System Log File Reduction and Detection of Malicious Behavior

A Thesis
Submitted to the Faculty of
The Department of Electrical and Computer Engineering
Villanova University

By

Ralph Ritchey

In Partial Fulfillment
Of the Requirements for the Degree of
Master of Science in Cybersecurity

VILLANOVA
UNIVERSITY

April 27, 2021

Copyright © 2021 by Ralph Ritchey
All Rights Reserved

Abstract

Cybersecurity relies heavily on data for the detection of malicious behavior. Historically, intrusion detection systems (IDS) utilized sensors placed strategically within a network to monitor and capture network traffic. Various tools then process the network traffic in real-time or batch mode, generating alerts which security analysts review. This methodology worked effectively until the use of encryption became prevalent for network traffic. Encrypted network traffic prevents signature-based IDS tools from inspecting packet payload contents to detect signatures indicating malicious activity or intent. As an alternative data source, system logs and web server logs capture the indicators at the system level required by cybersecurity tools and analysts to detect possible malicious behavior. The research presented here examines the viability of performing log file size reduction while retaining and indicating log entries containing malicious activity using truncated singular value decomposition (TSVD) and k-means clustering.

Contents

List of Tables

List of Figures

Chapter 1
Introduction

Network-based intrusion detection historically formed the backbone for protecting an organization's infrastructure. Strategically placed IDS sensors, as shown in Figure 1.1, monitored all network traffic flowing between the organization and the rest of the world. IDS sensor software, such as Snort [1], examined each network packet, including performing deep packet inspection into each network packet's payload. Examining the payload allowed the IDS software to detect malicious activity and output an alert for an analyst to review. This sensing capability heavily relied on network traffic being unencrypted. The lack of packet encryption was native to the Internet, which was "…never intended to be secure or open for commercial use." [2]

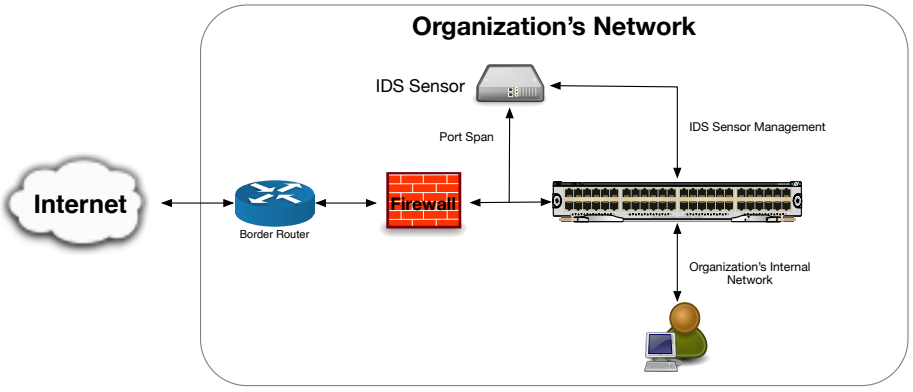

Figure 1.1: IDS Sensor Network Placement.

A study performed by Fortinet, Inc. in 2018 claims, "…encrypted traffic now represents over 72% of all network traffic…". [3] The study also indicates this represents an increase in encrypted traffic by 17% from the previous year. As encryption usage continues to increase, an IDS sensor's ability to perform deep packet inspection decreases, thereby reducing its effectiveness each year. Sensors cannot inspect encrypted packet payloads unless the sensor has the necessary information to complete the decryption or dynamically break the encryption ("break and inspect"). Dynamically breaking encryption is complex and computationally expensive, making this an impractical option, especially for high bandwidth environments. Cybersecurity must now look at alternative data sources to fulfill security monitoring.

Alternative data sources are logs generated by an organization's servers, desktops, and other computer-based devices. System logs "…consist of the voluminous intermixing of messages from many software components…" [4] executing on a device. These log files provide insight into the normal execution of software on the device. While used by systems administrators as indicators of a hardware or software fault, anomalies in log entries may also indicate defects triggered by malicious activity. For example,

Najafabadi et al. [5] used web server logs to train a machine learning algorithm to detect distributed denial of service (DDoS) attacks.

The research undertaken here and presented at the IEEE INFOCOM 2021 poster session [6] uses log files, coupled with chaining multiple machine learning algorithms, to reduce log file sizes and identify potentially malicious log entries. The following sections discuss the background of the problem, the programming approach used, a description of the data sets, and the experiments performed.

Chapter 2
Literature Review

Najafabadi et al. [5] used Principle Component Analysis (PCA) to detect DDoS attacks against a web server. The researchers split logged HTTP requests contained in log files based upon non-overlapping time windows and then sub-grouped the HTTP requests based on client IP addresses. The URLs accessed by the client during the time window processed populated the feature vector for that client record. Only URLs accessed by a minimum of 5 clients formed the feature vector. While successfully detecting DDoS attacks in web server log files, there are many other types of attacks requiring detection by cybersecurity analysts. Additionally, there is a wide range of log files from systems available as potential data sources beyond web server HTTP logs.

Xu's paper [4] focused on detecting anomalies in server system logs using PCA. In their use case, an anomaly is a change in behavior of a system indicated by log entries not conforming to prior learned behavior by a machine learning algorithm. Although the researchers do not apply their technique to detecting reportable cybersecurity events, anomaly detection can be a valuable approach for identifying potentially malicious activity. To create feature vectors, the researchers statically analyzed the source code generating log entries, creating templates used to parse log files. The researchers acknowledged static analysis of source code is critical to the "…accuracy of our approach…" While feasible in small environments, access to source code generating log files across all devices deployed in an organization is highly unlikely. Additionally, new software or updates to software may break templates created from prior static analysis, requiring continual static analysis at a rate not sustainable in some environments. In a follow-up paper, their source code static analysis revealed "…20,000 different possible message types…" [7], forcing them to "…heuristically choose the message variables that are likely to indicate problems." Cybersecurity requires accurate detection, using a technique "likely" determining problems will result in too many false positives and overwhelm analysts in large organizations.

Juvonen et al. [8] applied random projection, PCA, and diffusion maps to HTTP log files for cybersecurity anomaly detection. The authors broke URL requests contained in log entries into 2-grams to form the feature vector, which, in the worst-case scenario, can result in a vector of 256^2 dimensions. However, in a "…real-world situation, the actual number is much lower." Their results, using a small initial data set, show diffusion mapping, although taking longer to execute, identifying more attacks than PCA and random projection. After performing a similar experiment using more extensive data, the researchers only discussed performance speed. The researchers concluded diffusion maps and random projection work best as part of an ensemble configuration for anomaly detection.

A paper by Lee et al. [9] applied two PCA variations for anomaly detection using the KDD Cup 1999 data set. Although detailed data set transformation specifics did not appear in the paper, the authors claim 94-99.87% accuracy in identifying four malicious activity types. Execution time, critical for application in the real world, was approximately 34 seconds for osPCA and around 3 seconds for Online osPCA using 76,813 samples.

Although not using system log files, Abdulhammed et al. [10] applied PCA to the CICIDS2017 intrusion detection data set as part of an ensemble method to detect and classify malicious network traffic. After transforming the data, PCA reduced the number of features from 81 down to 10. The reduced feature set then fed into several machine learning classifier algorithms (random forest, Bayesian network, linear discriminant analysis, and quadratic discriminant analysis) to perform detection and classification.

Ritchey et al. [11] applied PCA to features extracted from a production Apache HTTP log file. PCA reduced the feature set permitting generation of a 3d graph, as shown in Figure 2.1, indicating possible outliers. The authors encountered difficulties attempting to map graph points back to specific log lines. This challenge caused the authors to switch to a Naïve Bayes based approach where they mapped results back to particular log lines. The Naïve Bayes algorithm trained using the same features used for PCA and generated a model. The trained model was then used to identify unusual log lines in the same log file falling below a probability threshold. Although useful for Apache HTTP logs, the approach was not as practical for outlier detection for Linux syslog files due to their highly repetitive nature.

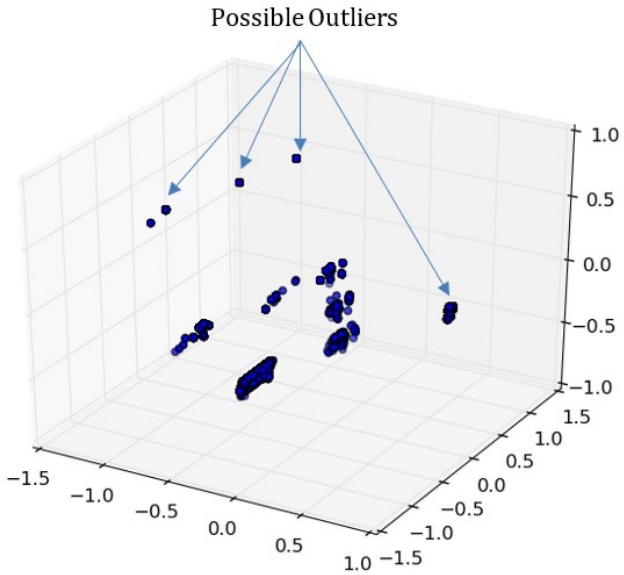

Figure 2.1 [11]: PCA Identification of Outliers (Anomalies).

Chapter 3
Research Overview

Based on the literature search results, there are alternative approaches available for exploring the application of PCA to log files that may be more suitable for cybersecurity purposes. Additionally, most published research focuses on identifying errors, faults, or variances, indicating a failure is imminent. Cybersecurity presents a unique challenge as the malicious activity may generate log entries appearing to be perfectly normal, routine entries. An approach used for cybersecurity purposes must identify malicious activity logged as regular log entries. This section provides the overarching goals of the proposed research, the coding approach, the hardware used, and the general experimental procedures.

3.1 Research Goals

A research project should expand the body of knowledge and contribute to the scientific understanding of the research topic. Research provides additional value when results are directly applicable to real-world implementation. The goals listed below add to the scientific knowledge gained and increase the real-world applicability of the research project:

- *Deployable without significant manual tuning and maintenance:* A tool requiring substantial amounts of time to install, manually tune, and manually maintain increases costs for an organization. Organizations are more likely to install and utilize solutions providing reliable detection results not requiring constant, manual labor. The detection technique should self-tune as much as possible, requiring a minimum of manual intervention.

- *Scalable to support large enterprises and large log file sizes:* Organizations range from small network environments with a few servers and desktops to enterprises with hundreds or thousands of systems deployed globally. A tool resulting from the research should work effectively for small and large organizations.

- *Reasonable resource (CPU, memory, time, and disk I/O) utilization:* Systems capable of generating logs used by organizations vary from small, battery-powered portable devices to large, multi-CPU based servers and clusters installed in data centers. A tool applying the approach presented in this paper should impact each device's available resources as little as possible. Systems administrators and users will immediately delete or disable tools utilizing a significant amount of resources affecting a device's usability.

- *Ability to reduce log file sizes while retaining lines with potential malicious behavior:* System usage may result in log files becoming vast, complicating storage and

transport. A tool developed based on the research should facilitate reducing a log file's size while still retaining log lines indicating malicious behavior.

3.2 Coding Approach

Coding for the research project used the Python [12] scripting language. Python, an interpreted, scripting-based programming language, facilitated fast code changes without the need to recompile after every minor code change. Although compiled languages such as C/C++ perform better for specific use cases, the ability to quickly adjust code was more important for quickly performing new experiments and testing minor code adjustments. Using Python allowed the focus to remain on the experiments and results without spending valuable time on potentially lengthy and complicated code compilation. Experiment execution times indicate the code is deployable into production environments with acceptable runtimes using the existing Python code.

Python provided access to numerous libraries extending the language's natively provided capabilities. Libraries such as scikit-learn [13], numpy [14], and pandas [15] provided access to well-tested, performant, and publicly reviewed implementations of common machine learning algorithms and highly efficient data manipulation and management capabilities. These libraries allowed the research to focus on feature extraction, feature engineering, machine learning tuning, and analysis of the results of code changes.

The framework (Figure 3.1) developed while performing the research used a lightweight plugin approach [16] for the data extraction/transformation, machine learning, graphing (output), and anomaly identification/reduction workflow steps. The plugin architecture allowed numerous variations of the same plugin to exist and quick creation of new ones to test new ideas or slight permutations.

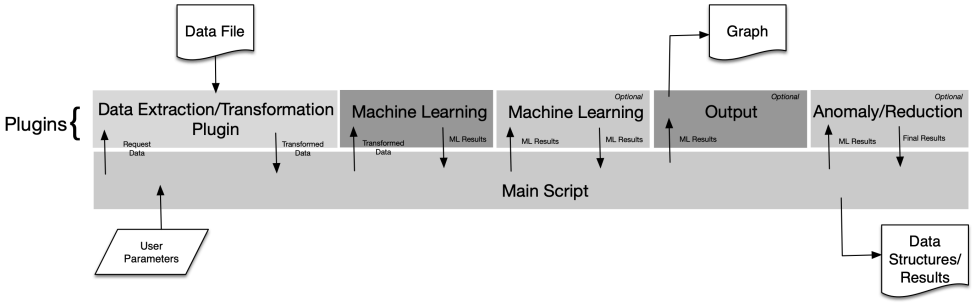

Figure 3.1: General Script Design

The framework's main script (Appendix E) provides command-line options, allowing the user to dynamically specify which plugin to use for each step in the workflow

from the available pool of plugins. General help information displays with the "`-h`" option, which includes a list of plugins for each workflow step.

3.3 Hardware and Software

The hardware used for the execution of the experiments consisted of an Apple desktop Mac Pro with the following configuration:

- Mac Pro (Late 2013)
- 3.5GHz 6-Core Intel Xeon E5
- 16GB 1866 MHz DDR3 memory
- MacOS 11.1 (Big Sur)
- 4TB SSD

Coding and testing also used an Apple MacBook Pro for portability reasons, with final experimental results generated using the desktop Mac Pro. Both systems used the Homebrew [17] package manager to access Python's latest version (3.9.x).

A small collection of libraries not natively included with Python provided the necessary functionality to efficiently perform data transformation, machine learning, and graphing. The Python package management system `pip` provided access to these libraries. The libraries used for the research project included:

- scikit-learn
- pandas
- numpy
- pympler
- apachelogs
- plotly
- pyhash

3.4 Experimental Procedure

Careful planning and execution of experiments included the incorporation of the following guidelines:

- Reboot the system used for experiment execution just before executing the experiment. Rebooting ensures all experiment executions begin with the system in the same state.
- Capture and store all script experimentation output (logs, etc.) for re-analysis and verification. In addition to providing necessary data for the paper, this permitted a careful review of execution to ensure no mistakes were made during each experiment.
- Ensure no extraneous processes or applications were running on the system before executing experiments. Irrelevant processes or applications impact memory, CPU,

and disk utilization as they competed for resources. This would introduce misleading differences in resource utilization and execution performance, affecting the final analysis of results.

- Each experiment was executed three times, gathering the same statistics and information generated from each run for inclusion in charts, graphs, and any calculations provided in this paper.

Using different feature sets, experiments used truncated SVD (Appendix G) as the first machine learning algorithm in the chain followed by k-means. Truncated SVD performed reduced dimensionality of the data down to two-dimensions. The dimensionality reduction results then fed into k-means for anomaly identification/file reduction. Another experiment using the same features and machine learning chain used cosine similarity to perform anomaly identification/file reduction for comparison to k-means built-in use of Euclidean distance.

This approach for anomaly identification/file reduction satisfies several of the extended research goals for real-world implementation. The method proposed uses unsupervised machine learning algorithms. These algorithms eliminate the initial and periodic training required to generate deployable models, which may take considerable time and effort to create and deploy. The final results, using k-means or cosine similarity, provides the information necessary to determine which log lines to retain or filter out, satisfying the primary research goal.

Chapter 4
Machine Learning

The application of machine learning proposed here uses a PCA variant (truncated SVD) as the selected machine learning algorithm. PCA is an unsupervised machine learning algorithm, making it an ideal candidate and suitable for potential real-world deployment. Unsupervised machine learning algorithms do not require labeled data. Data in the real-world is unlabeled and requires a significant investment in resources (people and time) for an organization to process, review, and generate labels for training. Experiments, on the other hand, may use both labeled and unlabeled data sets depending on the experiment and data set(s) meeting the experiment requirements. Labeled data allows verification of experiment results against expected results, whereas unlabeled data provides insights into how well SVD identifies potentially malicious log entries under a more realistic scenario.

PCA transforms a given data set by reducing the data set's dimensionality (number of features). The premise is if features in a data set are highly correlated, it "…implies that the 'true' dimension of the dataset is less…" [18], therefore, removal of those highly correlated features (dimensionality reduction) will still "…convey virtually all of the information in the original…" [18] data set. For example, Figure 4.1 [19] shows the graph of a 2-dimensional data set on the left. By rotating the axes as shown on the right, the "…potential for dimensionality reduction is in the fact that the *y* dimension does not now demonstrate much variability, and so it might be possible to ignore it…". [19] The reorientation of the axes in this example permits the reduction of the dimensionality of the feature set from 2-dimensions to 1-dimension. Elimination of dimensions reduces noise in the data, which "…can make the results better…". [19]

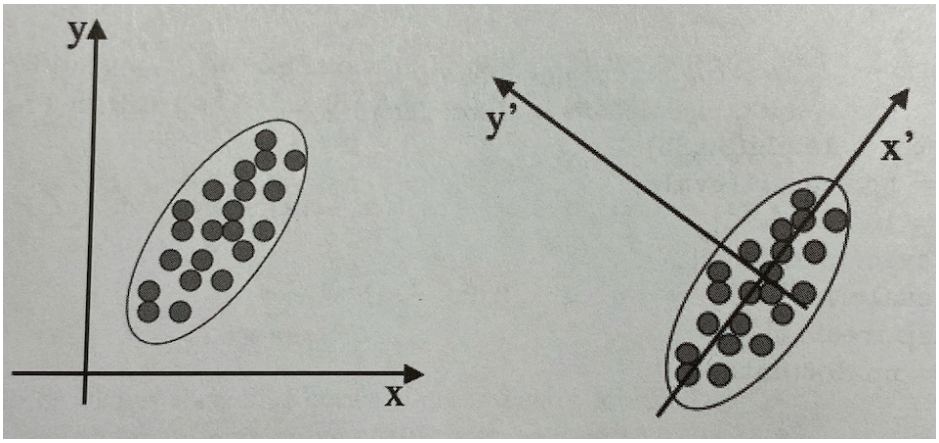

Figure 4.1 [19]: Visual Example of PCA

The first step in performing PCA calculates the mean-deviation form. For a matrix X with m samples and n features (axes), calculate the mean for each axis by summing all the sample values and dividing by the number of samples:

$$M_j = \frac{1}{m}\left(X_{1j} + \cdots + X_{mj}\right)$$

Create a new $m \times n$ matrix B with the normalized values:

$$B_{ij} = X_{ij} - M_j$$

The next step calculates the singular-value decomposition (SVD) of B, resulting in the $m \times n$ column-orthonormal matrix U, the $n \times n$ column-orthonormal matrix V, and the $n \times n$ diagonal matrix S containing the singular values of B (assuming $m > n$).

$$B = U\,S\,V^T$$

The columns of V are eigenvectors of the covariance matrix $B^T B$ and the corresponding eigenvalues are the squares of the singular values.

After computing the singular-value decomposition, the t smallest singular-values can be ignored, and the remaining $r = n - t$ columns of U and V form the reduced dimensional model.

Chapter 5
Data Sources

A challenge for cybersecurity-based research is the availability of suitable, publicly accessible data sets. For example, researchers heavily use the KDD Cup 99 [20] dataset for network-based intrusion detection research. While publicly available and used in published papers, criticism varies from the data set being derived from "…previous datasets, which had additional identified flaws…" [21] to the age of the dataset as it "…is not representative of the types, scale or complexity of modern network traffic." [21] Looking more broadly at sharing cybersecurity-related data in general, privacy concerns come to the forefront as "…sharing could expose customer and employee personal data to increased privacy risk." [22] Synthetically generated data sets facilitate sharing and publishing, however, they present similar challenges, such as not realistically representing real-world environments or current attacks.

To account for these challenges and limitations, the research used two types of log data for experiments. The first data set is a publicly accessible, laboratory synthesized log set. This data set permits other researchers to recreate and validate the results contained in this paper. The synthesized data set includes labels, facilitating validation of the approach regarding accurately identifying malicious versus non-malicious log entries. The second data set originates from real-world, publicly accessible servers. Use of real-world data in the experiments aided in validating that the approach effectively transitions to real-world application.

5.1 AIT Synthetic Data Set

The Austrian Institute of Technology (AIT) provided the publicly accessible, 5.61GB, AIT Log Data Set V1.0 [23] spanning six days. The data, synthetically generated in a lab environment, provides logs from mail server daemons, Apache webserver daemons, system syslogs, and other types of logs from a Linux-based environment. The data set includes over half a million labeled Apache log lines, along with just under half a million labeled syslog lines. The experiments presented in this research project focused on Apache web server logs due to availability.

Using the `mail.cup.com` directory as an example, the Apache access log file contained 148,534 lines, with labels identifying specific malicious entries in a separate file. The label file identified 6,789 log lines as indicators of malicious activity. The remaining servers used in the laboratory environment each have identical directory structures containing their respective raw log and label data. Table 5.1 provides basic statistics for each server's Apache log files. Similar to logs encountered in the real-world, the data set is highly imbalanced.

Server	Log Lines	Malicious Lines	% Malicious	File Size
mail.cup.com	148,534	6,789	5%	36MB
mail.insect.com	169,340	6,973	4%	43MB
mail.onion.com	81,963	6,429	8%	22MB
mail.spiral.com	100,445	7,370	7%	24MB

Table 5.1: AIT Apache Server Basic Log File Statistics.

The AIT Apache log file used the *Combined Log Format*, which includes the referrer and user agent fields. Figure 5.1 shows the format of the log file, broken into individual components. These components formed the initial, basic features extracted from the log file and transformed into data suitable for machine learning algorithm use.

```
<srcip> - - [dd/mon/yyyy:hh:mm:ss] "<request>" <statuscode> <bytes> "<referrer>" "<useragent>"
```

Figure 5.1: AIT Apache log file format.

A concern with synthetic data sets regards how closely the synthetic data compares to real-world data. Table 5.2 shows the cardinality of unique source IP addresses extracted from the synthetic log files as a feature. Compared to the cardinality of source IP addresses (Table 5.3) extracted from the private data sets, the synthetic data set contains significantly fewer unique source IP addresses. This difference validates using a synthetic data set that permits the publishing of full details and a private, real-world data set where publishing cannot include full details but experiments validate that the approach transfers to real-world data.

Server	Unique Src IPs
mail.cup.com	4
mail.insect.com	3
mail.onion.com	3
mail.spiral.com	4

Table 5.2: AIT Unique Source IPs.

Server	Unique Src IPs
alpha	3,817
beta	5,725
gamma	6,494
delta	36,647

Table 5.3: Private Data Sets Unique Source IPs.

5.2 Real-World Private Data Sets

Real-world Apache log files from several publicly accessible websites were provided for experimental use. To protect the owners' identity and the websites, each data set is generically described and referenced using Greek alphabet letters as the server names. The real-world log files used the *Common Log Format* and *Combined Log Format* depending on the server. The *Common Log Format* (Figure 5.2), very similar to the format used by the AIT synthetic data logs, does not include the referrer or user agent fields. Although the log data contained two fewer features, experimental results did not indicate this impacted identification of malicious log entries.

```
<srcip> - - [dd/mon/yyyy:hh:mm:ss] "<request>" <statuscode> <bytes>
```

Figure 5.2: Real-world Apache log file format.

Table 5.4 shows similar imbalanced data consistency seen in the synthetic log files. Although the log files contain more lines, they are smaller in file size because they do not include the referrer and user agent fields.

Server	Log Lines	Malicious Lines	% Malicious	File Size
alpha	180,782	18,990	11%	18MB
beta	72,488	15,671	22%	15MB
gamma	68,442	18,476	27%	13MB
delta	438,208	57,505	13%	36MB

Table 5.4: Real-world Apache Server Basic Log File Statistics.

Chapter 6
Experiments

The Python framework used for coding experiments allowed the execution of many experiments as new ideas, different approaches to feature extraction, and implementation of other code changes naturally occurred as part of the research process. This chapter describes several of the experiments performed using the synthetic and private, real-world data sets. Additionally, additional experiments captured the Python code's resource utilization to support the extended research goal of ensuring the approach transfers to real-world application.

6.1 Feature Extraction and Machine Learning Chain

This section describes the feature extraction, transformation, and application of the chained machine learning algorithms used for the experiments. The machine learning chain is also described, as the same chain was used throughout all experiments.

6.1.1 Feature Extraction

All file processing plugins process log files using the same general steps (Figure 6.1). The input file is read line by line, extracting individual features from the raw log file components. These components are immediately added to the *raw data dataframe*. At the completion of feature extraction from a log line, any necessary transformation occurs, adding a new line to the *transformed dataframe*. After log file processing, processing of an optional label data file occurs if specified by the user on the command line, resulting in a third dataframe, the *label dataframe*.

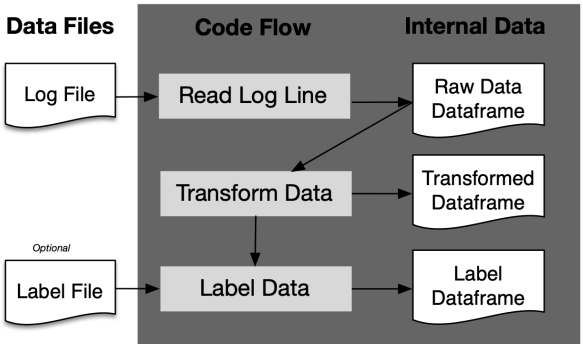

Figure 6.1: Input File Processing.

The `apachelogs` Python library provided a convenient parsing capability for Apache logs in both the common and combined formats. The library provided direct access to most of the desired log components, however, a few (command, URL, parameters, and version) were extracted from the `request_line` data structure provided during line parsing. The complete list of basic features from each log line included remote host, bytes returned by the server, status, command, URL, parameters, version, and user agent.

The category feature derives from the HTTP status code, immediately transformed during log parsing into the final value used for machine learning (Figure 6.2). HTTP status codes form groups based on the meaning of the code – success, redirection, client errors, and server errors. These groups map to specific values from 0.1 to 0.5.

```
if entry.final_status >= 100 and entry.final_status < 200:
    category = 0.1 # Informational
elif entry.final_status >= 200 and entry.final_status < 300:
    category = 0.2 # Success
elif entry.final_status >= 300 and entry.final_status < 400:
    category = 0.3 # Redirection
elif entry.final_status >= 400 and entry.final_status < 500:
    category = 0.4 # Client Error
elif entry.final_status >= 500 and entry.final_status < 600:
    category = 0.5 # Server Error
```

Figure 6.2: HTTP Status Code Transformation.

During parsing of the log file additional, unused features are extracted, such as the timestamp. These features are removed when creating a new data frame used exclusively for machine learning. By using discrete data frames, the raw data extracted from the log file and the machine learning ready dataframe are savable to disk as CSV files when using the `-s` command-line option. This allows the user to verify log file parsing correctly extracts raw data from the log file, and properly transforms the data for machine learning.

Transformation of the raw data occurs immediately after log file processing (Figure 6.3). A Fowler-Noll-Vo hash algorithm implemented in the `pyhash` library (`fnv1_64`) generates a 64-bit hash value for all string values. Scaling occurs when generating the hash value by dividing the hash by 2^{26}, adjusting values to fall between zero and one.

```
# Use FVN hash to transform string values into a numerical representation.
print("\n\tBeginning:  Hashing and scaling string values.")
hash_alg = pyhash.fnv1_64()

# Examine the Pandas data frame column by column and hash/scale only
# columns detected as strings.
for i in self.df.columns:
    if self.df.dtypes[i] == np.object:
        print("\t\tHashing and scaling column: %s" % i)
        self.df[i] = self.df[i].map(lambda a: (hash_alg(str(a).encode('utf-8'))) / 2**64)
```

Figure 6.3: Hash Transformation.

6.1.2 Machine Learning Chain

The machine learning chain (Figure 6.4) uses two algorithms, with an optional third step supplying an alternative calculation performing classification. Using the transformed data dataframe generated during file processing, the first step applies truncated SVD reducing dimensionality down to two dimensions. Once dimensionality reduction completes, k-means

performs classification by identifying two centroids based upon clustering of the data. Data points closer to centroid zero are non-malicious, and data points closer to centroid one are considered malicious or interesting enough for retention during log reduction.

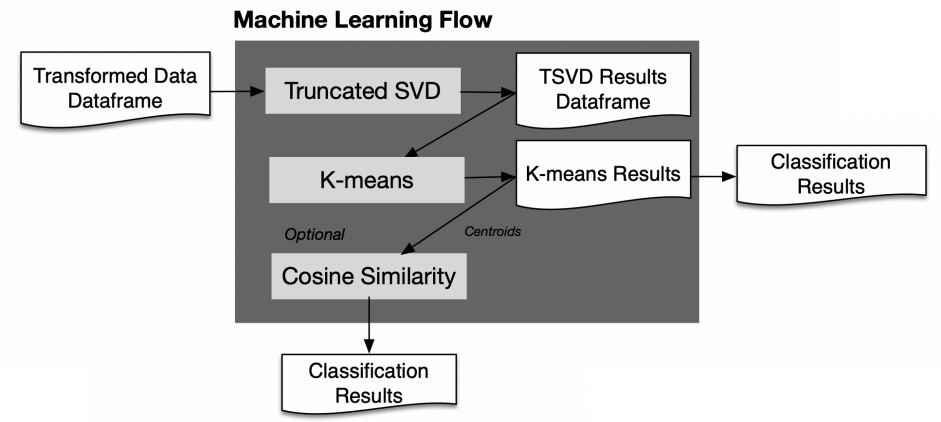

Figure 6.4: Machine Learning Processing.

If the user specifies the command line option triggering the optional third step, the second step using k-means instead outputs the cluster centroids and does not perform classification. A secondary experiment used cosine similarity as an alternative detection methodology to the default Euclidean distance used in the `sklearn` library's implementation of k-means. (While performing research, a plugin for the optional third step using Euclidean distance for classification matched the classification results when using k-means. These results are not included here.)

6.2 Synthetic Data Experiments

AIT provided labels with the synthetic data set. Accurately labeled data allowed comparison between experimental results and known ground truth. Table 6.1 contains the basic label information statistics used to compare experiment results against when determining the accuracy of identifying potentially malicious log entries and potential log reduction. The true-positive (TP) and true-negative (TN) columns contain the total number of log lines labeled as malicious and benign, respectively.

Apache access log entries include a status code indicating the success or failure of the client's request. In addition to indicating a legitimate, routine error processing a client's request, an error code may indicate a failed attack or probe against the server. These types of log entries indicating malicious attacks are easy to extract using non-machine learning algorithms because of the error status code. Of particular interest are the 200 status codes indicating success – a client-initiated a request, the server processed and returned results. A successful attack against the server may not always trigger an error code but instead trigger a success code. The synthetic data set includes maliciously labeled 200 status code log entries. Successfully identifying these types of log entries would indicate the approach is not entirely relying on the status code and

able to identify successful malicious attacks not triggering an error status code. Table 6.1 includes how many of the labeled malicious log lines are of this type.

AIT Data Set	TP	TN	# Log Lines	Malicious/200 Status
mail.cup.com	6,789	141,745	148,534	475
mail.insect.com	6,973	162,367	169,340	665
mail.onion.com	6,429	75,534	81,963	129
mail.spiral.com	7,370	93,075	100,445	1,047

Table 6.1: Synthetic Data Labeled Data Statistics.

6.2.1 Experiment #1: URL Splitting

The first experiment involved splitting the URL component into additional features. During the processing of each log line, the URL split into a new feature at each occurrence of the forward-slash ("/") character. A maximum number of splits limited the additional feature generation to eight splits, with the last feature containing any remainder of the unsplit URL. The machine learning chain first applied truncated SVD to the transformed data, reducing the dimensionality of the transformed dataframe to two dimensions. K-means, the next algorithm in the chain, then identified to which cluster centroid (named "0" or "1") each log line belongs. An extension of the basic machine learning chain incorporated a third set performing anomaly identification. An anomaly identification plugin used the centroids calculated by the k-means plugin to calculate the cosine similarity between the centroids and each log line. The cosine similarity value determined which cluster the log line belongs. For all detection methodologies, assignment to centroid "1" indicated the line logged potentially malicious or anomalous behavior. Assignment to centroid "0" indicated the logged line was not malicious or anomalous. Log file reduction occurs by removing log lines belonging to centroid "0".

Table 6.2 and Table 6.3 contain results captured for experiment #1 using the `mail.cup.com` server using both detection methodology variants. The experiment results show 48% maximum accuracy correctly predicting true positives using the k-means based detection method. Out of the true positives, the experiment correctly identified 74% of the log lines with a "success" status code, indicating the approach does not rely on error-based status codes alone to identify malicious log lines. The ability to identify success status code log lines as malicious is critical to identifying successful attacks that do not trigger an error status code. Using cosine similarity with the k-means centroids resulted in 33% of the true positives correctly identified and 50% of the successful status code malicious log lines being identified.

Server	Detection Methodology	TP	FP	TN	FN	# 200 Status
mail.cup.com	Kmeans + Cosine Similarity	2,267	78,452	63,293	4,522	237
mail.cup.com	Kmeans	3,227	81,609	60,136	3,562	353

Table 6.2: Classification Performance.

Using the better performing k-means detection methodology results in a retention of 42.88% of the original log file (Table 6.3). Although the experiment showed significant file reduction, the true positive rate coupled with the large number of false positives indicates this plugin combination is not sufficiently performant for cybersecurity use.

Server	Detection Methodology	Possible File Reduction %
mail.cup.com	Kmeans + Cosine Similarity	45.66%
mail.cup.com	Kmeans	42.88%

Table 6.3: File Reduction Performance.

Conducting the same experiment across the remaining servers included in the AIT data set showed similar results. (Refer to Appendix A for experiment results collected for all servers in the AIT data set.) In some cases, such as the `mail.onion.com` server, results were significantly poorer. File reduction for this server equated to removing only 0.04% of the log lines from the file using k-means as the detection methodology. Coupling k-means with cosine similarity for detection resulted in a retention of approximately 66% of the file, however, the true positive rate was 1%.

Two-dimensionally graphing the detection results (Figure 6.5) shows a lack of distinct clusters and clear separation between non-malicious and malicious log lines. Although this experiment's results improved upon a previous experiment where the URL was not split, the results and graph clearly indicate improvements are required before the approach could be utilized in a real-world application. Based on the increased performance seen when incorporating URL splitting, another feature extracted from the log file was chosen for splitting in a follow-up experiment described in the next section.

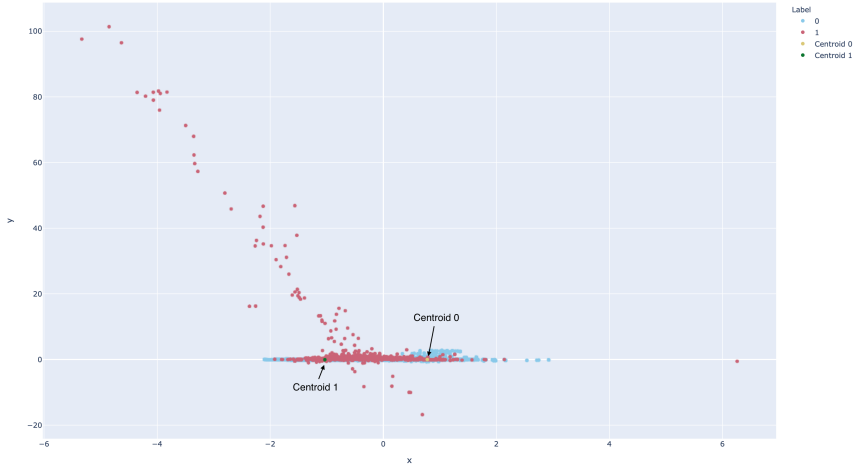

Figure 6.5: URL splitting, K-Means Detection.

6.2.2 Experiment #2: URL and User-Agent Splitting

The second experiment built on the first, performing the same feature extraction as the first experiment but adding user agent splitting (Appendix F) to create additional features. Before splitting the user agent, semicolons (";") replaced all occurrences of open and close parenthesis ("(", ")") characters. Splitting then occurs on all semicolons in the user agent string for a maximum of 10 splits. If the user agent contains more than ten semicolons, the remaining portion of the user agent string forms the last new feature.

Table 6.4 contains the performance results after incorporating user agent splitting using the `mail.cup.com` server data. True positive identification increased to 99%, with a slight increase in false positives. False negatives decreased significantly down to 98 from 3,562 as previously seen using URL splitting only. Identification of malicious log lines triggering a success status code increased slightly from 353 to 383 (81%). The results were identical using either detection methodology. A slight improvement in file reduction occurred, resulting in only 41.44% retention of the original log file (Table 6.5).

Server	Detection Methodology	TP	FP	TN	FN	# 200 Status
mail.cup.com	Kmeans + Cosine Similarity	6,691	80,296	61,449	98	383
mail.cup.com	Kmeans	6,691	80,296	61,449	98	383

Table 6.4: Classification Performance Incorporating User-Agent Splitting.

Server	Detection Methodology	Possible File Reduction %
mail.cup.com	Kmeans + Cosine Similarity	41.44%
mail.cup.com	Kmeans	41.44%

Table 6.5: File Reduction Incorporating User-Agent Splitting.

Splitting the user agent and the URL improved separation between non-malicious and malicious log entries, creating more distinct clusters (Figure 6.6). These improvements contributed to the improved performance in true positive identification. Similar improvements were seen with the remaining servers' log files in the synthetic data set (Appendix A and Appendix B).

Figure 6.6: URL and User-Agent Splitting, K-Means Detection

6.3 Private Data Experiments

The next research phase applied the detection methodologies used with the synthetic data experiments to the real-world data. Recreating the experiments using real-world data determines if the approach can potentially transfer to real-world application if similar or better performance occurs. The real-world data does not include the user agent field, which required slight alterations to the synthetic log data experiments. The first experiment does not perform any splitting of the components extracted from the log files as features. The second experiment incorporates URL splitting, matching synthetic data experiment one.

Table 6.6 contains basic statistics for the Apache log files for each of the servers in the real-world data set. The data set did not include labels; therefore, an automated labeling approach was developed to verify results. Based on the servers' use case, log lines containing HTTP status codes of greater than and equal to 400 but less than 600 and not equal to 401 were considered to contain potential indicators of malicious attack. A second log analysis phase used a manual review of the logs to identify malicious log entries not identified using the automated labeling script. Careful, manual examination of the log files did not reveal successful attacks with HTTP success status codes as the servers are well maintained and updated. Therefore, with this real-world data set, the approach's performance in correctly identifying successful attacks resulting in success status codes cannot be determined.

Server	TP	TN	# Log Lines
alpha	18,990	161792	180,782
beta	15,671	56817	72,488
gamma	18,861	49581	68,442
delta	57,505	380703	438,208

Table 6.6: Real-world Basic Log Statistics.

Execution of experiments with the real-world data showed an additional check needed in the cosine similarity detection plugin. Results using the synthetic data always resulted in centroid one identifying the cluster containing malicious log lines. Using the real-world data, cluster one did not always correlate to the malicious cluster. An additional check during anomaly detection determined which cluster contained the most error status code log lines. If centroid one does not have the greater number, the check reverses the classifications (Figure 6.7). In real-world environments, log lines are heavily imbalanced – malicious attacks occur far less frequently than routine, non-malicious client requests.

```
if centroidzero > centroidone:
    print("\t\tCENTROIDS REVERSED:  flipping labels")
    classes = np.apply_along_axis(self.flip_classes, axis=1, arr=classes)
return classes
```

Figure 6.7: Cosine similarity classification results check.

6.3.1 Experiment #1: No Splitting

The first experiment using the real-word data did not perform splitting on any extracted components from the log file to create additional features. Processing each log line simply extracted the essential components from the file, transformed them as needed, and performed detection of log entries indicating malicious behavior. Table 6.7 contains the performance results using k-means alone or k-means combined with cosine similarity for detection. Using k-means combined with cosine similarity resulted in a true positive identification rate of 99.99%, with only one false negative. Although the false positive rate increased significantly, for cybersecurity purposes, it is offset by the significant and vital reduction in false negatives.

Server	Detection Methodology	TP	FP	TN	FN
alpha	Kmeans + Cosine Similarity	18,989	7,938	153,854	1
	Kmeans	17,543	1,705	160,087	1,447

Table 6.7: Classification Performance Without Splitting.

As shown in Table 6.8, file reduction using either detection methodology is significant. Although a slight decrease of approximately 4% occurs using cosine similarity with k-means (due to increased false positives), a file reduction rate of 85% is still significant.

Server	Detection Methodology	Possible File Reduction %
alpha	Kmeans + Cosine Similarity	85.11%
	Kmeans	89.35%

Table 6.8: File Reduction Without Splitting.

Graphing the two-dimensional data results (Figure 6.8) shows a good separation between non-malicious (blue) and malicious (red) log lines. Although generally acceptable performance was obtained, real-world application requires as a low false-positivity rate as possible. Decreased false positivity rates lead to improved log file reduction and less work later for other tools or cybersecurity analysts reviewing the results.

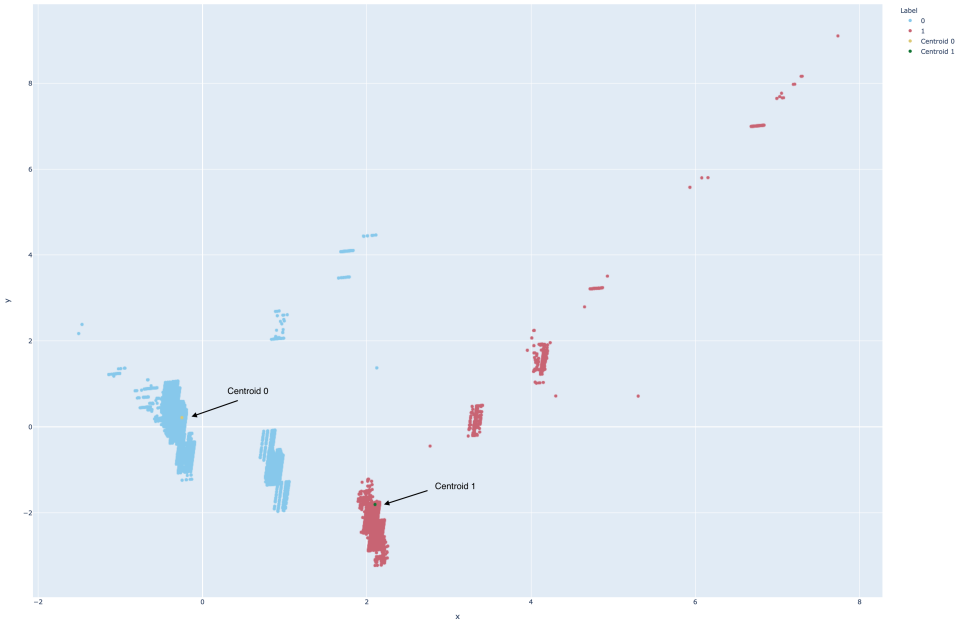

Figure 6.8: No Splitting, K-Means Detection.

6.3.2 Experiment #2: URL Splitting

The next experiment split the URL extracted from the real-world log data. Like experiment #1 using synthetic data, the URL split into a new feature at each occurrence of the forward-slash ("/") character. A maximum number of splits limited the additional feature generation to eight splits, with the last feature containing the remainder of the unsplit URL. Table 6.9 shows the performance is similar to experiment #1, however a significant reduction in false positives and false negatives using k-means only as the detection methodology indicates significantly better performance for cybersecurity application.

Server	Detection Methodology	TP	FP	TN	FN
alpha	Kmeans + Cosine Similarity	18,990	8,346	153,446	0
	Kmeans	18,990	978	160,814	0

Table 6.9: Classification Performance with URL Splitting.

Although file reduction is slightly decreased when compared to the experiment not using URL splitting (Table 6.10), this reduced file contains all true positives and minimal false positives.

Server	Detection Methodology	Possible File Reduction %
alpha	Kmeans + Cosine Similarity	84.88%
	Kmeans	88.95%

Table 6.10: Potential File Reduction using URL Splitting.

The resulting two-dimensional graph (Figure 6.9) shows the improved orientation of the clusters, centroids, and grouping over the previous experiment. This explains the improved performance URL splitting introduces over not splitting the URL in the previous experiment. Similar improvement occurred when executing the same experiments for the remaining real-world server data (Appendix C and Appendix D).

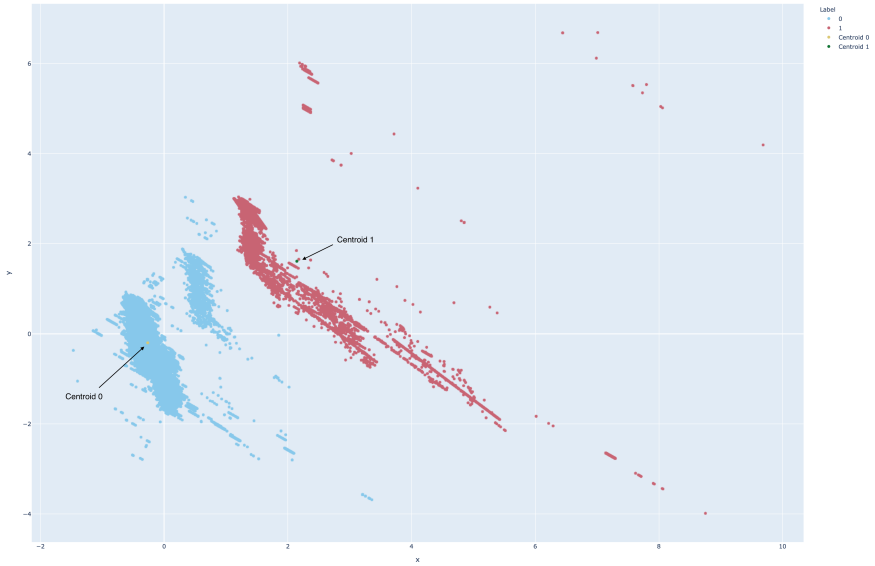

Figure 6.9: URL Splitting and K-means Detection.

6.4 Resource Utilization Experiments

An important factor involved in the decision to transition research to real-world application is the consumption of available resources. Resource consumption for software encompasses execution time, memory, CPU utilization, and data storage. If resource utilization is too excessive, successful research may not successfully transition to real-world utilization due to the monetary

costs of the necessary resources. For example, for cybersecurity applications, execution time is essential as delays in receiving notifications of malicious behavior may allow hackers the time needed to burrow deeper into a corporation's network, making it significantly more complex, and therefore costly, to recover. Delays in notification may also grant an attacker the time needed to exfiltrate business or government sensitive data and documents. This section discusses the resource utilization of the approach implemented in this research project.

Two experiments using one of the real-world Apache access log files and two of the file processing plugins were executed to examine resource utilization. Experiments were performed three times, capturing execution time (in seconds) and memory utilization. The average execution time for each workflow stage and total execution time was calculated to determine time resource utilization. A similar process was followed for calculating memory utilization.

Experiment one used the file processing plugin that only split the request URL. The processed data then ran through the TSVD and k-means machine learning plugins before the cosine similarity plugin calculated which k-means centroid each row was most similar. For each significant step in the workflow, the Python framework captured and displayed the execution time shown in Table 6.11 below.

The average total execution time was 50.5 seconds for 438,208 log lines, equating to a throughput of 8,685 log lines per second. The speed with which the experiment executed indicates the single-threaded Python implementation is suitable for actual use. Execution time may be reduced slightly by optimizing some code sections, multi-threading file processing, or possibly re-implementing the code in a compiled language such as C.

Execution Run	File Processing	TSVD	K-Means	Cosine Similarity	Total
1	49.504	1.359	0.796	0.718	52.377
2	47.169	1.227	0.782	0.690	49.869
3	46.418	1.238	0.785	0.684	49.125
Average Time:	**47.697**	**1.275**	**0.788**	**0.697**	**50.457**

Table 6.11: Experiment one execution times (in seconds).

Memory utilization did not vary between each execution run (Table 6.12), indicating exact, consistent data processing for each run. As currently implemented for research and experimentation, the total amount of memory used is 1,309MB. A real-world implementation would not maintain all data structures throughout execution, resulting in the brief maximum usage of 771MB (raw data plus initial data frame) before dropping to consistent use of 538MB while the machine learning and anomaly identification steps are performed. Memory utilization is not extensive, well below the maximum amount of memory typical installed in servers and even modern desktop systems. Memory utilization would not be a consideration preventing the approach used in this research project from real-world use.

The *raw data* column in Table 6.12 represents the amount of memory used in the data structure initially holding the log data read in from the log file and broken into features. The *data frame*

column is the amount of memory used after transforming the raw data in the initial data structure into numerical values suitable for machine learning and stored in a Pandas data frame. The last column, *reduced data frame*, shows the amount of memory used after dropping features (such as the date and time) from the Pandas data frame that were not used for the machine learning algorithms.

Execution Run	Raw Data	Data Frame	Reduced Data Frame
1	173MB	598MB	538MB
2	173MB	598MB	538MB
3	173MB	598MB	538MB
Average Memory:	**173MB**	**598MB**	**538MB**

Table 6.12: Experiment one memory usage (in MB).

CPU utilization was consistently 100% throughout the experiments' execution, with a brief spike exceeding 100%. This is unsurprising as the framework is single-threaded and should execute at the maximum speed of one CPU core until finished as there are no external, remote data dependencies or dependencies on user input, potentially causing pauses while executing. The brief spike exceeding 100% occurs when the machine learning algorithms execute (TSVD and K-Means), indicating the scikit-learn implementations may be multithreaded. Although using 100% of a single core continuously throughout execution, this would not negatively affect a decision to implement a real-world version. The implementation in Python executes in short enough time to not detrimentally impact other processes running simultaneously on the same system.

6.5 Additional Log Files and Plugins

Additional experiments applied the approach to alternative log files, such as `imapd.log` contained in the private log data. Unlike httpd log files forming two clusters when graphed, imapd log files formed three. K-means did not perform well positioning centroids (Figure 6.10), requiring exploration of alternative clustering machine learning plugins such as density-based spatial clustering of applications with noise (DBSCAN). DBSCAN performed much better positioning centroids within the clusters (Figure 6.11), however execution time significantly increased.

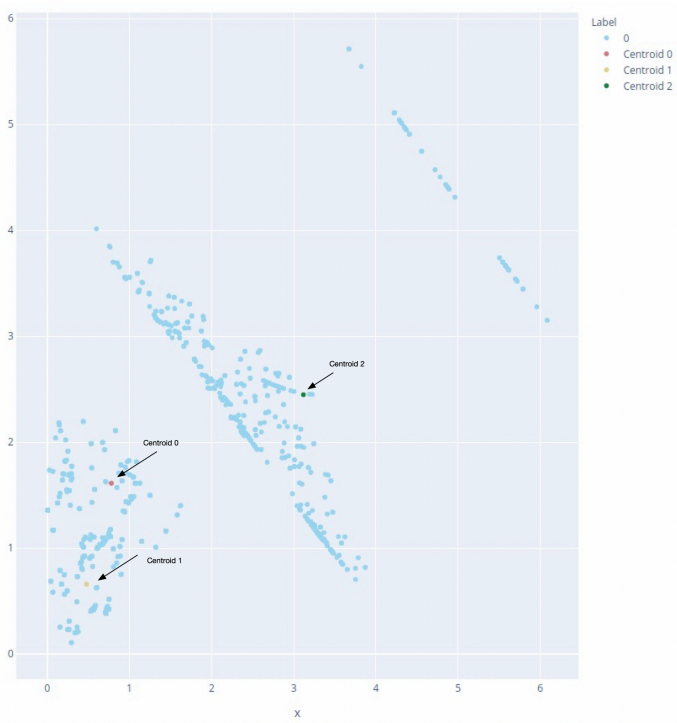

Figure 6.10: K-means and Three Clusters

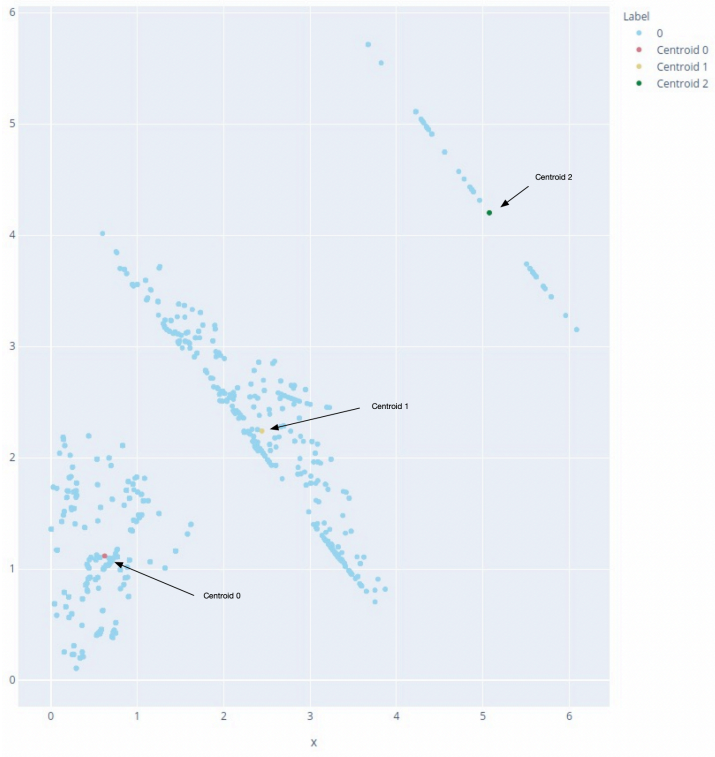

Figure 6.11: DBSCAN and Three Clusters

Chapter 7
Conclusion

The approach presented in this paper applying an unsupervised, machine learning chain achieved true-positive, false-negative, and file reduction performance rates sufficient for real-world cybersecurity application. Cybersecurity analysts require accurate tools, otherwise they risk putting their organizations at risk by missing identification of malicious activity. Successfully retaining these types of log entries during log reduction now adds a new data source for cybersecurity analysts as encryption continues to impact network-based detection methods

Additionally, low resource utilization (CPU, memory, and time) eliminates potential resistance to deployment because of increased production environment costs. Tuning of the implementation presented here and available at the GitHub repository [24] may further reduce resource utilization.

Unfortunately, other log file types were not available with log entries triggered by malicious actions. This prevented the complete development and testing of file processing plugins. Based upon the successful use of the approach with httpd log files from different real-world servers, the technique should transfer with proper feature extraction and transformation. Rudimentary testing using imapd log files does show promise as distinct clusters formed using the same k-means plugin.

Chapter 8
Future Work

Although the application of TSVD coupled with k-means performed sufficiently for transfer to real-world use, there are additional opportunities to extend the research further. The first exploration area examines the technique's ability to successfully identify attacks resulting in httpd 200 ("OK") status codes using real-world web server log data. Ensuring the detection of these log entries is critical to identifying and retaining the entries in the reduced log file for analysis by cybersecurity experts or other tools. If not retained, compromises may go undetected.

While important to retain log entries indicating malicious activity, cybersecurity analysts may benefit from related but non-malicious log entries. A further refinement would identify and keep additional log entries allowing cybersecurity analysts to thoroughly analyze, deconflict, and report malicious activity without retrieving additional log entries from the originating device. This speeds the analysis process, and reduces requests to other personnel outside the security group.

The approach requires testing with other system log file types. Although part of the research and experiments tested the approach using real-world log files of various kinds, none contained compromises. This prevented a precise determination if the technique applies as successfully to other log file types as it does to httpd access log files. Log files containing entries indicating benign and malicious activity need to be synthesized or obtained in an appropriate quantity. Feature engineering, employing the splitting technique used with URLs and user agent fields, would be an extensive part of adapting the approach to new types of log files.

While performing the research, experiment results occasionally reversed the k-means centroids, so centroid one indicated malicious log lines and centroid zero indicated non-malicious log lines. As documented in this paper, a methodology detected and reordered the centroids. Additional research on centroid reversals would determine if more robust methods exist for detecting these reversals to ensure accuracy.

References

[1] Cisco, "Snort - Network Intrusion Detection & Prevention System," [Online]. Available: https://www.snort.org. [Accessed 26 June 2020].

[2] Y. Y. Al-Salqan, "Future Trends In Internet Security," in *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Tunis, 1997.

[3] Fortinet, Inc., "As the Holiday Season Draws Near, Mobile Malware Attacks Are Prevalent," 14 November 2018. [Online]. Available: https://www.fortinet.com/blog/industry-trends/as-the-holiday-season-draws-near--mobile-malware-attacks-are-pre. [Accessed 26 June 2020].

[4] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating System Principles*, Big Sky, MT, 2009.

[5] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert and C. Kemp, "User Behavior Anomaly Detection for Application Layer DDoS Attacks," in *2017 IEEE International Conference on Information Reuse and Integration*, San Diego, CA, 2017.

[6] R. Ritchey and R. Perry, "Machine Learning Toolkit for System Log File Reduction and Detection of Malicious Behavior," in *IEEE International Conference on Computer Communications (INFOCOM)*, Virtual (Online), 2021.

[7] W. Xu, L. Huang, A. Fox, D. Patterson and M. Jordan, "Experience Mining Google's Production Console Logs," in *SLAML'10: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, Vancouver, BC, 2010.

[8] A. Juvonen, T. Sipola and T. Hamalainen, "Online Anomaly Detection Using Dimensionality Reduction Techniques for HTTP Log Analysis," *Computer Networks,* vol. 91, pp. 45-56, November 2015.

[9] Y.-J. Lee, Y.-R. Yeh and Y.-C. F. Wang, "Anomaly Detection via Online Oversampling Principal Component Analysis," *IEEE Transactions on Knowledge and Data Engineering,* vol. 25, no. 7, pp. 1460-1470, 2013.

[10] R. Abdulhammed, M. Faezipour, H. Musafer and A. Abuzneid, "Efficient Network Intrusion Detection Using PCA-Based Dimensionality Reduction of Features," in *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, Istanbul, Turkey, 2019.

[11] R. P. Ritchey, G. G. Shearer and K. D. Renard, "Naive Bayes Log File Reduction and Analysis," U.S. Army Research Laboratory, Adelphi, MD, 2019.

[12] Python Software Foundation, "Welcome to Python," [Online]. Available: https://www.python.org. [Accessed 6 July 2020].

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau and Brucher, "scikit-learn: Machine Learning in Python," [Online]. Available: https://scikit-learn.org/stable/index.html. [Accessed 06 July 2020].

[14] NumPy, "NumPy," [Online]. Available: https://numpy.org. [Accessed 6 July 2020].

[15] NumFOCUS, "pandas - Python Data Analysis Library," [Online]. Available: https://pandas.pydata.org. [Accessed 6 July 2020].

[16] R. P. Ritchey and T. W. Parker, "Simple Plugin Methodology in Python," U.S. Army Research Laboratory, Adelphi, MD, 2014.

[17] M. Howell, "The Missing Package Manager for macOS (or Linux)," [Online]. Available: https://brew.sh. [Accessed 30 January 2021].

[18] I. Jolliffe, "Principal Component Analysis: A Beginner's Guide - I. Introduction and Application," *Weather,* vol. 45, pp. 375-382, 1990.

[19] S. Marsland, "6.2 Principal Component Analysis (PCA)," in *Machine Learning An Algorithmic Perspective*, Boca Raton, FL, CRC Press, 2015, pp. 133-137.

[20] T. U. K. Archive, "KDD Cup 1999 Data," 28 October 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. [Accessed 2 July 2020].

[21] L. F. Sikos and K.-K. R. Choo, Data Science in Cybersecurity and Cyberthreat Intelligence, Springer, 2020, p. 118.

[22] J. Bhatia, T. D. Breaux, L. Friedberg, H. Hibshi and D. Smullen, "Privacy Risk in Cybersecurity Data Sharing," in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, Vienna, Austria, 2016.

[23] L. Max, S. Florian, W. Markus, H. Wolfgang and R. Andreas, "AIT Log Data Set v1.0 | Zenodo," 21 March 2020. [Online]. Available: https://zenodo.org/record/3723083#.Xv4HVy2z3UK. [Accessed 2 July 2020].

[24] R. P. Ritchey and R. Perry, "MLTK-Log-Reduction-Detection," 17 February 2021. [Online]. Available: https://github.com/pritchey/MLTK-Log-Reduction-Detection. [Accessed 17 February 2021].

Appendix A
Synthetic Data Experiment #1 URL Splitting Results

Table A.1 contains experiment #1 results for all servers included in the AIT log data set. The data presented covers the first portion of the experiment applying the two detection methodologies and splitting the URL into additional features.

Server	Detection Methodology	TP	FP	TN	FN	# 200 Status
mail.cup.com	Kmeans + Cosine Similarity	2,267	78,452	63,293	4,522	237
	Kmeans	3,227	81,609	60,136	3,562	353
mail.insect.com	Kmeans + Cosine Similarity	3,987	69,175	93,192	2,986	216
	Kmeans	3,380	63,729	98,638	3,593	204
mail.onion.com	Kmeans + Cosine Similarity	84	54,595	20,939	6,345	28
	Kmeans	6,395	75,534	0	34	129
mail.spiral.com	Kmeans + Cosine Similarity	6,466	30,948	62,127	904	250
	Kmeans	30	0	93,075	7,340	0

Table A.1: Splitting URL

The table below (Table A.2) shows the log file reduction possible and the percentage of correctly classified 200 status log lines for the first portion of the experiment, which only split the URL.

Server	Detection Methodology	Possible File Reduction %	% of 200 Identified
mail.cup.com	Kmeans + Cosine Similarity	45.66%	49.89%
	Kmeans	42.88%	74.32%
mail.insect.com	Kmeans + Cosine Similarity	56.80%	32.48%
	Kmeans	60.37%	30.68%
mail.onion.com	Kmeans + Cosine Similarity	33.29%	21.71%
	Kmeans	0.04%	100.00%
mail.spiral.com	Kmeans + Cosine Similarity	62.75%	23.88%
	Kmeans	99.97%	0.00%

Table A.2: Splitting URL File Reduction and 200 Status

Appendix B
Synthetic Data Experiment #2 URL and User-Agent Splitting Results

Table B.1 contains results from the second portion of the first experiment. This portion of the experiment also applied the two detection methodologies but added splitting of the user agent information to create additional features.

Server	Detection Methodology	TP	FP	TN	FN	# 200 Status
mail.cup.com	Kmeans + Cosine Similarity	6,691	80,296	61,449	98	383
	Kmeans	6,691	80,296	61,449	98	383
mail.insect.com	Kmeans + Cosine Similarity	442	91,578	70,789	6,531	416
	Kmeans	443	91,601	70,766	6,530	417
mail.onion.com	Kmeans + Cosine Similarity	6,399	4,603	70,931	30	102
	Kmeans	6,350	2	75,532	79	53
mail.spiral.com	Kmeans + Cosine Similarity	6,575	8,559	84,516	794	253
	Kmeans	6,296	0	93,075	1,074	19

Table B.1: Splitting URL and User-Agent

The table below (Table B.2) shows the log file reduction possible and the percentage of correctly classified 200 status log lines for the second portion of the experiment splitting both the URL and user agent.

Server	Detection Methodology	Possible File Reduction %	% of 200 Identified
mail.cup.com	Kmeans + Cosine Similarity	41.44%	0.00%
	Kmeans	41.44%	80.63%
mail.insect.com	Kmeans + Cosine Similarity	45.66%	62.56%
	Kmeans	45.65%	62.71%
mail.onion.com	Kmeans + Cosine Similarity	86.58%	79.07%
	Kmeans	92.25%	41.09%
mail.spiral.com	Kmeans + Cosine Similarity	84.93%	24.16%
	Kmeans	93.73%	1.81%

Table B.2: Splitting URL and User-Agent File Reduction and 200 Status

Appendix C
Real-World Data Experiment #1 No Splitting

Table C.1 contains results using two detection methodologies with the real-world httpd log files. The results do not include the identification performance of 200 status malicious log entries. The real-world data did not include successful attacks resulting in a malicious attack triggering the logging of a 200 status code line in the log files. No additional features were created by splitting extracted fields from the log files during this experiment.

Server	Detection Methodology	TP	FP	TN	FN
alpha	Kmeans + Cosine Similarity	18,989	7,938	153,854	1
	Kmeans	17,543	1,705	160,087	1,447
beta	Kmeans + Cosine Similarity	15,671	1,162	55,655	0
	Kmeans	0	56,125	692	15,671
gamma	Kmeans + Cosine Similarity	18,476	807	49,159	0
	Kmeans	18,476	385	49,581	0
delta	Kmeans + Cosine Similarity	57,505	7,326	373,377	0
	Kmeans	57,505	3,707	376,996	0

Table C.1: No Splitting

The table below (Table C.2) contains performance results regarding file reduction accomplished for each detection methodology for each server. No additional features were created by splitting extracted fields from the log files during this experiment.

Server	Detection Methodology	Possible File Reduction %
alpha	Kmeans + Cosine Similarity	85.11%
	Kmeans	89.35%
beta	Kmeans + Cosine Similarity	76.78%
	Kmeans	22.57%
delta	Kmeans + Cosine Similarity	71.83%
	Kmeans	72.44%
gamma	Kmeans + Cosine Similarity	85.21%
	Kmeans	86.03%

Table C.2: No Splitting File Reduction

Appendix D
Real-World Data Experiment #2 URL Splitting

Table D.1 contains results using two detection methodologies with the real-world httpd log files. The results do not include the identification performance of 200 status malicious log entries. The real-world data did not contain successful attacks resulting in a malicious attack triggering the logging of a 200-status code line in the log files. This experiment performed additional feature extraction by splitting the URL into additional features.

Server	Detection Methodology	TP	FP	TN	FN
alpha	Kmeans + Cosine Similarity	18,990	8,346	153,446	0
	Kmeans	18,990	978	160,814	0
beta	Kmeans + Cosine Similarity	15,671	3,895	52,922	0
	Kmeans	61	56,665	152	15,610
gamma	Kmeans + Cosine Similarity	18,476	807	49,159	0
	Kmeans	18,861	0	49,581	0
delta	Kmeans + Cosine Similarity	56,965	136,488	244,215	540
	Kmeans	643	284,244	96,459	56,862

Table D.1: Splitting URL

The table below (Table D.1) contains performance results regarding file reduction accomplished for each detection methodology for each server. This experiment performed additional feature extraction by splitting the URL into additional features.

Server	Detection Methodology	Possible File Reduction %
alpha	Kmeans + Cosine Similarity	84.88%
	Kmeans	88.95%
beta	Kmeans + Cosine Similarity	73.01%
	Kmeans	21.74%
gamma	Kmeans + Cosine Similarity	71.83%
	Kmeans	72.44%
delta	Kmeans + Cosine Similarity	55.85%
	Kmeans	34.99%

Table D.2: Splitting URL File Reduction

Appendix E
log_anomaly_identifier.py

The code below forms the main script file, `log_anomaly_identifier.py`, for the framework. The researcher executes this script, passing command line parameters indicating which plugins to use and the log file for processing. The public git code repository [24] contains all source code, including the code listing below, developed as part of the research project.

```
import glob # Imported for file directory listing capabilities.
import sys # Imported for access to command line parameters
import os # Imported for file path manipulation capabilities.
import argparse # Imported for handling command line arguments.
import time # Imported so execution time can be captured
import pandas # Imported to access built in ability to easily save data frames to
files.
import numpy as np # Imported to access numpy data types.
from pathlib import Path # Imported to facilitate autocreation of path to save data
into.

# Dictionaries to store the various plugins
file_processor = {}
machine_learning = {}
output = {}
anomaly_reduction = {}

plugin_parameters = {}

label_data = pandas.DataFrame([])
ml_results = {}
ml_results2 = {}
anomaly_reduction_results = {}

# Integers used to store execution time for various steps performed as
# part of normal execution.
file_processor_time = 0
machine_learning_time = 0
machine_learning_time2 = 0
save_time = 0
label_time = 0
output_time = 0
anomaly_reduction_time = 0

def load_plugins(directory, whichDictionary):
    """Load available plugins into separate dictionaries based upon their plugin type.
```

```
    Args:
        directory (String): Path/directory containing the plugins to load for a
particular type.
        whichDictionary (Dictionary): Empty dictionary to load the plugins into.
    """
    myPlugIns = sorted(glob.glob(directory + "/*.py"))
    for file in myPlugIns:
        # Extract just the first part of the .py file name.
        name = file.split("/")[1].split(".")[0]
        print ("Loading %s plugin: %s" % (directory, name))

        # Dynamically set the PYTHONPATH so the user doesn't have to. It assumes
        # the plugins are contained in subdirectories where the main file lives.
        path = os.path.dirname(sys.argv[0])
        if len(path) == 0:
            path = "."
        sys.path.append(path + "/" + directory)

        # Import the plugin module temporarily long enough to instantiate an object
        # which is stored in a globally accessible dictionary.
        tempModule = __import__(name)
        whichDictionary[name] = tempModule.simPlugin()

def available_plugins():
    """When the user uses the help command line option, this builds a formatted text
string describing
    all of the different plugins and what they do.

    Returns:
        String: Formatted string containing all the help output from each plugin.
    """
    text = "Available 'file processing' plug-ins:\n"
    for i in file_processor:
        text += file_processor[i].print_help() + "\n"

    text += "\n\nAvailable 'machine learning' plug-ins:\n"
    for i in machine_learning:
        text += machine_learning[i].print_help() + "\n"

    text += "\n\nAvailable 'output' plug-ins:\n"
    for i in output:
        text += output[i].print_help() + "\n"

    text += "\n\nAvailable 'anomaly' plug-ins:\n"
    for i in anomaly_reduction:
```

```
        text += anomaly_reduction[i].print_help() + "\n"
    return text

def save_data(save_path, save_data):
    """Loops through the supplied dictionary of internal data and saves each to a CSV
file.

    Args:
        save_path (String): Path where the data should be saved.
        save_data (Dictionary): Dictionary containing all the Pandas DataFrames and
dictionaries of internal data to be saved to disk.
    """
    print("--Beginning:  Saving Machine Learning Data")
    print("\tSaving data to path: %s" % save_path)
    Path(save_path).mkdir(parents=True, exist_ok=True)
    for i in save_data:
        # Ignore the "graph" entry if present - it's used to indicate which matrix
        # is used for graphing from the results returned by an ML plugin.
        if i != "graph":
            print("\tSaving Pandas data frame: %s" % i)
            pandas.DataFrame(save_data[i]).to_csv(save_path + "/" + i + ".csv")

    print("--Finished:  Saving Machine Learning Data")

def get_labels(label_data_file):
    """Extract label data from the file specified by the user on the command line.
The assumed
    format for the label file is a CSV file where several different patterns can
trigger a "1"
    label instead of a "0" label:

    <string>,0
    <string>,<string>
    0,<string>

    Where <string> can be any value other than a 0 (zero).

    Args:
        label_data_file (String): Path and name of the file containing label data.

    Returns:
        Pandas DataFrame: Vector containing the label data with 0's and 1's.
    """
    label_data = []
    print("--Beginning:  Reading label data")
    source_label_data = pandas.read_csv(label_data_file, header=None)
```

```
    print("--Finished:  Reading label data")

    # There are two different label files being used:
    # Type 1:
    #   Uses purely numeric values in comma separated file.  A "1"
    #   appears in the 2nd column to indicate malicious/interesting row.
    #
    # Type 2:
    #   Provided by the AIT data set using two columns.  If both column entries
    #   are "0", that row is benign.  If either column contains text, that
    #   row is malicious/interesting.
    #
    # Handle both situations.
    print("--Beginning:  Processing label data")
    for i in source_label_data.index:
        if (type(source_label_data.iloc[i,0]) is np.int64) and
(type(source_label_data.iloc[i,0]) is np.int64):
            # If both values are a 0, append 0 indicating non-malicious entry.
            if (source_label_data.iloc[i,0] == 0) and (source_label_data.iloc[i,1] ==
0):
                label_data.append(0)
            else:
                label_data.append(1)
        else:
            # We're processing AIT file because the datatype is not an np.int64.
            # label data AND we found a malicious line.
            if (source_label_data.iloc[i,0] == "0") and (source_label_data.iloc[i,1]
== "0"):
                label_data.append(0)
            else:
                label_data.append(1)
    print("--Finished:  Processing label data")
    return pandas.DataFrame(label_data)

def process_arguments():
    """Setup and parse command line arguments using Python's built-in capability.
This includes
    command line help to aid the user in what options are available.

    Returns:
        Dictionary: Command line options and parameters as specified by the user on
the command line.
    """
    # Setup the command line arguments supported by the Python script.
```

```
    parser =
argparse.ArgumentParser(formatter_class=argparse.RawDescriptionHelpFormatter,
epilog=available_plugins())
    parser.add_argument("-f", "--file_processor", help="The plugin to use for
processing a log file.  Required command line option.", type=str, required=True)
    parser.add_argument("-l", "--log_file", help="The log file (including directory
path if needed) to process.  Required command line option.", type=str, required=True)
    parser.add_argument("-m", "--machine_learning", help="The machine learning
algorithm to use.  Required command line option.", type=str, required=True)
    parser.add_argument("-n", "--next_machine_learning", help="The second machine
learning algorithm to use, using the results from the first.", type=str,
required=False)
    parser.add_argument("-s", "--save_path", help="The path and directory to save data
generated during execution.", type=str, required=False)
    parser.add_argument("-o", "--output", help="The plugin to use for generating some
type of output (graph, etc.).", type=str, required=False)
    parser.add_argument("-a", "--anomaly_reduction", help="The plugin to use for
identifying anomalies and reducing log file size.", type=str, required=False)
    parser.add_argument("-d", "--data_labels", help="Labels for the data used in --
log_file.", type=str, required=False)
    parser.add_argument("-p", "--plugin_options", help="Supply key=value pair options
to a plugin.  See plugin help for supported options.", nargs="*")
    return parser.parse_args()

def save_internal_data(args, raw_data, df):
    """If the command line option is specified by the user, saves internal data
structures as CSV files
    for late examination.  Useful for debugging and seeing what the algorithms are
doing with the data
    through various workflow steps.

    Args:
        args (dictionary): Command line options specified by the user
        raw_data (Pandas DataFrame): Dataframe containing the original data as read
from input file.
        df (Pandas DataFrame): Dataframe containing the transformed, machine learning
ready version of the data.
    """
    global ml_results, ml_results2, save_time
    # Grab the command line options used so they can be saved with the rest of the
data.
    v = vars(args)
    if (bool(v['plugin_options'])):
        v['plugin_options'] = " ".join(v['plugin_options']) # flatten plugin options
array
    ml_results["command_line_options"] = pandas.DataFrame(v,index=[0])
```

```
    # If saving, we want to include the raw data as well as the ML ready data.
    ml_results["raw_data"] = raw_data
    ml_results["ml_ready_data"] = df

    # If label data was provided, include that in the output.
    if (args.data_labels):
        ml_results["label_data"] = label_data

    # Check to see if a second machine learning algorithm was applied, if so we need
to grab
    # that output for saving.
    if (args.next_machine_learning):
        ml_results = {**ml_results, **ml_results2}


    # Be sure to capture the time taken to save data to disk as this can contribute to
    # overall execution time.
    start_time = time.time()
    save_data(args.save_path, ml_results)
    save_time = time.time() - start_time

def generate_output(args):
    """Depending on the output plugin specified by the user, generate output (graph,
etc.)
    as part of the workflow.

    Args:
        args (dictionary): Contains command line arguments specified by the user.
    """
    global ml_results, label_data, output_time
    if "graph" in ml_results:
        start_time = time.time()
        # Typecast the label data to a string.  This automatically triggers plotly to
use
        # discrete colors instead of gradient.
        if args.data_labels:
            label_data[0] = label_data[0].astype(str)

        # If KMeans, DBSCAN, or OPTICS was used as the 2nd machine learning algorithm,
        #  grab the centroids to plot them in the plot points colored by label.
        centroids = None
        if "KMeans_Centroids" in ml_results2:
            centroids = ml_results2["KMeans_Centroids"]
        elif "DBSCAN_Centroids" in ml_results2:
            centroids = ml_results2["DBSCAN_Centroids"]
```

```
        elif "OPTICS_Centroids" in ml_results2:
            centroids = ml_results2["OPTICS_Centroids"]
        if centroids is not None:
            if (len(label_data) == 0):
                # we have no labels, but we have centroids, build a default label set.
                label_data =
pandas.DataFrame(np.full((len(ml_results[ml_results['graph']]),1),"0", dtype =
np.str))
            # Loop through and append enough label values for the centroids as
provided.
            # pylint: disable=unused-variable
            i = 0
            for centroid in centroids:
                label_data = label_data.append([("Centroid " + str(i))],
ignore_index=True)
                i+=1
            # Append the centroids to the main data for plotting.
            ml_results[ml_results['graph']] =
np.append(ml_results[ml_results['graph']], centroids, axis=0)

        output[args.output].output(ml_results[ml_results['graph']], label_data)
        output_time = time.time() - start_time
    else:
        print("\n\nWARNING:  Machine learning algorithm does not include graphable
results.")
        print("WARNING:  Skipping graph generation.")

def output_time_stats():
    """Prints basic execution statistics about major steps in the workflow.
    """
    print("\n\n\n")
    print("------------------------------------")
    print("\tGENERAL STATISTICS")
    print("File Processing Time: %.4f seconds" % file_processor_time)
    if (args.data_labels):
        print("Data Label Processing Time: %.4f seconds" % label_time)
    print("Machine Learning Time: %.4f seconds" % machine_learning_time)
    if (args.next_machine_learning):
        print("2nd Machine Learning Time: %.4f seconds" % machine_learning_time2)
    if (args.save_path):
        print("Save ML Data Time: %.4f seconds" % save_time)
    if (args.anomaly_reduction):
        print("Anomaly Identification/File Reduction Time: %.4f seconds" %
anomaly_reduction_time)
    if (args.output):
        print("Output Generation Time: %.4f seconds" % output_time)
```

```
    print("\nTotal Execution Time: %.4f seconds" % (file_processor_time + label_time +
machine_learning_time + machine_learning_time2 + save_time + anomaly_reduction_time +
output_time))
    print("-------------------------------------\n")

if __name__ == "__main__":
    # Load plugins for reading and processing files
    load_plugins("file_processor", file_processor)
    load_plugins("machine_learning", machine_learning)
    load_plugins("output", output)
    load_plugins("anomaly_reduction", anomaly_reduction)

    # Process command line arguments
    args = process_arguments()

    # Now handle any (optional) plugin command line parameters
    if args.plugin_options:
        for pair in args.plugin_options:
            plugin_option, value = pair.split('=')
            plugin_parameters[plugin_option] = value

    # Based on the user selected plug-in, read the log file into a DataFrame.
    if (args.file_processor) and (args.file_processor in file_processor):
        # Capture start time.
        start_time = time.time()
        # update after changing file processors to take plugin_options argument:
        # raw_data, df =
file_processor[args.file_processor].process_file(args.log_file, plugin_parameters)
        raw_data, df = file_processor[args.file_processor].process_file(args.log_file)

        # Capture time taken to process the input file.
        file_processor_time = time.time() - start_time
    else:
        print("\n\nERROR:  Unknown file processor specified: %s\n\n" %
args.file_processor)
        exit()

    # See if the user is supplying labels for the data.
    if (args.data_labels):
        # Capture start time.
        start_time = time.time()
        label_data = get_labels(args.data_labels)
        label_time = time.time() - start_time

    if (args.machine_learning) and (args.machine_learning in machine_learning):
        # Capture start time.
```

```
        start_time = time.time()
        ml_results = machine_learning[args.machine_learning].machine_learning(df,
plugin_parameters)

        # Capture time taken to perform machine learning.
        machine_learning_time = time.time() - start_time
    else:
        print("\n\nERROR:  Unknown machine learning algorithm specified: %s\n\n" %
args.machine_learning)
        exit()

    # See if the user is chaining multiple machine learning algorithms together
    # (ensemble).  If so, we will use the results from the previous machine learning
    # algorithm as input to the second.
    if (args.next_machine_learning):
        if (args.next_machine_learning in machine_learning):
            # Capture start time.
            start_time = time.time()
            # The first machine learning algorithm in the chain most likely set the
"graph" entry to its final
            # output, which identifies the matrix to use as the input for the second
algorithm to use as input.
            ml_results2 =
machine_learning[args.next_machine_learning].machine_learning(ml_results[ml_results["g
raph"]], plugin_parameters)

            # Capture time taken to perform machine learning.
            machine_learning_time2 = time.time() - start_time
        else:
            print("\n\nERROR:  Unknown machine learning algorithm specified for second
algorithm: %s\n\n" % args.next_machine_learning)
            exit()
    else:
        # User did not specify a 2nd ML algorithm to use, provide empty results.
        ml_results2 = {}

    # See if the user wants to perform anomaly identification/file reduction
    # (not updated for DBSCAN or OPTICS Centroids)
    if (args.anomaly_reduction):
        if (args.anomaly_reduction in anomaly_reduction):
            # Capture start time.
            start_time = time.time()
            if "KMeans_Centroids" in ml_results2:
                if "status" in raw_data:
```

```
                    results, anomalies =
anomaly_reduction[args.anomaly_reduction].anomaly_reduction(ml_results[ml_results['gra
ph']], ml_results2["KMeans_Centroids"], raw_data["status"])
                else:
                    results, anomalies =
anomaly_reduction[args.anomaly_reduction].anomaly_reduction(ml_results[ml_results['gra
ph']], ml_results2["KMeans_Centroids"])

            # Capture time taken to perform machine learning.
            anomaly_reduction_time = time.time() - start_time

            ml_results["anomaly_reduction_results"] = results
            ml_results["identified_anomalies"] = anomalies
        else:
            print("\n\nERROR:  Unknown anomaly/reduction plugin specified: %s\n\n" %
args.anomaly_reduction)
            exit()

    # Check to see if the user wants to save data generated during the machine
learning
    # part of the workflow.
    if (args.save_path):
        save_internal_data(args, raw_data, df)

    # Check to see if the user wants some type of output generated during execution.
    if (args.output):
        # If so (because it's optional), make sure the output plugin exists.
        if (args.output in output):
            generate_output(args)
        else:
            print("\n\nERROR:  Unknown output plugin specified: %s\n\n" % args.output)
            exit()

    # Output basic execution stats.
    output_time_stats()
```

Figure E.1: log_anomaly_identifier.py Source Code

Appendix F
fp_apache_access_log_split_request_url_useragent .py

Figure F.1 below contains the file processor plugin code `fp_apache_access_log_split_request_url_useragent.py`. The plugin reads an Apache httpd access log, splitting both the URL and user-agent components. The public git code repository [24] contains all source code, including the code listing below, developed as part of the research project.

```
import apachelogs # Provides a nice parser for the Apache access log files.
import time # Imported so execution time can be captured.
import pyhash # pyhash used for access to FNV hash algorithm.
import numpy as np # Imported to access Numpy data types which underly Pandas.
import pandas as pd # Using Pandas dataframes for performance and code clarity.
from pympler.asizeof import asizeof # Used to get more accurate memory utilization.
from sklearn.preprocessing import MinMaxScaler # Used to scale values so they fall
between 0 and 1.

class simPlugin(object):
    state = ""

    # The following are used to capture features formed by splitting features
contained in
    # the log file.  The matrices will eventually be appended to main matrix, the
column name
    # list is dynamically built to match the number of split features extracted and
will be
    # used to provide meaningful names when the matrix is typecast to a Pandas data
frame.
    max_url_splits = 8
    max_user_agent_splits = 10
    dynamic_column_names=[]

    # Defines the Apache log file format we want to read.
    # See here:  https://apachelogs.readthedocs.io/en/stable/utils.html
    # And here:  http://httpd.apache.org/docs/current/mod/mod_log_config.html
    parse_format = apachelogs.COMBINED #  "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
\"%{User-Agent}i\""

    raw_data = []
    df = []

    url_splits = []
```

```
    user_agent_splits = []

    def __init__(self):
        self.state = "Initialized"

    def print_help(self):
        """Builds a formatted help text string for the plugin that's displayed when
the user specifies the help command line option.

        Returns:
            String: Formatted text string containing the plugin's help information.
        """
        text = "\tName: fp_apache_access_log_split_request_url\n"
        text += "\t\tThis file parser processes Apache access log files.  After
splitting the request into base\n"
        text += "\t\tcomponents (command (GET, PUT, ...), URL, and HTTP protocol
version), the URL portion is\n"
        text += "\t\tsplit on the '/' character into additional features.\n"
        return text

    def build_Column_Names(self, stock_names):
        """Dynamically build the list of column names for the Pandas DataFrames.  This
must be done due to dynamic splitting
        of some values extracted from the log file, and to provide meaningful columns
names in the CSV file if the user
        opts to save internal data structures for review.

        Args:
            stock_names (Array): Array containing the default, base (static) column
names that dynamically generated names will
            be appended to.
        """
        # Build our initial feature column names from the base set as extracted from
        # the log file which are not dynamically generated.
        self.dynamic_column_names = stock_names

        # To get the proper count of features, go ahead and typecast the Python matrix
to
        # a Pandas data frame.  (We'll need this later for concatenation anyhow.)
        self.url_splits = pd.DataFrame(self.url_splits)
        for i in range(self.url_splits.shape[1]):
            self.dynamic_column_names.append("url_split_" + str(i))

        # Now do the same for the split user agent string.
        self.user_agent_splits = pd.DataFrame(self.user_agent_splits)
        for i in range(self.user_agent_splits.shape[1]):
```

```
            self.dynamic_column_names.append("user_agent_split_" + str(i))

    def split_string(self, split_on, the_string, max_splits):
        """Perform deeper splitting on the URL using a specified character, but
limiting the number of splits
        to a maximum.

        Args:
            split_on (String): Character to split on.
            the_string (String): String to be split.
            max_splits (Integer): Maximum number of splits to perform

        Returns:
            Array: Array containing the values obtained by splitting the string.
        """
        # First, remove trailing split character if present so we don't end up with a
blank
        # split result at the end.
        the_string = the_string.lstrip(split_on).rstrip(split_on)

        # Now split the string the specified maximum number of times with the
remainder contained
        # in the final value if it exceeds the maximum number of splits.
        return the_string.split(split_on, max_splits)

    def url_splitter(self, url):
        """Split a given URL into pieces based on the locations of '/'.

        Args:
            url (String): String containing the requested URL from the Apache log
file.
        """
        self.url_splits.append(self.split_string("/", url, self.max_url_splits))

    def user_agent_splitter(self, user_agent):
        """
        user_agent: The user agent string from the request to be split apart.
        """
        # Make sure we have a user agent - if we don't, set it to a default value.
        if type(user_agent) != str:
            user_agent = "No_User_Agent"
        self.user_agent_splits.append(self.split_string(";", user_agent.replace("(",
";").replace(")", ";"), self.max_user_agent_splits))

    def transform(self):
```

```
        """Transform the data contained in the raw data Pandas DataFrame into
something a machine learning algorithm can use.
        """
        print("\t--Beginning:  Pandas dataframe transformation")

        # Capture start time.
        start_time = time.time()

        print("\nPre-transformation:\n")
        print(self.df.describe(include='all'))

        # Use FVN hash to transform string values into a numerical representation.
        print("\n\tBeginning:  Hashing and scaling string values.")
        hash_alg = pyhash.fnv1_64()

        # Examine the Pandas data frame column by column and hash/scale only
        # columns detected as strings.
        for i in self.df.columns:
            if self.df.dtypes[i] == np.object:
                print("\t\tHashing and scaling column: %s" % i)
                self.df[i] = self.df[i].map(lambda a: (hash_alg(str(a).encode('utf-
8'))) / 2**64)

        print("\tFinishd:  Hashing and scaling string values.")

        print("\n\nPost-transformation:")
        print(self.df.describe(include='all'))

        print("\t\tTransformation Time: %.4f seconds" % (time.time() - start_time))
        print("\t--Finished:  Pandas dataframe transformation")

    def process_line(self, entry):
        """Perform additional processing on a single Apache log line than apachelogs
natively provides.

        Args:
            entry (Dictionary): apachelogs dictionary structure containing the parsed
log line.
        """
        # Split the request, taking care as there are three different structures for
the request in the current
        # data set.
        command = (str(entry.request_line).split(" "))[0].strip()
        if command == "OPTIONS":
            url = "None"
            version = (str(entry.request_line).split(" "))[2]
```

```
            parameters = "None"
        elif command == "None":
            url = "None"
            version = "None"
            parameters = "None"
        else:
            temp = (str(entry.request_line)).split(" ")[1]
            version = (str(entry.request_line)).split(" ")[2]
            parameters = "None"
            # Check to see if there are parameters, if so we need to perform another
split...
            if "?" in temp:
                # There are requests where a parameter contains a URL and parameter
itself.  This
                # technique maintains the integrity of the original
request/parameters.
                url = temp[:temp.find("?")]
                parameters = temp[temp.find("?") + 1:]
            else:
                # No parameters were included in the request.
                url = temp

        # Analyze the status code and translate to a numeric version used as a
        # new feature in the dataset.
        # https://ci.apache.org/projects/httpd/trunk/doxygen/group__HTTP__Status.html
        category = 0.0
        if entry.final_status >= 100 and entry.final_status < 200:
            category = 0.1 # Informational
        elif entry.final_status >= 200 and entry.final_status < 300:
            category = 0.2 # Success
        elif entry.final_status >= 300 and entry.final_status < 400:
            category = 0.3 # Redirection
        elif entry.final_status >= 400 and entry.final_status < 500:
            category = 0.4 # Client Error
        elif entry.final_status >= 500 and entry.final_status < 600:
            category = 0.5 # Server Error

        # Extract the raw values out of the parsed Apache log file line.
        # See here for documentation on extractable values:
https://apachelogs.readthedocs.io/en/stable/directives.html
        temp = [entry.request_time.year, entry.request_time.month,
entry.request_time.day, entry.request_time.hour,
        entry.request_time.minute, entry.request_time.second, entry.remote_host,
entry.bytes_sent, category,
        entry.headers_in["Referer"], command, parameters, version]
```

```
        # Split the URL into smaller components which will be appended later to the
raw_data feature matrix.
        self.url_splitter(url)

        # Split the user agent into smaller components which will be appended later to
the raw_data feature matrix.
        self.user_agent_splitter(entry.headers_in["User-Agent"])

        # Append the new row to the current raw data matrix.
        self.raw_data.append(temp)

    def process_file(self, fileName):
        """Read the specified file and create a raw data Pandas DataFrame, and a
DataFrame with the transformed data ready for use
        in a machine learning algorithm.

        Args:
            fileName (String): Path and name of the file to be processed.

        Returns:
            Pandas DataFrames: A DataFrame containing the raw data as read from the
file and parsed.  Another DataFrame containing the
            transformed data ready for use in a machine learning algorithm.
        """
        print("\n")
        print("--Beginning:  File Processing")
        print("\tFile being processed: %s" % fileName)

        # Instantiate the apachelog parser object - this parses the log file
automatically
        # for us.  No need to write our own code.
        parser = apachelogs.LogParser(self.parse_format)

        # Make sure we can open the log file - if we can't let the user know and exit
immediately.
        try:
            fp = open(fileName)
        except Exception as e:
            print("\n\nERROR:  Unable to open log file -  %s\n\n" % str(e))
            exit()

        # Capture start time.
        start_time = time.time()
        print("\tBeginning to read file...")
        for entry in parser.parse_lines(fp):
            self.process_line(entry)
```

```
        print("\tFinished reading file...")

        print("\tFile Reading Time: %.4f seconds" % (time.time() - start_time))

        print("\tBeginning Creating Pandas dataframe...")

        # Before constructing the final data frame, build out the feature (column)
names.  Part of
        # this is static, part of it is dynamic based on splitting features contained
in the log file
        # which can vary.
        self.build_Column_Names(["year", "month", "day", "hour", "minute", "second",
"remote_host", "bytes_sent", "status", "referer", "command", "parameters", "version"])

        # Take the raw data, contained in a native Python data structure and typecast
to data frame.
        self.df = pd.DataFrame(self.raw_data)

        print("\t\tConcatenating dynamically split features to data frame...")
        self.df = pd.concat([self.df, self.url_splits, self.user_agent_splits],
axis=1, ignore_index=True)

        # Apply column names - both the static "as taken from the log file" and
dynamically generated from
        # splitting feature(s) into additional features.
        self.df.columns = self.dynamic_column_names

        # Now that the complete matrix containing raw data has been created, save it
for later reference.
        self.raw_data = self.df.copy()
        print("\tFinished Creating Pandas dataframe...")

        print("\tSample Entry: \n----------\n%s\n---------\n" % self.df.iloc[0])

        # Remove the date/time features.  Unfortunately we do it now because Python
list does not support multi-dimensional slicing.
        print("\tPre-feature reduction data frame dimensions: %i x %i" %
(self.df.shape[1], self.df.shape[0]))
        print("\tPre-feature reduction data frame memory: %iMB" % (asizeof(self.df) /
1024 / 1024))
        self.df = self.df.iloc[:, 6:]

        print("\tPost-feature reduction data frame dimensions: %i x %i" %
(self.df.shape[1], self.df.shape[0]))
        print("\tPost-feature reduction data frame memory: %iMB" % (asizeof(self.df) /
1024 / 1024))
```

```
        # Transform the data contained in the Pandas dataframe into something usable
by a
        # machine learning algorithm.
        self.transform()
        print("\tTransformed data frame memory: %iMB" % (asizeof(self.df) / 1024 /
1024))

        return self.raw_data, self.df
```

Figure F.1: fp_apache_access_log_split_request_url_useragent.py Source Code

Appendix G
ml_scikit_tsvd.py

Figure F.1 below contains the file processor plugin code `ml_scikit_tsvd.py`. plugin performs dimensionality reduction on a dataframe using truncated singular value decomposition. The public git code repository [24] contains all source code, including the code listing below, developed as part of the research project.

```
import time # Imported so execution time can be captured
from pympler.asizeof import asizeof # Used to get more accurate memory utilization
from pandas import DataFrame # Using Pandas dataframes for performance and code
clarity
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np

class simPlugin(object):
    state = ""
    random_state = 12345
    n_iter = 5
    n_components = 2
    with_mean = True

    fitting_time = 0

    def __init__(self):
        self.state = "Initialized"

    def print_help(self):
        """Builds a formatted help text string for the plugin that's displayed when
the user specifies the help command line option.

        Returns:
            String: Formatted text string containing the plugin's help information.
        """
        text = "\tName: ml_scikit_tsvd"
        text += "\n\t\tThis machine learning plugin uses scikit-learn's truncated SVD
algorithm.\n"
        text += "\n\t\tOptional Parameters:"
        text += "\n\t\t\ttsvd_skip_normalization:  Do NOT perform normalization
(scaling) of data, skip this step."
        text += "\n\t\t\ttsvd_number_of_components:  Specify the number of components
to retain (default is 2)."
```

```
        return text

    def machine_learning(self, df, plugin_options):
        """Apply the scikit-learn truncated singular value decomposition (TSVD)
machine learning algorithm to the supplied data set.

        Args:
            df (Pandas DataFrame): DataFrame containing the machine learning ready
version of the dataset to be processed.
            plugin_options (dictionary):  Dictionary containing any optional
parameters for plugins being used.

        Returns:
            Dictionary: Dictionary containing final machine learning results and other
internal data that user may want to save for review.
        """
        print("\n")
        print("--Beginning:  Machine Learning")
        print("\tMachine learning algorithm:  scikit-learn TSVD")

        if ("tsvd_number_of_components" in plugin_options):
            self.n_components = int(plugin_options["tsvd_number_of_components"])
            print("\tOverriding default number of components, it is set to: %i" %
self.n_components)
        else:
            print("\tUsing default setting for number of components: %i" %
self.n_components)

        #---------BEGIN TEMPORARY CODE---------
        # In order to save the full US matrix as calculated by TruncatedSVD, we need
to add
        # a column of zeros due to an API limitation that prevents us from retaining
all columns.
        # This is temporary to allow review of the calculated outputs. This code
should be remove
        # prior to executing final experiments.
        if self.n_components == df.shape[1]:
            print("\tUsing fix to retain all columns due to number of components
setting...")
            df["Zero"] = 0
            #self.n_components = df.shape[1] -1
        #----------END TEMPORARY CODE----------

        # Capture start time.
        start_time = time.time()
```

```
        # Create an instance of the Truncated SVD, a normalizer, and create a pipeline
for
        # automatic execution of both.
        svd =
TruncatedSVD(self.n_components,random_state=self.random_state,n_iter=self.n_iter)
        normalizer = StandardScaler(copy=False,with_mean=self.with_mean,with_std=True)

        # Build out the pipeline depending if the user opted to bypass normalization
        if "tsvd_skip_normalization" not in plugin_options:
            lsa = make_pipeline(normalizer, svd)
        else:
            print("\t\tNOT normalizing data as requested by user...")
            lsa = make_pipeline(svd)

        print("\tBeginning:  fitting")
        start_time_fitting = time.time()
        # In order to perform a diff for sanity checking, we must typecast the
        # Pandas dataframe to a Numpy array even though "copy=false" is set for the
        # normalizer.  The results will otherwise be incorrect.
        df_np = df.to_numpy()
        US = lsa.fit_transform(df_np)
        self.fitting_time = time.time() - start_time_fitting
        print("\tFinished:  fitting")

        S = svd.singular_values_
        VT = svd.components_
        variance = svd.explained_variance_ratio_

        print("\nU*S =\n", US)
        print("\nS =\n", S)
        print("\nVT =\n", VT)

        AA = US @ VT # np.matmul(US,VT) # reconstruct A
        print("\nAA =\n", AA)
        print("\ndiff =", np.linalg.norm(AA-df_np))

        T = S/S[0]
        T *= T
        T = np.cumsum(T) / sum(T)
        print("\nT =\n", T)

        print("\n\tFitting Time: %.4f seconds" % self.fitting_time)
        print("\tMachine Learning Total Time: %.4f seconds" % (time.time() -
start_time))

        print("--Finished:  Machine Learning")
```

```
        # Return a dictionary containing specific components created or calculated
        # as part of the machine learning process.  These may be used to perform
        # additional tasks (saving data to files, graphing, etc.).
        return {"US": US, "S": S, "VT": VT, "T": T, "graph": "US", "Variance_Ratios":
variance}
```

Figure G.1: ml_scikit_tsvd.py Source Code