insight into which features are more relevant for the representation. Hence, we will be defining simple metrics that will quantify the spread of the dictionary elements in each of the features.

Figure 4.2a illustrates the representation of the input signals with the learned dictionary atoms in a two-dimensional input feature domain with sparsity, $S = 2$. y_1, y_2 are the input signals belonging to two separate classes. With LC-KSVD the algorithm forces dictionary atoms $d_1, \ldots, d_3$ to be used for y_1 in class 1 and $d_4, \ldots, d_6$ to be used for y_2 in class 2. Fig.

4.2b illustrates the learned classifier boundary in the (sparse) domain of d_1 and d_4. Due to the class constraints in the dictionary learning process, the majority of the input data will only be represented by the dictionary atoms corresponding to their class. Thus in the sparse domain, most signals will lie in a class-specific subspace of the entire dictionary. Hence, a linear classifier can be utilized for the classification with satisfactory results. Further, there is a direct relationship between the classifier domain and the input domain.

Figure4.4 shows the linear mapping of the dictionary elements and weighted dictionary elements to the original input space. Even though the metrics are simple projections of the dictionary elements into input space, Learning dictionary elements are complex. The dictionary elements are learned through the KSVD algorithm iteratively optimizing, considering all the training samples. Hence the learned dictionary elements capture the common patterns in the dataset. Figure 4.3 provides and overview of how the two presented metrics relate to the input features and the sparse embedding space.

Dictionary Mapping

First *Dictionary mapping*, $\mathbf{D}_{\text{map}} \in \mathbb{R}^{1\times d}$, which calculates the sum of the squares of the projections of the dictionary atoms for each feature as given in (4.2), where Proj_j is the projection operator into feature F_j.

$$\mathbf{D}_{\text{map}}(j) = \sum_{i=1}^{m} \underset{j}{\text{Proj}}(d_i)^2 = \sum_{i=1}^{m} \mathbf{D}_{(j,i)}{}^2 \tag{4.2}$$

Figure 4.4.a shows the projections used for *dictionary mapping*. The Sum of the squares of the projections is used to calculate the metric. More dictionary elements will be concentrated in subspaces where the data is highly distributed. We hypothesize features with high variance should have a higher *dictionary mapping* score. This proposal will try to find mathematical and experimental validation for this claim.

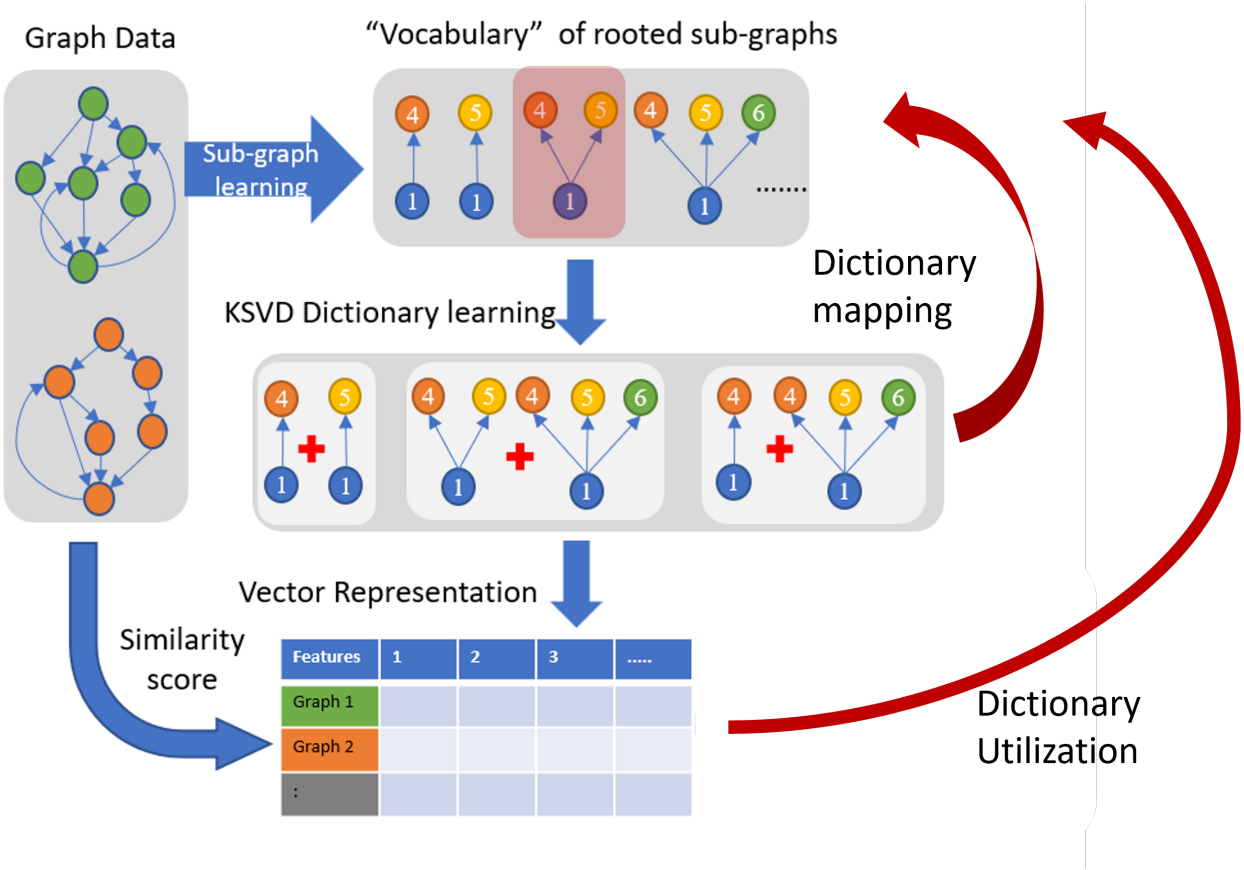


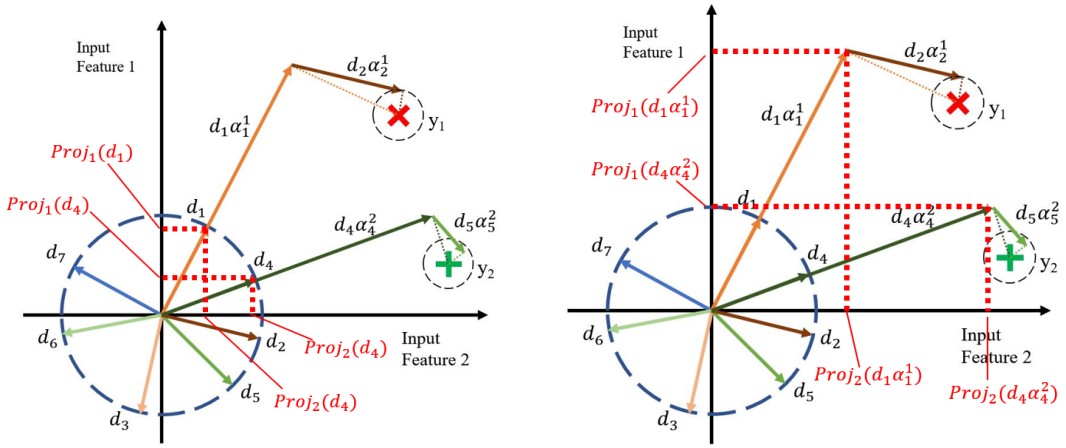
Figure 4.3: Overview of the presented Feature Ranking techniques

Dictionary Utilization

The second metric is the *Dictionary utilization*, $\mathbf{D}_{\text{util}} \in \mathbb{R}^{1\times d}$, which calculated the utilization of the dictionary elements by the sparse coefficients. This acts as a weighted measure of the *dictionary mapping*. Finally, the weighted dictionary atoms are projected back to their original features. The equation is given in (4.3).

$$\mathbf{D}_{\text{util}}(j) = \sum_{i=1}^{m} \underset{j}{\text{Proj}}(d_i \cdot \sum_{k=1}^{n} |\alpha_{j,p}|) \tag{4.3}$$

Figure Figure 4.4.b shows the projections used for *dictionary utilization*. Since we are learning an over-complete dictionary and representing data sparsely, Not all the dictionary



(a) *dictionary mapping* (b) *dictionary utilization*

Figure 4.4: Projection of dictionary elements and weights into input space. (a) *dictionary mapping* only projects the dictionary atoms. (b) *dictionary utilization* projects the weighted dictionary atoms according to the sparse representation.

atoms are used all the time. Hence, understanding which dictionary atoms are used by the sample can give insight into common patterns and rare occurrences. If identify the dictionary atom usage b class, we can develop class-wise feature importance. The more the dictionary atom is used the higher the score for its relevant feature.

Implementation

The presented two metrics are implemented as a python public Python package. hosted at `https://github.com/BMW-lab-MSU/dictionary_feature_ranking`. This package is designed to be compatible with *sci-kit learn* packages API calls. The primary function of the package is to calculate the two feature ranking metrics. It also provides supporting functions to learn a dictionary, calculate sparse coefficients, and compare with other feature ranking metrics. It has the option to learn different dictionaries that are present in *sci-kit* package as well as to learn *ksvd* and *random* dictionaries. The documentation and the instructions are in the Github repository.

The KSVD algorithm learns the dictionary in an unsupervised manner, hence we cannot get dictionary elements that are optimized for class discrimination. Therefore several other methods have been proposed to learn a more discriminative dictionary by learning in a supervised manner, giving us a strong association of dictionary atoms with each class. Here we will be exploring two such methods, LC-KSVD and Frozen KSVD. By doing so we can decompose the proposed metrics into classes, which gives more insight into feature behavior concerning class labels. In LC-KSVD the algorithm enforces two extra constraint terms related to dictionary association with each class and linear classifier performance. Hence, samples are forced to utilize a subset of the dictionary atoms for their representation leading to a more discriminatory dictionary. However, when imbalanced data is presented the LC-KSVD algorithm does not change the dictionary atom distribution. Therefore, Frozen KSVD is proposed to consider class imbalances in dictionary learning. It first learns a dictionary only considering the largest class. Then next largest class is trained with added dictionary atoms while keeping the previously learned dictionary atoms fixed (frozen). This forces the algorithm to learn new dictionary atoms which are associated with only the new class. This is repeated for all the classes such that the added number of dictionary atoms and the sparsity of each class are reduced depending on the class size.

Experiment

The presented FR metrics will be tested and compared according to the different combinations and methods in Table 4.1. Some commonly used FR methods are selected from the *sci-kit* package. For each dataset, the FR methods are independently evaluated and a ranking is produced. Then features are added in order of the ranking and classifiers are trained to observe the performance improvements. Then the performance is compared for the mean accuracies of the existing FR methods to the proposed methods.

Some of the common FR methods that will be considered are mRMR [31], ANOVA

F-value (f_classify)[1], χ^2-test [2], Mutual Information (MI) [3], and Variance Threshold (VT) [4]. These methods will be tested using common datasets[5] such as *iris*, *wine*, *breast cancer*, and *digits*. Each of the considered FR methods evaluates a different aspect of the features and each dataset has a different number of features.

Table 4.1: Feature ranking methods to be evaluated against

Dataset	FR methods	Classifier method
iris	mRMR	Gaus SVM
wine	ANOVA (f _ classify)	Lin SVM
breast cancer	χ^2-test	Quad SVM
digits	Mutual Info (MI)	Pol SVM
	Select K-best	Logistic regression
	Variance threshold (VT)	K-nearest Neighbors (KNN)
	D_{map}	Random Forrest (RF)
	D_{util}	Ridge

Performance Evaluation and Results

We will perform the test with common FR methods and common data sets as shown in table 4.2. As an example *wine* dataset with 13 features is considered henceforth for discussion [56][6] . The *wine* dataset consists of chemical analysis results of 178 wine samples

[1] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html

[2] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_classif.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.VarianceThreshold.html

[5] https://scikit-learn.org/stable/datasets/toy_dataset.html

[6] https://archive.ics.uci.edu/dataset/109/wine

grown in a region in Italy. The wine samples are labeled as three classes which belong to three different wine cultivars. The 13 features are shown in table 4.3. The ranking given by different FR methods is for the *wine* dataset is shown in table 4.4 and figure 4.5. Since different FR methods give scores in various ranges, all the scores are normalized to get a relative score.

Table 4.2: Dataset summary

Dataset	iris	wine	breast cancer	digits
Samples	150	178	569	1797
Features	4	13	30	64
Classes	3	3	2	10
Balanced	Y	N	N	Y
Missing Data	N	N	N	N

Next for each FR method, in the order of the highest feature score, features are added to a custom dataset. These training sets are used to train ML classifiers with holdout validation. Table 4.5, 4.6 shows the accuracy for each ML classifier for the number of features added according to the *dict_map* and *dict_util*FR methods respectively.

Table 4.7 shows the accuracy variation for the Random Forrest classifier with different FR method rankings. Figures 4.6, 4.7, 4.8, and 4.9 show the accuracy variation for Random Forrest classifier for datasets *iris*, *wine*, *breast cancer*, and *digits*. It can be observed that the proposed FR metrics behave as expected and on par with the existing FR metrics.

To quantify the performance comparison, the average accuracy gained for each feature added by existing FR methods is summarized and statistics are calculated. Table 4.8 provides the mean, standard deviation, min, max accuracy reported by existing FR methods. It also provides the accuracy difference from the mean for proposed FR methods. It can be observed

Table 4.3: Features in *wine* dataset

	Features
1	Alcohol
2	Malic Acid
3	Ash
4	Alcalinity of Ash
5	Magnesium
6	Total Phenols
7	Flavanoids
8	Nonflavanoid Phenols
9	Proanthocyanins
10	Colour Intensity
11	Hue
12	OD280/OD315 of diluted wines
13	Proline

that the *dictionary mapping* is almost on par with existing methods and *dictionary utilization* has a slight sub-par performance. Both proposed methods perform worse when a limited number of features are present. This is due to the nature of learned dictionaries. The learned dictionary elements usually span multiple features hence consider them important. The full experiment results can be found at `https://github.com/BMW-lab-MSU/dictionary_feature_ranking`.

Table 4.4: Normalized feature scores given by different FS methods for Wine dataset

	VT	KBest	chi2	f_classif	MI	mRMR	dict_map	dict_util
0	0.0005	0.5774	0.0003	0.5774	0.6905	0.5148	0.0007	0.0187
1	0.0009	0.1579	0.0017	0.1579	0.4333	0.1689	0.0001	0.0042
2	0.0001	0.0569	0.0000	0.0569	0.0974	0.1149	0.0000	0.0033
3	0.0079	0.1529	0.0018	0.1529	0.3464	0.1795	0.0016	0.0258
4	0.0656	0.0531	0.0027	0.0531	0.3080	0.0921	0.0332	0.1374
5	0.0003	0.4007	0.0009	0.4007	0.6227	0.2828	0.0000	0.0029
6	0.0007	1.0000	0.0038	1.0000	1.0000	0.3342	0.0000	0.0033
7	0.0000	0.1179	0.0001	0.1179	0.2033	0.1165	0.0000	0.0005
8	0.0002	0.1294	0.0006	0.1294	0.4364	0.1302	0.0001	0.0021
9	0.0038	0.5158	0.0066	0.5158	0.8345	1.0000	0.0001	0.0073
10	0.0000	0.4331	0.0003	0.4331	0.6259	0.3733	0.0000	0.0013
11	0.0004	0.8121	0.0014	0.8121	0.7880	0.5326	0.0001	0.0039
12	1.0000	0.8888	1.0000	0.8888	0.8283	0.7331	1.0000	1.0000
7	0.0000	0.1179	0.0001	0.1179	0.2033	0.1165	0.0000	0.0005
8	0.0002	0.1294	0.0006	0.1294	0.4364	0.1302	0.0001	0.0021
9	0.0038	0.5158	0.0066	0.5158	0.8345	1.0000	0.0001	0.0073
10	0.0000	0.4331	0.0003	0.4331	0.6259	0.3733	0.0000	0.0013
11	0.0004	0.8121	0.0014	0.8121	0.7880	0.5326	0.0001	0.0039
12	1.0000	0.8888	1.0000	0.8888	0.8283	0.7331	1.0000	1.0000

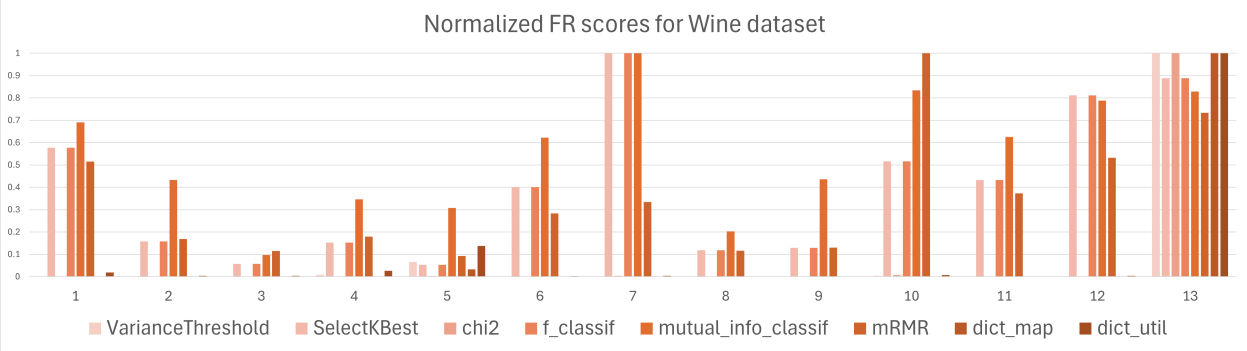


Figure 4.5: Normalized feature scores generated from different FR methods for the wine dataset.

Table 4.5: Classification Accuracy variation with number of features selected according to *Dictionary Mapping* vs different classifiers for Wine dataset

	Ridge	Log. Reg.	SVM	RF	KNN	lSVM	G. SVM	PolySVM	Q.SVM
0	0.777	0.75	0.777	0.638	0.694	0.388	0.777	0.833	0.75
1	0.777	0.722	0.805	0.777	0.722	0.5	0.805	0.833	0.75
2	0.833	0.861	0.805	0.861	0.722	0.666	0.805	0.833	0.75
3	0.888	0.694	0.805	0.916	0.722	0.666	0.805	0.833	0.861
4	0.916	0.861	0.805	0.972	0.722	0.833	0.805	0.8333	0.861
5	0.972	0.916	0.805	0.972	0.722	0.555	0.805	0.833	0.861
6	0.972	0.916	0.805	0.972	0.722	0.944	0.805	0.833	0.861
7	1.0	0.944	0.777	0.972	0.722	0.916	0.777	0.833	0.861
8	1.0	0.972	0.777	1.0	0.722	0.916	0.777	0.833	0.861
9	1.0	0.972	0.777	1.0	0.722	0.805	0.777	0.833	0.861
10	1.0	0.972	0.777	1.0	0.722	0.916	0.777	0.833	0.861
11	1.0	0.972	0.777	1.0	0.722	0.944	0.777	0.833	0.861
12	1.0	0.972	0.805	1.0	0.722	0.666	0.805	0.833	0.861

Table 4.6: Classification Accuracy variation with number of features selected according to *Dictionary Utilization* vs different classifiers for Wine dataset

	Ridge	Log. Reg.	SVM	RF	KNN	lSVM	G. SVM	PolySVM	Q.SVM
0	0.777	0.75	0.777	0.666	0.694	0.222	0.777	0.833	0.75
1	0.777	0.722	0.805	0.777	0.722	0.555	0.805	0.833	0.75
2	0.833	0.861	0.805	0.888	0.722	0.666	0.805	0.833	0.75
3	0.888	0.694	0.805	0.916	0.722	0.611	0.805	0.833	0.861
4	0.916	0.861	0.805	0.944	0.722	0.833	0.805	0.833	0.861
5	0.916	0.916	0.805	0.972	0.722	0.833	0.805	0.833	0.861
6	0.972	0.916	0.805	0.944	0.722	0.916	0.805	0.833	0.861
7	1.0	0.972	0.777	1.0	0.722	0.888	0.777	0.833	0.861
8	1.0	1.0	0.777	1.0	0.722	0.638	0.777	0.833	0.861
9	1.0	1.0	0.777	1.0	0.722	0.861	0.777	0.833	0.861
10	1.0	0.972	0.777	1.0	0.722	0.638	0.777	0.833	0.861
11	1.0	0.972	0.777	1.0	0.722	0.861	0.777	0.833	0.861
12	1.0	0.972	0.805	1.0	0.722	1.0	0.805	0.833	0.861

Table 4.7: Accuracy improvement with the number of selected features for Wine dataset with RandomForrest Classifier

Features	chi2	**dict_map**	**dict_util**	f_classif	mRMR	mut. info.	KBest	VT
0	0.666	0.638	0.666	0.666	0.722	0.638	0.638	0.666
1	0.916	0.777	0.777	0.888	0.916	0.944	0.916	0.75
2	0.944	0.861	0.888	0.888	0.944	0.944	0.916	0.861
3	0.944	0.916	0.916	0.888	0.944	0.944	0.916	0.888
4	0.944	0.972	0.944	0.972	0.944	0.972	0.972	0.916
5	0.972	0.972	0.972	1.0	0.972	1.0	1.0	1.0
6	1.0	0.972	0.944	1.0	1.0	1.0	1.0	1.0
7	1.0	0.972	1.0	1.0	0.972	1.0	1.0	1.0
8	0.972	1.0	1.0	1.0	1.0	1.0	1.0	1.0
9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 4.8: Classification Accuracy Statistics of *Other* FR methods compared to the difference from mean for *Dict_util* and *Dict_map* for Wine dataste with Random Forest classifier

Features added	mean	std	min	max	dict_util diff	dict_map diff
0	0.666	0.030	0.63	0.722	0.0	-0.0277
1	0.888	0.070	0.75	0.944	-0.111	-0.1111
2	0.916	0.035	0.861	0.944	-0.027	-0.0555
3	0.921	0.027	0.888	0.944	-0.004	-0.0046
4	0.953	0.022	0.916	0.972	-0.009	0.01851
5	0.990	0.014	0.972	1.0	-0.018	-0.018
6	1.0	0.0	1.0	1.0	-0.055	-0.0277
7	0.995	0.011	0.972	1.0	0.0046	-0.0231
8	0.995	0.011	0.972	1.0	0.0046	0.0046
9	1.0	0.0	1.0	1.0	0.0	0.0
10	1.0	0.0	1.0	1.0	0.0	0.0
11	1.0	0.0	1.0	1.0	0.0	0.0
12	1.0	0.0	1.0	1.0	0.0	0.0

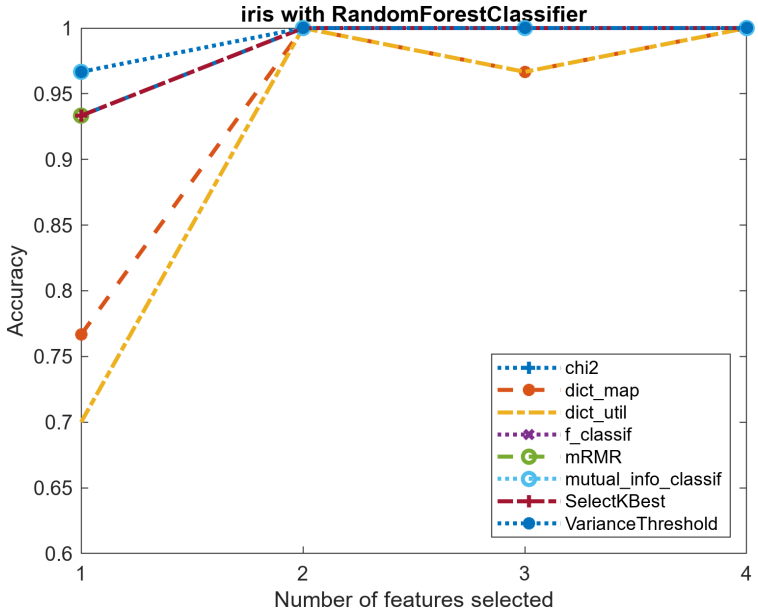


Figure 4.6: Accuracy improvement with the number of selected features for Iris dataset with RandomForrest Classifier

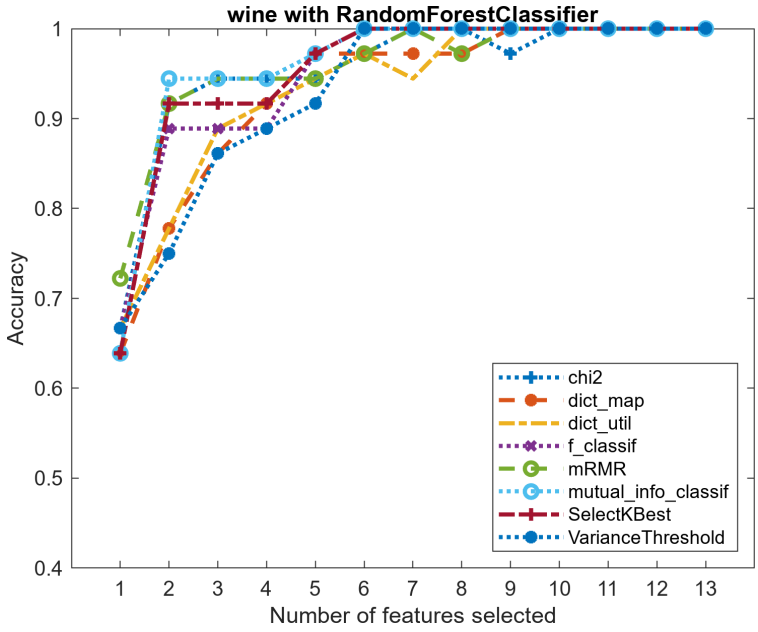


Figure 4.7: Accuracy improvement with the number of selected features for Wine dataset with RandomForrest Classifier

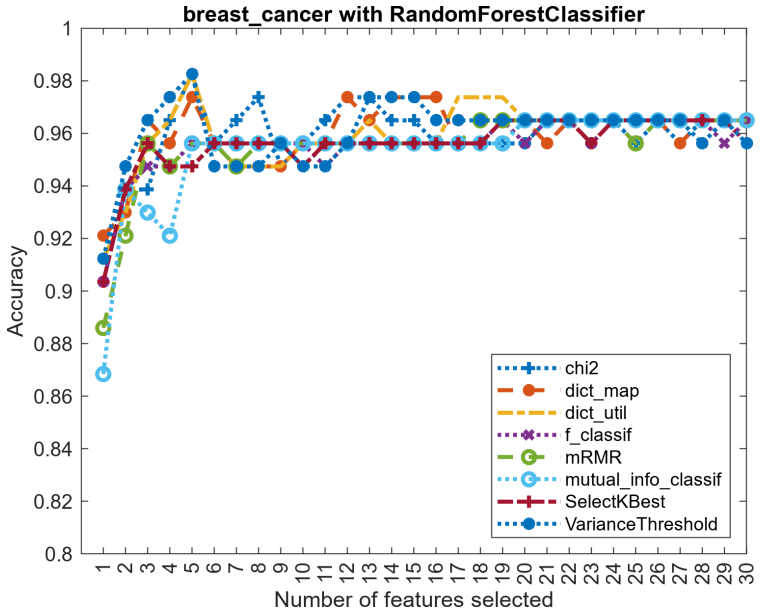


Figure 4.8: Accuracy improvement with the number of selected features for Breast cancer dataset with RandomForrest Classifier

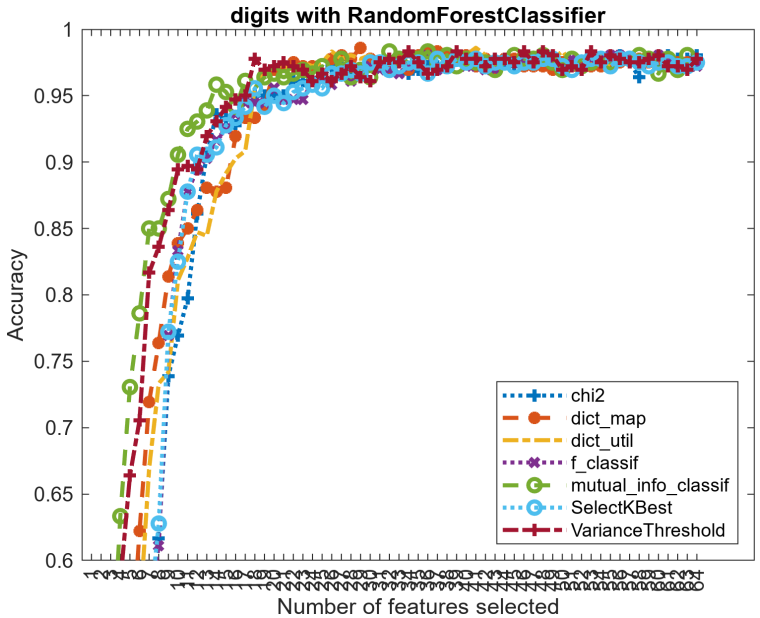


Figure 4.9: Accuracy improvement with the number of selected features for Digits dataset with RandomForrest Classifier

Application in Satellite Image Classification

Satellite image classification is a challenging problem in land survey and management due to the high variability and associated noise of satellite/airborne imaging. In recent years several deep learning methods have been proposed that achieve very high accuracies. However, deep learning methods lack an intuitive relationship between the learned features and the input layer. Also, they require significant computational resources for training and testing. The preliminary work done with the dataset was published in [34].

Dataset and Features

The satellite data set used in this proposal was created by the authors in [57]. It contains labeled satellite image patches of size 28×28 pixels of the continental United States. Images consist of four bands: Red, Green, Blue, and Near Infrared (NIR). The complete data set is divided into two subsets. Sat-4 and Sat-6 contain four and six classes, respectively. In this paper, we only consider the Sat-4 dataset. Fig. 4.10 shows the first 25 sample images of the training set, along with their respective class labels.

Authors in [57] analyzed 150 total features using the Distribution Separability Criterion (DSC) and identified 22 prominent features. These features include the mean, standard deviation, 2^{nd} moment, and variance of Hue (H), Saturation (S), and Intensity (I). The color co-occurrence matrix (CCM, described in [58]) is calculated for each H, S, and I. Various statistics are calculated using each CCM, including the sum of squares for variance (SSOVH), auto-correlation (AUTOC), covariance, as well as the mean, standard deviation, 2^{nd} moment, and variance. Several established vegetation indices are used to differentiate vegetation. Which include Simple Ratio or Ratio Vegetation Index (SR or RVI) [59], Atmospherically Resistant Vegetation Index (ARVI) [60], Normalized Vegetation Index (NDVI) [61], and Enhanced Vegetation Index(EVI) [62]. Finally, the Discrete Cosine Transformation (DCT)

Table 4.9: DeepSat dataset properties

Parameters	**Sat-4**	**Sat-6**
Total images	500,000	405,000
Classes	4	6
Class names	1) Barren Lands	1) Barren Lands
	2) Trees	2) Trees
	3) Grasslands	3) Grasslands
	4) Others (none)	4) Roads
		5) Buildings
		6) Water bodies
Training Data	400,000	324,000
Testing Data	100,000	81,000

is calculated for the image.

Our work utilizes the same 22 features as in [57], with some modifications. For a given image, we use the mean values of the vegetation indices. We use the second-highest frequency component position of the DCT, as this value contains the most discriminatory information. The feature vector obtained from each image, F_i, has a dimension of 22×1. This small dimension significantly reduces the computational workload of sparse coding and classification while maintaining high accuracy.

In addition to modifying some of the extracted features, another significant difference in our feature analysis involves the normalization process. The authors in [57] normalized the features using the whole set of images (both training and testing). Since this would assume the availability of the testing images at the time of the model training, our work normalized

Table 4.10: Handcrafted features for the Sat-4 & Sat-6 data.

	Indices		**Indices**
1	I CCM mean	12	I std
2	H CCM sosvh	13	H std
3	H CCM autoc	14	H mean
4	S CCM mean	15	I mean
5	H CCM mean	16	S mean
6	Simple Ratio (SR) mean	17	I CCM covariance
7	S CCM 3rd moment	18	NIR mean
8	I CCM 3rd moment	19	Atmospherically Resistant Vegetation Index (ARVI) mean
9	I 3rd moment	20	Normalized Vegetation Index (NDVI) mean
10	I variance	21	DCT highest non-DC component
11	NIR std	22	Enhanced Vegetation Index (EVI) mean

the features using only the *min* and *max* values of the training set. A five percent buffer is set to the *min* and *max* values when normalizing to account for possible variability in the testing image features. When normalizing, the 2^{nd} moment and the variance become the same. Hence, the 3^{rd} moment is used instead of the 2^{nd} moment. The features are then converted to a logarithmic scale since we have observed that the data is more concentrated toward the origin. This leads to more distributed features and increased classification accuracy. The final list of features is listed in table 4.10.

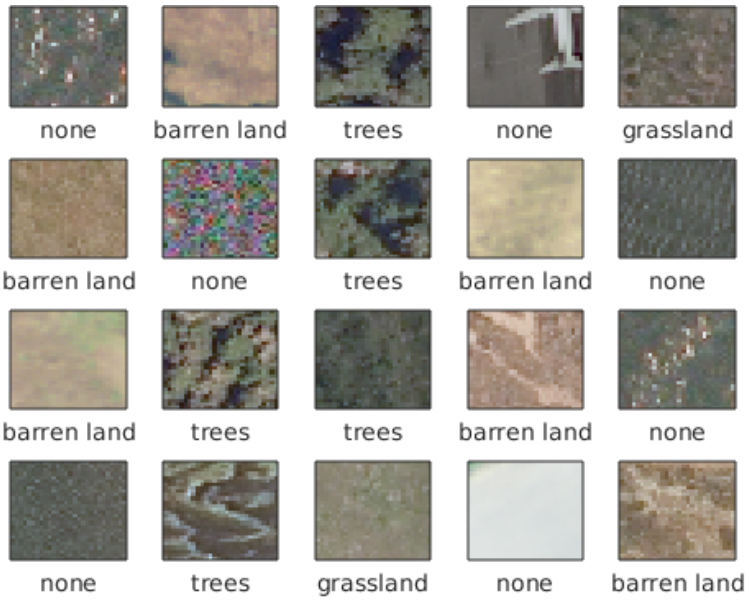


Figure 4.10: Sample images from Sat-4 Data set.

Dictionary Elements Analysis

When we examine the dictionary atoms used by correctly and incorrectly classified images it can be observed that incorrectly classified images share some of the dictionary atoms that are mostly allocated to the other class. When we transform these dictionary elements back to the feature space we can identify the features that give similar parameters to the two classes, indicating that these features are not able to distinguish between the two classes. Hence, we can decide whether to replace the features or add new features that are capable of separating the two classes. Since our feature domain is considerably small this will not drastically increase the computational burden of the system for this example. This ability to identify the features responsible for misclassifications is not intuitively simple with deep networks and other FR methods.

As an example, we can take a closer look at the most prevailing confusion between classes *barren land* and *grassland* in the Sat-6 dataset from the Frozen dictionary model. Most of the misclassified images belonging to these classes contain soil and grass patches.

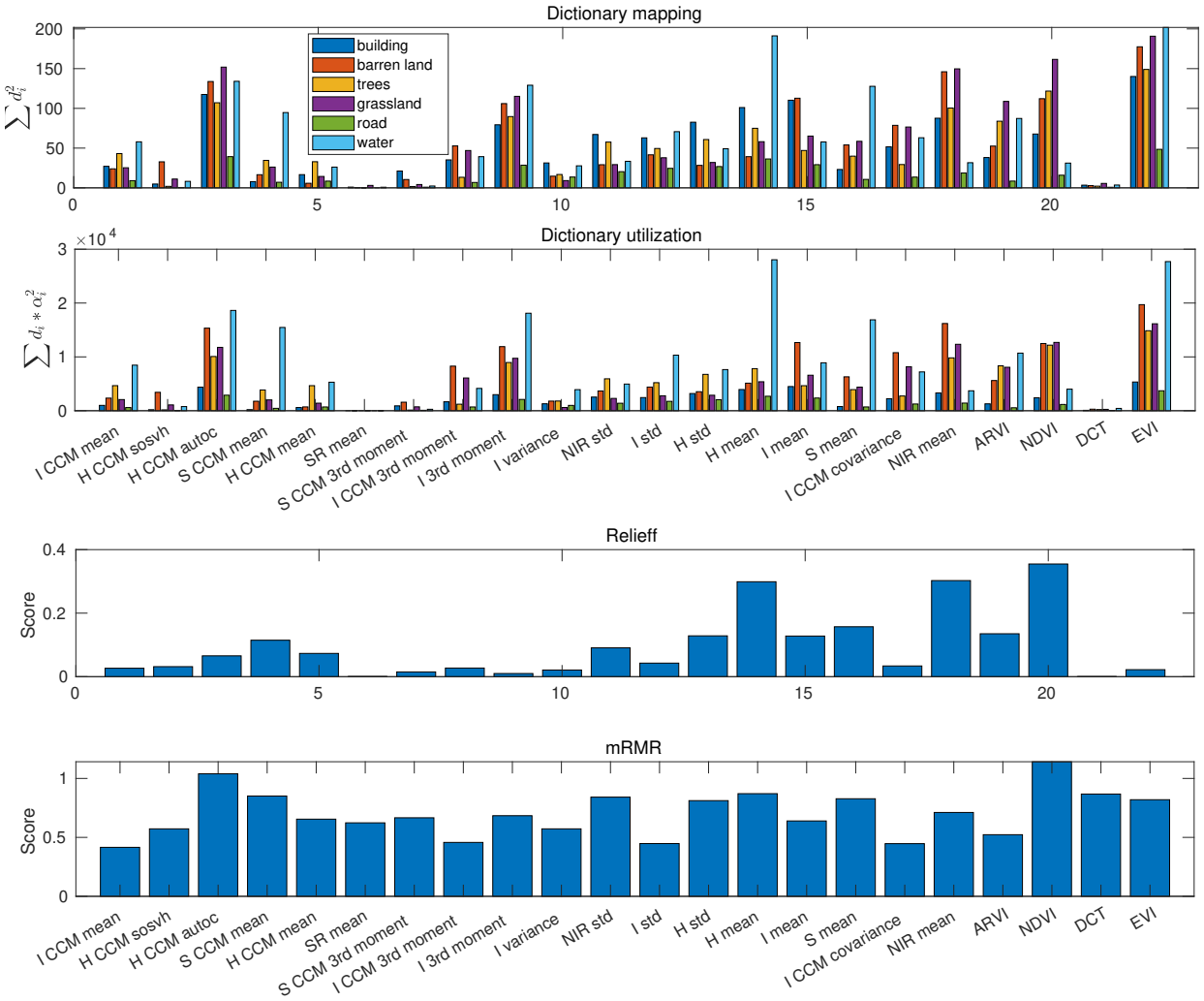


Figure 4.11: FR scores obtained using various methods on the Sat-6 dataset. The dictionary scores were obtained using frozen KSVD sparse representation.

For brevity, we do not present images of the misclassified data points, but the difficulty in producing accurate ground-truth labels likely contributes to confusion and lower accuracy. This can be observed also in the misclassified *building* class, where many of the images misclassified as *roads* were parking lots. This, combined with the fact that *roads* have very small training data, would accumulate to the low performance of the *road* class. While a similar analysis could be done for images classified with a deep network, the

linearity employed by dictionary learning and classification in this work allows for a more comprehensive analysis.

Fig.4.11 shows the *dictionary mapping* and the *dictionary utilization* of the sparse coefficients used by dictionary elements projected into the original feature space from the frozen KSVD method on Sat-6 data. Since each dictionary atom is associated with a class, the class-wise metric analysis is performed on the original features. It also shows the FR done by the non-class-specific Releiff and mRMR methods. Both proposed metrics perform similarly. While their scores are closer to the *Relieff* method they are also able to capture some important features predicted by *mRMR* method (e.g. *H CCM autoc* and *EVI*). The proposed methods also match the *Relieff* method in identifying SR mean and DCT as underutilized features.

The shortcomings of SR are well recorded in cases where an abundance of bare soil is present, and indices like Soil Adjusted Vegetation Indexes (SAVI) can be used to mitigate this problem [63]. Due to the presence of similar texture in both *barren land* and *grassland* classes, its DCT components also give similar results. Since we have used a modified and more statistical version of the 22 features selected by [57], some features do not perform well with the sparse model. Using sparse coding and linear classifiers, we can identify, evaluate, and improve the necessary input features that might increase the performance of the classification.

Concerning classification performance between the two SR methods used, Frozen KSVD performs slightly better than the LC-KSVD in the Sat-6 dataset. This is possibly due to the strategy Frozen KSVD employs in taking class sizes into account and learning dictionary elements hierarchically. This enables the algorithm to produce dictionary atoms that specifically model the smaller classes. For the Sat-4 dataset, Frozen KSVD performs notably better than LC-KSVD. This may be due to the high variability of the *none* class.

Conclusion

In this work, we have presented two metrics for feature ranking. Said metrics were compared with some popular FR methods to demonstrate that the ranking yields similar results. These metrics are ML model-independent and can thus be used with a wide variety of classifier models. Additionally, with the class decomposition of the metrics, better insight can be gained into the model behavior. As a demonstration, metrics were used with satellite/ airborne scene classification using a sparse representation-based method with the use of linear classifiers. Two discriminative dictionary learning methods, Frozen KSVD and LC-KSVD were tested. We were able to achieve relatively high-end results using an ensemble of simple linear classifiers. The main advantage of utilizing these methods is the ability to gain an intuitive understanding of the relationship between the learned sparse coefficients and the original feature space. We have shown an example that analyses the backward interpretation of the learned sparse coefficients and the classifier to the original feature space which enables an intuitive analysis of the model.

Potential Limitations

Although preliminary results show similar results to the existing FR method the presented methods have to be validated both theoretically and experimentally. There is still unknown behavior with different types of data. Especially the effect of assumptions like the sparsity of the original data has to be investigated. Further, these metrics are sensitive to data pre-processing operations such as normalization, shrinkage, and mapping. Currently, these metrics cannot be used with non-numeric and discrete data. We are hoping to conduct experiments to answer the above questions. Ultimately researchers can use these metrics as a guide to decide which features to alter for machine learning applications.

MALWARE DETECTION USING CONTROL FLOW GRAPHS (CFG)

The majority of the work reported in this section is published as *Improving Malware Detection from Binary Control Flow Graphs Using Supervised Learning* by Portillo et al. in the *2024 Intermountain Engineering, Technology and Computing (IETC)* conference [64].

Introduction

In recent years the number of attacks by malware shared through the internet has increased. To avoid damage to the computer systems effective identification and classification of malicious programs from benign programs are necessary. This chapter will focus on the analysis of binary files using control flow graphs (CFGs) and exploring the viability of using unsupervised clustering and supervised classification techniques for detecting Malware. It has not been established that there is a significant difference between CFGs belonging to malware and benign programs. Hence, we hypothesize that there is a significant difference between the graph structures of malware and benign programs.

This research is funded through the Department of Homeland Security (DHS) under the project "Cyber QR Ops: Improving the Quality and Resiliency of Critical Computing Infrastructure". The research focus is to explore the viability of ML as an effective tool in identifying malware through graph analysis of CFGs. This work is an extension of the preliminary work done by Veronika et al. [65]. An extended discussion on the dataset, graph generation, and malware types is reported in the master thesis by Pearsall [66]. The initial experiments are published in a reproducible code capsule at `https://codeocean.com/capsule/6978226/tree`.

Goals and Contributions

- **Benign and Malware dataset.** As the initial step, we would curate a dataset of benign and malicious programs. The dataset consists of dataset operating system files as benign and malware dataset provided by *Hoplite industries*[1]. This would allow us to explore real malware and benign files. Publicly available combined (Benign + Malware) datasets are rare, hence this would help future researchers to test and evaluate new algorithms. The Generated CFGs can be published publicly for a wider audience.

- **A framework for unsupervised clustering and supervised classification.** A robust end-to-end workflow is proposed to minimize model over-fitting and model contamination. The framework is flexible for swapping and trying different methods for research purposes.

The overview of the typical analysis is shown in figure 5.1. the binary files are converted into CFGs. This is carried out by *CFGEmulated()* function of the *angr* python library. Then the graphs are converted to a vector representation using *Graph2Vec* algorithm[12]. It can be seen that usually a dictionary of rooted sub-graphs is learned and then the similarity is calculated between the sub-graphs and the main graph representation. Typical implementations of the *Graph2Vec* are shown in figure 2.3, where the graphs are first converted into a word document using the Weisfeiler-Lehman graph kernel [13] and then uses *Doc*2*Vec* [14] to create a fixed length vector representation of the graph embedding. Finally, several unsupervised clustering techniques were used on the dataset. This allows for determining the viability of using CFGs as a static malware detection method with unsupervised clustering. The source code and the necessary files will be available at `https://github.com/MSUSEL/unsupervised-graph`.

[1] `https://www.hopliteindustries.com/`

The developed WL+KSVD and the Dictionary-based FR metrics for unsupervised clustering and important sub-tree structure identification. The preliminary work seems to show some distinct grouping for each category. Hence, identifying which sub-tree structures or a combination of structures leads to this grouping is helpful. If we can identify sub-tree structures associated with applications, we can identify vulnerable/harmful coding practices.

The hypothesis is that the *Malware and Benign CFGs have some distinct sub-tree structures associated with each category.* To test the hypothesis, the methods discussed in Chapter 3 and 4 will be utilized with other graph embedding and feature ranking metrics.

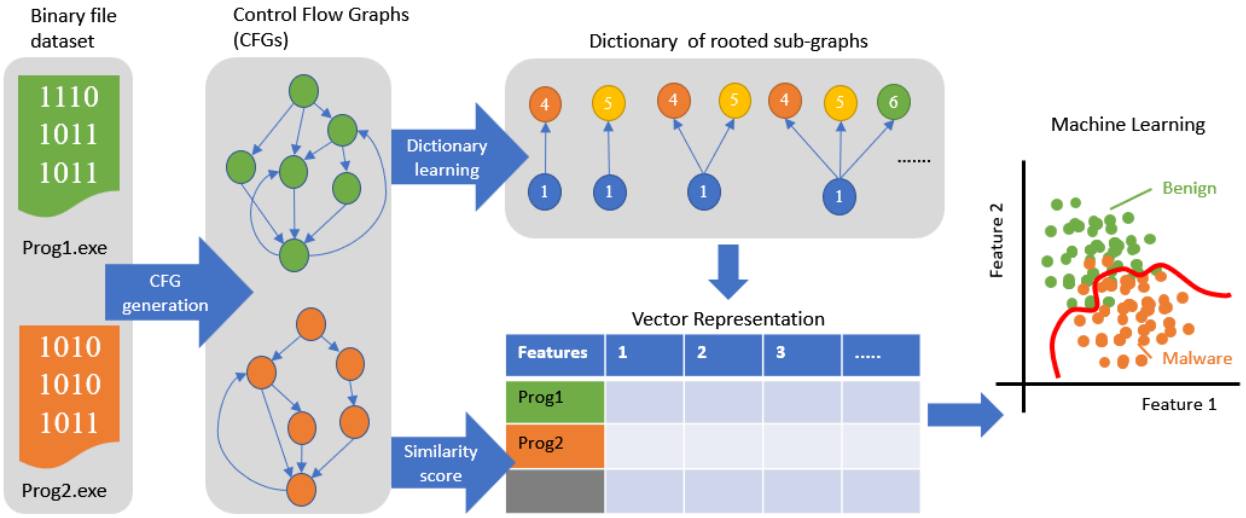


Figure 5.1: Overview of binary file analysis using CFGs

Dataset

The dataset consisted of 3000 malicious files and 3000 benign files. Malware samples were provided for this research by local cybersecurity company Hoplite Industries [2]. This collection of malware is an aggregation of samples collected from VirusTotal.com[3], honeypots, and personal cybersecurity connections. The collection of malware includes several different

[2]https://www.hopliteindustries.com/
[3]https://www.virustotal.com

types of malware, such as trojans, viruses, and droppers. The malware samples consisted of many different file types, but only PE executable and ELF binaries were evaluated in this research. For benign programs, operating system files were collected from versions of Windows and Linux. This includes trusted DLL files and benign Linux executables. The summary of types of malware used is shown in table 5.1, where the types are highly unbalanced.

Control Flow Graph Generation

CFG generation was done using the Python library from *angr*. Binary files are first disassembled (using the *angr* python library [37]) into a control flow graph structure using the symbolic execution method. This disassembles binary files into an assembly-level language whose execution structure can be more readily observed [67]. Once in assembly, code blocks with their respective address jumps are discernible and become graph nodes and links respectively. Figure 5.2 is an image of such an interpretation generated using the *angr-management* python package. *Angr* provides two different methods for generating control flow graphs: *CFGFast*, and *CFGEmulated*[4]. *CFGFast* is a static analysis method where the program is evaluated in random positions. *CFGFast* accuracy is theoretically bounded but much faster. *CFGEmulated* is a dynamic analysis method where the program is emulated using symbolic execution to identify the flow paths. Although more accurate than *CFGFast*, this method takes more time. Further, the accuracy is bounded by emulation restraints like missing hardware modules, system calls, and the choice of hyperparameters. For this proposal, we have chosen *CFGEmulated* as the CFG generation method. This is because we have prioritized the accuracy of the CFGs that are generated rather than the time it takes to generate them. One of the parameters for the *CFGemulated* is the context sensitivity. In this proposal, we chose context sensitivity level 3.

[4] `https://docs.angr.io/built-in-analyses/cfg`

Table 5.1: Types of malware in the dataset

Type	Train	Test	Type	Train	Test
adware	443	163	krypt	1	0
artemis	1	1	kryptik	1	1
banker	3	1	midgare	0	1
bechiro	0	1	morstar	3	3
bundler	1	1	pua	5	2
bundlore	3	1	qakbot	1	0
casino	1	0	ransomware	2	0
cinmus	1	0	sefnit	1	0
dialer	0	1	softpulse	3	1
domaiq	8	3	solimba	8	4
downloader	44	16	squarenet	1	0
dropper	2	10	suspect	1	0
fakeav	0	2	symmi	1	0
eurezo	1	0	susppack	0	1
file	1	0	trojan	1792	652
firseria	2	1	ulpm	1	0
firseriainstaller	4	0	unwanted	1	0
hacktool	5	0	virus	114	49
installcore	5	2	worm	202	68
installerex	1	0	Unspecified	333	23
kazy	2	0			

In figure 5.2, each node contains a lot of information like, memory address, functional name, *assembly* instructions, instruction values, instruction sequence, and number of instructions. Ideally, all of the information available in a node should be utilized for proper

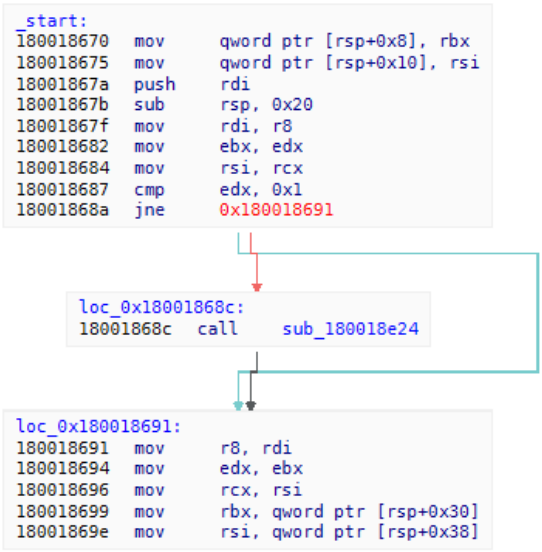


Figure 5.2: A portion of the CFG generated by the *angr-management* tool of a Windows system file *wintrust.dll*

analysis. However, this means that each subtree pattern could become a unique structure which increases the number of input features. In this thesis, to identify common subtree structures, All the node information is discarded and the nodes are replaced with a unique integer number. This skeleton version of graphs is combined to create a CFG graph dataset. The CFG dataset is archived on Zenodo[5]. Strategies for incorporating the node information is discussed in future works section.

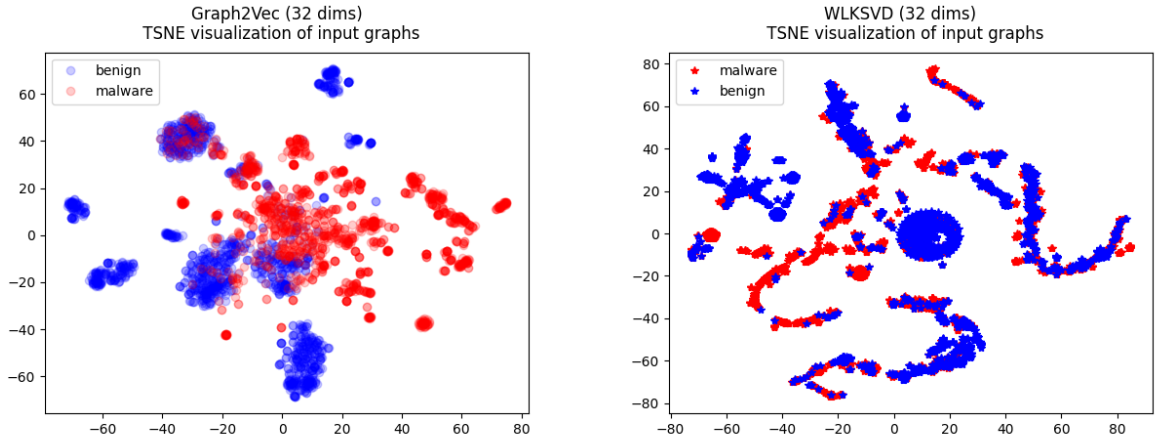
Graph Embedding

The generated graphs have to be represented with a fixed-length vector. First WL hash relabelling is conducted for each graph independently. This is computationally expensive since, for each graph, an exhaustive search is conducted. However, this step can be conducted independently for each graph. Two iterations of the WL hash are chosen as a compromise

[5]https://doi.org/10.5281/zenodo.7630371

of computational cost and sub-tree structure.

Then a portion of the training set is used to build the vocabulary and train the *Doc2Vec* model. The model dimensions considered are $(2, 4, 8, 16, 32, 64, 128, 256)$. Next, the trained *Doc2Vec* model is used to infer the training set for the unsupervised clustering. As an example fig. 5.3 shows the 2d tSNE representation. Some clear groupings of CFGs can be observed in the figures. Some groupings seem to consist of the majority class. However, there is a significant overlap also present in the figures. Further investigation is required to identify which features or CFG sub-tree patterns correspond to each grouping.



(a) (b)

Figure 5.3: Two dimensional t-sne visualization of 32-dimensional *Graph2Vec* and *WL+KSVD* embedding

Experiment

Unsupervised Clustering

To identify the clusters in the generated vectors, four unsupervised clustering methods were employed. For each method to find the hyper-parameters a grid search is conducted with hold-out validation. Several clustering evaluation metrics were measured for each clustering

algorithm to determine the "best" hyper-parameters. Since this is not an exhaustive search "best" is among the considered options. The considered cluster evaluation metrics are the RAND index, Adjusted RAND index (ARI), Mutual information score (MIS), Adjusted mutual information score (AMIS), Normalized mutual information (NMI), Homogeneity (Hmg), Completeness (Cmp), V-measure (V-meas)[6].

Finally, the clustering model trained using the best hyper-parameters are utilized to predict the clustering for the training test.

Results The Hyper-parameter tuning on the validation data for WL+KSVD 128 dimensional embedding for K-means algorithm is shown in table 5.2. This process is repeated for all the dimensions considered for both *WL+KSVD* and *Graph2Vec*. The best-performing hyperparameter is chosen for each clustering method. The best parameter predicted by each metric is shown in Table 5.3 , 5.4 for *Graph2Vec* and *WL+KSVD* embeddings respectively.

The choice of which metric represents the best clustering decision must be studied further. Table 5.5, and Table 5.6 list the contingency matrices for the validation dataset using 64 dim 3 clusters and 128 dim 5 clusters respectively. If we want to incorporate the labels, we can assign manual labels to each cluster to evaluate the test set. As the number of clusters increases, intuitive manual labeling is not feasible. Hence, we limited the maximum number of clusters to 30.

It can be observed that even with the best parameters some groupings have significant overlap. This means our vector representation must be further improved to achieve better results. Finally, the clustering model trained using the best hyper-parameters are utilized to predict the clustering for the training test. Considering the above predictions 128 dimensions with 5 clusters were chosen for K-means clustering. Figure 5.4 shows the ground truth of the test dataset and the predicted classes with the K-means algorithm. Table 5.7 shows

[6] `https://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation`

Table 5.2: Hyperparameter tuninig with number of clusters for K-means clustering for 128 dimenstion *WL+KSVD* embedding

n_clusters	RAND	ARAND	MIS	AMIS	NMIS	Hmg	Cmplt	V_meas	FMs
2	0.500	0.001	0.011	0.027	0.027	0.015	0.135	0.027	0.697
3	0.500	0.000	0.006	0.016	0.017	0.009	0.109	0.017	0.700
4	0.500	0.000	0.016	0.037	0.038	0.023	0.108	0.038	0.688
5	0.500	0.001	0.018	0.038	0.040	0.025	0.099	0.040	0.685
6	0.500	0.001	0.027	0.056	0.059	0.039	0.118	0.059	0.678
7	0.501	0.002	0.022	0.046	0.049	0.031	0.113	0.049	0.684
8	0.500	0.001	0.031	0.060	0.063	0.045	0.103	0.063	0.669
9	0.500	0.001	0.032	0.063	0.066	0.046	0.116	0.066	0.673
10	0.500	0.001	0.030	0.058	0.061	0.044	0.102	0.061	0.670
11	0.500	0.001	0.033	0.061	0.065	0.047	0.103	0.065	0.668
12	0.500	0.001	0.030	0.055	0.059	0.043	0.096	0.059	0.669
13	0.501	0.002	0.036	0.065	0.069	0.052	0.103	0.069	0.665
14	**0.519**	0.038	0.080	0.114	0.116	0.115	0.117	0.116	0.599
15	0.501	0.003	0.042	0.071	0.076	0.061	0.102	0.076	0.657
16	0.501	0.002	0.036	0.063	0.068	0.052	0.098	0.068	0.663
17	0.501	0.002	0.034	0.061	0.065	0.049	0.097	0.065	0.665
18	0.501	0.002	0.036	0.062	0.067	0.051	0.097	0.067	0.663
19	0.501	0.002	0.040	0.067	0.073	0.058	0.098	0.073	0.659
20	0.502	0.004	0.050	0.075	0.081	0.073	0.090	0.081	0.640
⋮									

Table 5.3: Best hyper-parameters predicted through different metrics for *Graph2Vec* embeddings

	K-means		Spectral		Agglomerative		DBSCAN	
Metric	dim	# Clusters	dim	# Clusters	dim	# Clusters	dim	density
RAND	128	5	8	19	128	2	8	0.80
ARAND	128	5	8	19	128	2	8	0.80
MIS	64	29	4	28	128	26	64	0.25
AMIS	64	3	8	14	128	2	8	0.75
NMIS	64	3	8	14	128	2	8	0.75
Hmg	64	29	4	28	128	26	64	0.25
Cmplt	64	3	8	14	128	2	2	0.45
V_meas	64	3	8	14	128	2	8	0.75
Best	128	5	8	14	128	2	8	0.75

the contingency matrix for the test results. As seen the methodology was able to perfectly group cluster 1 and cluster 2 for only benign data. About 62% of the benign programs can be safely grouped.

Table 5.8 and 5.4 shows the evaluation scores for the test set for all the clustering algorithms with the best hyperparameters for *Graph2Vec* and *WL+KSVD* respectively. The evaluating metrics do not have high values and *WL+KSVD* performance is significantly lower than expected. The two-dimensional representation of clustering on the test dataset is shown in Figure 5.5, Figure 5.6, and Figure 5.8 with Agglomerative, Spectral, and DBSCAN algorithms.

Table 5.4: Best hyper-parameters predicted through different metrics for *WL+KSVD* embeddings

	K-means		Agglomerative		DBSCAN	
Metric	dim	# Clusters	dim	# Clusters	dim	density
RAND	16	26	256	26	4	0.05
ARAND	16	26	256	26	4	0.05
MIS	16	26	16	28	512	0.05
AMIS	256	27	256	26	4	0.05
NMIS	256	27	256	26	512	0.05
Hmg	16	28	16	27	512	5
Cmplt	8	6	64	7	4	0.05
V_meas	256	26	256	26	512	0.05
Best	16	26	256	26	4	0.05

Table 5.5: Contingency matrix for the validation set with 64 dimensions and 3 clusters for *Graph2Vex*

Clusters No	1	2	3
Benign	493	412	34
Malware	619	3	359
Cluster label (manual)	Suspicious	Benign	Malware

Table 5.6: Contingency matrix for the validation set with 128 dimensions and 5 clusters

Cluster	1	2	3	4	5
Benign	44	374	167	14	340
Malware	0	0	87	383	511
Cluster label (manual)	Benign	Benign	Sus.	Malware	Malware

Table 5.7: Contingency matrix for test set with 128 dimensions 5 clusters.

Clusters No	1	2	3	4	5
Benign	89	531	113	13	254
Malware	0	0	90	401	509
Total	89	531	203	414	763

Supervised Classification

Three classifiers were trained for each model: a Linear Support Vector Classifier, a Random Forest, and a Histogram Gradient Boosting Classifier [68][7]. All were tuned empirically and according to each vector model by its given dimensions to maximize accuracy. A Linear Support Vector classifier is a support vector machine that attempts to insert a hyperplane to divide data into classes. The shape of the decision boundary can vary according to the chosen kernel but in the case of the Linear Support Vector classifier, a hyperplane is used as the boundary. A Random Forest Classifier is a bagging ensemble method classifier that trains decision tree classifiers on random subsets of the given data to build robust final collections that are not over-fitted to the given data. Final classification decisions

[7] `https://scikit-learn.org/stable/modules/ensemble.html`

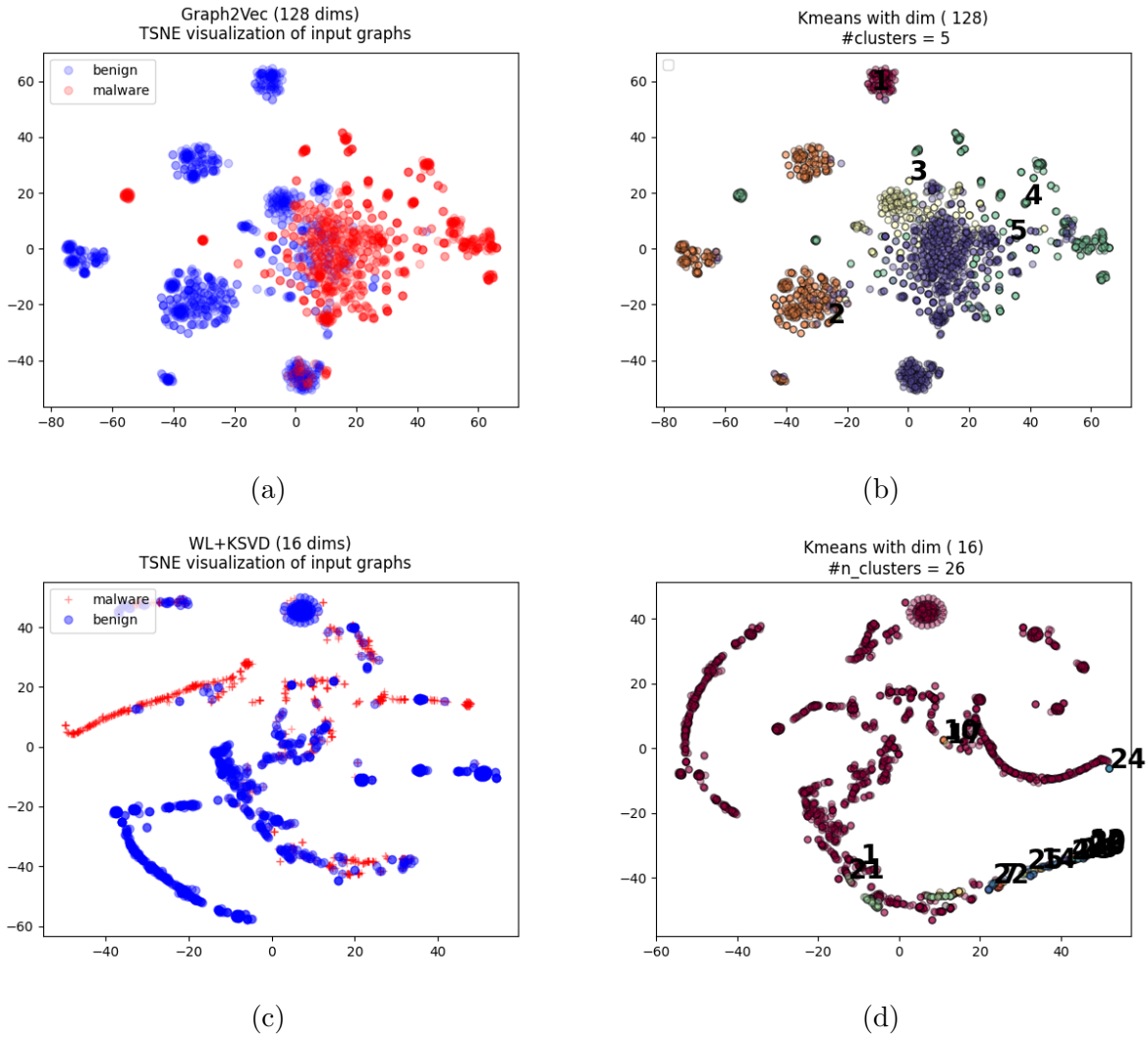


Figure 5.4: K-means clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions. Top row is the *Graph2Vec* and bottom row is *WL+KSVD*

are made by taking an average of the decisions made by individual trees. A Histogram Gradient Boosting Classifier is a boosting ensemble method that sequentially trains decision tree classifiers where the current tree being trained is trained on the data that previous trees performed most poorly on. In this implementation, the negative gradient of the loss function is computed and the next tree is trained while attempting to minimize this gradient. In this

Table 5.8: Evaluation metric scores for Test dataset with best hyper-parameters for *Graph2Vec*

Metric	K-means	Spectral	Agglomerative	DBSCAN
RAND	0.626	0.611	0.687	0.521
ARAND	0.252	0.221	0.374	0.042
MIS	0.352	0.333	0.282	0.323
AMIS	0.333	0.318	0.431	0.144
NMIS	0.333	0.321	0.431	0.194
Hmg	0.507	0.480	0.407	0.467
Cmplt	0.248	0.240	0.458	0.122
V_meas	0.333	0.321	0.431	0.194

Table 5.9: Evaluation metric scores for Test dataset with best hyper-parameters for *WL+KSVD*

Metric	K-means	Agglomerative	DBSCAN
RAND	0.502	0.503	0.516
ARAND	0.005	0.007	0.0326
MIS	0.014	0.041	0.212
AMIS	0.030	0.061	0.046
NMIS	0.032	0.075	0.142
Hmg	0.021	0.059	0.306
Cmplt	0.065	0.102	0.092
V_meas	0.044	0.075	0.142

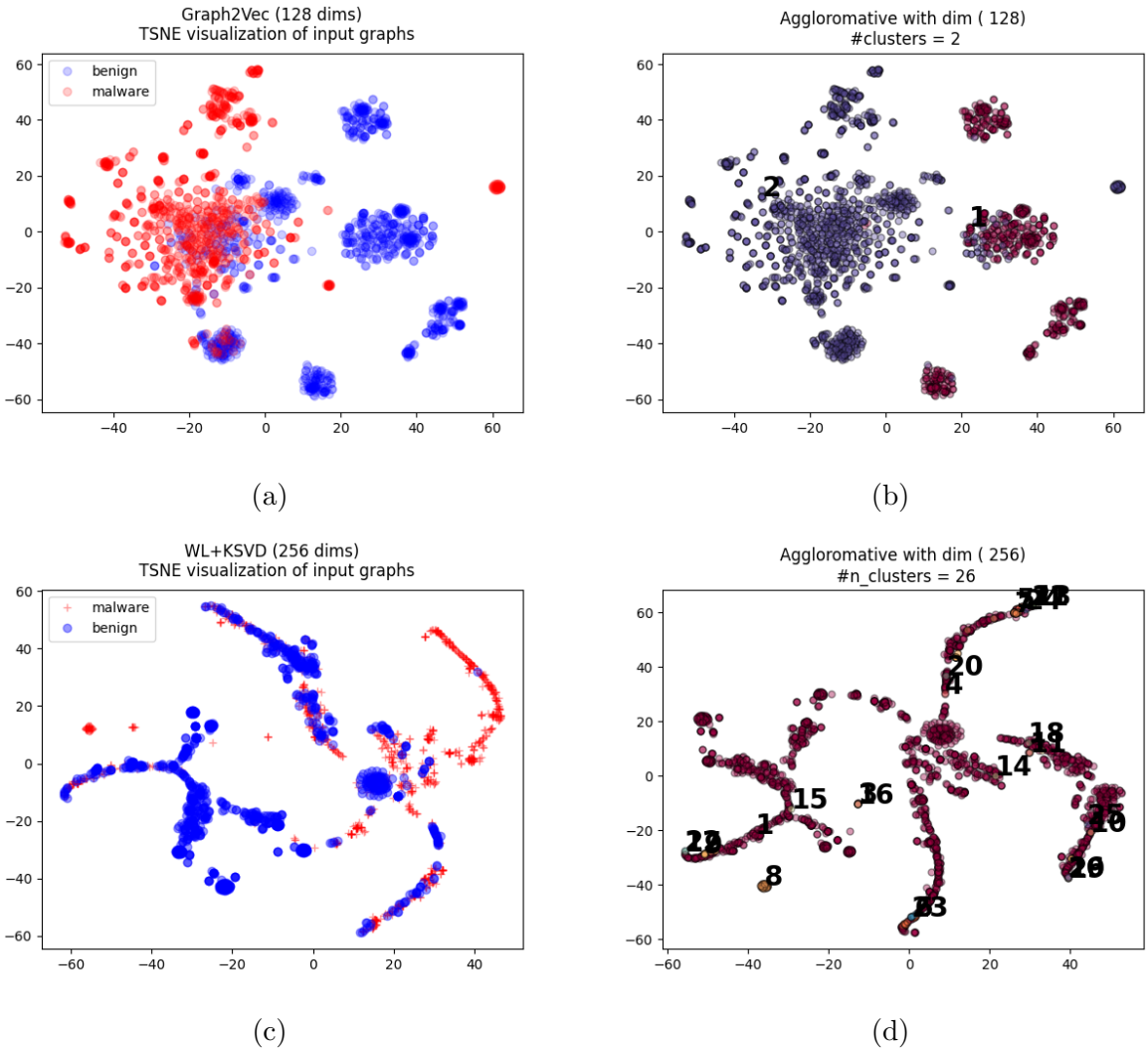
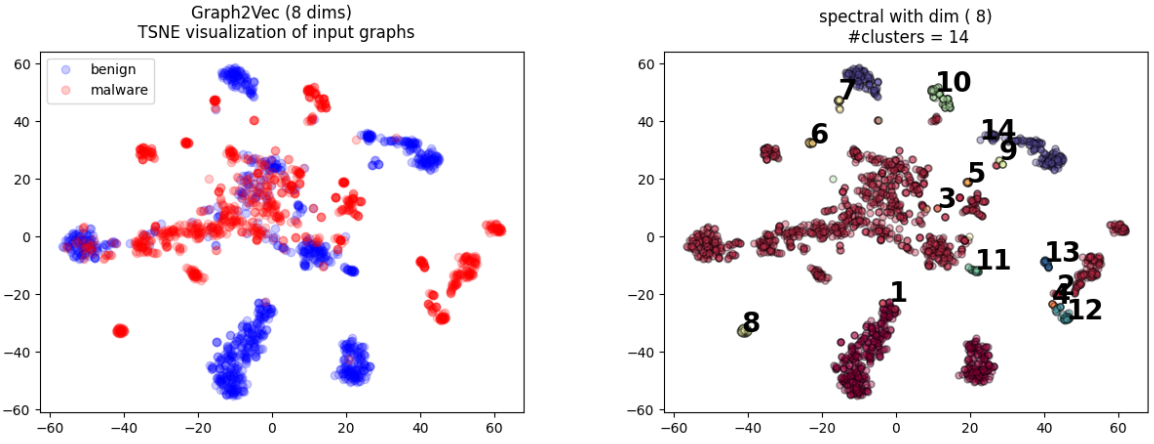


Figure 5.5: Agglomerative clustering predictions for the test set with best hyper-parameters. Left are the ground truth labels of the test set. Right, cluster predictions.Top row is the *Graph2Vec* and bottom row is *WL+KSVD*

implementation, a histogram is used to store discretized feature values for efficiency during training. The classifier efficacy is measured according to prediction accuracy on a test set, Area Under the Receiver Operating Characteristic Curve (ROC AUC) score, and finally F1 scores. For all three measures the closer the value is to one the better the result.



(a) (b)

Figure 5.6: Spectral clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.

Results Table 5.10 and 5.11 provide the supervised classification results with default parameters for different embedding dimensions for *Graph2Vec* and *WL+KSVD*. It can be observed that there seems to be an optimal embedding dimension that achieves high accuracy. Table 5.13 and 5.12 shows the optimized classifier results for *Graph2vec* and *Wl+KSVD* embeddings. The results show that *WL+KSVD* performance is sub-par to the *Graph2vec* performance. In the next section, we can use the introduced feature ranking techniques to analyze what led to this sub-par performance.

Sub-tree Structure Identification

It would be interesting to analyze the relationship between the learned sub-tree structures and the groupings in the vector representation. Since these sub-tree structures correspond to certain coding patterns, it would be interesting to identify and evaluate which patterns are common or have a high occurrence in malware groupings. An example of an idealized backdoor vulnerability is shown in Figure 5.7 [69][8].

[8] `https://www.ndss-symposium.org/wp-content/uploads/2017/09/11Firmalice.slide_.pdf`

Table 5.10: Classification results on *Graph2Vec* embeddings with default ML settings

Method	Dimensions	2	4	8	16	32	64	128	256	512
lSVC	Accuracy	0.759	0.796	0.792	0.836	**0.893**	0.847	0.808	0.826	0.840
	ROC AUC	0.844	0.84	0.922	0.858	0.966	0.926	0.911	0.892	0.913
	F1 Score	0.750	0.792	0.761	0.823	0.885	0.852	0.791	0.815	0.850
RF	Accuracy	0.838	0.715	0.857	0.906	0.927	**0.935**	0.929	0.899	0.900
	ROC AUC	0.918	0.797	0.926	0.949	0.977	0.969	0.976	0.962	0.965
	F1 Score	0.835	0.741	0.856	0.906	0.925	0.931	0.929	0.900	0.902
HGBC	Accuracy	0.850	0.726	0.811	0.872	**0.927**	0.858	0.893	0.881	0.846
	ROC AUC	0.928	0.782	0.915	0.943	0.981	0.959	0.965	0.951	0.949
	F1 Score	0.850	0.753	0.824	0.876	0.924	0.867	0.893	0.881	0.858

The introduced two feature ranking metrics were used on the learned embedding to identify the important sub-tree patterns. The ranking given by each of the FR metrics and the sub-tree patterns' position on the vocabulary is shown in table 5.14. By analyzing the results it can be seen that vocabulary trimming had a large effect on the rankings. It shows that the introduced FR metrics do not perform well with the current trimming rule. This is because the vocabulary is by default trimmed by identifying the highest frequency sub-tree patterns. Hence, the resulting vector is dense and violates an assumption of sparsity when applying sparse methods. Also, since the vocabulary is sorted by the order of frequency, both FR metrics prioritized the first few vocabulary atoms.

Hence, explain the low performance and the need to apply different trimming rules to identify sub-tree patterns. The trimming rule has to preserve sparsity and low-occurring subtree patterns unique to different malware types. Further to improve the representation,

Table 5.11: Classification results on *WL+KSVD* embeddings with default ML settings

Method	Dimensions	2	4	8	16	32	64	128	256	512
lSVC	Accuracy	0.5	0.74	0.640	0.564	0.724	0.718	0.795	**0.881**	0.864
	ROC AUC	0.498	0.628	0.634	0.523	0.706	0.674	0.811	0.895	0.862
	F1 Score	0.0	0.685	0.476	0.489	0.688	0.691	0.807	0.875	0.857
RF	Accuracy	0.491	0.817	0.812	0.875	0.904	0.921	0.926	**0.929**	0.928
	ROC AUC	0.493	0.897	0.896	0.945	0.970	0.975	0.977	0.981	0.980
	F1 Score	0.475	0.809	0.804	0.868	0.902	0.918	0.924	0.927	0.926
HGBC	Accuracy	0.490	0.833	0.848	0.853	0.892	0.920	0.923	**0.934**	0.928
	ROC AUC	0.485	0.916	0.919	0.943	0.963	0.973	0.975	0.981	0.981
	F1 Score	0.463	0.820	0.840	0.846	0.890	0.918	0.921	0.933	0.927

we can customize node labeling. By default, the node label is replaced with the number of neighbors the node has. Doing so, discards some information about the CFG. However, it is challenging to come up with a unifying node labeling system. In CFG generation, when clear function names are not present the *angr* assigns the memory location or the function name plus the number of steps as the node name. However, the memory location is dependent on architecture and the compiler the program is compiled on. Also, the node labels can be the function names given by the programmer. Therefore a unifying node labeling procedure is required to label similar functions with similar names. At least keeping the basic function names (eg. *puts()*) intact for the WL hash procedure will preserve some information.

Table 5.12: Classification results on *Graph2Vec* embeddings with optimized ML models

Method	Dimensions	2	4	8	16	32	64	128	256
	Test Accuracy	0.473	0.836	0.802	0.881	0.910	0.928	0.940	**0.946**
lSVC	ROC AUC	0.501	0.935	0.885	0.959	0.975	0.984	0.987	**0.986**
	F1 Score	0.532	0.824	0.793	0.874	0.907	0.927	0.939	**0.945**
	Test Accuracy	0.826	0.923	0.934	0.944	0.953	0.955	**0.957**	0.955
RF	ROC AUC	0.913	0.981	0.985	0.988	0.989	0.988	**0.986**	0.985
	F1 Score	0.822	0.922	0.933	0.944	0.953	0.955	**0.956**	0.954
	Test Accuracy	0.823	0.919	0.931	0.947	0.953	0.959	0.958	**0.961**
HGBC	ROC AUC	0.911	0.978	0.983	0.989	0.990	0.990	0.989	**0.989**
	F1 Score	0.820	0.917	0.931	0.946	0.953	0.958	0.957	**0.960**

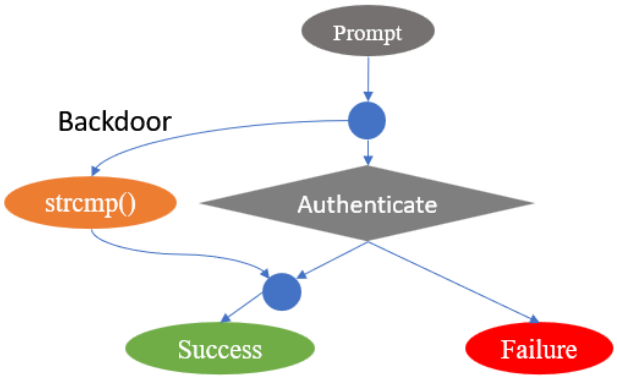


Figure 5.7: Conceptual authentication bypass vulnerability

Table 5.13: Classification results on *WL+KSVD* embeddings with optimized ML models

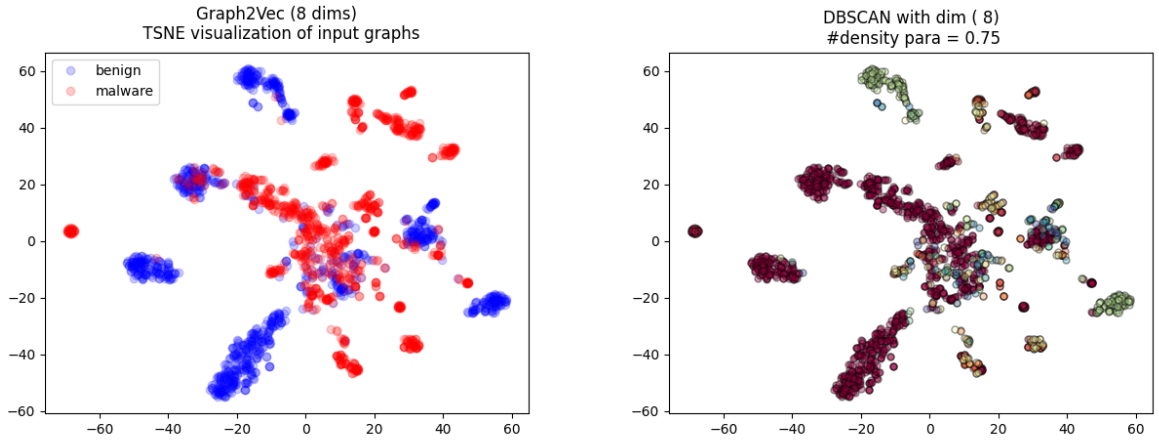
Method	Dimensions	256	512	1024	2048	4096
lSVC	Test Accuracy	0.842	**0.912**	0.904	0.912	0.893
	ROC AUC	0.862	**0.935**	0.921	0.927	0.898
	F1 Score	0.819	**0.911**	0.903	0.910	0.892
RF	Test Accuracy	0.940	**0.941**	0.929	0.928	0.913
	ROC AUC	0.985	**0.986**	0.982	0.982	0.977
	F1 Score	0.939	**0.940**	0.927	0.926	0.910
HGBC	Test Accuracy	0.946	**0.949**	0.938	0.942	0.931
	ROC Score	0.988	**0.989**	0.986	0.987	0.983
	F1 Score	0.945	**0.948**	0.937	0.941	0.930

Table 5.14: Dictionary rankings from Malware datset

Dict mapping		Dict Utilization	
Vocab number	subtree	Vocab number	subtree
3	$(5, 2008725)$	0	$(4, 3199272)$
0	$(4, 3199272)$	1	$(15ee1\ldots, 2510253)$
4	$(6, 703922)$	2	$(05b50\ldots, 2415126)$
6	$(58c70\ldots, 217596)$	3	(5, 2008725)
1	$(15ee1\ldots, 2510253)$	4	$(6, 703922)$
7	$(3a4d\ldots, 209575)$	7	$(3a4d\ldots, 209575)$
5	$(7, 228068)$	5	$(7, 228068)$
159	$(05b50\ldots, 2415126)$	6	$(58c7\ldots, 217596)$

Conclusion

The generated *Graph2Vec* an d*WL+KSVD* embeddings in fig.5.3 shows some form of the inherent grouping of programs. However, when clustering algorithms were employed they could not perform well. In summary, while the visual representation of clusters shows that the methods presented have promise, the numerical performance of unsupervised clustering is not impressive. Supervised clustering and transfer learning have stronger training procedures, resulting in more accurate classifiers. However, unsupervised methods can be trained without having any prior knowledge of which specific data samples (if any) are malware. Thus, unsupervised clustering may generalize better to never-before-seen types of malicious software.



(a) (b)

Figure 5.8: Spectral clustering predictions for the test set with best hyper-parameters. Left, ground truth labels of the test set. Right, cluster predictions.

Although the clustering results are not significant, the methods seem promising. Even with the default parameters and general settings, the vector representation seems to show a clear grouping of programs. Some of the malware and benign programs settle into defined groups. However, there are groups with significant overlap and there is a need for further

study to identify the causes. The unsupervised clustering algorithms were only capable of uniquely distinguishing just over half of the programs with high separation. This means the vector representation must be more customized to be able to achieve higher separability between the classes. The findings will be published as a research paper.

It should be noted that even though the introduced *WL+KSVD* method had sub-par performance compared to *Graph2vec* embedding for this specific application setup, the linearity and the interpretability of the *WL+KSVD* method allows us to analyze and understand why the performance is low. This is the strong point of the introduced sparse methods. The trade-off is the performance vs intuition when choosing a sparse descriptor method.

Potential Limitations

The dataset consists of CFGs generated with *cfgEmulated* function in *angr* package. Hence, for some applications, the CFG is not properly generated due to the limitations in the *cfgEmulated* method. In that case, sub-tree generation will be limited to the generated CFG only. Further, the amount unique sub-trees grow rapidly with each iteration of the WL relabelling process. Hence, vocabulary trimming plays a major part in graph embedding. More unique sub-tree structures might get trimmed.

Future Directions

In the case of incorrect CFG generation, *cfgFast* method can be used to generate CFGs as the data set. Also, node label information can be incorporated to improve the information contained in sub-tree vocabulary learning. This node labeling can be challenging since the *angr* CFG generation method has to be studied to get an insight into the attributes of each node.

In typical CFGs, the edges do not contain any information or weight hence each function call or node connection is considered equal. However, we could incorporate a weight for the

edges that represent the calling functions' computational complexity. It would help identify if a program keeps calling a computationally expensive function. One possibility is to consider the count of assembled codes in a function as the weight of its incoming edge.

Currently, dictionary learning is performed in an unsupervised manner. Since most malware has some components that are the same as benign programs, it is hard to identify which sub-tree structures are unique to malware. In the future, we will investigate using discriminatory dictionary learning methods like frozen dictionary learning to isolate unique sub-tree structures.

SUMMARY

Conclusion

This dissertation focuses on employing sparse representation methods to identify important sub-tree patterns in Control Flow Graphs generated by binary files. In the preliminary work related to malware detection, several opportunities were identified for improvement of the existing graph embedding and feature ranking methods. Sparse representation is identified as an under-utilized technique in this domain. To exploit these opportunities, a novel graph descriptor method and two novel feature ranking metrics are introduced. This dissertation presents the definitions and the experiments conducted to evaluate the effectiveness of the sparse representation-based techniques compared to other graph embedding and FR metrics. Although processing time is high compared to other methods, it could be considered a trade-off to achieve an intuitive, linear, and simple model.

Sparse Graph Descriptors

In existing graph embedding methods, a lack of intuition was identified as a main drawback. Sparse representation is selected as a potential solution due to its linear and intuitive relationship between the input domain and the embedded domain. Inspired by *Graph2Vec*, the WL+KSVD algorithm was developed to incorporate sparse representation methods. The method consists of two parts; sub-tree encoding and dictionary learning. In the presented method WL hashing and KSVD are selected for the two parts respectively, but there is flexibility to replace them with other techniques depending on the user's needs. The implemented public Python package utilizes the *scikit-learn* API structure and has the option to use different sub-tree encoding and dictionary learning methods. The presented method was compared with several existing graph embedding methods to evaluate the performance. Although the performance accuracy is shown to have an improvement

compared to *Graph2Vec*, the computational cost and time are expensive. Hence, the presented method is mostly suitable for applications that require analysis and intuition of input features.

Dictionary-based Feature Ranking

The sparse representation creates a linear mapping between the input and the output domain. Two feature ranking metrics were presented to exploit the linear relationship to gain an understanding of the input features. The first metric, *Dictionary mapping,* looks at the dictionary element distribution in the input space. The second metric, *Dictionary utilization,* calculates how much each of the dictionary elements is used by the input data. Both of the metrics are model-agnostic and can be trained in a supervised or unsupervised manner. However, learning the initial dictionary is expensive; hence, it is useful when a dictionary is already trained. The metrics are implemented as a Python package and can use different dictionary-learning methods. When used with supervised dictionary learning methods, class-wise feature ranking can be calculated. The presented metrics were compared against several Feature Ranking metrics to evaluate the performance. The presented metrics show similar performance to existing methods.

Malware Analysis

A case study of the application of the presented methods is shown in the Malware analysis domain. A dataset was curated consisting of benign and malware binary portable executable files. The Control Flow Graphs (CFG) generated from the binary files were published as a public dataset using *CFGfast()* method. The presented sparse graph descriptor method was applied to the CFG dataset and supervised and unsupervised ML methods were applied and compared with *Graph2Vec* method. The presented Feature Ranking metrics were used to identify important sub-tree patterns. Different node labeling strategies were discussed.

Threat to Validity

Several threats to validity are identified. The proposed methods are highly sensitive to the input data pre-processing techniques. Further, the input data has to be continuous, real-valued, and should not contain missing data. Dictionary learning is an expensive process hence suitable for intrinsically sparse data and small datasets. The experiments were conducted using default hyper-parameters of the ML and FR methods, which might not yield the best results. The Python implementations are not optimized for time or memory management, hence for larger datasets memory issues might arise. Also, the datasets and the methods compared in this dissertation are not exhaustive.

Future Direction

Several avenues for potential future development can be identified. Improvements in the Python implementation can lead to improved time/memory usage and stability. The possibility of using parallel processing and graphic processors can be explored to improve performance. Other efficient sub-tree encoding and dictionary learning methods can be explored. To identify patterns, further experiments can be conducted with different node and edge features.

REFERENCES CITED

[1] Ines Chami et al. "Machine Learning on Graphs: A Model and Comprehensive Taxonomy". In: *Journal of Machine Learning Research* 23.89 (2022), pp. 1–64. DOI: 10.48550/ARXIV.2005.03675.

[2] Christopher Morris et al. "TUDataset: A collection of benchmark datasets for learning with graphs". In: *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*. 2020. arXiv: 2007.08663. URL: www.graphlearning.io.

[3] Fenxiao Chen et al. "Graph representation learning: a survey". In: *APSIPA Transactions on Signal and Information Processing* 9.1 (2020). DOI: 10.1017/atsip.2020.13.

[4] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010. ISBN: 978-1-4419-7011-4. DOI: 10.1007/978-1-4419-7011-4.

[5] Ryotaro Matsuo, Ryo Nakamura, and Hiroyuki Ohsaki. "Sparse Representation of Network Topology with K-SVD Algorithm". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, July 2019. DOI: 10.1109/compsac.2019.00050.

[6] Jinbiao Chen, Xiao Qi, and Yongcai Wang. "An efficient solution to locate sparsely congested links by network tomography". In: *2014 IEEE International Conference on Communications (ICC)*. IEEE, June 2014. DOI: 10.1109/icc.2014.6883497.

[7] Kaveen Liyanage et al. "Dictionary Learning on Graph Data with Weisfieler-Lehman sub-tree kernel and KSVD". In: *IEEE International conference on Acoustics, Speech and Signal Processing* (2023). submitted Oct. 2022.

[8] Kaveen Liyanage and Bradley M. Whitaker. "Feature Analysis in Satellite Image Classification Using LC-KSVD and Frozen Dictionary Learning". In: *2022 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, May 2022. DOI: 10.1109/ietc54973.2022.9796892.

[9] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. "A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* 30.9 (Sept. 2018), pp. 1616–1637. DOI: 10.1109/tkde.2018.2807452.

[10] Peng Cui et al. "A Survey on Network Embedding". In: *IEEE Transactions on Knowledge and Data Engineering* 31 (2019), pp. 833–852.

[11] L. Maddalena et al. "On Whole-Graph Embedding Techniques". In: *Trends in Biomathematics: Chaos and Control in Epidemics, Ecosystems, and Cells*. Springer International Publishing, 2021, pp. 115–131. DOI: 10.1007/978-3-030-73241-7_8.

[12] Annamalai Narayanan et al. "graph2vec: Learning distributed representations of graphs". In: *arXiv preprint arXiv:1707.05005* (2017). URL: https://arxiv.org/pdf/1707.05005.pdf.

[13] Nino Shervashidze et al. "Weisfeiler-lehman graph kernels." In: *Journal of Machine Learning Research* 12.9 (2011). URL: https://www.jmlr.org/papers/volume12/shervashidze11a/shervashidze11a.pdf.

[14] Quoc Le and Tomas Mikolov. "Distributed Representations of Sentences and Documents". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research. PMLR. Bejing, China: PMLR, June 2014, pp. 1188–1196. URL: https://proceedings.mlr.press/v32/le14.html.

[15] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. "A survey on graph kernels". In: *Applied Network Science* 5.1 (2020), pp. 1–42.

[16] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *arXiv preprint arXiv:1301.3781* (2013). DOI: 10.48550/ARXIV.1301.3781.

[17] Xin Rong. "word2vec Parameter Learning Explained". In: *arXiv preprint* (2014). DOI: 10.48550/ARXIV.1411.2738.

[18] M. Aharon, M. Elad, and A. Bruckstein. "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation". In: *IEEE Transactions on Signal Processing* 54.11 (Nov. 2006), pp. 4311–4322. DOI: 10.1109/tsp.2006.881199. URL: https://doi.org/10.1109/TSP.2006.881199.

[19] Ron Rubinstein, Tomer Peleg, and Michael Elad. "Analysis K-SVD: A Dictionary-Learning Algorithm for the Analysis Sparse Model". In: *IEEE Transactions on Signal Processing* 61.3 (Feb. 2013), pp. 661–677. DOI: 10.1109/tsp.2012.2226445. URL: https://doi.org/10.1109/TSP.2012.2226445.

[20] Y.C. Pati, R. Rezaiifar, and P.S. Krishnaprasad. "Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition". In: *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers*. IEEE. IEEE Comput. Soc. Press, 1993, pp. 40–44. DOI: 10.1109/acssc.1993.342465. URL: https://doi.org/10.1109/ACSSC.1993.342465.

[21] G. Davis, S. Mallat, and M. Avellaneda. "Adaptive greedy approximations". In: *Constructive Approximation* 13.1 (Mar. 1997), pp. 57–98. DOI: 10.1007/bf02678430. URL: https://doi.org/10.1007/BF02678430.

[22] Zhuolin Jiang, Zhe Lin, and Larry S. Davis. "Learning a discriminative dictionary for sparse coding via label consistent K-SVD". In: *CVPR 2011*. IEEE, June 2011. DOI: 10.1109/cvpr.2011.5995354. URL: https://doi.org/10.1109/CVPR.2011.5995354.

[23] Zhuolin Jiang, Zhe Lin, and L. S. Davis. "Label Consistent K-SVD: Learning a Discriminative Dictionary for Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.11 (Nov. 2013), pp. 2651–2664. DOI: 10.1109/tpami.2013.88. URL: https://doi.org/10.1109/TPAMI.2013.88.

[24] Brandon T. Carroll et al. "Outlier Learning via Augmented Frozen Dictionaries". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25.6 (June 2017), pp. 1207–1215. DOI: 10.1109/taslp.2017.2690567.

[25] Kaveen Liyanage, Fereshteh Ramezani, and Bradley M. Whitaker. "Covid-19 Chest CT Scan Image Classification Using LCKSVD and Frozen Sparse Coding". In: *Proceedings of 2021 International Conference on Medical Imaging and Computer-Aided Diagnosis (MICAD 2021)*. Springer Singapore, Aug. 2021, pp. 272–281. ISBN: 9789811638800. DOI: 10.1007/978-981-16-3880-0_28.

[26] Khadija Abdullah Uthman, Fadl Mutaher Ba-Alwi, and Suad Mohammed Othman. "A Survey on Feature Selection in Microarray Data: Methods, Algorithms and Challenges". In: *International Journal of Computer Sciences and Engineering* (2020).

[27] Anbuselvan Sangodiah, Rohiza Ahmad, and Wan Fatimah Wan Ahmad. "A review in feature extraction approach in question classification using Support Vector Machine". In: *2014 IEEE International Conference on Control System, Computing and Engineering (ICCSCE 2014)*. IEEE, Nov. 2014, pp. 536–541. DOI: 10.1109/iccsce.2014.7072776.

[28] Dimitrios Effrosynidis and Avi Arampatzis. "An evaluation of feature selection methods for environmental data". In: *Ecological Informatics* 61 (Mar. 2021), p. 101224. DOI: 10.1016/j.ecoinf.2021.101224.

[29] A. Jovic, K. Brkic, and N. Bogunovic. "A review of feature selection methods with applications". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, May 2015. DOI: 10.1109/mipro.2015.7160458.

[30] Igor Kononenko, Edvard Šimec, and Marko Robnik-Šikonja. "Overcoming the myopia of inductive learning algorithms with RELIEFF". In: *Applied Intelligence* 7.1 (1997), pp. 39–55. DOI: 10.1023/a:1008280620621.

[31] Chris Ding and Hanchuan Peng. "Minimum redundancy feature selection from microarray gene expression data". In: *Journal of Bioinformatics and Computational Biology* 03.02 (Apr. 2005), pp. 185–205. DOI: 10.1142/s0219720005001004.

[32] Yishi Zhang et al. "Feature assessment and ranking for classification with nonlinear sparse representation and approximate dependence analysis". In: *Decision Support Systems* 122 (July 2019), p. 113064. DOI: 10.1016/j.dss.2019.05.004.

[33] Shutao Li et al. "Deep Learning for Hyperspectral Image Classification: An Overview". In: *IEEE Transactions on Geoscience and Remote Sensing* 57.9 (Sept. 2019), pp. 6690–6709. DOI: 10.1109/tgrs.2019.2907932.

[34] Kaveen Liyanage and Bradley M. Whitaker. "Satellite Image Classification Using LC-KSVD Sparse Coding". In: *2020 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, Oct. 2020. DOI: 10.1109/ietc47856.2020.9249174.

[35] Ori Or-Meir et al. "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey". In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3329786. URL: https://doi.org/10.1145/3329786.

[36] Zahra Bazrafshan et al. "A survey on heuristic malware detection techniques". In: *IKT 2013 - 2013 5th Conference on Information and Knowledge Technology* (May 2013), pp. 113–120. DOI: 10.1109/IKT.2013.6620049.

[37] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)* (2016).

[38] Boris Weisfeiler and Andrei Leman. "The reduction of a graph to canonical form and the algebra which appears therein". In: *NTI, Series* 2.9 (1968), pp. 12–16.

[39] Pinar Yanardag and S.V.N. Vishwanathan. "Deep Graph Kernels". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '15. Sydney, NSW, Australia: Association for Computing Machinery, 2015, pp. 1365–1374. ISBN: 9781450336642. DOI: 10.1145/2783258.2783417. URL: https://doi.org/10.1145/2783258.2783417.

[40] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs". In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM. 2020, pp. 3125–3132.

[41] Gesina Schwalbe and Bettina Finzel. "A comprehensive taxonomy for explainable artificial intelligence: a systematic survey of surveys on methods and concepts". In: *Data Mining and Knowledge Discovery* 38.5 (Jan. 2023), pp. 3043–3101. ISSN: 1573-756X. DOI: 10.1007/s10618-022-00867-8.

[42] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[43] Tyler Neylon. "Sparse Solutions for Linear Prediction Problems". AAI3221982. PhD thesis. USA: New York University, 2006. ISBN: 9780542752384.

[44] Yael Yankelevsky and Michael Elad. "Finding GEMS: Multi-Scale Dictionaries For High-Dimensional Graph Signals". In: *IEEE Transactions on Signal Processing* 67.7 (Apr. 2019), pp. 1889–1901. DOI: 10.1109/tsp.2019.2899822.

[45] Cédric Vincent-Cuaz et al. "Online Graph Dictionary Learning". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 10564–10574. URL: http://proceedings.mlr.press/v139/vincent-cuaz21a.html.

[46] Chenglong Bao et al. "Dictionary Learning for Sparse Coding: Algorithms and Convergence Analysis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.7 (July 2016), pp. 1356–1369. ISSN: 2160-9292. DOI: 10.1109/tpami.2015.2487966.

[47] Bradley Efron et al. "Least Angle Regression". In: *The Annals of Statistics* 32.2 (Apr. 2004). ISSN: 0090-5364. DOI: 10.1214/009053604000000067.

[48] Fadil Santosa and William W. Symes. "Linear Inversion of Band-Limited Reflection Seismograms". In: *SIAM Journal on Scientific and Statistical Computing* 7.4 (Oct. 1986), pp. 1307–1330. ISSN: 2168-3417. DOI: 10.1137/0907087.

[49] Robert Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 00359246. URL: http://www.jstor.org/stable/2346178 (visited on 11/04/2024).

[50] Federico Errica et al. *A Fair Comparison of Graph Neural Networks for Graph Classification*. 2019. DOI: 10.48550/ARXIV.1912.09893.

[51] Christopher Morris, Gaurav Rattan, and Petra Mutzel. "Weisfeiler and Leman go sparse: Towards scalable higher-order graph embeddings". In: *Advances in Neural*

Information Processing Systems. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 21824–21840. URL: https://proceedings.neurips.cc/paper/2020/file/f81dee42585b3814de199b2e88757f5c-Paper.pdf.

[52] Hong Chen and Hisashi Koga. "GL2vec: Graph Embedding Enriched by Line Graphs with Edge Features". In: *Neural Information Processing*. Springer International Publishing, 2019, pp. 3–14. DOI: 10.1007/978-3-030-36718-3_1.

[53] Nathan de Lara and Edouard Pineau. "A Simple Baseline Algorithm for Graph Classification". In: *arXiv preprint arXiv:1810.09155* (2018). DOI: 10.48550/ARXIV.1810.09155.

[54] Lars Buitinck and et. el. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[55] Julien Mairal et al. "Online dictionary learning for sparse coding". In: *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*. ACM Press, 2009. DOI: 10.1145/1553374.1553463.

[56] Stefan Aeberhard and M. Forina. *Wine*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5PC7J. 1992.

[57] Saikat Basu et al. "Deepsat: a learning framework for satellite imagery". In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '15*. ACM Press, 2015, pp. 1–10. DOI: 10.1145/2820783.2820816. URL: https://dl.acm.org/doi/pdf/10.1145/2820783.2820816?download=true.

[58] Edward Boyda et al. "Deploying a quantum annealing processor to detect tree cover in aerial imagery of California". In: *PLOS ONE* 12.2 (Feb. 2017). Ed. by Shijo Joseph, e0172505. DOI: 10.1371/journal.pone.0172505. URL: https://doi.org/10.1371/journal.pone.0172505.

[59] Carl F. Jordan. "Derivation of Leaf-Area Index from Quality of Light on the Forest Floor". In: *Ecology* 50.4 (July 1969), pp. 663–666. DOI: 10.2307/1936256. URL: https://doi.org/10.2307/1936256.

[60] Y.J. Kaufman and D. Tanre. "Atmospherically resistant vegetation index (ARVI) for EOS-MODIS". In: *IEEE Transactions on Geoscience and Remote Sensing* 30.2 (Mar. 1992), pp. 261–270. DOI: 10.1109/36.134076. URL: https://doi.org/10.1109/36.134076.

[61] JW Rouse et al. "Monitoring vegetation systems in the Great Plains with ERTS". In: *NASA special publication* 351 (1974), p. 309. URL: https://ntrs.nasa.gov/search.jsp?R=19740022614.

[62] A Huete et al. "Overview of the radiometric and biophysical performance of the MODIS vegetation indices". In: *Remote Sensing of Environment* 83.1-2 (Nov. 2002), pp. 195–213. DOI: 10.1016/s0034-4257(02)00096-2. URL: https://doi.org/10.1016/S0034-4257(02)00096-2.

[63] A.R Huete. "A soil-adjusted vegetation index (SAVI)". In: *Remote Sensing of Environment* 25.3 (1988), pp. 295–309. ISSN: 0034-4257. DOI: https://doi.org/10.1016/0034-4257(88)90106-X. URL: https://www.sciencedirect.com/science/article/pii/003442578890106X.

[64] Bryan Portillo, Kaveen Liyanage, and Bradley Whitaker. "Improving Malware Detection from Binary Control Flow Graphs Using Supervised Learning". In: *2024 Intermountain Engineering, Technology and Computing (IETC)*. IEEE, May 2024, pp. 174–179. DOI: 10.1109/ietc61393.2024.10564451.

[65] Veronika Strandova-Neeley et al. "Graph-based analysis of binary code for malware detection and vulnerability identification". In: *Cyber QR ops report* (2021).

[66] Reese Andersen Pearsall. "An evaluation of graph representation of programs for malware detection and categorization using graph-based machine learning methods". PhD thesis. Montana State University-Bozeman, College of Engineering, 2023.

[67] Shushan Arakelyan et al. "Bin2vec: learning representations of binary executable programs for security tasks". In: *Cybersecurity* 4.1 (July 2021). ISSN: 2523-3246. DOI: 10.1186/s42400-021-00088-4.

[68] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[69] Yan Shoshitaishvili et al. "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware". In: *NDSS* (2015).