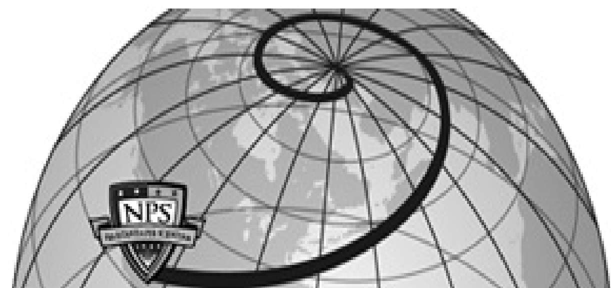
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2022-03

A MACHINE LEARNING APPROACH FOR CLASSIFYING JAVASCRIPT USING STATIC CODE ANALYSIS

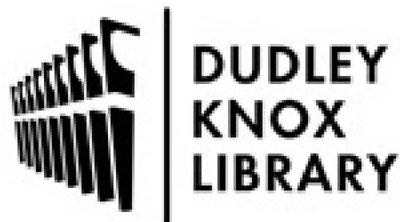
Miller, Michael D.

Monterey, CA; Naval Postgraduate School

http://hdl.handle.net/10945/69686

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A MACHINE LEARNING APPROACH FOR CLASSIFYING JAVASCRIPT USING STATIC CODE ANALYSIS

by

Michael D. Miller

March 2022

Thesis Advisor:	John C. McEachen
Co-Advisor:	Murali Tummala

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE				*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.				
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis		
4. TITLE AND SUBTITLE A MACHINE LEARNING APPROACH FOR CLASSIFYING JAVASCRIPT USING STATIC CODE ANALYSIS				**5. FUNDING NUMBERS**
6. AUTHOR(S) Michael D. Miller				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000				**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A				**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.				**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This thesis develops a machine learning approach to classify normal and anomalous JavaScript based on a static analysis of select features derived from the top 30 000 webpages on the internet. A dataset of 136 features was extracted from 100 000 raw JavaScript files. Nine test groups were created and tested using 10 subsets of features. K-means clustering was used to group the data and manually translate into binary classification. The results from the K-means clustering show moderate performance with distortions less than 1.0 from elbow plot analysis and average silhouette scores between 0.3 and 0.8 using silhouette analysis of the clustering. The classification of each JavaScript file was then examined using naïve Bayes algorithm to re-create and examine the performance of the highest performing classifiers using a less processing intensive method. Naïve Bayes was not a good model to re-create the K-means classifier. The best performing classifiers had a Matthews correlation coefficient of 0.75 when examining small JavaScript, and less that 0.38 when examining the medium or large JavaScript. The results show that most JavaScript files were small in file size, and file size was the only defining feature. No features tested effectively categorize the vast majority of JavaScript other than file size. Further research is needed to find features that more accurately encompass the majority of JavaScript to define normal JavaScript.

14. SUBJECT TERMS JavaScript, machine learning, static, behavior, large data sets, Jupyter Notebook, real world data			**15. NUMBER OF PAGES** 117
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A MACHINE LEARNING APPROACH FOR CLASSIFYING JAVASCRIPT USING STATIC CODE ANALYSIS

Michael D. Miller
Lieutenant, United States Navy
BS, San Jose State University, 2015

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2022

Approved by: John C. McEachen
Advisor

Murali Tummala
Co-Advisor

Douglas J. Fouts
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis develops a machine learning approach to classify normal and anomalous JavaScript based on a static analysis of select features derived from the top 30 000 webpages on the internet. A dataset of 136 features was extracted from 100 000 raw JavaScript files. Nine test groups were created and tested using 10 subsets of features. K-means clustering was used to group the data and manually translate into binary classification. The results from the K-means clustering show moderate performance with distortions less than 1.0 from elbow plot analysis and average silhouette scores between 0.3 and 0.8 using silhouette analysis of the clustering. The classification of each JavaScript file was then examined using naïve Bayes algorithm to re-create and examine the performance of the highest performing classifiers using a less processing intensive method. Naïve Bayes was not a good model to re-create the K-means classifier. The best performing classifiers had a Matthews correlation coefficient of 0.75 when examining small JavaScript, and less that 0.38 when examining the medium or large JavaScript. The results show that most JavaScript files were small in file size, and file size was the only defining feature. No features tested effectively categorize the vast majority of JavaScript other than file size. Further research is needed to find features that more accurately encompass the majority of JavaScript to define normal JavaScript.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
  A. OBJECTIVE 1
  B. RELATED WORK 1
  C. ORGANIZATION 3

II. BACKGROUND AND LITERATURE REVIEW 5
  A. CURRENT STATE OF JAVASCRIPT RESEARCH 5
    1. Dynamic Behavior 5
    2. JavaScript Obfuscation 6
    3. Malicious JavaScript 6
    4. JavaScript Optimization 7
  B. MACHINE LEARNING 8
    1. K-Means Clustering 8
    2. Naïve Bayes Algorithm 12
  C. PRECISION, RECALL, ACCURACY, F1, AND MCC 14

III. METHODOLOGY 17
  A. PROPOSED SCHEME 17
  B. K-MEANS CLUSTERING 18
  C. NAÏVE BAYES 20

IV. IMPLEMENTATION AND RESULTS 21
  A. TESTING ENVIRONMENT 21
    1. Preprocessing 21
    2. Subdividing The Data Frame 26
  B. K-MEANS CLUSTERING 28
    1. K-Means Clustering and Elbow Plot 28
    2. K-Means Silhouette Plot 31
  C. NAÏVE BAYES APPLICATION 39
  D. RESULTS FROM SIZE COMPARISON 46

V. CONCLUSION 53
  A. SUMMARY OF WORK 53
  B. FUTURE WORK 54

APPENDIX A: DATAFRAME FEATURE LIST 57

APPENDIX B: DATA SUBSET FEATURE SELECTION....59
- A. SUBSET 1 – ALL DATA....59
- B. SUBSET 2 – CONTAINS ALL FEATURES EXCLUDING FILE_SIZE AND FILE_LINE_COUNT....60
- C. SUBSET 3 – ALL DATA EXCLUDING TYPE 1 TERMS, VALUES THAT INCLUDE BLANKS....60
- D. SUBSET 4 - ALL DATA EXCLUDING TYPE 2 TERMS, VALUES THAT DON'T INCLUDE BLANKS....60
- E. SUBSET 5 – CONTAINS THE INDIVIDUAL KEYWORDS AS FEATURES....61
- F. SUBSET 6 – ONLY CONTAINS LOW CORRELATION, CALCULATED VALUES....61
- G. SUBSET 7 – CONTAINS TOTAL ELEMENTS, UNIQUE ELEMENTS, BLANK ELEMENTS AND TYPE-2 TERMS....61
- H. SUBSET 8 –CONTAINS TOTAL ELEMENTS, KEYWORD CATEGORIES, UNIQUE ELEMENTS AND ENTROPY OF TYPE-2 TERMS....62
- I. SUBSET 9 – CONTAINS UNIQUE ELEMENTS, ENTROPY, AND KEYWORD CATEGORIES....62
- J. SUBSET 10 – CONTAINS ALL FEATURES WITH LESS THAN 0.5 CROSS CORRELATION....62

APPENDIX C: PYTHON SCRIPT FOR PREPROCESSING RAW JAVASCRIPT....63

APPENDIX D: PYTHON SCRIPT FOR K-MEANS CLUSTERING ANALYSIS....73

APPENDIX E: PYTHON SCRIPT CLASSIFICATION AND NAÏVE BAYES MODELING....89

LIST OF REFERENCES....97

INITIAL DISTRIBUTION LIST....101

LIST OF FIGURES

Figure 1. Example of K-means clustering with three centroids. Source: [16]. 9

Figure 2. Example of an elbow plot with distortion. Source: [19]. 10

Figure 3. Example of a silhouette plot. Each cluster in (b) is represented as a horizontal bar plot in (a). The width from top to bottom of each cluster represents the number of data points assigned to that cluster. The dotted line represents the average silhouette score or coefficient, which is a simple indication of how good a clustering scheme may be. The closer to 1, the better the clustering. Source: [20]. 11

Figure 4. Step by step methodology for extracting features from raw JavaScript, apply ML techniques, classification, and verification............17

Figure 5. Flow diagram for converting JS to word frequency counts using *regular expressions* and *collections*..22

Figure 6. Illustration of the three size categories of JavaScript using the first 1 000 JS files. Three distinct size groups exist for JavaScript based on the distribution of *file_size* versus *keyword uses*. All data is assigned a size category based on how close its file size is to the mean of these three categories. 25

Figure 7. K-means selection via elbow plot using small data pulled from 100 000 datapoints. We see that subset 1 through subset 5 and subset 10 have a significantly larger mean distortion and will be eliminated from consideration for developing the binary classifier. 30

Figure 8. K-means selection of the four best performing datasets. Observe the elbows for subset 6 with six clusters, subset 9 with four clusters, and subset 8 with eight clusters. 31

Figure 9. Comparing silhouettes of four chosen SCP's. Note the distribution of SCP 8-8 and SCP 9-4 show multiple broad groups with moderate silhouette scores as well as thin groups which are the indications desired for our classification. Note SCP 8-2 had the highest observed AvSS. SCP 6-6 may be too evenly distributed to make the desired classifier. 33

Figure 10. Converting K-clusters into a binary classifier for SCP 8-8. Observe the 5% threshold is 4 802 files. Since clusters 2, 0, 3, and 6 on the right each contain less than the threshold, they are assigned as a 1 for

anomalous. Likewise, clusters 1, 4, 7, and 5 are classified as 0 for normal. 35

Figure 11. Feature correlation comparison for subset 6 and subset 8. In (a) we see the features of subset 6 have low to moderate correlation with the largest correlation magnitude of 0.41. In (b) we see that most features of subset 8 have a correlation over 0.5 to every other feature. 40

Figure 12. Confusion Matrix for SCP 8-8 naïve Bayes results. The four values represent the number of true positives, false negatives, false positives, and true negatives. This shows that the model is proportionally split on misclassifying zeros as ones, and vice versa.........42

Figure 13. Comparison of *Keyword Uses* and *Total Elements2* features versus classification. The 89 306 normal datapoints in (b) are highly concentrated in terms of size represented by total elements and keyword uses, while the 6 736 anomalous datapoints are relatively vast in the distribution of data...45

LIST OF TABLES

Table 1. Comparison of three size categories: (a) *small*, (b) *medium*, (c) *large*. Examining the mean *file_size* and *total_elements* of the three size categories for the three categories we see the separation illustrated in Figure 6. All data is rounded to nearest integer, and file size is measured in bytes. 26

Table 2. Binary classification distribution of four SCPs. Observe the final distribution of normal to anomalous JS files is similar for SCP 6-6, SCP 8-2, and SCP 8-8, while SCP 9-4 has a more skewed distribution due to the classifier rules. Only one of four clusters from SCP 9-4 was classified as anomalous. 35

Table 3. SCP 8-8 binary classification header data comparison. Note the mean *file_size* of the normal data in (a) is much smaller than in (b), considering that file_size was not a feature of subset 8. The overlap of maximum values indicates that there is more to the clustering than just *file_size*. 36

Table 4. SCP 8-8 binary classification feature data comparison. Examining the mean of all features in (a) for the normal data, we see that they are all significantly lower than in (b) representing the anomalous classified data. The overlap in max values for each feature suggests that *file_size*, is not the only association for the clustering. 37

Table 5. SCP 6-6 binary classification header data comparison. In contrast to SCP 8-8, the normal data in (a) is significantly larger in *file_size* compared to the anomalous data in (b). While classification is heavily dependent on *file_size*, the feature set is varying in what size outliers are being grouped to form the normal and anomalous data. All values rounded to the nearest integer. 38

Table 6. Train/test split (70/30) of SCP 8-8 classifier for training the naïve Bayes model. The entries contain the number of datapoints assigned to train/test groups by classification. 41

Table 7. Naïve Bayes results for predicting the SCP 8-8 classification. The testing set metrics are the results of predicting new data, compared to the training data that was used to build the naïve Bayes model. The maximum value is 1.0. 41

Table 8. Train test split and metrics for four SCP models. Based on naïve Bayes model performance, model 6-6 was the worst classifier, model 9-4 performed moderately, while model 8-2 and model 8-8 had

similar results across all metrics. Due to classification rules, there were only about half as many anomalous files for model 9-4 which may have impacted model performance. 43

Table 9. Distribution of data into three size groups for varying number of total datapoints. The mean and standard deviation of the file_size of each group is provided along with the number of datapoints that each category contains. 47

Table 10. K-means and naïve Bayes results for all nine DF groups. The table displays the group assignment in the left most column in terms of number of datapoints used. The second column gives the chosen SCPs from K-means analysis, supplemented with the distortion and AvSS for the chosen SCPs in column 3 and 4. Columns 5-9 give the results of the naïve Bayes modeling on the 30% test set for the binary classifier derived from each tested SCP in column 2. 48

LIST OF ACRONYMS AND ABBREVIATIONS

DF	data frame
JS	JavaScript
KM	K-means
MCC	Mathews correlation coefficient
ML	machine learning
MNBA	multinomial naïve Bayes algorithm
NB	naïve Bayes
SCP	subset cluster pair

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

JavaScript is the predominant language of the web as the internet has become innately tied with modern society. JavaScript is used in all manner of web applications from news to commerce to entertainment and continues to grow in its proliferation in the makeup of the web. Most JavaScript performs similar functions across all forms of the most popular web applications while having countless different implementations. JavaScript is a versatile language that allows dynamic freedom for any given web application, which makes it ideal for development in a growing internet space and inherently difficult to analyze. Understanding JavaScript and establishing a baseline classification of normal JavaScript is key to any web-based analysis supporting network security or the execution of a web application. Establishing a baseline classification for normal JavaScript can be instrumental when observing new data of unknown purpose or intention when it comes to security and efficacy.

A. OBJECTIVE

The objective of this thesis is to create a database of features for raw JavaScript files and then feed these features into a machine learning (ML) algorithm to classify normal JavaScript versus anomalous JavaScript. The raw JavaScript files will be read sequentially, converting each file into a vector of the word contents, and the desired features will be extracted from the word vector. Subsets of features are provided to a clustering algorithm to establish commonalities in the files. The resulting clusters are then translated to a binary classification of normal for large clusters of data, that represent common JavaScript, or anomalous for small clusters of uncommon and outlier data that represent different combinations of features than the large clusters. After the data is assigned classification, it will then be examined using a second ML tool to identify how well the classification could be identified using less process intensive methods and classify future data more effectively.

B. RELATED WORK

JavaScript classification using ML is a well-explored field of study. There are two primary directions for research on the topic. The first is the detection and classification of

malicious JavaScript. This is explored from the perspective of identifying JavaScript that is vulnerable to external malicious JavaScript based on function level vulnerabilities [1]. Another approach is focused on classifying the malicious JavaScript directly [2], [3]. Liang et al. [3] and Ndichu et al. [2] explore the idea of breaking JavaScript down into vectors of various lengths and using sets of keywords to identify if a particular chunk of code is malicious. One approach is to compare the word vectors to a set of chosen keywords labeled as malicious or benign with the result classifying a larger set of code based on the use of these keywords. The other approach uses abstract syntax trees (AST's) to examine patterns of code and using neural network models to classify patterns as malicious or benign. The idea of breaking JavaScript into word vectors was inspired by these two works, albeit a real-world dataset, and different features and different ML tools are used.

A significant barrier in any JavaScript analysis is caused by obfuscation. Obfuscation is not inherent to just malicious JavaScript, as it can be used for security, or caused by compression in order to improve performance, but it makes malicious JavaScript detection more difficult to identify since the code is often unreadable to humans [4]. Some methods of obfuscation can include using broad functions such as *eval* or asynchronous code that may not be recognized as malicious from a static perspective because they only occur at runtime. The focus of obfuscation in most research is related to identifying malicious JavaScript. The approaches from these topics inspired the keyword/feature selection used in this thesis.

One other approach to JavaScript analysis is for the purpose of efficiency in JavaScript implementation. One method of improving JavaScript is to reduce the amount of code that is within a script that does not have an application. Many JavaScript pages use functions or code from various libraries that never end up running at all on the page. Kupoluyi et al. [5] use an approach to classify blocks of JavaScript code as *essential* or *non-essential* and recreate web applications with the removed non-essential (dead) code. A similar approach is by Chaqfeh et al. to reduce load times of JavaScript pages on low bandwidth networks and mobile devices. With the goal of creating a classifier to remove non-essential and undesired JavaScript from loading on a page [5]. Both papers similarly explored real world data, and classification of JavaScript in alternate ways than looking for

malicious JavaScript. These papers explore the idea of creating multiple labels for JavaScript and then converting them to a binary classifier, which has been adopted for this thesis.

Recent research on the study of JavaScript has primarily dealt with analysis for the purpose of identifying malicious JavaScript but does not provide a baseline of normal JavaScript for comparison. Several ML JavaScript classifiers are effective at identifying malicious JavaScript using control data but are not thoroughly explored using real world data. Alternatively, the research regarding optimization has a more established approach to working with real world data. This thesis will demonstrate an approach to establish a baseline classification method for real-world JavaScript.

C. ORGANIZATION

The remainder of this thesis will detail the following. Chapter II details the state of JavaScript-related research currently, including the background information on the two ML methods used in the thesis, and covers the metrics used to quantify the effectiveness of the resulting classifications. Chapter III contains a detailed description of the steps taken to go from raw data to classified data, while exploring the challenges, assumptions and compromises made for the thesis. Chapter IV contains the results of processing the data, and the overall classification of the data for the largest group of data tested and compares the results of all groups of data tested. Chapter V concludes the thesis with a summary of the work performed and suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND LITERATURE REVIEW

This chapter will discuss the current state of research when it comes to analyzing JavaScript. It will also detail the two ML methods, K-means clustering and naïve Bayes algorithm, that are used to classify JavaScript and then determine its replicability. Included is how to measure the effectiveness of the K-means clustering using silhouette and elbow plots as well as measuring the effectiveness of the naïve Bayes model using precision, recall, accuracy, F1, and Matthew's correlation coefficient metrics. These topics will provide background for why this research topic was chosen as well as necessary background needed to understand how the results were obtained.

A. CURRENT STATE OF JAVASCRIPT RESEARCH

There are two primary focus areas of JavaScript research, and two aspects of JavaScript that make analysis difficult. One reason JavaScript is difficult to analyze is because it is a dynamic programming language with loose syntax, with endless ways to change predefined elements to suit any need. The second challenge is due to obfuscation. Obfuscation, whether intentional or due to compression, is common and makes direct analysis difficult due to asynchronous code that may change at run-time. Obfuscation analysis is often linked to malicious JavaScript, which is one of the two primary areas of JavaScript research. Malicious JavaScript detection continues to be a highly researched topic since security is becoming more important in the always-connected world. The other focus area for JavaScript research is used for optimization. Removing unwanted or unnecessary JavaScript from a web application is important for improving performance over slow network connections or to improve the user experience.

1. Dynamic Behavior

One primary area in JavaScript research is focused on the dynamic nature of JavaScript. JavaScript as a programming language is dynamic because it has loose syntax that does not require type declarations, and since it is only compiled at run-time, there are limited checks to the accesses the code is making. The methods and objects can inherit properties from other prototypes that can be endlessly modified for the creator's purpose,

which makes static analysis and optimization difficult [6]. The use of the *eval* function in JavaScript is powerful, which can run any arbitrary code as a string at runtime, creating or modifying objects [7]. The dynamic nature of JavaScript has led to far more specific research efforts and smaller scale analysis, such as tracking the state of JavaScript objects as they change when code is executed [8].

2. JavaScript Obfuscation

While JavaScript is a dynamic programming language, obfuscation challenges are also present when attempting to analyze JavaScript. Obfuscation is the process of transforming the JavaScript to obscure its true content by a reader or by a machine. It has many legitimate uses, such as data compression and legitimate code protection, but is also used to disguise malicious code as well. Some techniques include name randomization, word substitution and inserting meaningless code to obscure the meaningful data. Some research aims at identifying malicious JavaScript by identifying obfuscated malicious JavaScript features [9] while others attempt to incorporate several levels of de-obfuscation into their classifiers [10].

3. Malicious JavaScript

Detecting malicious JavaScript is one of the most researched topics when it comes to JavaScript analysis. Identifying vulnerabilities and malicious JavaScript code are needed to ensure security. Malicious JavaScript may be found in mobile apps or run in a browser when accessing a web application that can execute content or access private data on a user machine [11]. Some research has a specific focus, such as identifying injection attacks on mobile applications based on modeling the regular behavior of applications [12]. There are several analysis methods currently being explored in order to identify malicious JavaScript by converting files into word and feature vectors in order to identify malicious code by processing it with a neural network trained model [2]. Other approaches for securing JavaScript involve identifying vulnerabilities in otherwise benign JavaScript. One method built a ML model that could classify JavaScript vulnerabilities by training the model on known vulnerable JavaScript pulled from GitHub libraries [1].

Another method of JavaScript analysis is classification using abstract syntax trees (ASTs). The process involves breaking a JavaScript file into various lengths and then examining the pattern of the types of code called nodes, from the starting node to a terminal node to classify the line of code as whole. A model can then be built in order to predict method names and summarize the underlying JavaScript [13]. A similar AST approach is used to establish syntactic and semantic features of JavaScript in order to identify malware based on patterns among the features [3]. He et al. [14] combine several aspects discussed, using static features of malicious JavaScript, extracting features using ASTs, identifying obfuscated code based on established obfuscation features, and then building a model to classify JavaScript [14].

4. JavaScript Optimization

Optimization is also the focus of much JavaScript research. The overall goal of optimization is improving web application performance in poor network locations, such as rural areas, or poorly networked countries and is essential for getting these areas online. One way to do so is by stripping a webpage of the JavaScript down into the essential code only. Chaqfeh et al. [5] do this by using ML to define what parts of a webpage JavaScript is essential and blocking all other elements to include undesired elements such as advertisements. The performance on slow networks was greatly improved while maintaining the essential functions of the web applications tested for real world users operating in developing regions [5]. In a similar way, Kupoluyi et al. [15] focus on removing unused, *dead code* from a JavaScript file. They do this by testing the functionality of a web application from a user perspective, such as mousing over buttons, or advertisements and clicking on various elements. They examine what JavaScript code runs for each user interaction, and afterwards they can see what code was never run for all tested user interactions and remove any JavaScript that was not called during the testing. The results showed overall improvement in load times for most pages without any loss in content functionality. Additionally they showed that their model did not need to be frequently run since most JavaScript that was actively used did not change over the period of a week [15].

B. MACHINE LEARNING

1. K-Means Clustering

K-means is an unsupervised ML algorithm that tries to group *n* total datapoints into a specified number of clusters *(K)* of equal variances by minimizing the sum-of-squares error between the cluster centers (centroids) represented by the mean μ_j, and the individual datapoints assigned to that cluster represented by x_i. The algorithm will minimize the Euclidean distance, sometimes referred to as inertia (*J*) as:

$$J = \sum_{i=0}^{n} \min_{\mu_j \in K} \left(\| x_i - \mu_j \|^2 \right) \tag{1.1}$$

from each centroid μ_j [16]. The inertia is the sum of square differences between the mean and each datapoint within a cluster. By averaging the inertia across all datapoints and all clusters, we have a measure called the mean distortion, which can be used as a measure of how close datapoints of all clusters are to their associated centroids.

K-means algorithm starts by arbitrarily assigning all cluster centers to an equal number of random data. For each datapoint afterwards, the datapoint is assigned to the nearest cluster center by distance, and then the cluster center means μ_j are re-calculated. This continues until the change in cluster center means for each new point falls below a threshold, and all datapoints are assigned into one of the centroids as in Figure 1. K-means is run multiple times with different cluster centers to ensure they converge on a global minimum instead of a local minimum, which is dependent on what data the centroids are initially assigned. K-means can also be used with specified sample weights if needed [16].

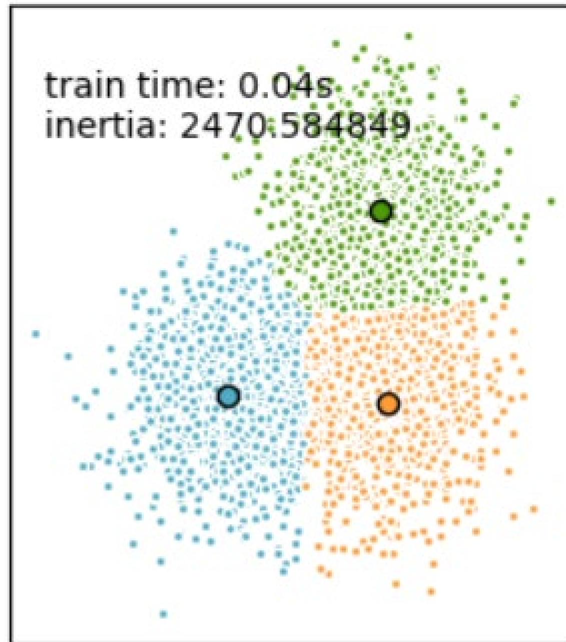


Figure 1. Example of K-means clustering with three centroids. Source: [16].

Implementation of K-means in Python is simple using a library imported from SK-Learn [17], with the only required parameter *n*_clusters. By default, the implementation choses datapoints that are far from each other for initialization, and the algorithm runs for ten different seed values for choosing the centroids. The returned results are based on the best seed used. The algorithm will run until 300 iterations, or the change in mean for all centroids from one iteration to the next falls below a threshold, (10^{-4}) by default [17].

Unsupervised ML algorithms need human analysis to validate their results, since they are used for unlabeled data [18]. Two ways that we can judge the results of K-means clustering are with elbow plots and silhouette plots. Figure 2 is an example of an elbow plot for an arbitrary dataset.

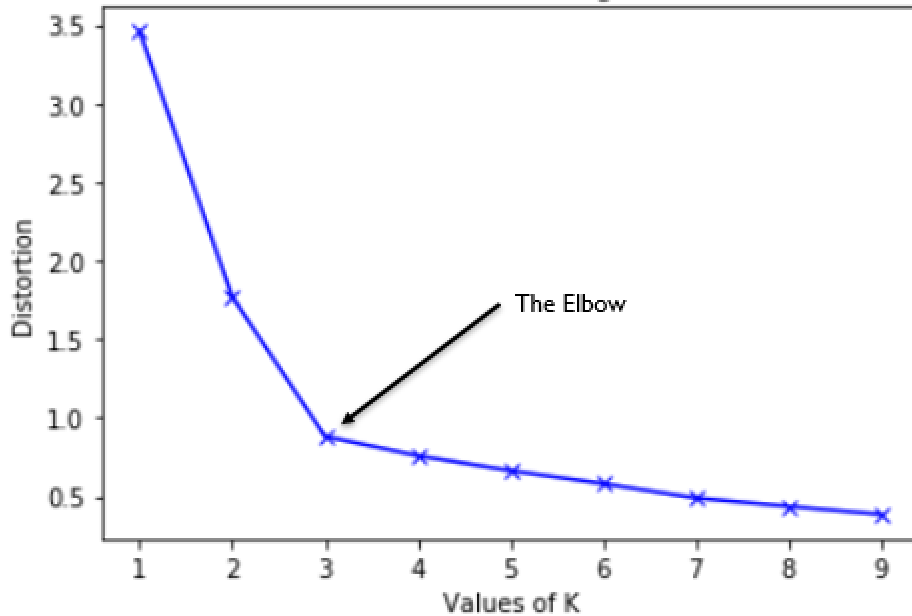


Figure 2. Example of an elbow plot with distortion. Source: [19].

The plot in Figure 2 shows the average distortion between all datapoints assigned to their nearest cluster versus the number of clusters. Distortion is an average while inertia, mentioned previously is a sum, but both can be used in measuring the quality of the clusters. The closer to zero on the y-axis a cluster is, the closer the data of each cluster is to each other. The subjective analysis of an elbow plot is in deciding what number of clusters to choose for the best clustering assignment. It is best to limit the number of clusters used because while choosing more clusters may have a lower mean distortion, there is increased complexity for each additional cluster center that may divide the data unnecessarily and increase the time it takes to run the model. The term *elbow* refers to the points that represent a good choice for the number of clusters to use since they represent where there is a depreciation on how much improvement is gained for the added complexity. We see in Figure 2 that there are two distinct elbows at K = 2 and K = 3 clusters. Considering the cluster at K = 3, we see that there is the most significant change in the slope of the line from K = 2 to K = 3 than from K = 3 to K = 4 and so on. This significant change makes it the best candidate for classification, since we see only moderate improvement in the distortion using more than K = 3 clusters. Not all datasets are as clearly formulated as the

example in Figure 2 when it comes to selecting an ideal number of clusters, and so it is most often a subjective process in deciding how many clusters to pick given the data that is being analyzed.

Another way to analyze K-means is by using a silhouette plot. A silhouette plot is a graphical representation of how each datapoint fits in its assigned cluster vs neighboring clusters defined by a silhouette score [20]. The total plot can also have an average silhouette score that describes the average clustering assignment for all the *K* clusters tested, i.e., the plot's average silhouette score is a summary of the total clustering scheme for a clustering scheme. A silhouette score of 1 represents a perfectly assigned datapoint to a cluster. A score of 0 represents a weak cluster assignment, as the datapoint may be equally distant from two cluster centers. A value of -1 represents a datapoint assigned to the wrong cluster [20]. Overall, it is ideal to have an average silhouette score as close to 1 as possible indicating perfect clustering. Figure 3 shows an example of a well-defined number of clusters for a set of data.

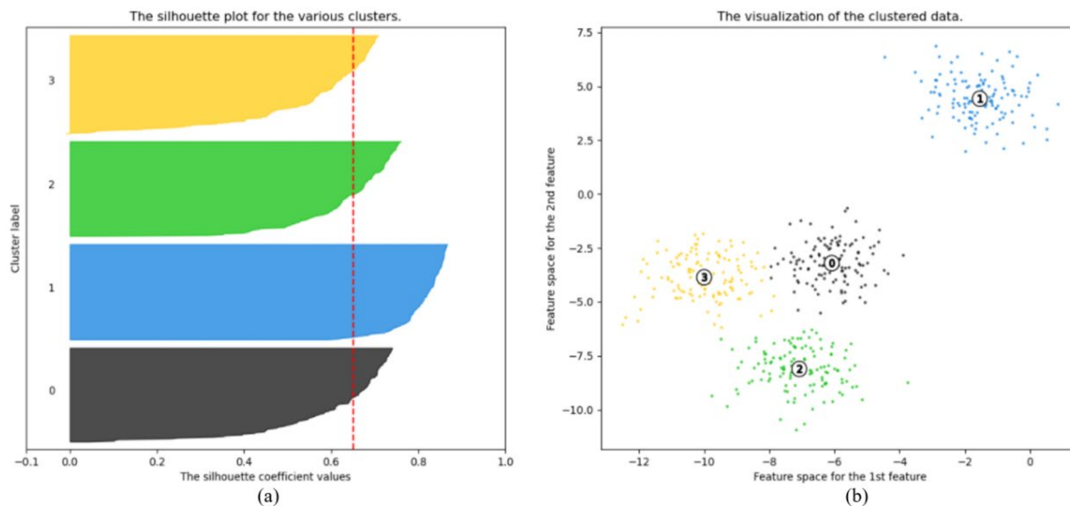


Figure 3. Example of a silhouette plot. Each cluster in (b) is represented as a horizontal bar plot in (a). The width from top to bottom of each cluster represents the number of data points assigned to that cluster. The dotted line represents the average silhouette score or coefficient, which is a simple indication of how good a clustering scheme may be. The closer to 1, the better the clustering. Source: [20].

In Figure 3, we see four distinct clusters in the scatter plot in (b) and the corresponding silhouette plot in (a). Each cluster in Figure 3a is an amalgam of bar plots for each datapoint, grouped by the cluster the datapoint is assigned. In Figure 3a, each cluster group is about of the same width, which tells us that each cluster has approximately the same number of datapoints. Each cluster group exceeds the average silhouette score represented by the red dotted line around 0.65 and there are no datapoints with negative values. While the average silhouette score may give some measure of how well a clustering scheme works, it does not tell the whole story when working with multi-dimensional data that cannot be easily verified. Both the elbow plot and the silhouette plot are tools to give a sense to the effectiveness of a clustering scheme of *n* clusters.

K-means can be implemented in Python with the number of desired clusters as the only required parameter [17]. The deficiency of K-means is that it assumes data to be symmetric about the clusters and can perform poorly with elongated clusters or other data distributions [16].

2. Naïve Bayes Algorithm

Naïve Bayes (NB) is a supervised ML algorithm that uses Bayes probabilities to determine the classification of data based on a chosen feature set. What makes this algorithm supervised versus unsupervised as seen with K-Means is that the training data is already labeled with a chosen classification, therefore the effectiveness of the classification can be measured. The term naïve is used because it works under the assumption that the features tested are independent from each other, which is hardly the case when working with real world data. Despite the realities of independence, naive Bayes is still an effective tool at classifying data [21].

Bayes theorem given by [21] is

$$P(y \mid x_1, x_2, x_3, ..., x_n) = \frac{P(y)P(x_1, x_2, x_3, ..., x_n \mid y)}{P(x_1, x_2, x_3, ..., x_n)} \tag{1.2}$$

where $P(y \mid x_1, x_2, x_3, ..., x_n)$ represents the probability of classification *y* occurring given features $x_1, x_2, x_3, ..., x_n$ are present, and $P(y)P(x_1, x_2, x_3, ..., x_n \mid y)$ represents the probability of observing classification *y* multiplied by the probability of features $x_1, x_2, x_3, ..., x_n$ given that classification *y* is present. The denominator on the right-hand side of Equation 1.1 represents the probability of features $x_1, x_2, x_3, ..., x_n$ occurring.

In this thesis, we are using a variant of naïve Bayes, called multinomial naïve Bayes, which is regularly used in text classification based on word vector counts. For this application, each probability $\theta_{yi} = P(x_i \mid y)$ is calculated as

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n} \tag{1.3}$$

where N_{yi} is the count for one feature that appears in data labeled as *y*, N_y is the total count for all features labeled as *y*, α is a smoothing factor used in certain applications equal to 1 by default, and *n* is the total number of features. If we remove the rightmost terms in the numerator and the denominator, we can easily see that the probabilities are a ratio of how often a feature appears in data with the specified label [21].

Implementation in Python is straightforward and requires a user to first create a training set of data. The training set of data must contain enough datapoints, and a set of measurable feature vectors containing the information to build the model, and the classification vector or *true* output of the model for the selected training data. The model then calculates the probability of each datapoint obtaining its assigned classification based on the input features [22].

After applying the model to a training set, we can then apply the developed model to predict a testing set that contains known classifications. We can then compare the known classifications and the predicted classifications for the test data to determine how well the

model can correctly classify the data. If the model is successful, and the training set large enough, it can then be used to predict unlabeled data with the same effectiveness.

C. PRECISION, RECALL, ACCURACY, F1, AND MCC

To quantify the results of the naïve Bayes application to the dataset, we will utilize statistical figures for measuring how well each model performs. The five statistics we will look at are: precision (*p)*, recall (*r)*, accuracy (*a)*, F1 score, and the Matthews correlation coefficient (MCC). These five statistics are calculated from four values; the number of true positives (n_{TP}), false positives (n_{FP}), true negatives (n_{TN}) and the number of false negatives (n_{FN}). Precision is the measure of true positives divided by the number of total positives

$$p = \frac{n_{TP}}{n_{TP} + n_{FP}} \tag{1.4}$$

and represents the ability of the classifier to not misclassify a positive sample [23]. The recall is the measure of true positives divided by the sum of all positives

$$r = \frac{n_{TP}}{n_{TP} + n_{FN}} \tag{1.5}$$

and represents the percentage of correctly identified positive samples [23]. The accuracy is

$$a = \frac{n_{TP} + n_{TN}}{n_{TP} + n_{FP} + n_{FN} + n_{TN}} \tag{1.6}$$

and is the measure of correctly classified data over all data [23]. The F1 score (ρ_{F1}) is given by,

$$\rho_{F1} = \frac{2(p \bullet r)}{p + r} \quad (1.7)$$

which is a weighted score calculated between precision and recall [24]. The reason ρ_{F1} is often used instead of accuracy is because it does not need to know all four values $n_{TP}, n_{TN}, n_{FP}, n_{FN}$ and can be calculated from precision and recall statistics. All four of these metrics are coded in Python as a built-in library, which can be accessed as a function by using the true output vector and the predicted output vector. The final statistic to cover is the MCC.

The MCC is considered an exceptional measure of the effectiveness for a binary classification model. More specifically, the MCC defined as [24]

$$\rho_{MCC} = \frac{n_{TP} \cdot n_{TN} - n_{FP} \cdot n_{FN}}{\sqrt{(n_{TP} + n_{FP}) \cdot (n_{TP} + n_{FN}) \cdot (n_{TN} + n_{FP}) \cdot (n_{TN} + n_{FN})}} \quad (1.8)$$

which takes all four quantities mentioned into account, giving it greater sensitivity than other metrics like the diagnostic odds ratio (DOR) and results in a single continuous value between [-1, 1]. A value of 1 represents a perfect classification model, 0 represents a random classification model, and -1 represents an inverse classification model. The MCC is coded in Python as a built-in library, which can be accessed as a function by providing the true output vector and the predicted output vector [25], [26].

This concludes the background section of the thesis. The next chapter focuses on the methodology used as it relates to the information provided in this chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

This chapter will discuss the approach taken, starting with raw JavaScript files, and detailing the process in creating the feature data frame or DF. We will discuss the techniques and ML applications used to classify the JavaScript as normal or anomalous and discuss how the results can be evaluated for overall model effectiveness.

A. PROPOSED SCHEME

In this thesis, we propose a method for taking raw JavaScript text from a database, extracting a set of features based on the static JavaScript, applying K-means clustering to the data, and assigning a classification to the JavaScript files as normal or anomalous. We then take our newly classified data and use a naïve Bayes algorithm to determine how well it matches with the classifications obtained using K-means and compare those results across multiple iterations, using different feature selections and different clustering choices as well as for datasets of various length divided by size categories of data as in Figure 4.

Figure 4. Step by step methodology for extracting features from raw JavaScript, apply ML techniques, classification, and verification.

The provided database of JavaScript used in this thesis consists of over 2.7 million JavaScript files that were collected from the list of Alexa.com top 30 000 webpages in 2019. They are saved in the JSON (JavaScript Object Notation) file format, which is converted to text and parsed using common Python library functions. A Python script was written to access this database and extract features in the form of word vector frequency counts, which are then compiled into a Python DF. This approach is inspired by related research that converts JavaScript into vectors of words, and then analyzes the syntax in order to determine if the JavaScript is malicious [2], [13]. A DF functions similarly to an

excel spreadsheet and allows easy manipulation of and access to the features of each JavaScript file and will simplify implementation of later ML applications.

The list of features includes *file name*, *file path*, *site rank* on the Alexa top 30 000 from 2019, *file size*, an estimated *line count*, the *average length of a line* in terms of file size, the *total number of elements*, the *number of unique elements*, the *entropy* of those unique elements, and 111 JS *keywords* separated into four categories. Additionally, several statistical parameters were derived from these feature columns that will be explored in Chapter IV in more detail. A full list of features with descriptions is included in Appendix A.

After building the DF with the desired features and number of datapoints, we need to add a categorical variable to each datapoint. This is because the file size of each JavaScript file varies widely, and we wish to reduce the impact of file size on the classification implemented. We assign every datapoint in the DF to one of three categories; *small*, *medium,* and *large*, based on two features; *file_size* and *total elements*. These two features are linearly related and provide a clear differentiation in the three categories. This categorization is done to limit the effect that file size has on the classification in further analysis and we will compare the results across these three categories at the end of our analysis.

B. K-MEANS CLUSTERING

We are using K-means to build the classifier because it is a method that allows the grouping of data, thus allowing us to identify data that is alike, which is needed for developing our binary classification. K-means is an *unsupervised* ML method, which means the data it is grouping has no predefined labels, versus a *supervised* ML method, which uses a known classification or label to determine how to group data [18]. Since we are working with raw, real-world JavaScript and the goal is to create labels for the data, we are limited to unsupervised methods since we have no predefined labels for our data. K-means is a widely used algorithm; it is simple to implement and understand while being effective at grouping similar types of data given appropriate features [16].

Using K-means, we will assign each JS file a classification of normal based upon whether the underlying features fit into a dominant (well-populated cluster) or a classification as anomaly if the features place it in a smaller, less populated group, or outlier cluster. We do this by first dividing the DF into 3 smaller DFs based on its categorical feature (*small*, *medium*, *large*) and examine one group at a time to limit the effect of size on the classification. With the chosen category DF, we scale all data using a Python function called z-score, which scales all data in each column by the mean and standard deviation of that column as discussed in Chapter II. This is done because we are working with a wide range of data values between all the chosen features, from a few hundred bits up to several megabits of data in terms of the JS file sizes. From the three size-based DFs, we develop ten subsets of the DF with different combinations of features. We then use two methods of evaluating K-means to choose the best number of clusters, and the most appropriate features to base our classification upon. The first is the elbow plot, which visualizes the average distortion between datapoints and their assigned clusters versus the number of cluster centers or centroids used [19]. Based on the elbow plot, we will narrow the number of subsets of data and cluster centers to examine using the second analysis method, the silhouette plot. The silhouette plot provides a way to visualize the distance each datapoint has between its assigned cluster versus other clusters. Silhouette plots can also provide a visualization as to how large each cluster is and the distance between each centroid [20]. Using these two tools, we will choose a selection of DF subsets and the number of cluster centers to use, in order to create our classification.

After choosing the subsets and the number of cluster centers we want to test, we will apply K-means to the chosen subsets and desired number of cluster centers, then manually reassign the clusters as a binary classification. Depending on the number of clusters used, we will assign the largest clusters as normal, and smaller clusters we will classify as anomalous. The threshold value for classification will be set at 5% of the total number of datapoints being tested and will have exceptions based on the number of clusters, or overly skewed clustering that will be fully explored in Chapter IV.

We are limited in ways to measure how good K-means is at classifying data since we do not have a true classification to compare the results of the clustering/classification

against. If we treat the classification derived from K-means as the true label to the data, we can then use a supervised ML model to the data and see how well we can reproduce the results from K-means using a less intensive process that would be easy to implement on new data in the future.

C. NAÏVE BAYES

The supervised method we have used to check our results is the naïve Bayes algorithm [21]. Naïve Bayes uses probabilities to determine the classification of data based on features chosen. Specifically, we will use the multinomial naïve Bayes algorithm (MNBA), which is commonly used for text classification using word vector frequency counts, as we have in our DF. MNBA is a fast and simple algorithm compared to other ML methods, which makes it preferable for working with large datasets. One downside of MNBA is that it is meant to be used with discrete values, and performance may be worse when using non-discrete inputs [22]. Naïve Bayes works under the assumption that each feature is independent from other features as discussed in Chapter II.

To use MNBA and verify its effectiveness, we will break our highest performing data subsets from K-means analysis into a 70/30 train test split dataset each: 70% of the dataset being used to train the MNBA model with 30% reserved to test the model. We will create the MNBA model for each data subset using all features of the subset as the input features to the model. We will create a confusion matrix to visualize the results of the NB modeling and compare precision, recall, accuracy, F1 and MCC scores from the NB model for each data subset and cluster value chosen. We will also examine the descriptive statistics, such as mean, standard deviation, min, max, and quartile values from the data classified as normal and compare it to the data classified as anomalous.

The methodology explored in this section will be implemented in the next chapter as it relates to the data of 100 000 JavaScript files used for testing. We will detail how the methodology proposed was implemented, the limitations and assumptions made to fit the proposed methodology and discuss the performance of the ML classifier used to classify each JavaScript file.

IV. IMPLEMENTATION AND RESULTS

This section will discuss the implementation of the methodology discussed in Chapter III as it relates to the data tested. We will discuss the environment used to conduct testing, the three stages of building the DF, creating the K-means classifier, and analyzing the proposed scheme and classification efficacy using naïve Bayes. We will conclude by comparing the nine combinations of datapoints to file size and comparing the results of these cases to determine the effect of file size on the results and determine how much data is needed before seeing a plateau in the naïve Bayes model performance of the proposed classifier.

A. TESTING ENVIRONMENT

All coding and analysis of work in this thesis were done using a Jupyter Notebook integrated development environment (IDE) using Python. Both the IDE and the raw JS data were accessed remotely. The raw data consists of over 2.7 million JavaScript files from the Alexa list of top 30 000 webpages that was collected in 2019 and saved in a JSON format for easy integration into the Python programming language, which was used for the entirety of this thesis work.

1. Preprocessing

To streamline access to the data and features, all data features are converted to lists and then appended and stored as a Python DF. The first 100 000 JavaScript files of the 2.7 million were pulled from the storage directory, stripped of its JSON format, and the raw JavaScript text was read to develop a set of features. The features included the file name, file path, file size of each file as it appears in the server directory, and the rank that each JS file's originating webpage had on the list of Alexa's top 30 000 webpages. These four features are compiled into a DF, and all subsequent features will be developed as lists that can be added onto the DF.

We developed a script that would read each JavaScript file one at a time, convert the file into a list of words, converting all spaces and special characters into a "" (blank

character) within the list. This is done using a Python tool called regular expressions (REGEX).

Regular expressions exist in multiple programming languages and allows us to search a text file for specific patterns, remove specific elements from consideration and put the results in the form of a Python list. In this analysis, it is used to remove all special characters and convert the raw JavaScript text file into a list that only contains words and characters defined by the JS file. All removed elements are replaced by a blank character within the list. Each script is then a list of characters or words separated by blanks; for example, the JS text: "Hello World!" would be converted to list: ['Hello', '', 'World', ''] consisting of two words, "Hello" and "World" separated by two blanks. One blank was created by replacing the space between the two words, and the second from replacing the exclamation point (!) special character.

Another Python tool called *Collections* is used to count every unique word/character that appears in the list and the frequency of that word/character. The result is a Python object that can be converted into two lists. List A contains the *element* of the original JavaScript file after converting to a list, and list B is a list of the *frequency* of each word. Figure 5 illustrates the process of using REGEX and Collections to convert each script into a list of elements and frequency counts using the previous code example.

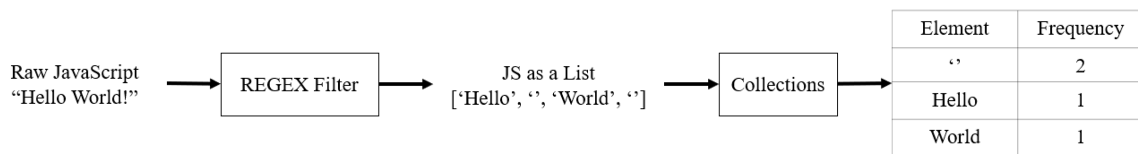


Figure 5. Flow diagram for converting JS to word frequency counts using *regular expressions* and *collections*.

In Figure 5, the combination of REGEX and Collections allows the fast breakdown of every JavaScript file into two lists. List A contain the names of all unique elements in the JS file and List B contains the frequency of each element in List A as shown in the table in Figure 5. One downside of this method is that it does not consider whether a word

appears as functional code or as a comment within the JavaScript, which may have a different meaning. After looking through several files to determine if this would be a significant issue, the cases in which a word only appeared or primarily occurred in a comment and had a different meaning were infrequent, as most observed files did not contain comments. The choice was made to sacrifice some precision in only counting the functional uses of JavaScript to have more features, more data, and run more efficiently.

From these two lists, we can create the first set of features to add to the DF. The length of List A (or List B) tells us the number of *unique elements* for each script. The sum of values in List B gives the *total elements* in each script. If we remove the blank elements (those leftovers after removing all special characters), we have one less unique element but have reduced the total elements by a large amount. This is because every removed element by the regular expression search is treated as a blank element, so the sum of all removed elements is the number of blanks in the file, which for some files was near 50% of the total number of elements. Both sets of features, the *total elements*, *unique elements*, as well as the total elements and unique elements after removing all blanks are features in the DF. These four features are similar, but not identical, and will be used to define other features later. For clarity, the feature names with removed blanks will end in the number 2 such as *total elements2* versus *total elements* in the list of DF features in Appendix A. The number of blanks contained in each file is also a feature that we extract from these two lists.

List A and List B can easily be used to calculate the entropy of each file. From information theory, entropy H is the average information (in bits) contained in a character a based on the frequency of characters [27]. We are instead using words in list A instead of characters to make up our alphabet A. A random variable f can be any word a in A, with the probability $p_f(a)$. In this case, we are calculating the average word entropy as in [27] as,

$$H(f) = -\sum_{a \in A} p_f(a) \ln p_f(a) \tag{1.9}$$

using each unique word in List A and its frequency in List B to calculate the probability of each element. This allows us to calculate the entropy in each JS file [27].

Next, we compare List A to a defined set of keywords. These keywords are a non-exhaustive list of chosen JavaScript words that fall into four categories; reserved words, objects, properties, and methods (OPM) words, event words and asynchronous words. *Reserved words* are those that can or should not be used outside of their explicit purpose. The *OPM* keywords chosen, are those that are reserved by the language but act as objects properties or methods, such as “isFinite” versus the reserved keyword “Boolean” [28]. *Event words* are used to program JavaScript events such as causing a sound to play when a user clicks a button on a webpage or preview a video when a user places their mouse cursor over the thumbnail of a video [29]. *Asynchronous words* are used to cause some type of action to occur either after an amount of time or at a time other than when the line of code is run [30]. These four categories of keywords are features of the DF as the combined frequency of all keywords assigned to the category as in Appendix A. Each individual keyword in these four categories is a feature, and the sum of every keyword makes up a feature called *keyword uses*. The words were chosen to consider common behavior, as with the reserved word and OPM word categories, and specific behavior with the event, and asynchronous categories. This feature list is not comprehensive, and many more words and categories could be added to look for other JS behavior.

With all the keywords and word vector frequencies added to the DF as features, we wish to calculate other metrics that may be of interest. Python DFs allow easy manipulation and mathematical operations among entire columns of data. Treating each feature column as a vector, four additional features are calculated. A *unique element ratio* is the number of unique elements divided by the total number of elements. This is calculated with and without considering blank elements. A *blank element percentage* is the percentage of total elements that are blanks in case we consider them. A *keyword percentage* or the number of keywords uses divided by the total number of elements, both with and without considering blanks. Finally, a *blank element ratio* is the number of blanks divided by the number of words in a JS file.

We now have all desired features in the dataset. One last thing to add is a categorical variable that will represent the relative size of the JavaScript file. Each JavaScript file will be given a category of *small*, *medium,* or *large* based on two features: *file_size* and *total elements*. Figure 6 illustrates these size groups for the first 1 000 JS files.

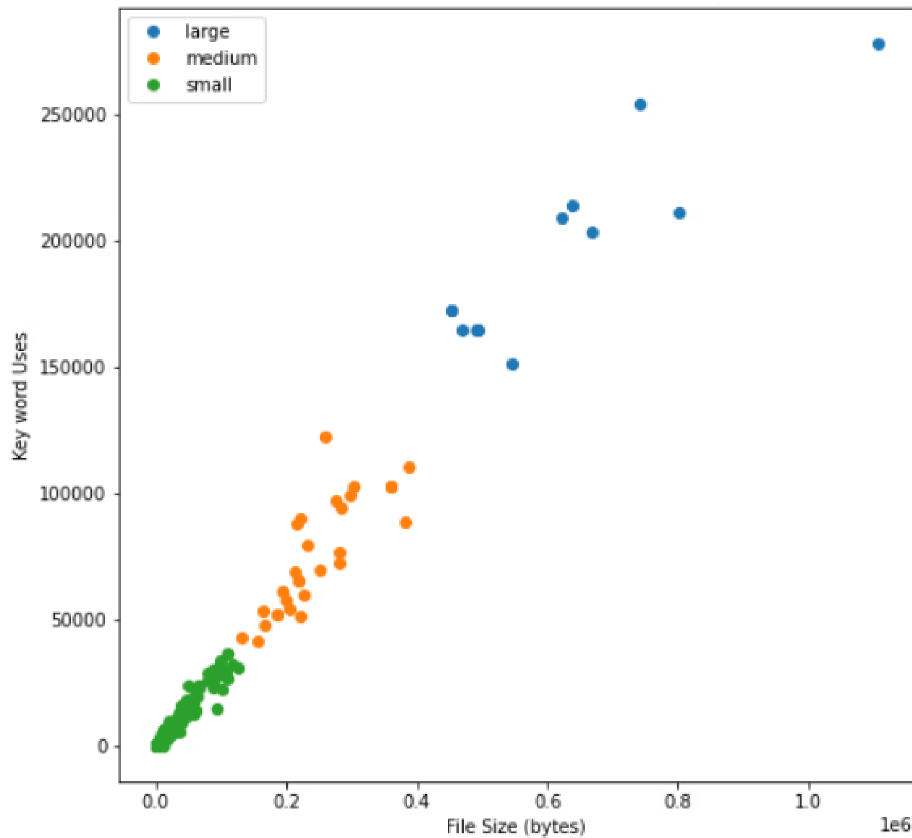


Figure 6. Illustration of the three size categories of JavaScript using the first 1 000 JS files. Three distinct size groups exist for JavaScript based on the distribution of *file_size* versus *keyword uses*. All data is assigned a size category based on how close its file size is to the mean of these three categories.

This categorical feature was added due to poor clustering when applying K-means even with z-score scaling as the relative size of each JS file was the dominant feature in clustering. The size categories are assigned to each datapoint based on how close each datapoint's file_size feature is to the mean of the three groups in Figure 6. The first 1 000

datapoints were used because they provided a clear separation between these three categories using these datapoints and using more datapoints introduced extreme outliers that skewed the three-category assignment. Since most features of the DF are word frequency counts, each feature is inherently correlated to file size; generally, larger files will have higher values on all features. Using the categorical variable as a means of reducing the impact of file size, clustering and classification should be less dependent on file size.

2. Subdividing The Data Frame

The total working DF of 100 000 datapoints is sub-divided into groups of the first 10 000, 50 000 and 100 000 datapoints and then separated into groups based on size category to create nine sub data frames that are tested individually. The distribution of the 100 000 datapoints by size category is given in Table 1.

Table 1. Comparison of three size categories: (a) *small*, (b) *medium*, (c) *large*. Examining the mean *file_size* and *total_elements* of the three size categories for the three categories we see the separation illustrated in Figure 6. All data is rounded to nearest integer, and file size is measured in bytes.

Feature	Count	Mean	STD	Min	25%	50%	75%	Max
File_size	96042	14241	87645	89	6407	13998	21642	30000
Total elements	96042	2145	6146	1	20	62	393	67681
(a)								
File_size	3232	227087	72051	127108	163361	215268	276600	435546
Total elements	3232	67248	22560	9795	51122	65338	81715	196115
(b)								
File_size	726	887864	606189	436101	510655	649702	1111303	5944747
Total elements	726	253447	163033	41481	163763	188244	307291	2022817
(c)								

Table 1 gives a description of two features of the DF for the 100 000 datapoints separated by size category. Included in the description for each feature is the count or number of datapoints the feature represents, which in this case is the total number of datapoints in the three size categories. The other description values are the mean, standard deviation, and quartile values of all datapoints for the listed features. The two features are *file_size*, and *total elements*. The description metrics are used to compare the size categories in Table 1 and will be used to compare data with different classifications further on.

In Table 1, comparing the three size categories, we observe that the *small* size category represents 96 042 datapoints with a *file_size* mean of 6 960 bits and 2 144 *total elements*. The *medium* size category represents 3 232 datapoints with a mean *file_size* of 227 087 bits and 67 248 *total elements*. The *large* size category represents just 726 datapoints and has a mean *file_size* of 887 864 bits and 253 447 *total elements*. Grouping data from the first 10 000, 50 000, and 100 000 datapoints split among three size categories creates nine sub-DF's that will be examined to determine if more than 10 000 datapoints are required to classify JavaScript files and what effect the method of classification has considering JavaScript pages of vastly different sizes. For all the following analysis, the largest sub-DF consisting of the *small* files from the 100 000 JS files contain a total of 96 042 files and we will compare this to the other eight groups when discussing results. After selecting a group to analyze, we will scale the data in preparation for K-means application.

We used a Python function called z-score to scale all 131 numeric data features since we are working with vectors that have a range in the thousands, such as frequency counts, while we also have percentages that have values between 0 and 1. The z-score does this by scaling each feature value X based on the mean μ and standard deviation σ of the column of data such as:

$$z = \frac{(X - \mu)}{\sigma} \tag{1.10}$$

where z is the new scaled value. The z-score will not permanently change the actual data but makes it easier to visualize at a later stage when we apply K-means [31].

B. K-MEANS CLUSTERING

K-means clustering is an unsupervised ML method for labeling data. K-means clustering allows us to identify *n* clusters of similar data that we will then convert into a binary classification of normal or anomalous.

1. K-Means Clustering and Elbow Plot

Using each of the sub-DF's mentioned in the previous section, the goal is to determine the following: what features in the DF can differentiate each JavaScript file and reduce the cluster distortion, i.e., the average distance between each cluster center and the datapoints associated with that cluster? Also, we want to consider using as few features as possible to reduce complexity and speed up the clustering process, and we would like the chosen features to have low correlation with each other.

Altogether, ten features were chosen for testing to identify what combination of factors may effectively group and classify the JavaScript files. These subset combinations are listed in Appendix B.

For each subset, we run an iterative K-means model built using the SK-Learn library in Python to calculate a K-means model using between two and nine cluster centers [16], [17]. We need at least two clusters since we are using the K-means results to build a binary classifier, and we are testing higher cluster values to reduce potential distortion and reveal the outliers in the data that will create the distinction between normal and anomalous JavaScript. We then plot the average distortion between all centroids versus the number of clusters used as in Figure 2. We will describe each K-means trial and classification for a particular subset and cluster value as the subset cluster pair (SCP) moving forward. The SCP will be followed by two digits separated by a hyphen. The first digit is the data subset

used, and the second digit refers to the number of clusters used from the K-means application. We can then compare the average distortion between each data subset and cluster value to determine the best subset and centroid count to use for classification.

From Figure 7, we can see subsets 6,7,8, and 9 are far better at reducing distortion compared to the other six subsets and will therefore be the only four subsets considered in further analysis. Figure 8 focuses on the elbow plot of just subsets 6,7,8 and 9. Subset 6 and subset 7 have similar performance even though subset 6 has the least number of features and subset 7 has the most features of the remaining four sets. Set 8 and set 9 have similar cluster values at four and eight clusters but subset 9 consistently has the smallest distortion. We observed that subset cluster pair (SCP) 9-4 appears to have the best ratio of distortion versus complexity. While SCP 9-8 has slightly lower distortion, the added complexity by doubling the number of clusters makes SCP 9-4 the preferred selection. We will consider both SCP 9-4 and SCP 8-8 since SCP 8-8 performs similarly to SCP 9-8 and provides a different subset of data to consider with our second analysis technique, the silhouette plot.

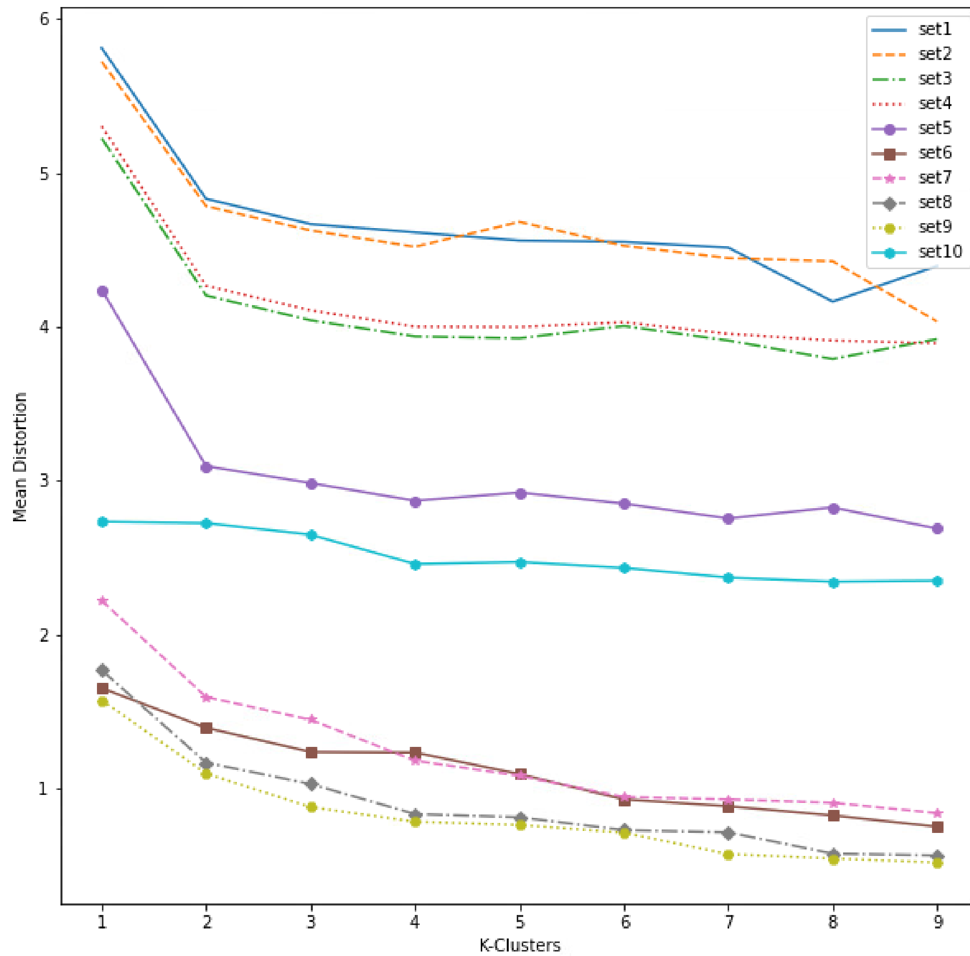


Figure 7. K-means selection via elbow plot using small data pulled from 100 000 datapoints. We see that subset 1 through subset 5 and subset 10 have a significantly larger mean distortion and will be eliminated from consideration for developing the binary classifier.

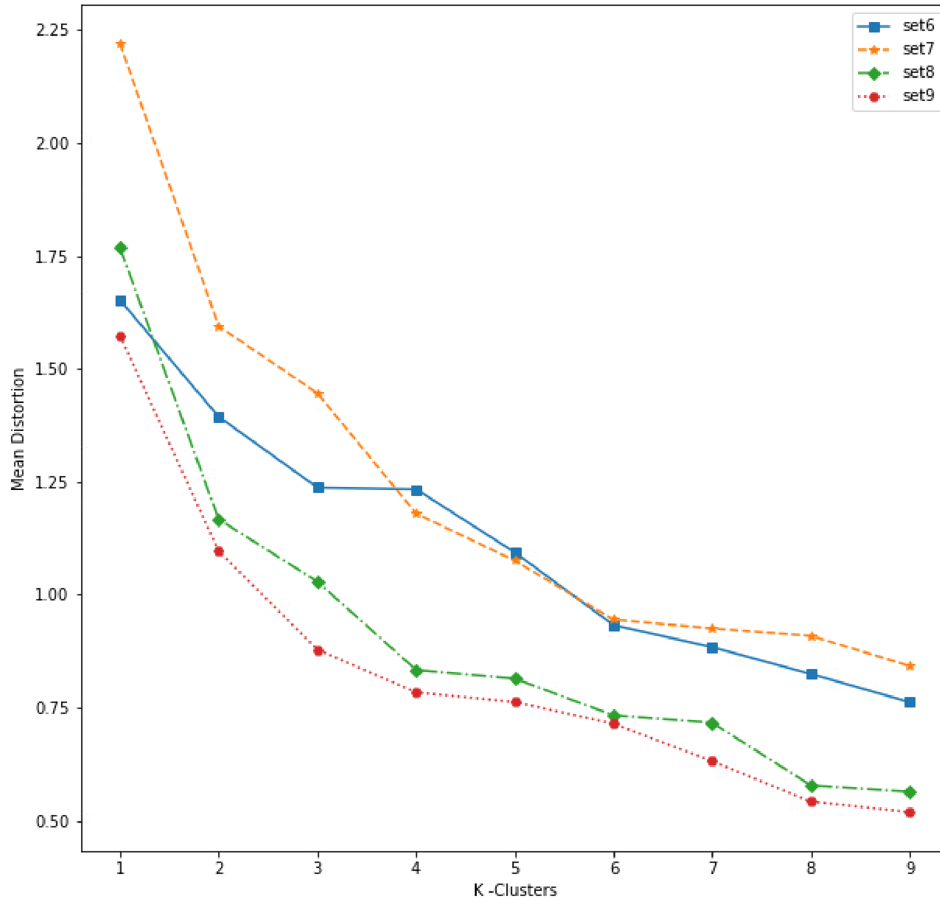


Figure 8. K-means selection of the four best performing datasets. Observe the elbows for subset 6 with six clusters, subset 9 with four clusters, and subset 8 with eight clusters.

The second method used to analyze the K-means clustering is by silhouette plot of each SCP for subsets 6-9. We will re-examine the silhouette of SCP 6-6, SCP 8-8, SCP 9-4 based on their apparent elbows in the elbow plot and look for any other SCPs with high silhouette scores to consider developing as a classifier.

2. K-Means Silhouette Plot

A silhouette plot is an alternative method to the elbow plot to decide on how good the K-means application is. While elbow plots visualize the within group variance, inertia, or distortion for a specific number of clusters, a silhouette plot allows us to visualize the inter group variations and the distribution of data within those clusters [20].

Each silhouette plot provides a numerical value for how good a cluster assignment is for both a datapoint and the entire K-means model. This value is called the silhouette score, which applies to every datapoint, and is summarized on each plot as an average silhouette score (AvSS). The silhouette score and AvSS are measured between -1 and +1, with +1 representing a perfect cluster classification and -1 representing an incorrect cluster classification. A 0 represents a weak classification or a split classification in which the datapoint is equidistant from two centroids, so the classification is not clear [20]. Visually, the thickness from top to bottom of each group in the silhouette plot represents the number of datapoints in the plot. If the right edge of a cluster group is flat, it means that all values of that plot are equally centered around that cluster center whereas sharp, pointed groups imply the assigned data varies in distance from the centroid.

Observing the silhouette plot of all cluster values of the four best subsets from the elbow plot in Figure 8, we chose four SCPs to examine further. The distribution of clusters was considered extensively in this thesis. To determine what silhouettes may be the best choice for the classification assignment, we are looking for the natural outliers of the data, the JS that makes up smaller, less-defined clusters to consider anomalous and contrast them against larger well-defined clusters that represent normal JavaScript. In this respect, we want to see some thin but moderately performing silhouettes for some clusters to represent anomalous JS, and thicker and better performing silhouettes for the normal JS. Other considerations used when examining silhouette plots were to have few values near or below zero in the silhouette plot as they are poorly defined and are more difficult to classify as normal or anomalous. For each silhouette plot, we want to consider the higher performing SCPs in terms of their AvSS.

Taking into consideration the distortion to cluster ratio from the elbow plot along with the silhouette score for all SCP's, four SCPs were chosen for further analysis. SCP 9-4 was chosen due to its desirable distortion to cluster ratio. SCP 8-2 was chosen because it had the highest AvSS. SCP 6-6 was chosen for a balance in both plots as well as having the least number of features, and SCP 8-8 was chosen for having a low number of datapoints with silhouette score near zero as well as a low distortion and moderately performing AvSS. The silhouettes for these four chosen SCPs are shown in Figure 9.

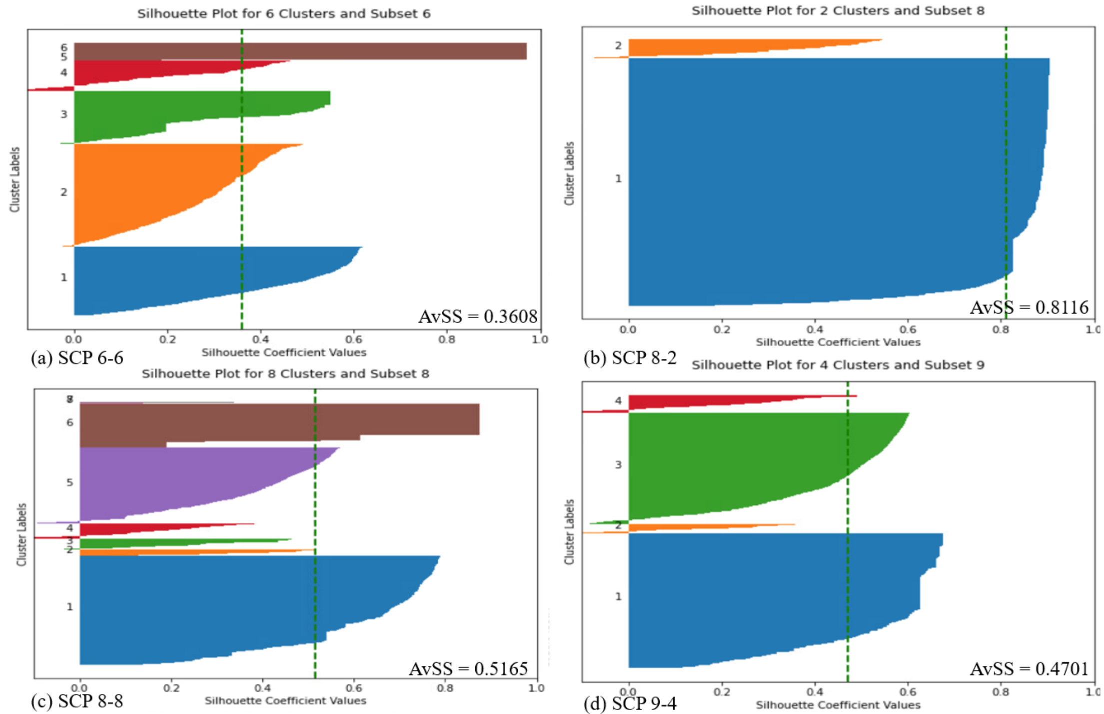


Figure 9. Comparing silhouettes of four chosen SCP's. Note the distribution of SCP 8-8 and SCP 9-4 show multiple broad groups with moderate silhouette scores as well as thin groups which are the indications desired for our classification. Note SCP 8-2 had the highest observed AvSS. SCP 6-6 may be too evenly distributed to make the desired classifier.

SCP 6-6 (a) has the worst AvSS of the four chosen SCPs at 0.3608. SCP 6-6 was chosen because subset 6 has the least number of features, the lowest correlation between features, which will be seen in a later section, and has a relatively low number of datapoints near or below the zero on the silhouette plot. We also see that five of the six groups have values exceeding the average, meaning that while the overall AvSS is low, five of the six groups are well defined relative to each other. SCP 8-2 (b) was chosen because it had the largest overall silhouette score of any subset. With an AvSS of 0.8116, there is a relatively clear distinction between the two clusters compared to other plots. There are a low number of datapoints at or around 0 or in the negative region; however, the smaller group 2 is not defined well since no datapoints have a high silhouette score relative to group 1. SCP 8-8 (c) was chosen due to its low distortion in the elbow plot, a relatively high AvSS at 0.5165, and most clusters are near the AvSS indicating that the clusters are somewhat distinct;

however, clusters 7 and 8 containing few datapoints fall below the AvSS. There are groupings of various sizes, which is the indication that the smaller groups may be outliers to classify as anomalous and larger groups to classify as normal. SCP 9-4 (d) was chosen because it had the best balance between distortion and cluster count based on the elbow plot. The AvSS of 0.4701 is moderate compared to the other plots shown. There are some datapoints with negative or zero silhouette scores but not significantly worse than other SCPs. The clustering pattern with two large clusters and two small clusters are clear and distinct sized groups that appear suitable for classifying normal and anomalous data just from observation.

There needs to be consideration for how we organize multiple clusters for a binary classification for all the JS files. Two considerations are to be made. In the case of a binary clustering scheme, for example, let the larger cluster containing up to 90% of the data be classified as normal with the smaller cluster containing the remaining 10% of data representing anomalous JS. The second consideration is for non-binary clustering schemes. We will use a 5% threshold such that if a cluster group contains more than 5% of the total data, it will be considered normal. If a cluster does not meet this 5% threshold, it will be considered anomalous. If there are SCPs in which all groups contain more than 5% of the total data, then the least populated groups will be considered anomalous until the total number of anomalous files exceeds 1% of the total files. This 1% requirement is to ensure that there is enough data classified as anomalous to ensure that the data can be split for use with naïve Bayes later.

Considering the small DF of 96 042 files, a 5% threshold is 4 802 datapoints, and the 1% threshold is 960 datapoints. This means that any cluster group with more than 4 802 files will be considered normal, and any groups below this value will be considered anomalous, and we must have at least 960 JS files labeled as anomalies. Figure 10 shows the conversion of SCP 8-8 into binary classification, and Table 2 lists the resulting binary classifier counts for the four SCPs we are testing.

CLUSTER (K)	JS FILE COUNT
1	39846
4	27519
7	16189
5	5752
2	3636
0	1967
3	870
6	263

CLASSIFICATION	JS FILE COUNT
0	89306
1	6736

Figure 10. Converting K-clusters into a binary classifier for SCP 8-8. Observe the 5% threshold is 4 802 files. Since clusters 2, 0, 3, and 6 on the right each contain less than the threshold, they are assigned as a 1 for anomalous. Likewise, clusters 1, 4, 7, and 5 are classified as 0 for normal.

Table 2. Binary classification distribution of four SCPs. Observe the final distribution of normal to anomalous JS files is similar for SCP 6-6, SCP 8-2, and SCP 8-8, while SCP 9-4 has a more skewed distribution due to the classifier rules. Only one of four clusters from SCP 9-4 was classified as anomalous.

	CLASS 6-6	CLASS 8-2	CLASS 8-8	CLASS 9-4
ZEROS	89902	89172	89306	92764
ONES	6140	6870	6736	3278

With all four classifiers assigned to the data, we look at all features of the entire DF by classification or just the subset features for the four classifiers. For example, in Table 3, we take into account the description statistics of the *first five* features of the normal data versus the anomalous data. In Table 4, we compare the *subset specific* feature statistics for the normal and anomalous data, i.e., the features that were used in the clustering process. Note that in further discussion on classification, *zeros* will be used to refer to *normal* data, and *ones* will be used to refer to anomalous data when referencing the figure results.

Table 3. SCP 8-8 binary classification header data comparison. Note the mean *file_size* of the normal data in (a) is much smaller than in (b), considering that file_size was not a feature of subset 8. The overlap of maximum values indicates that there is more to the clustering than just *file_size*.

Feature	Count	Mean	STD	Min	25%	50%	75%	Max
File_size	89306	2457	7238	89	159	266	775	126955
File_line_count	89306	32	111	1	1	3	12	3628
Average_line_size	89306	200	1222	15	57	105	161	125366
Total elements	89306	701	2217	1	17	51	200	64017
Blank elements	89306	375	1427	0	9	26	108	55374
(a)								
File_size	6736	66663	26627	11634	46112	64036	89261	126979
File_line_count	6736	1045	618	1	620	857	1327	6554
Average_line_size	6736	200	3415	13	44	58	114	118329
Total elements	6736	21283	8920	2447	14420	19102	28678	67681
Blank elements	6736	8871	4849	7	5586	7636	10953	59110
(b)								

Table 4. SCP 8-8 binary classification feature data comparison. Examining the mean of all features in (a) for the normal data, we see that they are all significantly lower than in (b) representing the anomalous classified data. The overlap in max values for each feature suggests that *file_size*, is not the only association for the clustering.

Feature	Count	Mean	STD	Min	25%	50%	75%	Max
Total elements2	89306	326	984	0	6	24	90	24040
Unique_elements2	89306	66	160	0	5	13	39	3574
Unique_word_entropy2	89306	3.5566	2.0687	0	2.2999	3.4992	4.9232	11.2571
Keyword_uses	89306	51	156	0	1	3	13	1576
Event word uses	89306	0	1	0	0	0	0	28
Reserve word uses	89306	35	110	0	0	2	9	1437
OPM uses	89306	15	46	0	0	1	3	752
Async uses	89306	1	3	0	0	0	0	83
(a)								
Total elements2	6736	12411	5187	1501	8280	10750	17330	29261
Unique_elements2	6736	1115	476	203	869	1044	1300	13613
Unique_word_entropy2	6736	7.133	0.5393	5.2373	6.8372	6.9686	7.3466	13.4806
Keyword_uses	6736	2364	1007	1	1610	2170	3000	6336
Event word uses	6736	7	8	0	3	5	9	94
Reserve word uses	6736	1627	715	1	1125	1453	2076	4295
OPM uses	6736	691	317	0	459	690	854	2254
Async uses	6736	39	31	0	22	31	47	380
(b)								

From Table 3, comparing file sizes of (a) normal and (b) anomalous data, we see that the normal data has a smaller mean and standard deviation (2 457 bits, 7 239 bits), than the anomalous data (66 662 bits, 26 627 bits), which is less concentrated around it's mean. This shows that even while file size was not included as a feature, it is having a distinct effect on the classification. From Table 4, we see that the deciding features are all affected by the size, i.e., the anomalous data trends towards large files in terms of total elements and all metrics for this dataset. It is not a clear-cut trend in the classification; however, we see a significant overlap between the minimum and maximum values for all metrics regardless of classification. This shows that while the normal and anomalous classification is sensitive to size and an important factor in classification, it is not solely a factor of the size. Consider the description data for SCP 6-6 in Table 5. SCP 6-6 is classifying the extremely small data as anomalous in contrast to SCP 8-8 classifying the larger outliers as anomalous. With the anomalous data averaging just over one line of code compared to the

110-line average for the normal data. This means that depending on the features chosen, we can identify those extremely large outliers in the case of SCP 8-8 or the extremely small outliers as in SCP 6-6. Regardless of the features chosen, however, file size is a significant factor in determining the classification, despite not being directly included in the features used in the clustering from either subset. Note that subsets 8 and 9 have features that reflect file size since they are all frequency count features while subset 6 has far less size dependent features except for entropy. It appears then that choosing the less size dependent features is how SCP 6-6 identifies the smallest outliers while using the more size dependent features allows SCP 8-8 to identify the larger outliers.

Table 5. SCP 6-6 binary classification header data comparison. In contrast to SCP 8-8, the normal data in (a) is significantly larger in *file_size* compared to the anomalous data in (b). While classification is heavily dependent on *file_size*, the feature set is varying in what size outliers are being grouped to form the normal and anomalous data. All values rounded to the nearest integer.

Feature	Count	Mean	STD	Min	25%	50%	75%	Max
File_size	89902	7425	19717	89	173	326	1789	126979
File_line_count	89902	110	334	1	1	4	24	6554
Average_line_size	89902	205	1532	13	52	103	161	125366
Total elements	89902	2290	6326	1	28	75	493	67681
Blank elements	89902	1037	2957	0	14	37	270	59110
(a)								
File_size	6140	143	1701	91	101	104	109	120950
File_line_count	6140	1	<1	1	1	1	1	23
Average_line_size	6140	119	370	29	101	104	109	28046
Total elements	6140	12	396	1	1	3	6	24277
Blank elements	6140	6	88	0	1	3	6	6440
(b)								

We will be able to quantify the effectiveness of these two feature sets and classification schemes with naïve Bayes in the next section.

C. NAÏVE BAYES APPLICATION

K-means is a computationally intensive method of clustering as discussed in Chapter II. After computations are made, a user is needed to manually choose which SCPs to build the classifier, which may change based on the classifier rules the user has chosen. Therefore, a faster supervised ML method for approximating the same classification derived from K-means would be ideal for reducing computation time and quantifying the effectiveness of the model. Naïve Bayes algorithm allows us to meet both requirements if we consider the results from the K-means algorithm as a true classifier.

Naïve Bayes is a supervised ML tool that requires labeled data, meaning the data must already be classified as normal or anomalous. Naïve Bayes assumes that all the data input vectors used are independent [21]. Figure 11 shows that there is low correlation among the subset 6 features so there is almost no linear dependance between the features, while subset 8 has moderate to high correlation among features which shows represents a linear dependance between features. Note that subset 9 is omitted since subset 8 contains all the same features as subset 9. The contrast in feature correlation between these two subsets again demonstrates the relationship among the size dependent features of subsets 7, 8, and 9 while subset 6 has mostly independent features.

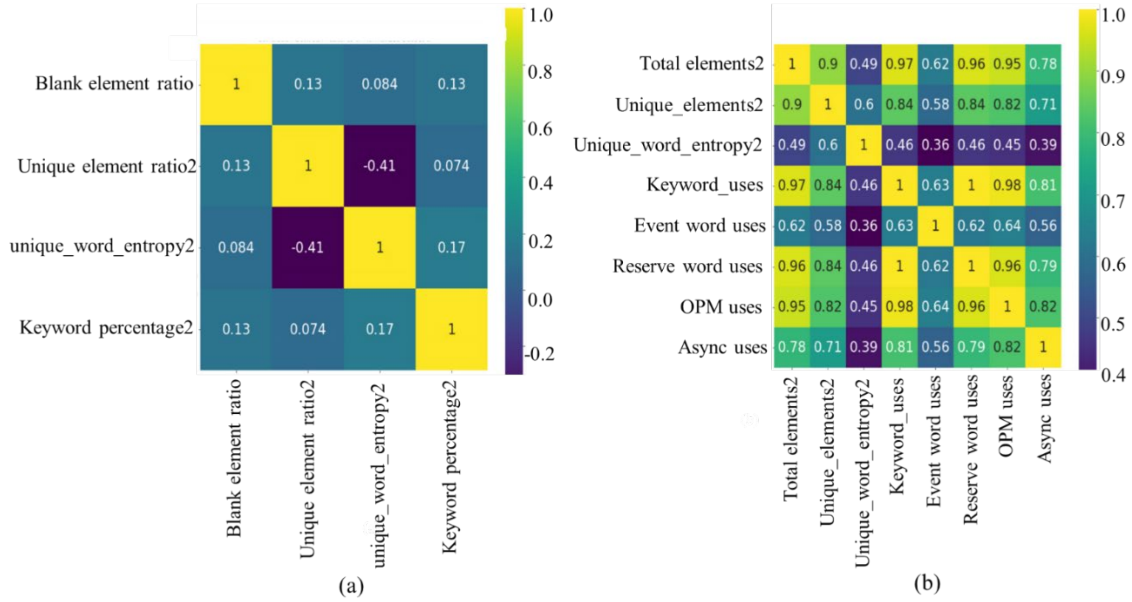


Figure 11. Feature correlation comparison for subset 6 and subset 8. In (a) we see the features of subset 6 have low to moderate correlation with the largest correlation magnitude of 0.41. In (b) we see that most features of subset 8 have a correlation over 0.5 to every other feature.

We now apply the multinomial naïve Bayes algorithm to our dataset in Python. We define the features used for each cluster subset as the input features with our K-means classification as the outcome vector. A built-in train-test split function is used to divide the data into training and testing groups at a 70/30 train-test ratio. Other ratios were tested as well with no discernible improvement in results. The model creates a predicted outcome vector for the training data (including the true classification), which can be compared to the training classification provided. The model can then be applied to the test data to predict the classification based on the input (excluding the true classification). We can now compare the results of the training and testing in the form of a confusion matrix as well as Precision, Recall, Accuracy, F1, and Matthews Correlation Coefficient (MCC) values. The MCC is a desirable metric for measuring the effectiveness of our binary classifier, but some later results will not allow us to calculate this metric, thus Precision, Recall, Accuracy and F1 are used as well [25]. In referring to the results of the naïve Bayes application, we will use the term *model* followed by the two-digit SCP value that the naïve Bayes model is based upon, i.e., the naïve Bayes results for the SCP 8-8 will be referred to as model 8-8.

In Table 6, we can see the break for our train test split data for SCP 8-8 in terms of the classification vector with 67 229 (70%) datapoints in the training set and 28 813 (30%) datapoints in the testing set.

Table 6. Train/test split (70/30) of SCP 8-8 classifier for training the naïve Bayes model. The entries contain the number of datapoints assigned to train/test groups by classification.

Classification counts	Train split	Test split
0 (Normal)	62480	26826
1 (Anomalous)	4749	1987

After training the model, we find that the F1 score is approximately 0.961 and the MCC is 0.752. Comparing the training data against our test data, we see similar performance, with an F1 score of 0.9621 and an MCC of 0.7517. The full list of metrics is shown in Table 7.

Table 7. Naïve Bayes results for predicting the SCP 8-8 classification. The testing set metrics are the results of predicting new data, compared to the training data that was used to build the naïve Bayes model. The maximum value is 1.0.

Dataset	Precision	Recall	Accuracy	F1	MCC
Training Set	0.9698	0.9572	0.9572	0.961	0.752
Testing Set	0.9703	0.9584	0.9584	0.9621	0.7517

It should be noted that the precision recall, accuracy and F1 metrics are using a weighted measurement in that the values are scaled based on the number of datapoints in each category, i.e., the data labeled as zeros have a much greater impact on the score than

the data labeled as ones. We can visualize the results of the model using the confusion matrix in Figure 12.

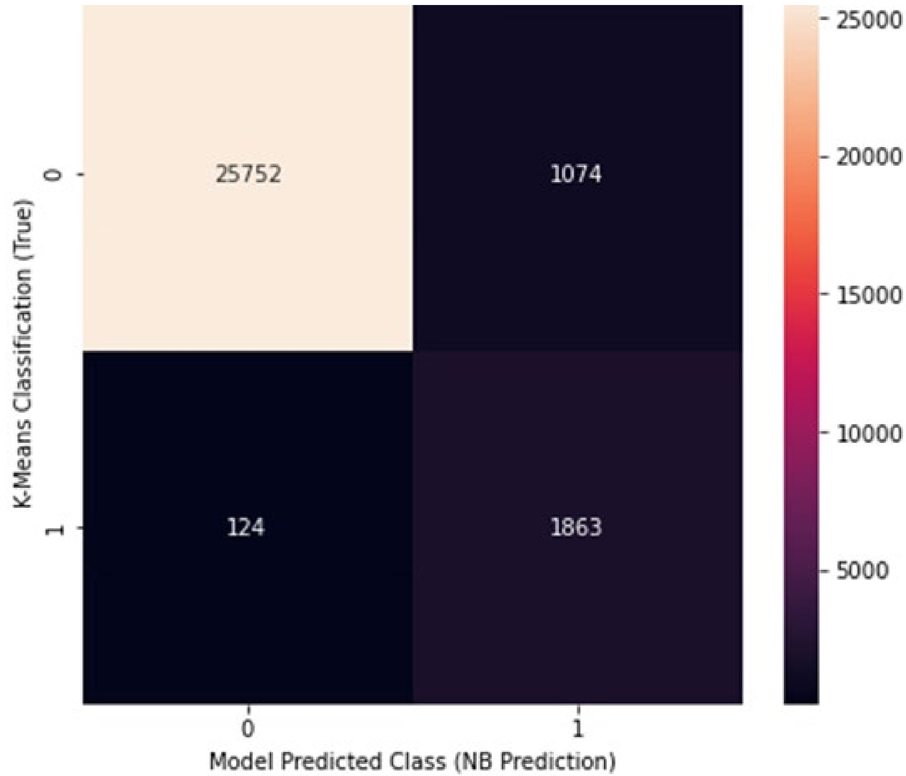


Figure 12. Confusion Matrix for SCP 8-8 naïve Bayes results. The four values represent the number of true positives, false negatives, false positives, and true negatives. This shows that the model is proportionally split on misclassifying zeros as ones, and vice versa.

The confusion matrix lets us examine the relations among true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) when comparing the results of the K-means classification on the Y-axis to the naïve Bayes predicted classification on the X-axis. Examining Figure 12, we see that of the 26 826 true zeros, 1 074 were predicted as ones, and of the 1 987 true ones, 124 were predicted as zeros. The model does a moderate job at successfully predicting the binary classification with the MCC of 0.7517. Table 8 shows the metrics for the four SCP cases chosen in the previous section.

Table 8. Train test split and metrics for four SCP models. Based on naïve Bayes model performance, model 6-6 was the worst classifier, model 9-4 performed moderately, while model 8-2 and model 8-8 had similar results across all metrics. Due to classification rules, there were only about half as many anomalous files for model 9-4 which may have impacted model performance.

Classification counts	Train split	Test split	Dataset	Precision	Recall	Accuracy	F1	MCC
0 (Normal)	62970	26932	Training Set	0.8763	0.9181	0.9181	0.8967	-0.0349
1 (Anomalous)	4259	1881	Testing Set	0.8725	0.9153	0.9153	0.8934	-0.0372
(a) SCP 6-6								
0 (Normal)	62388	26784	Training Set	0.969	0.9569	0.9569	0.9606	0.7514
1 (Anomalous)	4841	2029	Testing Set	0.9694	0.9579	0.9579	0.9614	0.7501
(b) SCP 8-2								
0 (Normal)	62480	26826	Training Set	0.9698	0.9572	0.9572	0.961	0.752
1 (Anomalous)	4749	1987	Testing Set	0.9703	0.9584	0.9584	0.9621	0.7517
(c) SCP 8-8								
0 (Normal)	64892	27872	Training Set	0.9768	0.9501	0.9501	0.9593	0.5955
1 (Anomalous)	2337	941	Testing Set	0.9776	0.9502	0.9502	0.9598	0.5837
(d) SCP 9-4								

Model 6-6 is the only model with a negative MCC of -0.0372 indicating that the data set may not translate well to a naïve Bayes model, which is surprising since subset 6 had the least correlated elements of all subsets tested, which is an assumption made when using naïve Bayes. Another possibility is that the classification rules may not work well for this SCP as most of the clusters were of similar size, which is bad for our use-case in which we want a mixture of large and small clusters that represent the large majority of JavaScript versus the small outlier clusters. Model 8-2 performs well overall with an MCC of 0.7501. Recall that this model had the highest performing AvSS of the four when comparing the silhouette plots. This may be due to the direct translation of SCP 8-2 from a binary clustering scheme to a binary classifier, i.e., the outcome vector is not being reduced from three or more clusters into a binary classification as with the other three SCPs making the model far simpler to predict.

Model 8-8 has the most interesting results. It performs similarly to model 8-2 in predicting both zeros and ones better than the other models, but it does so when coming from a non-binary clustering with eight centroids instead of two. This is peculiar since we

may have expected worse classification of the ones as we saw in model 6-6 or with model 9-4. The MCC of model 8-8 is similar to 8-2 with 0.7517 and 0.7501, respectively; similar performing precision, recall, accuracy and F1 scores. Model 9-4 has an MCC of 0.5837, which is below that of model 8-2 or model 8-8.

The results for models 6-6, 8-8, and 9-4 show that it may not simply be the conversion to a binary classifier that degrades naïve Bayes performance. There needs to be enough datapoints in the anomalous category to obtain a moderately performing classification with naïve Bayes. The SCP 6-6 classification using the established classification rules means that not enough data fell into the anomalous category, resulting in an MCC of zero due to insufficient data in the test set. Since model 8-8 performed on par with model 8-2 in all metrics and is not a direct binary classifier, we see that the reduction from K-Clusters to a binary classifier is not necessarily going to create a poor model with naïve Bayes. Model 9-4 shows that feature set 9 is not as easily modeled using naive Bayes as feature subset 8 was, thus leading to a worse MCC. Model 8-8 was the best overall performer, and we will explore this subset of data and results further. Recall Table 4 was used to compare the description data for the subset 8 features breakdown of SCP 8-8 for the zeros and ones.

One reason that dataset 8 performed better could be due to two features not included in subset 6 or 9. The two features are *total elements2* and *keyword_uses*, which are both highly correlated to the file size. From Figure 13, we can compare these two parameters for SCP 8-8 classification using scatterplots to identify how the classification is related to these two parameters. In Figure 13(a) we see the distribution of all datapoints, colored by classification. In Figure 13(b) we observe the 89306 zeros representing normal JS are far more concentrated in their distribution compared to the broad distribution of the 6736 ones representing the anomalous data in Figure 13(c). Seeing the two overlap Figure 13(a) shows that the size while influencing the classification, is not the only factor in classification, or we would see a more distinct division in the data as we saw in Figure 6, when creating our size categories. This shows that the classification of data as normal or anomalous is related to file size as the vast majority of JavaScript is relatively small. With a mean of just 326 total elements for the normal JS and over 12 000 total elements for the

anomalous JS shown in Table 4, we see that the total elements and size in general are still an important factor in classifying the data. This is in contrast to previous discussion for SCP 6-6 in which we saw the smaller mean data classified as anomalous and the larger mean data classified as normal.

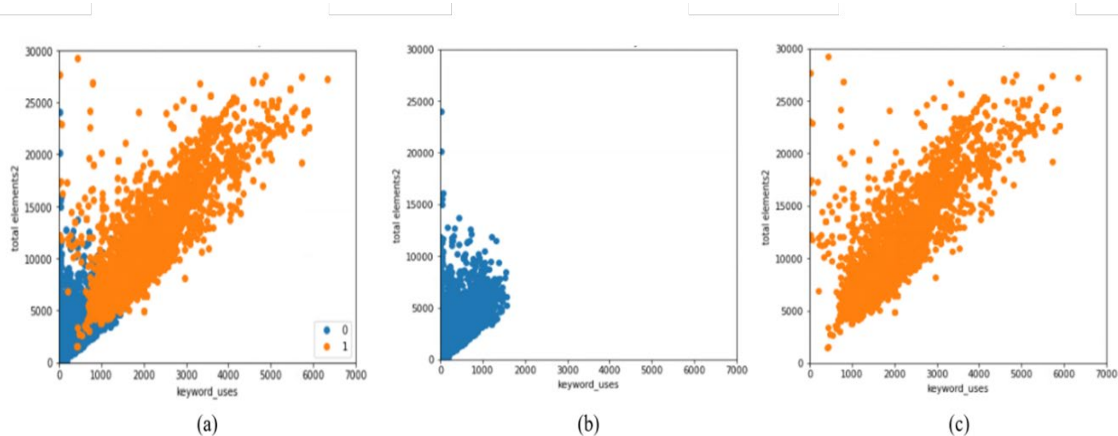


Figure 13. Comparison of *Keyword Uses* and *Total Elements2* features versus classification. The 89 306 normal datapoints in (b) are highly concentrated in terms of size represented by total elements and keyword uses, while the 6 736 anomalous datapoints are relatively vast in the distribution of data.

Model 6-6 was the worst performing of the four tested, but the poor performance was likely due to the chosen number of clusters given the classification rules established, and we will see that this performance is not consistent with similar SCP's when testing the 10 000 datapoint and 50 000 datapoint groups. With a different selection of cluster centers, subset 6 could perform better under the established rules, which we will see in the next section.

Considering the subset of data we described throughout the clustering, classification, and analysis process, it appears that for the small category of data, feature set 6 produces the least size dependent classification. Feature subset 8 was a better performer across all analyses in terms of metrics with the highest silhouette score for SCP 8-2 and the best MCC for Model 8-8, but the classification is more skewed towards overall file size, compared to subset 6. To say one feature subset is better at classifying data is

dependent on how heavily file size is to be weighed as a factor. By the numbers, the vast majority of JavaScript is in the small category, and a small percentage of that has a distinctly different size. Considering all models and SCPs tested with poor to moderate silhouette scores and moderate MCC scores, no feature subset tested proved to be ideal for classifying normal or anomalous JavaScript, but feature subset 6 may be the basis for developing further features if size is to be omitted from consideration as using word vector frequencies is entwined with file size.

While all analysis up to this point was conducted on the *small* category of data using 100 000 datapoints, the same process was run on 10 000, and 50 000-point datasets as well as for the *medium* and *large* size categories for a total of 9 sets of results. In the next section, we will discuss the impact of varying amounts of data on the results and what conclusions can be made for the significantly larger size categories compared to the small files we have examined up to this point.

D. RESULTS FROM SIZE COMPARISON

To determine the size of data points needed for classification into small, medium, and large, we broke the first 100 000 datapoints into divisions of size category and number of files. The breakout of these groups is in Table 9 with the number of associated JavaScript files as well as the mean and standard deviation of the file size provided within the parenthesis. From the table, we can see that over 96% of JavaScript falls into the *small* category based on our 100 000 datapoints case, and less than 1% in the large category. The low number of datapoints in the large size category would be classified as anomalous using the established classification rules based on the *file_size* and *total elements* alone. However, this factor is omitted in the interest of using content specific features versus file size directly for classification. This also led to the decision to create these size categories for analysis to limit the effect of file size on the classification.

Table 9. Distribution of data into three size groups for varying number of total datapoints. The mean and standard deviation of the file_size of each group is provided along with the number of datapoints that each category contains.

	10,000 datapoints	50,000 datapoints	100,000 datapoints
Small	9606 (6889, 19139)	48005 (6933,19063)	96042 (6960, 19164)
Medium	322 (229508, 72171)	1619 (227649, 72697)	3232 (227087, 72,051)
Large	72 (930965, 834907)	376 (906412, 625580)	726 (887864, 606,189)

By applying the K-means and naïve Bayes algorithms as we did with the 100 000 datapoint small category on the other eight groups, we obtained two or three SCPs for each by K-means and verified them using naïve Bayes. The choices for these included the SCP with the highest silhouette score for the group, The SCP with the best elbow based on the elbow plots, and then one additional SCP chosen by balancing the analysis of these two kinds of plots. Table 10 summarizes the results of K-means clustering of the chosen SCPs for all groups based on the size category and the number of total datapoints used. We will summarize the data by each size category, discussing the effect of changing the total datapoint pool from 10 000 to 100 000 on the results and then conclude with overall observations.

Table 10. K-means and naïve Bayes results for all nine DF groups. The table displays the group assignment in the left most column in terms of number of datapoints used. The second column gives the chosen SCPs from K-means analysis, supplemented with the distortion and AvSS for the chosen SCPs in column 3 and 4. Columns 5-9 give the results of the naïve Bayes modeling on the 30% test set for the binary classifier derived from each tested SCP in column 2.

Groups	SCP	Avg Distortion	AvSS	Test Precision	Test Recall	Test Accuracy	Test F1	Test MCC
10k small	C6-7	0.8976	0.3794	0.957	0.9618	0.9664	0.9561	0.5458
	C8-2	1.1688	0.8128	0.9668	0.9563	0.9563	0.9595	0.7627
	C9-4	0.7909	0.473	0.974	0.9455	0.9455	0.9552	0.6012
50k small	C8-2	1.1669	0.8122	0.97	0.9597	0.9597	0.9628	0.7624
	C8-8	0.5785	0.5174	0.9709	0.9598	0.9598	0.9632	0.7588
	C9-4	0.7843	0.4701	0.9765	0.9517	0.9517	0.9602	0.6077
100k small	C6-6	0.9290	0.3608	0.8725	0.9153	0.9153	0.8934	-0.0372
	C8-2	1.1674	0.8116	0.9694	0.9579	0.9579	0.9614	0.7501
	C8-8	0.5742	0.5165	0.9703	0.9584	0.9584	0.9621	0.7517
	C9-4	0.7842	0.4701	0.9776	0.9502	0.9502	0.9598	0.5837
10k medium	C6-2	1.3326	0.3707	0.7894	0.6907	0.6907	0.5912	0.2488
	C6-5	0.9648	0.3357	0.9078	0.8969	0.8969	0.863	0.3862
	C6-8	0.7633	0.3293	0.8996	0.9485	0.9485	0.9234	0
50k medium	C6-2	1.4524	0.4636	0.8255	0.7737	0.7737	0.6863	0.1997
	C6-5	0.9350	0.3253	0.9438	0.9403	0.9403	0.9191	0.3717
	C6-9	0.7300	0.3435	0.9755	0.9691	0.9691	0.9713	0.7395
100k medium	C6-2	1.4569	0.4887	0.8436	0.8052	0.8052	0.7285	0.2101
	C6-3	1.2408	0.3284	0.9511	0.9485	0.9485	0.9302	0.3803
	C6-6	0.9237	0.3250	0.9491	0.9742	0.9742	0.9615	0
10k large	C6-2	1.2629	0.4471	0.9091	0.9091	0.9091	0.9091	0.45
	C6-7	0.7914	0.3336	0.9112	0.9545	0.9545	0.9323	0
50k large	C6-2	1.1954	0.4801	0.803	0.7257	0.7257	0.6332	0.2517
	C6-4	1.0255	0.4235	0.8493	0.8142	0.8142	0.7458	0.2655
100k large	C6-4	1.0395	0.4379	0.863	0.8349	0.8349	0.7831	0.3674
	C8-2	2.3254	0.4954	0.7425	0.5092	0.5092	0.5376	0.1937
	C8-7	1.2485	0.2875	0.9107	0.7569	0.7569	0.8155	0.1925

Comparing the results of the small size category, we see that subset 8 consistently provides the largest MCC in the range of 0.75-0.77 when re-creating the K-means clustering using naïve Bayes. Subset 6 and subset 9 were worse performers. As mentioned earlier, we see that with 100 000 datapoints case, the MCC for Model 6-6 is not consistent

with what is expected as SCP 6-7 had an MCC of 0.55 only when using 10 000 datapoints and all models maintained a similar MCC when adding more datapoints in the small size category groups. This suggests that the poor MCC for Model 6-6 discussed earlier was skewed, likely due to the classification rules established. Overall, we see a small improvement in most metrics on the small dataset when adding more data, while the MCC maintains performance or becomes slightly worse. This suggests that there is no significant gain in performance with more than 9 600 datapoints for the small size category.

Comparing the results of the medium category in Figure 10, only subset 6 was used as it had the best performance in silhouette and elbow plot analysis. There is a drop in all naïve Bayes test metrics compared to the small category. Considering the transition from 10 000 to 100 000 datapoints for subset 6 among the medium categories, we see far higher naïve Bayes metrics in the higher cluster groups than the lower cluster groups, such as model 6-9 (9 clusters) performing far better than model 6-2 (2 clusters) from the 50 000 datapoint/medium group. Except for model 6-9 performance, we see small performance changes when transitioning from the 322 datapoints from the 10 000 datapoint group to the 3 232 datapoints of the 100 000 datapoint group, suggesting that there is no gain in performance when using more datapoints for the medium category.

Comparing the results of the large category in Figure 10, only subsets 6 and 8 were chosen based on K-means analysis with the 10 000 datapoint and 50 000 datapoint groups performing markedly better only with subset 6. Looking at the naïve Bayes metrics for subset 6, we see worse performance from the 10 000 datapoint case to the 100 000 datapoint case, and overall subset 6 had poor model performance in the large size category with a max MCC of 0.45. Subset 8 was only considered when using the 100 000 datapoint group and had dismal performance in all naïve Bayes metrics with a best performing MCC of 0.1937 which is only slightly better than a random classification, and notably, a higher cluster value in model 8-7 did not have a better performance as we observed in the medium category. Overall performance of the large dataset is difficult to confirm since of the 100 000 datapoints considered, only 726 JavaScript files fell in the large group; however, based on the performance trend, more data may not provide a significant improvement.

Considering the impact of adding more data to the K-means and naïve Bayes analysis, there is a mild improvement in some of the classification model performance when adding more data in the case of the small size category. The medium and large size category data exhibited no notable improvement when adding more data, and in the case of the large size category, the performance gets worse when adding more data. This is likely due to the dynamic nature of such large JavaScript files. The larger the file size, the less defined the JavaScript file is by the feature set chosen to examine. This is apparent in the decline of average silhouette scores from the k-means analysis, which show less effective clustering for the larger size files.

The best overall performing models from K-means are in the small size category, across the 50 000 and 100 000-datapoint groups. Model 8-2 had the highest average silhouette scores of all models tested. Model 9-4 had the best distortion to cluster ratio based on the associated elbow plot with a mean distortion less than one. Subset 6 had the best performing silhouette scores or elbow plots for all other groups.

The best models, according to the recreation of the classifier using naïve Bayes, are different based on the size category. For medium and large groups, Subset 6 with almost any clustering model performed better than any other subset albeit with poor overall model performance with no MCC greater than 0.5 except for one. For the small size category, subset 8 had the highest performing model when it came to replicating the classification using naïve Bayes with MCC greater than 0.75 while all other subsets and models performed worse.

The results above suggest that we cannot conclusively classify JavaScript as normal or anomalous in all cases based solely on the features we have examined in this thesis. While K-means classifier and naïve Bayes model verification perform well on the small JavaScript files, it is only using a small subset of features, which perform moderately when reproduced using naïve Bayes. The medium and large JavaScript files cannot be classified using the features we have used. The performance of the models in these categories is inconsistent as the SCPs for subsets 6-9 did not have well defined elbows or silhouettes compared to the small size category. This is likely due to the dynamic nature of JavaScript, which is highly flexible in its implementation and loose in its syntax compared to other

programming languages. This dynamic nature suggests that a more robust feature set would be needed for a static approach to classifying these files or a method to reduce the complexity of these files based on their size.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

Now that all methodology and results have been discussed, this chapter will provide a summary of work done and a discussion of the results. It will also include recommendations for future work.

A. SUMMARY OF WORK

This thesis focused on the process of converting raw, real-world JavaScript into a data frame consisting of word vector frequencies and statistics about the composition of each JavaScript file and then classifying each JavaScript file as normal or anomalous. To classify the JavaScript as normal or anomalous, we made a baseline assumption that 95% of all JavaScript was going to be normal with the remaining 5% considered anomalous. The 5% threshold was chosen as we found no alternative definition of normal JavaScript while researching the topic.

In creating the data frame, 132 features were developed for 100 000 JavaScript files using the Python language regular expressions by converting each JS file into a list of words and extracting the desired features. Several additional statistics, such as entropy, were calculated as an additional measure of what each JS file contained. After compiling all features, the data frame was broken into three size categories to examine each size category separately and examined how well the K-means classifier worked on the three sizes of JavaScript files.

Each size category was considered separately, and ten different subsets of the 132 features were chosen as input data to K-means clustering algorithm to group and classify all JavaScript files. Considering each SCP, the best combinations were determined based on analysis of elbow plots and silhouette plots. From the elbow plots, the best SCPs had a mean distortion of less than 1.0. The silhouette plots showed that the chosen SCPs were not well defined with silhouette scores mostly around 0.4 for the *small* data and 0.3 for the *medium* and *large* data. Ideally, a well-defined clustering will have an average silhouette score greater than 0.8, which was observed in a few SCPs for the *small* data category. The chosen SCPs were then manually converted into a binary classifier based on how many

files were assigned to each cluster. If a cluster contained more than the chosen 5% threshold of the total number of files, the cluster was classified as normal; otherwise, it was classified as anomalous. Additional criteria were added to ensure that there was at least 1% of data classified as anomalous for use with naïve Bayes.

To establish a measured effectiveness of the classification assignments from K-means, the newly classified JavaScript was then examined using naïve Bayes algorithm using the same data that the K-means classification was based upon. Naïve Bayes was used to measure how effective the K-means classifier can be reproduced using a faster supervised algorithm. The results showed that for the *small* category of data, a naïve Bayes model could map the K-means classifier with an MCC of 0.75 at best and less than 0.38 for the *medium* or *large* data category.

Overall, the results from testing all 9 groups showed that file size was the only defining feature in identifying normal and anomalous data regardless of whether it was used as a feature. Other features that were less size dependent, such as in subset 6, were able to identify outliers that were significantly smaller than the average JS file. Using features from subset 6 and subset 8 as well as discovering additional features may provide a more accurate representation of normal JavaScript through static analysis in further research.

B. FUTURE WORK

The analysis of JavaScript is an area of research with many branches as discussed earlier in this thesis. Continuing to establish a baseline definition for normal JavaScript is essential for improving malicious JavaScript detection [2]–[4], and JavaScript optimization [5], [15]. To expand on the research conducted in this thesis, a more in-depth examination of alternative statistics beyond primarily word vector counts is recommended. Alternate forms of scaling may alleviate the impact of file size on the K-means classifier. Alternate forms of clustering may lead to clustering that is less size dependent as well. Examining raw JavaScript to classify the types of operations being conducted or the type of web application the code belongs to can be a step towards creating a conditional definition of normal JavaScript.

Another method for further exploration to expand on the research done here is to apply pattern recognition techniques for JavaScript to establish what a line of JavaScript does based on the pattern or types of code that are commonly written together. This relates to research involving abstract syntax trees in defining the most probable word following a sequence of known words [3], [13]. Given the established method of converting raw JavaScript into a sequence or list of words in this thesis, these findings could provide a foundation for pattern recognition using a large pool of data as we have explored and could establish what normal JavaScript looks like in terms of how a particular line or file is structured.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: DATAFRAME FEATURE LIST

This appendix details the entire set of features used from the DF. A total of 136 features (132 used for analysis, four used for manipulating the data frame) are listed with brief descriptions of their application. The final four features (*event words uses*, *reserved word uses*, *O.P.M. uses* and *async uses*) are summations of specific categories of individual keywords. The individual keywords belonging to each category are given in the form of a list and have been adapted from multiple sources [27]–[29] as they are commonly used in JavaScript. Each keyword is a stand-alone feature whose value is the frequency that the keyword appears in a JS file.

file_name: The name of the file in a JSON format.
file_path: The location of the file in the directory.
site rank: The rank of the associated website on the Alexa top 30 000 webpages.
file_size: The size of the file in bytes.
file_line_count: The estimated number of lines of code the file contains.
average_line_size: The average size in bytes of each line in the file.
total elements: The number of words and blank elements the file contains.
blank elements: The number of empty and removed special characters from the file.
blank element percentage: The number of blank elements divided by the total number of elements.
unique_elements: The number of unique, author defined words in the file.
unique element ratio: The number of unique elements divided by the total number of elements.
unique_word_entropy: The average information for the unique words of the file in bits.
blank element ratio: The ratio of blank elements divided by the number of number of remaining elements.
total_elements2: The total number of words remaining after removing all blank elements.
unique_elements2: The number of unique, author defined words, removing blank element.
unique element ratio2: The ratio of unique elements divided by the total elements, without including any blank elements.
unique_word_entropy2: The average information of the unique words of the file in bits excluding the blanks.
keyword_uses: The number of specified keywords in the file.
keyword percentage: The percentage of keywords used compared to all elements.
keyword percentage2: The percentage of keywords used compared to all words used in the script.
size_cat: The categorical feature identifying the JS files as of *small, medium* and *large.*
event word uses: A category of keyword, that is the sum of JavaScript event words only.

reserve word uses: A category of keyword, that is the sum of JavaScript reserved words only.
o.p.m. uses: A category of keyword, that is the sum of JavaScript objects properties and methods only.
async uses: A category of keyword, that is the sum of JavaScript asynchronous words only.
Event Words: onAbort, onBlur, onChange, onClick, onDblClick, onDragDrop, onError, onFocus, onKeyDown, onKeyPress, onKeyUp, onLoad,onMouseDown, onMouseOverImage, onMouseUp, onMove, onReset, onResize, onSelect, onSubmit, onUnload, addEventListener [29].
Reserved Words: abstract, arguments, boolean, break, byte, case, char, class, const, continue, debugger, default, delete, do, double, enum, export, extends, false, final, finally, float, for, goto, if, implements, import, in, instanceof, int, interface, let, long, native, new, null, package, private, protected, public, short, static, super, synchronized, this, throw, throws, transient, true, try, typeof, void, volatile, while, with, yield, switch, else, return, var [28].
O.P.M. words: Array, Date, eval, function, hasOwnProperty, Infinity, isFinite, isNaN, isPrototypeOf, length, Math, NaN, name, Number, Object, prototype, String, toString, undefined, valueOf, map, constructor [28].
Async words: async, await, setTimeout, setInterval, then, fetch, Promise, catch [30].

APPENDIX B: DATA SUBSET FEATURE SELECTION

In this appendix, we list the features that belong to each data subset in the form of a vector or a Python list containing the names of all features between a pair of brackets. References to specific subsets mean that the data used contained the features in the corresponding subset in this section. References to a SCP A-B are referring to a K-means subset cluster pair with the A value representing one of the data subsets in this appendix. References to a model A-B also have the A value representing one of the data subsets in this appendix. Some subsets refer to a type-1 or type-2 term. A type-1 term refers to the features derived from the list of JavaScript terms including the blanks leftover after filtering out regular expressions. A type-2 term excludes these blanks leftover from the regular expression filtering, and therefore the features derived in this manner are correlated but different from type-1 terms. Several of these datasets list the same keywords discussed in Appendix A [27]–[29].

A. SUBSET 1 – ALL DATA

['file_size', 'file_line_count', 'average_line_size', 'total elements', 'blank elements', 'blank element percentage', 'unique_elements', 'unique element ratio', 'unique_word_entropy', 'blank element ratio', 'total elements2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2', 'keyword_uses', 'keyword percentage', 'keyword percentage2', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses', 'abstract', 'boolean', 'break', 'byte', 'case', 'char', 'class', 'const', 'continue', 'debugger', 'default', 'delete', 'do', 'double', 'enum', 'export', 'extends', 'false', 'final', 'finally', 'float', 'for', 'goto', 'if', 'implements', 'import', 'in', 'instanceof', 'int', 'interface', 'let', 'long', 'native', 'new', 'null', 'package', 'private', 'protected', 'public', 'short', 'static', 'super', 'synchronized', 'this', 'throw', 'throws', 'transient', 'true', 'try', 'typeof', 'void', 'volatile', 'while', 'with', 'yield', 'switch', 'else', 'return', 'var', 'Array', 'Date', 'eval', 'function', 'hasOwnProperty', 'Infinity', 'isFinite', 'isNaN', 'isPrototypeOf', 'length', 'Math', 'NaN', 'name', 'Number', 'Object', 'prototype', 'String', 'toString', 'undefined', 'valueOf', 'map', 'constructor', 'onAbort', 'onBlur', 'onChange', 'onClick', 'onDblClick', 'onDragDrop', 'onError', 'onFocus', 'onKeyDown', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseDown', 'onMouseMove', 'onMouseUp', 'onMove', 'onReset', 'onResize', 'onSelect', 'onSubmit', 'onUnload', 'addEventListener', 'async', 'await', 'setTimeout', 'setInterval', 'then', 'fetch', 'Promise', 'catch'] [27]-[29].

B. SUBSET 2 – CONTAINS ALL FEATURES EXCLUDING FILE_SIZE AND FILE_LINE_COUNT

['total elements', 'blank elements', 'blank element percentage', 'unique_elements', 'unique element ratio', 'unique_word_entropy', 'blank element ratio', 'total elements2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2', 'keyword_uses', 'keyword percentage', 'keyword percentage2', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses', 'abstract', 'boolean', 'break', 'byte', 'case', 'char', 'class', 'const', 'continue', 'debugger', 'default', 'delete', 'do', 'double', 'enum', 'export', 'extends', 'false', 'final', 'finally', 'float', 'for', 'goto', 'if', 'implements', 'import', 'in', 'instanceof', 'int', 'interface', 'let', 'long', 'native', 'new', 'null', 'package', 'private', 'protected', 'public', 'short', 'static', 'super', 'synchronized', 'this', 'throw', 'throws', 'transient', 'true', 'try', 'typeof', 'void', 'volatile', 'while', 'with', 'yield', 'switch', 'else', 'return', 'var', 'Array', 'Date', 'eval', 'function', 'hasOwnProperty', 'Infinity', 'isFinite', 'isNaN', 'isPrototypeOf', 'length', 'Math', 'NaN', 'name', 'Number', 'Object', 'prototype', 'String', 'toString', 'undefined', 'valueOf', 'map', 'constructor', 'onAbort', 'onBlur', 'onChange', 'onClick', 'onDblClick', 'onDragDrop', 'onError', 'onFocus', 'onKeyDown', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseDown', 'onMouseMove', 'onMouseUp', 'onMove', 'onReset', 'onResize', 'onSelect', 'onSubmit', 'onUnload', 'addEventListener', 'async', 'await', 'setTimeout', 'setInterval', 'then', 'fetch', 'Promise', 'catch'] [27]-[29].

C. SUBSET 3 – ALL DATA EXCLUDING TYPE 1 TERMS, VALUES THAT INCLUDE BLANKS

['file_size', 'file_line_count', 'average_line_size', 'blank element ratio', 'total elements2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2', 'keyword_uses', 'keyword percentage2', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses', 'abstract', 'boolean', 'break', 'byte', 'case', 'char', 'class', 'const', 'continue', 'debugger', 'default', 'delete', 'do', 'double', 'enum', 'export', 'extends', 'false', 'final', 'finally', 'float', 'for', 'goto', 'if', 'implements', 'import', 'in', 'instanceof', 'int', 'interface', 'let', 'long', 'native', 'new', 'null', 'package', 'private', 'protected', 'public', 'short', 'static', 'super', 'synchronized', 'this', 'throw', 'throws', 'transient', 'true', 'try', 'typeof', 'void', 'volatile', 'while', 'with', 'yield', 'switch', 'else', 'return', 'var', 'Array', 'Date', 'eval', 'function', 'hasOwnProperty', 'Infinity', 'isFinite', 'isNaN', 'isPrototypeOf', 'length', 'Math', 'NaN', 'name', 'Number', 'Object', 'prototype', 'String', 'toString', 'undefined', 'valueOf', 'map', 'constructor', 'onAbort', 'onBlur', 'onChange', 'onClick', 'onDblClick', 'onDragDrop', 'onError', 'onFocus', 'onKeyDown', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseDown', 'onMouseMove', 'onMouseUp', 'onMove', 'onReset', 'onResize', 'onSelect', 'onSubmit', 'onUnload', 'addEventListener', 'async', 'await', 'setTimeout', 'setInterval', 'then', 'fetch', 'Promise', 'catch'] [27]-[29].

D. SUBSET 4 - ALL DATA EXCLUDING TYPE 2 TERMS, VALUES THAT DON'T INCLUDE BLANKS

['file_size', 'file_line_count', 'average_line_size', 'total elements', 'blank elements', 'blank element percentage', 'unique_elements', 'unique element ratio', 'unique_word_entropy',

'keyword_uses', 'keyword percentage', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses', 'abstract', 'boolean', 'break', 'byte', 'case', 'char', 'class', 'const', 'continue', 'debugger', 'default', 'delete', 'do', 'double', 'enum', 'export', 'extends', 'false', 'final', 'finally', 'float', 'for', 'goto', 'if', 'implements', 'import', 'in', 'instanceof', 'int', 'interface', 'let', 'long', 'native', 'new', 'null', 'package', 'private', 'protected', 'public', 'short', 'static', 'super', 'synchronized', 'this', 'throw', 'throws', 'transient', 'true', 'try', 'typeof', 'void', 'volatile', 'while', 'with', 'yield', 'switch', 'else', 'return', 'var', 'Array', 'Date', 'eval', 'function', 'hasOwnProperty', 'Infinity', 'isFinite', 'isNaN', 'isPrototypeOf', 'length', 'Math', 'NaN', 'name', 'Number', 'Object', 'prototype', 'String', 'toString', 'undefined', 'valueOf', 'map', 'constructor', 'onAbort', 'onBlur', 'onChange', 'onClick', 'onDblClick', 'onDragDrop', 'onError', 'onFocus', 'onKeyDown', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseDown', 'onMouseMove', 'onMouseUp', 'onMove', 'onReset', 'onResize', 'onSelect', 'onSubmit', 'onUnload', 'addEventListener', 'async', 'await', 'setTimeout', 'setInterval', 'then', 'fetch', 'Promise', 'catch'] [27]-[29].

E. SUBSET 5 – CONTAINS THE INDIVIDUAL KEYWORDS AS FEATURES

['abstract', 'boolean', 'break', 'byte', 'case', 'char', 'class', 'const', 'continue', 'debugger', 'default', 'delete', 'do', 'double', 'enum', 'export', 'extends', 'false', 'final', 'finally', 'float', 'for', 'goto', 'if', 'implements', 'import', 'in', 'instanceof', 'int', 'interface', 'let', 'long', 'native', 'new', 'null', 'package', 'private', 'protected', 'public', 'short', 'static', 'super', 'synchronized', 'this', 'throw', 'throws', 'transient', 'true', 'try', 'typeof', 'void', 'volatile', 'while', 'with', 'yield', 'switch', 'else', 'return', 'var', 'Array', 'Date', 'eval', 'function', 'hasOwnProperty', 'Infinity', 'isFinite', 'isNaN', 'isPrototypeOf', 'length', 'Math', 'NaN', 'name', 'Number', 'Object', 'prototype', 'String', 'toString', 'undefined', 'valueOf', 'map', 'constructor', 'onAbort', 'onBlur', 'onChange', 'onClick', 'onDblClick', 'onDragDrop', 'onError', 'onFocus', 'onKeyDown', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseDown', 'onMouseMove', 'onMouseUp', 'onMove', 'onReset', 'onResize', 'onSelect', 'onSubmit', 'onUnload', 'addEventListener', 'async', 'await', 'setTimeout', 'setInterval', 'then', 'fetch', 'Promise', 'catch'] [27]-[29].

F. SUBSET 6 – ONLY CONTAINS LOW CORRELATION, CALCULATED VALUES

['blank element ratio', 'unique element ratio2', 'unique_word_entropy2', 'keyword percentage2'].

G. SUBSET 7 – CONTAINS TOTAL ELEMENTS, UNIQUE ELEMENTS, BLANK ELEMENTS AND TYPE-2 TERMS

['total elements', 'blank elements', 'total elements2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2', 'keyword_uses', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses'].

H. SUBSET 8 –CONTAINS TOTAL ELEMENTS, KEYWORD CATEGORIES, UNIQUE ELEMENTS AND ENTROPY OF TYPE-2 TERMS

['total elements2', 'unique_elements2', 'unique_word_entropy2', 'keyword_uses', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses'].

I. SUBSET 9 – CONTAINS UNIQUE ELEMENTS, ENTROPY, AND KEYWORD CATEGORIES

['unique_elements2', 'unique_word_entropy2', 'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses'].

J. SUBSET 10 – CONTAINS ALL FEATURES WITH LESS THAN 0.5 CROSS CORRELATION

['blank element percentage', 'unique element ratio', 'blank element ratio', 'unique element ratio2', 'keyword percentage2', 'abstract', 'byte', 'class', 'debugger', 'double', 'enum', 'export', 'extends', 'false', 'final', 'float', 'goto', 'int', 'interface', 'let', 'native', 'package', 'private', 'protected', 'public', 'short', 'synchronized', 'throws', 'transient', 'volatile', 'eval', 'map', 'onAbort', 'onDblClick', 'onKeyPress', 'onKeyUp', 'onLoad', 'onMouseMove', 'onMove', 'onReset', 'onResize', 'onSubmit', 'onUnload', 'await'] [27]-[29].

APPENDIX C: PYTHON SCRIPT FOR PREPROCESSING RAW JAVASCRIPT

This appendix contains the code used to create the DF containing all features for 100 000 datapoints. The keywords discussed in Appendix A are also used within this section when building the DF [27]–[29]. The code contains partial comments for easier understanding; file paths and locations are omitted for privacy/security reasons.

```
import shutil, os
import pandas as pd
source_directory = '...'
# chose the range of files
file_start = 1
file_end = 100000
file_count = 1
my_df = pd.DataFrame(columns = ['file_name','file_path','file_size']) # we initialize our
df and specify column names
for x in os.scandir(source_directory):
    if file_count < file_start:
        file_count +=1
    else:
        name = x.name
        path = x.path
        size = x.stat(follow_symlinks=False).st_size
        # This block actually creates the dataFrame iteratively, not efficient currently but
works
        my_df=my_df.append({'file_name' : name, 'file_path' : path, 'file_size' : size},
ignore_index=True)
        file_count +=1
        if file_count>file_end:
            break

# This block will copy the JavaScript from each json file and save it in a directory I have
labeled as outpath, will contain only .js files to be read later
# They are now in a folder that I can modify more easily
import random, shutil, os, json
total_lines = []
# directory to save to
for json_file in my_df['file_path']: # for all files in the dataframe
    with open(json_file) as f:
        text = json.load(f)["src"]
```

```
        temp = text.replace(';',';\n')
        sp_text = temp.splitlines()
        #print(sp_text)
        sp_text = [i for i in sp_text if i]
        l_count = len(sp_text)
        if l_count == 0:
            l_count = 1;
        #print(l_count, ':', sp_text)
        total_lines.append(l_count)
        #print(json_file)
        #print(text)
my_df['file_line_count'] = total_lines

my_df.to_csv(r'...', index=False)

# load the dataframe to work on a specific portion
import pandas as pd
import numpy as np
import random, shutil, os, json
my_df = pd.read_csv('my_dataFrame3.csv')

# This script uses regular expressions to remove almost all special characters as delimeters and add all remaining elements to a list. the elements are keywords, method names parameter names variables, basically all of the words used
# outside of the delimiters and operators used in the code. It allows us to see what names, methods, functions are used without needing to see all the intricacies. The purpose may be for counting specific uses of terms, or entropy calculation
import re,math,collections
unique_words = [] # the total number of unique words in a file, includes blanks left by delimiters
unique_words2 = [] # the total number of unique words in a file, does NOT include blanks left by a file, will be significantly smaller than unique words
total_elements = [] # count the total number of elements (letters, numbers, words, keywords, methods, properties, any non-special characters in the code, blanks included, indicates the number of removed delimiters)
total_elements2 = [] # count the total number of elements (letters, numbers, words, keywords, methods, properties, any non-special characters in the code, blanks NOT included)
unique_word_entropy = [] # calculates the entropy of the file, including the blanks and frequency
unique_word_entropy2 = [] # caclulates the entropy of the file, NOT including the blanks and frequency
for json_file in my_df['file_path']: # for all files in the dataframe
    with open(json_file) as f:
        text = json.load(f)["src"]
```

```
    text2 = re.split('[\@\*\$\~|^\(\)\{\}\.\[\]\n,;\|\s=\"\'\?\!\+\-\/:><&\%\\\\]',text)
    temp_counter = collections.Counter(text2) # temp counter acts as the list of
frequencies for type 1 including blanks
    temp2 = collections.Counter() # temp2 will be for the type 2 counts not using blanks
    temp2 = temp2 + temp_counter
    del temp2['']
    uniq_ele_count = len(temp_counter) # This is the number of unique elements in the
file for type 1 and 2
    uniq_ele_count2 = len(temp2)
    c1 = dict(temp_counter)
    c2 = dict(temp2)
    val1 = list(c1.values())
    val2 = list(c2.values())
    tot1 = sum(val1) # tot 1 and tot2 are the total number of elements for tye 1 and type
2
    tot2 = sum(val2)
    unique_words.append(uniq_ele_count)
    unique_words2.append(uniq_ele_count2)
    total_elements.append(tot1)
    total_elements2.append(tot2)
    #********* Now for entropy
    p_x,p_x2,entropy,entropy2,n = 0,0,0,0,0
    for i,x in enumerate(val1):
      p_x = (val1[i]/tot1)
      if p_x>0:
        n+=1
        entropy += - p_x*math.log(p_x,2)
    unique_word_entropy.append(entropy)
    for i,x in enumerate(val2):
      p_x2 = (val2[i]/tot2)
      if p_x2>0:
        n+=1
        entropy2 += - p_x2*math.log(p_x2,2)
    unique_word_entropy2.append(entropy2)
    #********* End of Entropy Calculations
# Assign our lists to the dataframe
my_df['unique_elements'] = unique_words
my_df['total elements'] = total_elements
my_df['unique_elements2'] = unique_words2
my_df['total elements2'] = total_elements2
my_df['unique_word_entropy'] = unique_word_entropy
my_df['unique_word_entropy2'] = unique_word_entropy2
my_df.to_csv(r'...', index=False)
```

```
# This block is creating a single list of all keywords by concatenating lists of words with
different categories [27]-[29].
eve_hand = ['onAbort','onBlur','onChange','onClick','onDblClick',
          'onDragDrop','onError','onFocus','onKeyDown','onKeyPress',
          'onKeyUp','onLoad','onMouseDown','onMouseMove','onMouseOverImage',

'onMouseUp','onMove','onReset','onResize','onSelect','onSubmit','onUnload','addEventLis
tener']
res_words = ['abstract','arguements','boolean','break','byte','case',
          'char','class','const','continue','debugger','default','delete',
           'do','double','enum','export','extends','false','final',
           'finally','float','for','goto','if','implements','import','in',
          'instanceof','int','interface','let','long','native','new','null','package',
          'private','protected','public','short','static','super','synchronized',
          'this','throw','throws','transient','true','try','typeof','void',
          'volatile','while','with','yield', 'switch', 'else', 'return', 'var'] # switch removed, else
removed, return removed, var removed
obj_prop_meth = ['Array','Date','eval','function','hasOwnProperty','Infinity',
             'isFinite','isNaN','isPrototypeOf','length','Math','NaN','name',

'Number','Object','prototype','String','toString','undefined','valueOf','map','constructor']
async_words = ['async','await','setTimeout','setInterval','then','fetch','Promise','catch'] #
catch removed
# await and catch are res words as well as asyncronous words, eval is a res word as well
as OPM
key_list2 = res_words + obj_prop_meth + eve_hand + async_words

import collections
import re
# This block goes through every file in js_path list, breaks the file into a list of terms,
counts the frequency of each term pulls out the counts for each keyword in
# key_list 2, and creates a frequency table dataFrame consisting of 117 keywords as
columns, for all datapoints
eve_uses = []
res_uses = []
OPM_uses = []
async_uses = []
#other_uses = []
total_key_uses =[] # count the number of keyword uses (includes duplicates) that match
the keyword_list2
df = pd.DataFrame(columns = key_list2)
#df
for json_file in my_df['file_path']: # for all files in the dataframe
    with open(json_file) as f:
        text = json.load(f)["src"]
```

```
        text2 = re.split('[\@\*\$\~|^\(\)\{\}\.\[\]\n,;\|\s=\"\'\?\!\+\-\/:><&\%\\\\]',text)
        temp_counter = collections.Counter(text2)
        temp_dict = dict(temp_counter)
        key_freq = []
        eve_frequency = []
        res_frequency = []
        OPM_frequency = []
        asy_frequency = []
        for word in key_list2:
            eve_freq,res_freq,OPM_freq,asy_freq = 0,0,0,0
            if word in temp_dict:
                #print(word,': key exists, uses is: ', zz[word])
                key_freq.append(temp_dict[word])
                if word in eve_hand:
                    eve_freq = temp_dict[word]
                elif word in res_words:
                    res_freq = temp_dict[word]
                elif word in obj_prop_meth:
                    OPM_freq = temp_dict[word]
                else:
                    asy_freq = temp_dict[word]
            else:
                #print(word, ':key doesnt exist')
                key_freq.append(0)
            eve_frequency.append(eve_freq)
            res_frequency.append(res_freq)
            OPM_frequency.append(OPM_freq)
            asy_frequency.append(asy_freq)
        total_key_word_uses = sum(key_freq) # sum up how many keywords are used total
in the file, passed to dataframe, this includes duplicates
        total_key_uses.append(total_key_word_uses)
        eve_uses.append(sum(eve_frequency))
        res_uses.append(sum(res_frequency))
        OPM_uses.append(sum(OPM_frequency))
        async_uses.append(sum(asy_frequency))
        dftemp = pd.DataFrame([key_freq],columns=key_list2)
        df = pd.concat([df,dftemp], ignore_index=True,axis=0)
my_df['keyword_uses'] = total_key_uses
my_df['event word uses'] = eve_uses
my_df['reserve word uses'] = res_uses
my_df['O.P.M. uses'] = OPM_uses
my_df['async uses'] = async_uses
# df is seperate for now and represents the dataframe of individual keywords
```

```
# Here we are saving both dataframes, my_df contains major metrics while df is the dataframe of just the keywords as features
my_df.to_csv(r'...', index=False)
df.to_csv(r'...', index=False)

# load the dataframe to work on a specific portion
import pandas as pd
import numpy as np
my_df = pd.read_csv('my_dataFrame3.csv')

# This block, is assigning the rank of each datapoint according to its rank on the list of top 30K websites
import pandas as pd
site_rank = [] # create list to add to dataframe
rank_df = pd.read_csv('top30k.txt',header = None) # read our list of ranked sites
rank = list(range(1,30001)) # create our rank list to add to the rank dataframe
rank_df['rank'] = rank
rank_df.columns = ['site','rank'] # establish rank dataframe
for sites in my_df['file_name']: # for all datapoints in working dataframe, we strip off the beginning and end of the site to match the name format in the rank dataframe
    name = sites.split('www.')[1]
    name = name.split('_')[0]
    result = rank_df.loc[rank_df['site'].str.contains(name,case=False),'rank'].iloc[0] # we match the name of the datafile to a name in the ranked list, and assign the rank to the datapoint
    site_rank.append(result) #
my_df['site rank'] = site_rank

# save the dataFrame for access later
my_df.to_csv(r'...', index=False)

# load the dataframe to work on a specific portion
import pandas as pd
import numpy as np
my_df = pd.read_csv('my_dataFrame3.csv')

# This block will take raw counts and convert them into ratios that may be more useful, we are also rounding the decimal places to 5 digits to be more managable
my_df['keyword percentage'] = my_df['keyword_uses']/my_df['total elements'] # percentage of total elements including blanks that are in the keyword list
my_df['keyword percentage2'] = my_df['keyword_uses']/my_df['total elements2'] # percentage of total elements NOT including blanks that are in the keyword list
my_df['average_line_size'] = my_df['file_size']/my_df['file_line_count'] # calculate the average byte size per line
```

```
my_df['blank elements']=my_df['total elements']-my_df['total elements2'] # calculate the total number of blank elements in the file
my_df['unique element ratio']=my_df['unique_elements']/my_df['total elements'] # calculate the ratio of the number of unique elements to the total number of elements blanks included
my_df['unique element ratio2']=my_df['unique_elements2']/my_df['total elements2'] # calculate the ratio of the number of unique elements to the total number of elements, blanks NOT included
my_df['blank element percentage'] = my_df['blank elements']/my_df['total elements'] # calculate the percentage of the file that is blank elements
my_df['blank element ratio'] = my_df['blank elements']/my_df['total elements2'] # calculate the ration of blank elements to non-blank elements in the file
my_df = my_df.round(5)
my_df = my_df.replace([np.inf],0)

# We are rearranging our dataFrame to put all data on the right and info on the left
my_df = my_df[['file_name','file_path','site rank','rank prelim class','file_size', 'file_line_count','average_line_size',
            'total elements','blank elements','blank element percentage','unique_elements','unique element ratio','unique_word_entropy','blank element ratio',
            'total elements2','unique_elements2','unique element ratio2','unique_word_entropy2',
            'keyword_uses','keyword percentage','keyword percentage2','event word uses','reserve word uses','O.P.M. uses','async uses',
            ]]

# Another Save
my_df.to_csv(r'...', index=False)


from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import matplotlib.pylab as plt
# load dataset called 'original'
original = pd.read_csv('my_dataFrame3.csv')
expanded = pd.read_csv('Expanded_DataFrame3.csv')
#**********
# We want to implement a check and remove for any columns that have all zero values
# the my_df != 0 creates a boolean dataframe which is true when value is non-zero
# (my_df !=0).any(axis=0) returns a series indicating which columns have non-zero values, my_df.loc selects those columns, and we can reassign my_df to this value to remove the empty column
original = original.loc[:,(original != 0).any(axis=0)]
expanded = expanded.loc[:,(expanded != 0).any(axis=0)]
```

```
#**********
original.fillna(0,inplace=True) # We use this command to fill in or replace the null or missing values with a zero
expanded.fillna(0,inplace=True)
# we chose zero because the two parameters, unique element ratio2 is derived by the ratio (unique elements 2)/(total elements 2) likewise, keyword percentage 2 is (keyword uses)/(total elements 2)
# the reason we are getting empty/null values is because the denominator is zero in both cases, the 'total elements 2' is 0, because the corresponding file only has special characters in it, ie () or {} and not other info
# so this causes the bad values. in this case we substitute zero for the total value since we cant divide by zero, but these files are very small overall and so any and all values should be almost zero
df = pd.concat([original,expanded],axis = 1) # here we create the base dataset,
df.to_csv(r'...', index=False)

# This block reads the current dataframe and creates the 3 size categories, small, medium and large based on relative size defined by the first 1000 datapoints. using more makes grouping worse due to extreme outliers.
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
import matplotlib.pylab as plt
import pandas as pd
import numpy as np
import seaborn as sns
df_o = pd.read_csv('Combined_DataFrame.csv')
#z = len(df_o)
#kmeans_class = ['NA']*z
#df_o['kmeans class'] = kmeans_class
#**********
df = df_o.iloc[:1000,:]
#**********
work = df.iloc[:,4:] # contains the dataset of only numeric values starting with file_size,
df_copy = df.iloc[:,4:] # The same as work, will be used for a different function later
#s_work = work.apply(zscore, nan_policy ='omit') # Normalizing the work dataset using a zscore,
#print(s_work.columns)on all columns, z = data (value-sample mean/standard deviation) for each column
#**********
# In this block we want to use a k-means to find the ideal split into large and small versions of the dataframe to reduce the impact of size on classification
meanDistortion = []
# we will use the work dataset and specifically columns, filesize, file_line_count and total elements [0,1,3] in the determination, and k = 2 clusters
```

```
df_split = work.iloc[:,[0,3]] # We are only actually using file_size elements here because it has the lowest variance vs using all 3
# df_split.columns
split_model = KMeans(n_clusters=3)
sp = split_model.fit(df_split)
split_prediction=sp.predict(df_split)
meanDistortion.append(sum(np.min(cdist(df_split,split_model.cluster_centers_,'euclidean'),axis=1))/df_split.shape[0])
plt.figure(figsize=(8,8))
# plt.plot(3,meanDistortion,'x-') # plotting cluster means vs variances
# plt.xlabel('n = 3 clusters')
# plt.ylabel('Mean Distortion (Variance)')
# plt.title('Cluster vs Variance Visualization')
df_copy['size_category'] = split_prediction
print(df_copy['size_category'].value_counts())
# The vast majority of files are in the small category, so we will assign whichever is the largest group to a df to a small dataset and the large data to a large dataset
# Since K-means starts with random data, it does not consistently group the same datapoints to the same values on repeated runs, so we need to make sure we always have
# The correct datasets
a = (df_copy[df_copy.size_category==0])
amean = a['file_size'].mean()
b = (df_copy[df_copy.size_category==1])
bmean = b['file_size'].mean()
c = (df_copy[df_copy.size_category==2])
cmean = c['file_size'].mean()
if (amean<bmean)&(amean<cmean): # if category a is larger than category b, we know that a is the dataset of smaller files, and b is the dataset with larger files and we can create the subsets
    small_mean = amean
    if(bmean<cmean):
        mid_mean = bmean
        large_mean = cmean
    else:
        large_mean=bmean
        mid_mean=cmean
elif (bmean<amean)&(bmean<cmean):
    small_mean = bmean
    if(amean>cmean):
        large_mean=amean
        mid_mean = cmean
    else:
        large_mean=cmean
        mid_mean=amean
else: #cmean is the smallest
```

```
    small_mean = cmean
    if(amean>bmean):
        large_mean=amean
        mid_mean=bmean
    else:
        large_mean=bmean
        mid_mean=amean
groups = df_copy.groupby('size_category')
for name,group in groups:
    plt.plot(group['file_size'],group['total elements'],marker="o",linestyle="",label =name)
plt.legend()
plt.xlabel('File Size (bytes)')
plt.ylabel('Key word Uses')
plt.title('Size Category Visualization')
l = []
km_class = []
for index,row in df_o.iterrows():
    temp = df_o['file_size'][index]
    dif1 = abs(temp-small_mean)
    dif2 = abs(temp-mid_mean)
    dif3 = abs(temp-large_mean)
    if (dif1<dif2)&(dif1<dif3):
        l.append('small')
        km_class.append('')
    elif (dif2<dif1)&(dif2<dif3):
        l.append('medium')
        km_class.append('')
    else:
        l.append('large')
        if dif3>(5*large_mean):
            print('abnormal based on size alone')
            km_class.append('1')
        else:
            km_class.append('')
df_o['size_cat'] = l
df_o['KM Class'] = km_class

# This block creates our three tiers of data and saves them for access later
df_10k = df_o.iloc[:10000,:]
df_50k = df_o.iloc[:50000,:]
df_100k = df_o
df_10k.to_csv(r'...', index=False)
df_50k.to_csv(r'...', index=False)
df_100k.to_csv(r'...', index=False)
```

APPENDIX D: PYTHON SCRIPT FOR K-MEANS CLUSTERING ANALYSIS

This appendix contains the code used to apply K-means clustering to the small category data from the 100 000 datapoints. All other groups have near identical code with exceptions due to the manual choice of SCP's based on performance but reflects the process of applying and selecting clusters used in all groups tested, thus they were not included. Within the code are two different blocks of repeating code that were adapted from external sources [32], [33] to create elbow plots and silhouette plots used in the analysis of the K-means clustering performance.

The first block of adapted code is used to create the elbow plot and was adapted from an online Udemy course for ML that is no longer available. The code used specifically calculates the mean distortion for a dataset for cluster values ranging between 1 and 10, and then plots them. This was adapted for the data subsets built for this thesis and repeated ten times to compare the elbow plots of all data subsets, and then four more times to compare the best performing data subsets [32]. The results appear in Figure 7 and Figure 8.

The second block of adapted code was adapted from an online forum. The page specifically covers how to effectively display a silhouette plot as a means of analyzing K-means clustering. The code calculates a silhouette score for all datapoints in the dataset provided, plots them on a horizontal bar plot that visualizes the clustering of K-means, displays the average silhouette score, and places it in a formatted window for easy analysis of the K-means performance [33]. The code was adapted to produce a plot for all cluster values between two and nine, and the code was repeated for the four best performing datasets from the elbow analysis. The results of the code are given in Figure 9.

The code contains partial comments to assist in understanding; file paths and locations are omitted for privacy reasons. The two blocks of adapted code have comments about their use within the code. The ten subset feature lists containing keywords from Appendix B are also in this section and explicitly use those keywords from Appendix A [27]–[29].

```
# Read our dataFrame CSV and implement K-means
import pandas as pd
import numpy as np
#import matplotlib.pylab as plt
%matplotlib inline
import seaborn as sns
from sklearn.cluster import KMeans
from scipy.stats import zscore
from scipy.spatial.distance import cdist
from numpy.linalg import norm
import sklearn.preprocessing
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.pyplot as plt
df_o = pd.read_csv('df_100k.csv')
# split by size category
small_df = df_o[df_o['size_cat']=='small']
medium_df = df_o[df_o['size_cat']=='medium']
large_df = df_o[df_o['size_cat']=='large']
# eliminate empty columns
small_df = small_df.loc[:,(small_df != 0).any(axis=0)]
medium_df = medium_df.loc[:,(medium_df != 0).any(axis=0)]
large_df = large_df.loc[:,(large_df != 0).any(axis=0)]
# replace na with zero due to division by zero in earlier metric calculations
small_df.fillna(0,inplace=True)
medium_df.fillna(0,inplace=True)
large_df.fillna(0,inplace=True)
# create scaled datasets for k-means
small_df_work = small_df.iloc[:,4:-2]
s_work_small = small_df_work.apply(zscore,nan_policy ='omit')
medium_df_work = medium_df.iloc[:,4:-2]
s_work_medium = medium_df_work.apply(zscore,nan_policy ='omit')
large_df_work = large_df.iloc[:,4:-2]
s_work_large = large_df_work.apply(zscore,nan_policy ='omit')

# choose which size category to work with
s_work = s_work_small
#**********
# This block creates 10 subsets of our normalized working dataset called 's_work_X'
where X is the set number, each set has a different selection of chosen datapoints.
#**********
# This block is for the 'no removal of columns case, all data used'
s_work_1 = s_work
# This block removes file size and line count
s_work_2 = s_work.iloc[:,3:]
```

```
# This block will remove all type 1 terms, total elements, blank elements, blank element percentage, keyword percentage,unique elements, unique element ratio, and unique word entropy
s_work_3 = s_work.drop(['total elements','blank elements','blank element percentage', 'keyword percentage','unique_elements','unique element ratio','unique_word_entropy'],axis=1)
# This block will remove all type 2 terms, total elements2, blank elements ratio, keyword percentage2, unique_elements2, unique element ratio2, unique_word_entropy2
#s_work_4 = s_work
#s_work_4.drop(['total elements2', 'blank element ratio', 'keyword percentage2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2'],axis=1)
s_work_4 = s_work.drop(['total elements2', 'blank element ratio', 'keyword percentage2', 'unique_elements2', 'unique element ratio2', 'unique_word_entropy2'],axis=1)
# This block will remove all values except the individual keywords
s_work_5 = s_work.iloc[:,21:]
# This block only contains low correlation calculated terms Keyword percent 2, unique elements ratio 2, unique entropy 2, and blank elements ratio
s_work_6 = s_work.iloc[:,[9,12,13,16]]
# This Block will remove any excess, or overly exlicit values, we will use;, total elements, blank elements, total elements2, unique_elements2, unique_word entropy2, events, reserve,OPM and other
s_work_7 = s_work.iloc[:,[3,4,10,11,12,13,14,17,18,19,20]]
# This block will remove all calculated statistics, keyword percentage 1 and 2, average line size, blank elements, blank element ratio and precentage, and unique element ratio 1 and 2
s_work_8 = s_work.iloc[:,[10,11,13,14,17,18,19,20]]
# this block has  reduced features as set 8
s_work_9 = s_work.iloc[:,[11,13,17,18,19,20]]
# all low correlation features
s_work_10 = s_work[['blank element percentage', 'unique element ratio',
    'blank element ratio', 'unique element ratio2', 'keyword percentage2',
    'abstract', 'byte', 'class', 'debugger', 'double', 'enum', 'export',
    'extends', 'false', 'final', 'float', 'goto', 'int', 'interface', 'let',
    'native', 'package', 'private', 'protected', 'public', 'short',
    'synchronized', 'throws', 'transient', 'volatile', 'eval', 'map',
    'onAbort', 'onDblClick', 'onKeyPress', 'onKeyUp',
    'onLoad', 'onMouseMove', 'onMove', 'onReset', 'onResize', 'onSubmit',
    'onUnload', 'await']]

# In this block we will use KMEANS algorithm to determine a how many cluster centers to consider for grouping. KMEANS is using a Euclidean distance between the datapoints in order to
# Minimize the distance between each datapoint and the cluster centers. We are using a range [1-10) clusters centers for the algoritm and we then plot the Variance or meanDistorition
```

```
# between the datapoints and the cluster centers for the number of cluster centers chosen.
We Are doing this for each dataset chosen in the previous cell, and chosing the model
with the
# lowest variance to analyze more closely and bring into the next phase of Learning.
# we will try to use elbow method to find appropriate K value

#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
clusters =range(1,10)
meanDistortions = []
for k in clusters:
    model=KMeans(n_clusters=k)
    model.fit(s_work_1)
    prediction=model.predict(s_work_1)

meanDistortions.append(sum(np.min(cdist(s_work_1,model.cluster_centers_,'euclidean'),
axis=1))/s_work_1.shape[0])
    # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values,
    # 'average variance between clusters'
plt.figure(figsize=(10,10))
plt.plot(clusters,meanDistortions,'-', label='set1') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method, set 1')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
    model=KMeans(n_clusters=k)
    model.fit(s_work_2)
    prediction=model.predict(s_work_2)

meanDistortions.append(sum(np.min(cdist(s_work_2,model.cluster_centers_,'euclidean'),
axis=1))/s_work_2.shape[0])
    # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'--', label='set2') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method, set 2')
#**********
```

```
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_3)
   prediction=model.predict(s_work_3)

meanDistortions.append(sum(np.min(cdist(s_work_3,model.cluster_centers_,'euclidean'),
axis=1))/s_work_3.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'-.', label='set3') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method, set 3')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_4)
   prediction=model.predict(s_work_4)

meanDistortions.append(sum(np.min(cdist(s_work_4,model.cluster_centers_,'euclidean'),
axis=1))/s_work_4.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,':', label='set4') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method, set 4')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_5)
   prediction=model.predict(s_work_5)
```

```
meanDistortions.append(sum(np.min(cdist(s_work_5,model.cluster_centers_,'euclidean'),
axis=1))/s_work_5.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'-o', label='set5') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method, set 5')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_6)
   prediction=model.predict(s_work_6)

meanDistortions.append(sum(np.min(cdist(s_work_6,model.cluster_centers_,'euclidean'),
axis=1))/s_work_6.shape[0])
plt.plot(clusters,meanDistortions,'-s', label='set6') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method')
# This last command creates a list of the average minimum distances between the cluster
centers and each point, calculating distances using euclidean values, 'average variance
between clusters'
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_7)
   prediction=model.predict(s_work_7)

meanDistortions.append(sum(np.min(cdist(s_work_7,model.cluster_centers_,'euclidean'),
axis=1))/s_work_7.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'--*', label='set7') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
```

```
#plt.title('Selecting k with the elbow method')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_8)
   prediction=model.predict(s_work_8)

meanDistortions.append(sum(np.min(cdist(s_work_8,model.cluster_centers_,'euclidean'),
axis=1))/s_work_8.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'-.D', label='set8') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method')
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_9)
   prediction=model.predict(s_work_9)

meanDistortions.append(sum(np.min(cdist(s_work_9,model.cluster_centers_,'euclidean'),
axis=1))/s_work_9.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,':H', label='set9') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method')
#**********
# All features have a correlation <0.5 with every other feature in set
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_10)
```

```
    prediction=model.predict(s_work_10)

meanDistortions.append(sum(np.min(cdist(s_work_10,model.cluster_centers_,'euclidean'
),axis=1))/s_work_10.shape[0])
    # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'-h', label='set10') # plotting cluster means vs variances
plt.xlabel('K-Clusters')
plt.ylabel('Mean Distortion')
#plt.title('Selecting K Via Elbow Method')
#**********
plt.legend(loc='upper right')
# Best performin  models are 4) set 7, all type 2 and keyword categories, ['total elements',
'blank elements', 'total elements2', 'unique_elements2', 'unique element ratio2',
'unique_word_entropy2', 'keyword_uses','event word uses', 'reserve word uses', 'O.P.M.
uses', 'async uses']
#.......................... 3) set 6, the smallest uncorrelated subset with 4 columns, 'blank
element ratio', 'unique element ratio2', 'unique_word_entropy2', 'keyword percentage2'
#.......................... 2) set 8, reduced version of 7, ['total elements2', 'unique_elements2',
'unique_word_entropy2', 'keyword_uses', 'event word uses', 'reserve word uses', 'O.P.M.
uses', 'async uses']
#.......................... 1) set 9, A small featureset with reduced but not insignificant
correlation,  ['unique_elements2', 'unique_word_entropy2', 'event word uses', 'reserve
word uses', 'O.P.M. uses', 'async uses']


# This code will plot the elbow for the four best perfroming k-means models only
clusters = range(1,10)
plt.figure(figsize=(10,10))
distortion_df =pd.DataFrame()
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
    model=KMeans(n_clusters=k)
    model.fit(s_work_6)
    prediction=model.predict(s_work_6)

meanDistortions.append(sum(np.min(cdist(s_work_6,model.cluster_centers_,'euclidean'),
axis=1))/s_work_6.shape[0])
plt.plot(clusters,meanDistortions,'-s', label='set6') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
```

```
#plt.title('Selecting k with the elbow method')
distortion_df['set 6 distortion']  = meanDistortions
# This last command creates a list of the average minimum distances between the cluster
centers and each point, calculating distances using euclidean values, 'average variance
between clusters'
#**********
print(meanDistortions)
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_7)
   prediction=model.predict(s_work_7)

meanDistortions.append(sum(np.min(cdist(s_work_7,model.cluster_centers_,'euclidean'),
axis=1))/s_work_7.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'--*', label='set7') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method')
distortion_df['set 7 distortion']  = meanDistortions
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_8)
   prediction=model.predict(s_work_8)

meanDistortions.append(sum(np.min(cdist(s_work_8,model.cluster_centers_,'euclidean'),
axis=1))/s_work_8.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,'-.D', label='set8') # plotting cluster means vs variances
plt.xlabel('k')
plt.ylabel('Mean Distortion')
#plt.title('Selecting k with the elbow method')
distortion_df['set 8 distortion']  = meanDistortions
```

```
#**********
# This method was adapted from class Notes taken from Udemy.com from a course in
Machine Learning [32].
meanDistortions = []
for k in clusters:
   model=KMeans(n_clusters=k)
   model.fit(s_work_9)
   prediction=model.predict(s_work_9)

meanDistortions.append(sum(np.min(cdist(s_work_9,model.cluster_centers_,'euclidean'),
axis=1))/s_work_9.shape[0])
   # This last command creates a list of the average minimum distances between the
cluster centers and each point, calculating distances using euclidean values, 'average
variance between clusters'
plt.plot(clusters,meanDistortions,':H', label='set9') # plotting cluster means vs variances
plt.xlabel('K -Clusters')
plt.ylabel('Mean Distortion')
plt.legend(loc='upper right')
#plt.title('Selecting k with the elbow method')
distortion_df['set 9 distortion']  = meanDistortions
# End of adapted code from class notes taken from Udemy.com.


# This block looks at the correlation of the 4 best subsets
plt.figure(figsize=(8,8))
sns.heatmap(s_work_6.corr(), vmax=1,square=True,annot=True,cmap='viridis')
plt.title('Correlation Between Features of Normalized Subset 6')
plt.show()
plt.figure(figsize=(8,8))
sns.heatmap(s_work_7.corr(), vmax=1,square=True,annot=True,cmap='viridis')
plt.title('Correlation Between Features of Normalized Subset 7')
plt.show()
plt.figure(figsize=(8,8))
sns.heatmap(s_work_8.corr(), vmax=1,square=True,annot=True,cmap='viridis')
plt.title('Correlation Between Features of Normalized Subset 8')
plt.show()
plt.figure(figsize=(8,8))
sns.heatmap(s_work_9.corr(), vmax=1,square=True,annot=True,cmap='viridis')
plt.title('Correlation Between Features of Normalized Subset 9')
plt.show()

# In this block we will look at Silhouette plots for the best sets and clusters noted above
   # Silhouette Analysis.
   # Thickness of each par is relative to the size of the cluster in terms of number of
datapoints
```

```
    # for each sample the average distance from all other datapoints in the cluster is
computed as ai
    # the average distance from the datapoint to all datapoints in nearest neighbor cluster is
calculated as bi
    # a coefficient is calculated as (bi - ai)/max(ai,bi)
    # so,
    # The closer a sample is to 1, the further away from neighboring clusters, good
    # If a sample is around 0, it is very close to neighboring clusters, these are fringe
datapoints
    # If a sample is near -1 it is assigned to the wrong cluster, bad
# The following code was adapted from Mukesh Chaudhary's post on medium.com
regarding the creation of silhouette plots [33].
X_std = sklearn.preprocessing.StandardScaler().fit_transform(s_work_6)
fig = plt.figure(figsize=(8,56))
for i,k in enumerate([2,3,4,5,6,7,8,9]):
    ax1=fig.add_subplot(8,1,i+1)
    km = KMeans(n_clusters = k)
    labels = km.fit_predict(X_std)
    centroids = km.cluster_centers_
    silhouette_vals=silhouette_samples(X_std, labels)
    y_ticks = []
    y_lower, y_upper = 0,0
    for i,cluster in enumerate(np.unique(labels)):
        cluster_silhouette_vals = silhouette_vals[labels == cluster]
        cluster_silhouette_vals.sort()
        y_upper += len(cluster_silhouette_vals)
        ax1.barh(range(y_lower,y_upper), cluster_silhouette_vals, edgecolor = 'none', height
= 1)
        ax1.text(-0.03, (y_lower + y_upper)/ 2, str(i + 1))
        y_lower += len(cluster_silhouette_vals)
    avg_score = np.mean(silhouette_vals).round(4)
    ax1.axvline(avg_score,linestyle='--', linewidth = 2,color='green')
    ax1.set_yticks([])
    ax1.set_xlim([-0.1, 1])
    ax1.set_xlabel('Silhouette Coefficient Values')
    ax1.set_ylabel('Cluster Labels')
    ax1.set_title('Silhouette Plot for ' + str(i+1) + ' Clusters and Subset 6', y=1.02)
    #ax1.text(avg_score-0.05,(i/2),'AvSS: ' + str(avg_score), rotation='vertical')
    ax1.text(1.02,(i),'AvSS: ' + str(avg_score), rotation= -90)
# The following code was adapted from Mukesh Chaudhary's post on medium.com
regarding the creation of silhouette plots [33].
X_std = sklearn.preprocessing.StandardScaler().fit_transform(s_work_7)
fig = plt.figure(figsize=(8,56))
for i,k in enumerate([2,3,4,5,6,7,8,9]):
    ax1=fig.add_subplot(8,1,i+1)
```

```
    km = KMeans(n_clusters = k)
    labels = km.fit_predict(X_std)
    centroids = km.cluster_centers_
    silhouette_vals=silhouette_samples(X_std, labels)
    y_ticks = []
    y_lower, y_upper = 0,0
    for i,cluster in enumerate(np.unique(labels)):
        cluster_silhouette_vals = silhouette_vals[labels == cluster]
        cluster_silhouette_vals.sort()
        y_upper += len(cluster_silhouette_vals)
        ax1.barh(range(y_lower,y_upper), cluster_silhouette_vals, edgecolor = 'none', height
= 1)
        ax1.text(-0.03, (y_lower + y_upper)/ 2, str(i + 1))
        y_lower += len(cluster_silhouette_vals)
    avg_score = np.mean(silhouette_vals).round(4)
    ax1.axvline(avg_score,linestyle='--', linewidth = 2,color='green')
    ax1.set_yticks([])
    ax1.set_xlim([-0.1, 1])
    ax1.set_xlabel('Silhouette Coefficient Values')
    ax1.set_ylabel('Cluster Labels')
    ax1.set_title('Silhouette Plot for ' + str(i+1) + ' Clusters and Subset 7', y=1.02)
    #ax1.text(avg_score-0.05,(i/2),'AvSS: ' + str(avg_score), rotation='vertical')
    ax1.text(1.02,(i),'AvSS: ' + str(avg_score), rotation= -90)
# The following code was adapted from Mukesh Chaudhary's post on medium.com
regarding the creation of silhouette plots  [33].
X_std = sklearn.preprocessing.StandardScaler().fit_transform(s_work_8)
fig = plt.figure(figsize=(8,56))
for i,k in enumerate([2,3,4,5,6,7,8,9]):
    ax1=fig.add_subplot(8,1,i+1)
    km = KMeans(n_clusters = k)
    labels = km.fit_predict(X_std)
    centroids = km.cluster_centers_
    silhouette_vals=silhouette_samples(X_std, labels)
    y_ticks = []
    y_lower, y_upper = 0,0
    for i,cluster in enumerate(np.unique(labels)):
        cluster_silhouette_vals = silhouette_vals[labels == cluster]
        cluster_silhouette_vals.sort()
        y_upper += len(cluster_silhouette_vals)
        ax1.barh(range(y_lower,y_upper), cluster_silhouette_vals, edgecolor = 'none', height
= 1)
        ax1.text(-0.03, (y_lower + y_upper)/ 2, str(i + 1))
        y_lower += len(cluster_silhouette_vals)
    avg_score = np.mean(silhouette_vals).round(4)
    ax1.axvline(avg_score,linestyle='--', linewidth = 2,color='green')
```

```
    ax1.set_yticks([])
    ax1.set_xlim([-0.1, 1])
    ax1.set_xlabel('Silhouette Coefficient Values')
    ax1.set_ylabel('Cluster Labels')
    ax1.set_title('Silhouette Plot for ' + str(i+1) + ' Clusters and Subset 8', y=1.02)
    #ax1.text(avg_score-0.05,(i/2),'AvSS: ' + str(avg_score), rotation='vertical')
    ax1.text(1.02,(i),'AvSS: ' + str(avg_score), rotation= -90)
# The following code was adapted from Mukesh Chaudhary's post on medium.com
regarding the creation of silhouette plots [33].
X_std = sklearn.preprocessing.StandardScaler().fit_transform(s_work_9)
fig = plt.figure(figsize=(8,56))
for i,k in enumerate([2,3,4,5,6,7,8,9]):
    ax1=fig.add_subplot(8,1,i+1)
    km = KMeans(n_clusters = k)
    labels = km.fit_predict(X_std)
    centroids = km.cluster_centers_
    silhouette_vals=silhouette_samples(X_std, labels)
    y_ticks = []
    y_lower, y_upper = 0,0
    for i,cluster in enumerate(np.unique(labels)):
        cluster_silhouette_vals = silhouette_vals[labels == cluster]
        cluster_silhouette_vals.sort()
        y_upper += len(cluster_silhouette_vals)
        ax1.barh(range(y_lower,y_upper), cluster_silhouette_vals, edgecolor = 'none', height
= 1)
        ax1.text(-0.03, (y_lower + y_upper)/ 2, str(i + 1))
        y_lower += len(cluster_silhouette_vals)
    avg_score = np.mean(silhouette_vals).round(4)
    ax1.axvline(avg_score,linestyle='--', linewidth = 2,color='green')
    ax1.set_yticks([])
    ax1.set_xlim([-0.1, 1])
    ax1.set_xlabel('Silhouette Coefficient Values')
    ax1.set_ylabel('Cluster Labels')
    ax1.set_title('Silhouette Plot for ' + str(i+1) + ' Clusters and Subset 9', y=1.02)
    #ax1.text(avg_score-0.05,(i/2),'AvSS: ' + str(avg_score), rotation='vertical')
    ax1.text(1.02,(i),'AvSS: ' + str(avg_score), rotation= -90)

# End of adapted code from Mukesh Chaudhary's post on medium.com.

# In this block we will create all clusters for the various datasets to examine further and
apply them to the work dataset
# For the naming scheme, the first number is the data_subset, the second number is how
many clusters
model2 = KMeans(2)
model3 = KMeans(3)
```

```
model4 = KMeans(4)
model5 = KMeans(5)
model6 = KMeans(6)
model7 = KMeans(7)
model8 = KMeans(8)
#********** Set6 Clusters
m6_6 = model6.fit(s_work_6)
p6_6 = m6_6.predict(s_work_6)
small_df['C6-6'] = p6_6
#********** Set8 Clusters
m8_2 = model2.fit(s_work_8)
p8_2 = m8_2.predict(s_work_8)
small_df['C8-2'] = p8_2
m8_8 = model8.fit(s_work_8)
p8_8 = m8_8.predict(s_work_8)
small_df['C8-8'] = p8_8
#********** Set9 Clusters
m9_4 = model4.fit(s_work_9)
p9_4 = m9_4.predict(s_work_9)
small_df['C9-4'] = p9_4
# Here we can visualize the groupings of set 7, the lowest variance set, and the
distribution of how many datapoints fit in all of the classifications
columns = list(['C6-6','C8-2','C8-8','C9-4'])
fig = plt.figure(figsize=(10,10))
for i,col in enumerate(small_df[columns]):
   ax=fig.add_subplot(2,2,i+1) # create a subplot
   small_df[col].hist(bins=8,ax=ax,align='mid') # create histogram with the column data
   ax.set_title(col + " Distribution") # create title
   ax.set_xlabel('Cluster Labels')
   ax.set_ylabel('Cluster Size')
   #large_df[i].hist(bins=10,figsize=(10,5), align='mid', label = ['test', 'test'])
plt.suptitle(' Distribution of Data into Clusters for Chosen Subset/Cluster Pairs')
# We can save the chosen KMEANS Classifier Columns to our dataframe, or a new
dataframe to bring to later analysis, note we are adding it to a copy of the df unscaled
dataset because
# we only wanted the datascaled to determine cluster centers
#small_df.to_csv(r'...', index=False)


# In this block we can visualize the distribution of data relative to size, and total elements
to see how size has effected grouping
plt.figure(figsize=(6,24))
groups = small_df.groupby('C6-6')
for name,group in groups:
   plt.subplot(411)
```

```
    plt.plot(group['file_size'],group['total elements'],marker="o",linestyle="",label =name)
plt.legend()
plt.xlabel('File Size (bytes)')
plt.ylabel('Key word Uses')
plt.title('Size Category Visualization Set 6 w/ 7 Clusters')
groups = small_df.groupby('C8-2')
for name,group in groups:
    plt.subplot(412)
    plt.plot(group['file_size'],group['total elements'],marker="o",linestyle="",label =name)
plt.legend()
plt.xlabel('File Size (bytes)')
plt.ylabel('Key word Uses')
plt.title('Size Category Visualization Set 8 w/ 2 Clusters')
groups = small_df.groupby('C8-8')
for name,group in groups:
    plt.subplot(413)
    plt.plot(group['file_size'],group['total elements'],marker="o",linestyle="",label =name)
plt.legend()
plt.xlabel('File Size (bytes)')
plt.ylabel('Key word Uses')
plt.title('Size Category Visualization Set 8 w/ 8 Clusters')
groups = small_df.groupby('C9-4')
for name,group in groups:
    plt.subplot(414)
    plt.plot(group['file_size'],group['total elements'],marker="o",linestyle="",label =name)
plt.legend()
plt.xlabel('File Size (bytes)')
plt.ylabel('Key word Uses')
plt.title('Size Category Visualization Set 9 w/ 4 Clusters')

# now we save the dataframe with the chosen clusters assigned to data as columns for
each SCP chosen
small_df.to_csv(r'...', index=False)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: PYTHON SCRIPT CLASSIFICATION AND NAÏVE BAYES MODELING

This appendix contains the code used to translate the cluster selections from K-means into a binary classification for the small category data from 100 000 datapoints using the process discussed in Chapter IV. It also contains the code for applying Naïve Bayes to include splitting the data, training the model, and calculating the metrics to compare the performance of all chosen SCPs for the nine groups of data. The code contains partial comments to assist in understanding, and file paths and locations are omitted for privacy/security reasons.

```
#**********
# Classification and Naive Bayes Application
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
import math
from sklearn.naive_bayes import MultinomialNB # primarily used for text classifcation
where data represented as word vector counts
from sklearn.metrics import precision_score, recall_score, f1_score,accuracy_score
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import matthews_corrcoef
from tabulate import tabulate
import time
# read data from k-means selection
df = pd.read_csv('small_df_100k_clusters.csv')
df = df.drop(['rank prelim class'], axis=1)
#df['KM Class'].sum()
df['KM Class'].replace(0.0,'', inplace=True)
fs_average = df['file_size'].mean()

# identify the cluster distribution of the clusters, for maual classification, identify
threshold
a = df['C6-6'].value_counts()
b = df['C8-2'].value_counts()
```

```
c = df['C8-8'].value_counts()
d = df['C9-4'].value_counts()
print(a)
print(b)
print(c)
print(d)
l = (df.shape[0]) # total number of datapoints
threshold = math.ceil(0.05*l) # set a five percent threshold rounded up to nearest int for
any clusters greater than 2
print('length of data: ', l)
print('threshold value for classification: ',threshold)

# Manually assign classification based on cluster counts and established cluster rules
temp_class = []
for classification in df['C6-6']:
    #print(classification)
    if (classification==3)|(classification==1)|(classification==2)|(classification==0):
         temp_class.append(0) # normal is zero
    else:
        temp_class.append(1) # unknown is one
df['Class 6-6'] = temp_class
for i in range(0,df.shape[0]):
    if df['KM Class'][i] == 1:
        #print(df['KM Class'][i])
        print(i, ': ', 'original class: ',temp_class[i], ' reclassified due to size as: ', '1', ' The file
size is: ', df['file_size'][i], ' vs an average of: ', fs_average)
        temp_class[i]=1
        #print(i, ': ', temp_class[i])
    #else:
        #print(i, ': ', temp_class[i])
df['Class 6-6'] = temp_class
temp_class = []
for classification in df['C8-2']:
    #print(classification)
    if (classification == 0):
         temp_class.append(0) # normal is zero
    else:
        temp_class.append(1) # unknown is one
df['Class 8-2'] = temp_class
for i in range(0,df.shape[0]):
    if df['KM Class'][i] == 1:
        #print(df['KM Class'][i])
        print(i, ': ', 'original class: ',temp_class[i], ' reclassified due to size as: ', '1', ' The file
size is: ', df['file_size'][i], ' vs an average of: ', fs_average)
        temp_class[i]=1
```

```
            #print(i, ': ', temp_class[i])
        #else:
            #print(i, ': ', temp_class[i])
df['Class 8-2'] = temp_class
temp_class = []
for classification in df['C8-8']:
    #print(classification)
    if (classification == 1)|(classification == 4)|(classification == 7)|(classification == 5):
         temp_class.append(0) # normal is zero
    else:
        temp_class.append(1) # unknown is one
df['Class 8-8'] = temp_class
for i in range(0,df.shape[0]):
    if df['KM Class'][i] == 1:
        #print(df['KM Class'][i])
        print(i, ': ', 'original class: ',temp_class[i], ' reclassified due to size as: ', '1', ' The file
size is: ', df['file_size'][i], ' vs an average of: ', fs_average)
        temp_class[i]=1
        #print(i, ': ', temp_class[i])
    #else:
        #print(i, ': ', temp_class[i])
df['Class 8-8'] = temp_class
temp_class = []
for classification in df['C9-4']:
    #print(classification)
    if (classification == 0)|(classification == 2)|(classification == 3):
         temp_class.append(0) # normal is zero
    else:
        temp_class.append(1) # unknown is one
df['Class 9-4'] = temp_class
for i in range(0,df.shape[0]):
    if df['KM Class'][i] == 1:
        #print(df['KM Class'][i])
        print(i, ': ', 'original class: ',temp_class[i], ' reclassified due to size as: ', '1', ' The file
size is: ', df['file_size'][i], ' vs an average of: ', fs_average)
        temp_class[i]=1
        #print(i, ': ', temp_class[i])
    #else:
        #print(i, ': ', temp_class[i])
df['Class 9-4'] = temp_class


# Here we want to create our different X and Y variables for using NB, X is the featureset
used in clustering, Y is the newly assigned classification
```

```
X6 = df[['blank element ratio', 'unique element ratio2', 'unique_word_entropy2', 'keyword
percentage2']] # X6 uses the subset 6 data from K-means
X8 = df[['total elements2', 'unique_elements2', 'unique_word_entropy2', 'keyword_uses',
'event word uses', 'reserve word uses', 'O.P.M. uses', 'async uses']]
X9 = df[['unique_elements2', 'unique_word_entropy2', 'event word uses', 'reserve word
uses', 'O.P.M. uses', 'async uses']]
Y6_6 = df['Class 6-6']
Y8_2 = df['Class 8-2']
Y8_8 = df['Class 8-8']
Y9_4 = df['Class 9-4']

# Create and display train test split
x_train,x_test,y_train,y_test = train_test_split(X6,Y6_6,test_size=0.3,random_state=1)
tt_split_table = [['Counts', 'Train Split','Test Split'],
                ['0',y_train.value_counts()[0],y_test.value_counts()[0]],
                ['1',y_train.value_counts()[1],y_test.value_counts()[1]]] # manually add zero,
otherwise error
print(tabulate(tt_split_table, headers = 'firstrow',tablefmt='pretty'))
# create, train, and apply NB model
NB_model = MultinomialNB()
NB_model.fit(x_train,y_train)
train_pred = NB_model.predict(x_train)
test_pred = NB_model.predict(x_test)
# show confusion matrix of training data
cm = metrics.confusion_matrix(y_train,train_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.figure(figsize=(15,6))
#plt.suptitle('Comparing Training and Testing Splits for Classification 6-6 \n Comparison
Plot, AvSS = 0.36')
plt.subplot(121)
plt.title('CM Training Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
# show confusion matrix of test data
cm = metrics.confusion_matrix(y_test,test_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.subplot(122)
plt.title('CM Test Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
# output the metrics for train/test
table = [['Dataset', ' Precision','Recall','Accuracy','F1 Score', 'MCC'],
```

```
    ['Training Set',precision_score(y_train,train_pred,average = 'weighted',
zero_division=0).round(4),
     recall_score(y_train,train_pred,average =
'weighted').round(4),accuracy_score(y_train,train_pred).round(4),
     f1_score(y_train,train_pred,average = 'weighted').round(4),
round(matthews_corrcoef(y_train,train_pred),4)],
    ['Testing Set',precision_score(y_test,test_pred,average =
'weighted',zero_division=0).round(4),
     recall_score(y_test,test_pred,average =
'weighted').round(4),accuracy_score(y_test,test_pred).round(4),
     f1_score(y_test,test_pred,average = 'weighted').round(4),
round(matthews_corrcoef(y_test,test_pred),4)]]
print(tabulate(table, headers = 'firstrow',tablefmt='pretty'))

#repeat train/test/model/metrics for next SCP
x_train,x_test,y_train,y_test = train_test_split(X8,Y8_2,test_size=0.3,random_state=1)
tt_split_table = [['Counts', 'Train Split','Test Split'],
          ['0',y_train.value_counts()[0],y_test.value_counts()[0]],
          ['1',y_train.value_counts()[1],y_test.value_counts()[1]]]
print(tabulate(tt_split_table, headers = 'firstrow',tablefmt='pretty'))
NB_model = MultinomialNB()
NB_model.fit(x_train,y_train)
train_pred = NB_model.predict(x_train)
test_pred = NB_model.predict(x_test)
cm = metrics.confusion_matrix(y_train,train_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.figure(figsize=(15,6))
#plt.suptitle('Comparing Training and Testing Splits for Classification 8-2 \n Best
Silhouette, AvSS = 0.812')
plt.subplot(121)
plt.title('CM Training Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
cm = metrics.confusion_matrix(y_test,test_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.subplot(122)
plt.title('CM Test Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
table = [['Dataset', ' Precision','Recall','Accuracy','F1 Score', 'MCC'],
    ['Training Set',precision_score(y_train,train_pred,average = 'weighted',
zero_division=0).round(4),recall_score(y_train,train_pred,average =
```

```
'weighted').round(4),accuracy_score(y_train,train_pred).round(4),f1_score(y_train,train_
pred,average = 'weighted').round(4), round(matthews_corrcoef(y_train,train_pred),4)],
        ['Testing Set',precision_score(y_test,test_pred,average =
'weighted',zero_division=0).round(4),recall_score(y_test,test_pred,average =
'weighted').round(4),accuracy_score(y_test,test_pred).round(4),f1_score(y_test,test_pred,
average = 'weighted').round(4), round(matthews_corrcoef(y_test,test_pred),4)]]
print(tabulate(table, headers = 'firstrow',tablefmt='pretty'))

#repeat train/test/model/metrics for next SCP
x_train,x_test,y_train,y_test = train_test_split(X8,Y8_8,test_size=0.3,random_state=1)
#print('training counts:\n',y_train.value_counts())
#print('testing counts:\n', y_test.value_counts())
tt_split_table = [['Counts', 'Train Split','Test Split'],
                ['0',y_train.value_counts()[0],y_test.value_counts()[0]],
                ['1',y_train.value_counts()[1],y_test.value_counts()[1]]]
print(tabulate(tt_split_table, headers = 'firstrow',tablefmt='pretty'))
NB_model = MultinomialNB()
NB_model.fit(x_train,y_train)
train_pred = NB_model.predict(x_train)
test_pred = NB_model.predict(x_test)
cm = metrics.confusion_matrix(y_train,train_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.figure(figsize=(15,6))
#plt.suptitle('Comparing Training and Testing Splits for Classification 8-8 \n Comparison
Plot, AvSS = 0.516')
plt.subplot(121)
plt.title('CM Training Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
cm = metrics.confusion_matrix(y_test,test_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.subplot(122)
plt.title('CM Test Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
table = [['Dataset', ' Precision','Recall','Accuracy','F1 Score', 'MCC'],
        ['Training Set',precision_score(y_train,train_pred,average = 'weighted',
zero_division=0).round(4),recall_score(y_train,train_pred,average =
'weighted').round(4),accuracy_score(y_train,train_pred).round(4),f1_score(y_train,train_
pred,average = 'weighted').round(4), round(matthews_corrcoef(y_train,train_pred),4)],
        ['Testing Set',precision_score(y_test,test_pred,average =
'weighted',zero_division=0).round(4),recall_score(y_test,test_pred,average =
```

```
'weighted').round(4),accuracy_score(y_test,test_pred).round(4),f1_score(y_test,test_pred,
average = 'weighted').round(4), round(matthews_corrcoef(y_test,test_pred),4)]]
print(tabulate(table, headers = 'firstrow',tablefmt='pretty'))

#repeat train/test/model/metrics for next SCP
x_train,x_test,y_train,y_test = train_test_split(X9,Y9_4,test_size=0.3,random_state=1)
#print('training counts:\n',y_train.value_counts())
#print('testing counts:\n', y_test.value_counts())
tt_split_table = [['Counts', 'Train Split','Test Split'],
        ['0',y_train.value_counts()[0],y_test.value_counts()[0]],
        ['1',y_train.value_counts()[1],y_test.value_counts()[1]]]
print(tabulate(tt_split_table, headers = 'firstrow',tablefmt='pretty'))
NB_model = MultinomialNB()
NB_model.fit(x_train,y_train)
train_pred = NB_model.predict(x_train)
test_pred = NB_model.predict(x_test)
cm = metrics.confusion_matrix(y_train,train_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.figure(figsize=(15,6))
#plt.suptitle('Comparing Training and Testing Splits for Classification 9-4 \n Ideal Elbow
Plot, AvSS = 0.47')
plt.subplot(121)
plt.title('CM Training Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
cm = metrics.confusion_matrix(y_test,test_pred,labels=[0,1])
NB_cm = pd.DataFrame(cm,index = [i for i in [0,1]],columns = [i for i in [0,1]])
plt.subplot(122)
plt.title('CM Test Data')
sns.heatmap(NB_cm,annot=True, fmt = 'd')
plt.xlabel('Model Predicted Class (NB Prediction)')
plt.ylabel('K-Means Classification (True)')
table = [['Dataset', ' Precision','Recall','Accuracy','F1 Score', 'MCC'],
    ['Training Set',precision_score(y_train,train_pred,average = 'weighted',
zero_division=0).round(4),recall_score(y_train,train_pred,average =
'weighted').round(4),accuracy_score(y_train,train_pred).round(4),f1_score(y_train,train_
pred,average = 'weighted').round(4), round(matthews_corrcoef(y_train,train_pred),4)],
    ['Testing Set',precision_score(y_test,test_pred,average =
'weighted',zero_division=0).round(4),recall_score(y_test,test_pred,average =
'weighted').round(4),accuracy_score(y_test,test_pred).round(4),f1_score(y_test,test_pred,
average = 'weighted').round(4), round(matthews_corrcoef(y_test,test_pred),4)]]
print(tabulate(table, headers = 'firstrow',tablefmt='pretty'))
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, "Challenging machine learning algorithms in predicting vulnerable JavaScript functions," in *2019 IEEE/ACM 7th Int. Work. on Re. Art. Int. Syn. in Soft. Eng.*, 2019, pp. 8–14 [Online]. Available: doi: https://doi.org/10.1109/RAISE.2019.00010.

[2] S. Ndichu, S. Ozawa, T. Misu, and K. Okada, "A machine learning approach to malicious JavaScript detection using fixed length vector representation," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8 [Online]. Available: doi: https://doi.org/10.1109/IJCNN.2018.8489414.

[3] H. Liang, Y. Yang, L. Sun, and L. Jiang, "JSAC: A novel framework to detect malicious JavaScript via CNNs over AST and CFG," in *2019 Int. Joi. Conf. on Neu. Net.*, 2019, pp. 1–8 [Online]. Available: doi: 10.1109/IJCNN.2019.8851760.

[4] M. F. Rozi, S. Kim, and S. Ozawa, "Deep neural networks for malicious JavaScript detection using bytecode sequences," in *2020 Int. Joi. Conf. on Neu. Net.*, 2020, pp. 1–8 [Online]. Available: doi: 10.1109/IJCNN48605.2020.9207134.

[5] M. Chaqfeh *et al.*, "To block or not to block: Accelerating mobile web pages on-the-fly through JavaScript classification," *ArXiv210613764 CsOH*, Jun. 2021 [Online]. Available: http://arxiv.org/abs/2106.13764

[6] S. Wei and B. G. Ryder, "State-sensitive points-to analysis for the dynamic behavior of JavaScript objects," in *ECOOP 2014 – Object-Oriented Programming*, Berlin, Heidelberg, 2014, pp. 1–26 [Online]. Available: doi: 10.1007/978-3-662-44202-9_1.

[7] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," *ACM SIGPLAN Not.*, vol. 45, no. 6, pp. 1–12, Jun. 2010 [Online]. Available: http://doi.org/10.1145/1809028.1806598

[8] S. Wei and B. G. Ryder, "Taming the dynamic behavior of JavaScript," in *Proceedings of the Com. Pub. of the 2014 ACM SIGPLAN Conf on Sys., Prog., and Apps.: Soft. for Hum.*, 2014, pp. 61–62 [Online]. Available: doi: 10.1145/2660252.2660393.

[9] H. Wu and S. Qin, "Detecting obfuscated suspicious JavaScript based on collaborative training," in *2017 IEEE 17th Int. Conf. on Comm. Tech.*, 2017, pp. 1962–1966 [Online]. Available: doi: 10.1109/ICCT.2017.8359972.

[10] S. Ndichu, S. Kim, and S. Ozawa, “Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement,” *CAAI Trans. Intell. Technol.*, vol. 5, no. 3, pp. 184–192, Jul. 2020 [Online]. Available: doi: 10.1049/trit.2020.0026.

[11] W. Song, Q. Huang, and J. Huang, “Understanding JavaScript vulnerabilities in large real-world Android applications,” *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 5, pp. 1063–1078, Sep. 2020 [Online]. Available: doi: 10.1109/TDSC.2018.2845851.

[12] J. Mao *et al.*, “Detecting malicious behaviors in JavaScript applications,” *IEEE Access*, vol. 6, pp. 12284–12294, Jan. 2018 [Online]. Available: doi: 10.1109/ACCESS.2018.2795383.

[13] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, p. 40:1-40:29, Jan. 2019 [Online]. Available: doi: 10.1145/3290353.

[14] X. He, L. Xu, and C. Cha, “Malicious JavaScript code detection based on hybrid analysis,” in *2018 25th Asia-Pac. Soft. Eng. Conf.*, 2018, pp. 365–374 [Online]. Available: doi: 10.1109/APSEC.2018.00051.

[15] T. Kupoluyi, M. Chaqfeh, M. Varvello, W. Hashmi, L. Subramanian, and Y. Zaki, “Muzeel: A dynamic JavaScript analyzer for dead code elimination in today’s web,” *ArXiv210608948 CsSE*, pp. 1–14, Jun. 2021 [Online]. Available: http://arxiv.org/abs/2106.08948

[16] Scikit-Learn, “2.3. Clustering,” *scikit-learn*. [Online]. Available: https://scikit-learn/stable/modules/clustering.html

[17] Scikit-Learn, “sklearn.cluster.KMeans — scikit-learn 1.0.1 documentation.” [Online]. Available: https://scikit-learn.org/stable/modules/generated /sklearn.cluster.KMeans.html

[18] J. Delua, “Supervised vs. unsupervised learning: What’s the difference?,” IBM, Mar. 12, 2021 [Online]. Available: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

[19] A. Gupta, “Elbow Method for optimal value of k in KMeans,” *GeeksforGeeks,blog*, Jun. 06, 2019 [Online]. Available: https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/

[20] Scikit-Learn, “Selecting the Number of Clusters with Silhouette Analysis on KMeans Clustering,” *scikit-learn*. [Online]. Available: https://scikit-learn/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html

[21] Scikit-Learn, "1.9. naive Bayes." [Online]. Available: https://scikit-learn/stable/modules/naive_bayes.html

[22] Scikit-Learn, "sklearn.naive_bayes.MultinomialNB," *scikit-learn*. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.naive _bayes.MultinomialNB.html

[23] Scikit-Learn, "sklearn.metrics.accuracy_score," *scikit-learn*. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.metrics.accuracy_score.html

[24] Scikit-Learn, "3.3. Metrics and scoring: quantifying the quality of predictions," [Online]. Available: *scikit-learn*. https://scikit-learn/stable/modules/model_evaluation.html

[25] D. Chicco, V. Starovoitov, and G. Jurman, "The benefits of the Matthews correlation coefficient (MCC) over the diagnostic odds ratio (DOR) in binary classification assessment," *IEEE Access*, vol. 9, pp. 47112–47124, 2021 [Online]. Available: doi: 10.1109/ACCESS.2021.3068614.

[26] Scikit-Learn, "sklearn.metrics.matthews_corrcoef," *scikit-learn*. [Online]. Available: https://scikit-learn/stable/modules/generated /sklearn.metrics.matthews_corrcoef.html

[27] R. M. Gray, *Entropy and Information Theory*. New York, NY, USA: Springer Science & Business Media, 2013.

[28] W3Schools, "JavaScript reserved words." [Online]. Available: https://www.w3schools.com/js/js_reserved.asp

[29] "JavaScript events and event handlers," *Matt Doyle | Elated Communications*, Dec. 11, 2001 [Online]. Available: https://www.elated.com/events-and-event-handlers/

[30] D. Flanagan, *Javascript: The Definitive Guide: Master the World's Most-Used Programming Language*. Sebastopol, CA, USA: O'Reilly Media, 2020.

[31] SciPy, "scipy.stats.zscore-SciPy v1.7.1 manual." [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.zscore.html

[32] "K-means application," class notes for Master PYTHON for data science | 12+ Projects, Udemy, 2021. [Online]. Available: https://www.udemy.com /course/draft/3835982/learn/lecture/25040834?start=0

[33] M. Chaudhary, "Silhouette analysis in K-means clustering," Medium, Jun. 05, 2020 [Online]. Available: https://medium.com/@cmukesh8688/silhouette-analysis-in-k-means-clustering-cefa9a7ad111

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California